Azure Data Factory (Advanced)

- Continuous Integration and Delivery with Azure DevOps
- Pipeline templates
- ARM Templates
- Integrating ADF with Azure Key Vault
- Copy Data activity
- Leveraging ADF Managed Identity to simplify access to other Azure resources
- Copy + Delete activities to archive or move files
- Databricks integration
  - ℹ️ ADF data flows are built on top of Azure Databricks
  - ADF data flows provide an easy-to-use GUI for developing ETL pipelines, whilst Databricks provides notebooks that you can use to develop ETL pipelines using code
  - ℹ️ You can call Azure Databricks notebooks from ADF pipelines to perform your ETL (i.e. the transformations)
  - 💡 The advantage of using Databricks instead of ADF data flows is sheer flexibility due to the code-based approach
    - 💡 Any time you can't get the job done using ADF data flows, you can almost certainly get it done using Databricks
  - Example: Databricks notebook to load multiple csv files (with same schema) into Spark dataframe and then save dataframe into JSON format, targeting directory with name specified via Databricks input parameter, and demonstrate calling this notebook from ADF pipeline, setting the input parameter in the process
    - In Databricks, we create a notebook
      - Cell 1 uses dbutils.widgets to create a string input parameter named Table and then set a local variable named table_name to that parameter value
      - Cell 2 sets up some Spark config so that the notebook can connect to ADLS Gen2 storage account using the access key (blacked out in screenshot)
        - More secure practice is to use secrets for the access key (i.e. Azure Key Vault or Databricks secret scope)
        - We also test out our connection to ADLS Gen2 by issuing a dbutils.fs.ls command
      - Cell 3 calls spark.read with some options to load the csv files in from the data lake, storing it in a variable, df, and then displaying the resultant dataframe
      - Cell 4 invokes the write.json method of the dataframe, appending the table_name variable to the output path
    - Test run the notebook in Databricks
      - 💡 Note that we set the Table parameter to "aaaa" for this test run
    - Check result in ADLS Gen2 using Storage Explorer
      - 💡 In this example, it ingested 21 csv files into a Spark dataframe and exported that dataframe as 6 JSON files, grouped logically in a Table folder labelled "aaaa"
    - Create a new Databricks access token
      - 💡 Copy the access token when it displays - if you close the window, you'll have to re-generate the access token
    - Create a new ADF pipeline and add a parameter for Table
      - In this example, we set the default value to "InternetSales_JSON"
    - Add a new pipeline, and drag on a Databricks Notebook activity
      - Create a new Databricks linked service and paste in the access key
        - I chose "Existing interactive cluster" here for speed but normally you would choose "New job cluster"
        - Pasting in the access key to the linked service config is not best practice - better to use Azure Key Vault
      - Under Settings, browse to your Databricks notebook and then add a parameter that you name Table and map via the pipeline parameter of same name
      - Debug the pipeline and verify that it completes successfully
        - Using Storage Explorer, verify that the "InternetSales_JSON" input parameter was used to capture the JSON files produced by the Databricks notebook
- Mapping data flows
- Dynamic content
- Alerts
- Wrangling data flows
- Log Analytics integration
- Pipeline execution

Azure Data Factory (Advanced)

1. Continuous Integration and Delivery with Azure DevOps

1.1. Azure DevOps is hosted at dev.azure.com

Link: https://dev.azure.com/

Link: https://dev.azure.com/

1.2. Within your DevOps account, you can create multiple organisations and within each organisation you can create multiple projects

1.2.1. When working on a real world ADF project, you will typically be working within a particular DevOps organisation and project

1.2.2. I have an organisation named argento-it-training and a project named Argento Training

1.2.3. As soon as you create a new project, a new Git repo is created, named after the project, and you have configuration options for the repo via Project Settings

1.3. In ADF, in the Manage page, you can link (and unlink) the data factory to your DevOps Git repo

1.3.1. By default, the branch where your changes will be committed is the master branch (as by default this is set as the "collaboration branch")

1.3.1.1. This is probably the wrong choice in most projects, and it will better to create feature branches for your own work

1.3.2. The adf_publish branch is just another branch to Git but in the ADF development environment, it's reserved for publishing changes to your data factory in the designated Azure development environment, and you can see it's greyed out in ADF so you can't link your data factory work directly to this

1.3.2.1. The adf_publish branch contains the ARM templates that represent the latest published version of the data factory you've designated for your development environment

1.3.3. You should only link the data factory designated for your Dev environment to source control - other data factories that represent QA/UAT/PROD should all be unlinked as they will get their changes from Azure DevOps CI/CD release pipelines

1.3.4. It is important to understand the difference between saving and publishing in ADF, as saving changes only becomes an option after you have linked your ADF instance to Git

1.3.4.1. Save (or Save All) allows you to make any kind of change and commit it to your chosen Git branch

1.3.4.1.1. This allows you to commit development work as you go without needing to complete all changes necessary for publishing a successful change

1.3.4.1.2. It also means that changes that are saved but not yet published are ignored when you run your pipelines (either via debug or via a trigger)

1.3.4.1.3. For example, we can make an edit to a pipeline and save the change

1.3.4.1.3.1. After saving the change we can see the change in the associated Json file in our feature branch

1.3.4.1.3.2. The change is not yet in our master branch because we didn't yet process a pull request from the feature to master branch

1.3.4.1.3.3. We can also compare diffs between our feature branch and master branch in DevOps

1.4. If you try to publish changes in ADF but are on a non collaboration branch, you will be prevented from doing so

1.4.1. There is only ever one branch designated for collaboration and by default it is the master branch

1.4.2. If you click the link to "merge the changes to master", this will automatically launch an Azure DevOps master branch pull request in a new browser tab

1.4.2.1. Once merge request is completed successfully, you can switch back to ADF change to your collaboration branch (e.g. master) and then you can publish your changes

1.4.2.2. You can also process the pull request directly from DevOps; there is no need to rely on clicking this link from ADF

1.4.2.3. You can also easily reverse out your changes here if you want by swapping the direction of pull request to be: master->feature

1.5. When you publish your changes in ADF (from your freshly merged collaboration branch) this will trigger a refresh of the ARM templates that are then automatically committed into your adf_publish branch

1.6. Another approach you can use is to capture individual ADF objects (such as pipelines) as Json files and upload the file using Azure DevOps (via a branch commit)

1.6.1. First thing to note is that every object in ADF can be serialised to Json using the Code button { } in the top right area of screen

1.6.1.1. You can copy the Json and save to file, noting that to be valid in ADF, the file must be named exactly the same as the name property with a .json file extension

1.6.1.1.1. If the filename and name property are misaligned and you upload to ADF, the pipeline will be flagged as being in error and won't be usable

1.6.2. Example: upload a pipeline to ADF via DevOps

1.6.2.1. Note that before starting, there is a single existing pipeline in ADF for our feature branch

1.6.2.2. In DevOps, we upload the pipeline Json file to the feature branch

1.6.2.2.1. The new file is committed to the feature branch

1.6.2.3. Back in ADF, after a refresh, we now have the newly added pipeline

1.7. DevOps Release Pipelines

1.7.1. A typical release pipeline for ADF will take two inputs, have a continuous deployment trigger tied to updates made to the adf_publish branch and include at least one stage that invokes one or more Azure tasks that get the data factory published to another ADF that represents a downstream environment

1.7.1.1. Inputs are the ARM templates in the adf_publish branch and a parameters file

1.7.1.1.1. Parameters file is for controlling the variables that are particular to an environment (QA/TEST/PROD, etc.)

1.7.1.2. Typical tasks include:

1.7.1.2.1. Azure Key Vault

1.7.1.2.2. Azure PowerShell

1.7.1.2.3. Azure resource group deployment

1.7.1.2.4. Azure Functions

1.7.2. Setting up a release pipeline for data factory

1.7.2.1. Our starting point is two data factories provisioned in Azure, one for Dev and one for QA

1.7.2.1.1. Only Dev data factory should be configured for Git integration

1.7.2.2. In DevOps, we create a new release pipeline, selecting "Empty Job" rather than a template

1.7.2.3. Add an artifact to the release pipeline, setting the source to be an Azure repo, and pointing it to the adf_publish branch

1.7.2.3.1. Think of artifacts as the pipeline source

1.7.2.3.2. Remember that the adf_publish branch holds the complete set of ARM templates required to deploy the entire Dev data factory, updated whenever we publish from the collaboration branch

1.7.2.4. Add a trigger to the artifact, enabling continuous automated deployment whenever a commit is made to adf_publish branch

1.7.2.5. Add a new task to the stage

1.7.2.5.1. Choose ARM template deployment

1.7.2.5.1.1. Authorize agent as Contributor for subscription

1.7.2.5.1.1.1. This adds a service principal to AAD and assigns it to Contributor RBAC role at subscription level

1.7.2.5.1.1.2. Note: for tighter security, you can restrict the Contributor scope of the agent down to a particular resource group

1.7.2.5.1.1.3. Set subscription, action, resource group and location

1.7.2.5.1.1.3.1. Action should be create or update resource group

1.7.2.5.1.1.3.2. Resource group should be the one that holds the QA data factory ???

1.7.2.5.1.1.3.3. Specify "Linked artifact" as template location and pick the ARMTemplateForFactory.json file from the adf_publish branch

1.7.2.5.1.1.3.3.1. Specify location for the template parameter file

1.7.2.5.1.1.3.3.1.1. Note: ARMTemplateParameterForFactory.json is automated parameter file that represents our Dev data factory - best practice would be to prepare separate parameter files per environment and commit these manually into adf_publish

1.7.2.5.1.1.3.3.1.2. In absence of separate pre-committed parameter files for each environment, we will need to override some parameter values from those tracking the Dev environment

1.7.2.5.1.1.3.3.1.2.1. Set Deployment mode to Incremental and click Save

1.7.2.5.1.1.3.3.1.2.1.1. If you choose Complete, it will wipe out everything in the target and re-deploy from scratch

1.7.2.5.1.1.3.3.1.2.1.2. You get a prompt on Save for a Folder and Comment - the Folder is mandatory and defaults to "\" (for root), the Comment is optional - I left both unchanged and clicked OK

1.7.2.6. Rename the stage from "Stage 1" to something that is more descriptive

1.7.2.6.1. e.g. QA Release

1.7.2.7. Set pre-deployment conditions for the stage

1.7.2.7.1. Note that the "After release" trigger means that stage deployment activity will automatically commence after the upstream artifacts trigger has fired

1.7.2.7.1.1. When combined with an enabled continuous deployment trigger on the artifacts, this makes for a fully automated, hands free deployment

1.7.2.7.2. In the real world, you may have stages for more controlled environments like Pre Prod and Prod, and in these cases you will probably choose the deployment to be manual and/or to have pre approvals before the deployment proceeds

1.7.2.8. Rename the pipeline from "New Release Pipeline" to something that is more descriptive

1.7.2.8.1. e.g. Argento_DataFactory_Release

1.7.3. Create a manual release

1.7.3.1. A release can be created in two ways:

1.7.3.1.1. via the continuous deployment trigger, if enabled on the artifacts section of the release pipeline

1.7.3.1.2. via a manual trigger

1.7.3.2. Ensure you've selected your release pipeline and click Create Release button

1.7.3.2.1. Optionally add description for release and click Create button

1.7.3.3. All being well, the release will succeed

1.7.3.3.1. First time I tried, it failed due to me misnaming the target data factory name parameter value

1.7.3.3.1.1. The logs help to diagnose the problem

1.7.3.3.1.2. After fixing, just create a new release

1.7.3.3.1.2.1. Note: fixing and then attempting to re-deploy the same failed release does not seem to work as it appears problematic parameter values are baked into the release

1.7.3.4. Check the target data factory to verify the changes are successfully deployed

1.7.4. Automated release

1.7.4.1. According to the way we configured the continuous deployment trigger and pre deployment conditions of our release pipeline, we can expect automated release from Dev to QA data factory whenever we publish a change in Dev

1.7.4.2. Imagine we have a change in our Dev data factory that is currently committed only to our feature branch

1.7.4.2.1. In DevOps, we create a pull request from our feature branch

1.7.4.2.1.1. The pull request is from our feature to our master branch (i.e. where master is the designated collaboration branch in the ADF Git configuration)

1.7.4.2.1.1.1. Complete pull request

1.7.4.2.2. In ADF, we now see the change (a new pipeline in this case) merged to the master branch, and we can click the Publish button

1.7.4.2.2.1. The pending changes are summarised for us to review and we click OK to accept and continue with the publication

1.7.4.2.3. In DevOps, we see that a new release from our release pipeline is automatically created and deployment in progress

1.7.4.2.4. In ADF, we now see the change in our QA data factory, automatically deployed there by our DevOps release branch

1.7.5. You can also re-deploy old releases, which is useful when a later release deployed a change that you want to undo

1.7.5.1. As our configured action for the ARM Template deployment task in the DevOps release pipeline is "Create or update" this will not result in deleting any unwanted data factory components

1.7.5.1.1. To delete an unwanted component in our downstream data factory, this can be done manually using eh ADF web interface or you could have a DevOps release pipeline with the ARM Template deployment task configured with the "Delete" action

1.7.6. Create individual parameter files for each data factory environment

1.7.6.1. Rather than configure the ARM Template deployment task in each stage (where stage typically has a 1-to-1 relationship with environment) to use the development parameter file + override values, it is better to set up separate parameter files for each environment and commit these directly into the adf_publish branch

1.7.6.2. From DevOps, you can download a copy of the development ADF parameter file

1.7.6.2.1. Once downloaded, rename file to put an environment suffix on

1.7.6.2.1.1. e.g. ARMTemplateParametersForFactory_QA.json

1.7.6.2.2. Edit the parameter file in Notepad++ and then upload/commit new file directly into adf_publish branch in same location as the development version

1.7.6.2.2.1. Note that if your release branch is configured for continuous deployment with no pre deployment conditions, the upload/commit action will trigger automated releases

1.7.6.2.2.1.1. Such releases should generally be harmless, but just to note that this can happen quite easily, so you have the option to temporarily disable the continuous deployment trigger on the release pipeline whilst you upload the environment specific parameter files

1.7.6.2.3. Once you have dedicated parameter files available in adf_publish, you can choose these in the ARM Template deployment task and remove parameter overrides

1.7.6.2.3.1. I imagine that one downside of this approach is the need to keep manually extending every copy of these separate parameter files as more and more components get added to your data factory in Dev

2. ARM Templates

2.1. ARM = Azure Resource Manager

2.2. Provides means of developing infrastructure as code

2.2.1. Templates all stored in JSON format

2.3. Although Azure DevOps offers a very good way to deploy ARM templates, there are other methods too

2.3.1. Importing ARM templates directly via Azure Portal will trigger a deployment

2.3.2. ARM templates can be deployed via PowerShell

2.4. Deploy ARM templates via Export->Import method

2.4.1. In ADF, click ARM template menu and choose the export option

2.4.2. Unzip the ARM template

2.4.2.1. You get the main ARM template that includes everything you need to deploy a copy of the data factory somewhere else and a paired parameter file that holds values particular to your data factory that you want to change when deploying to different targets

2.4.2.2. Every ARM template has a common JSON structure, consisting of a sequence of embedded objects, only some of which are mandatory - a typical ARM template for a data factory deployment consists of:

Link: https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/template-syntax

Link: https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/template-syntax

2.4.2.2.1. $schema

2.4.2.2.1.1. Mandatory. Location of the JSON schema file that describes the version of the template language. The version number you use depends on the scope of the deployment and your JSON editor.

2.4.2.2.2. contentVersion

2.4.2.2.2.1. Mandatory. Version of the template (such as 1.0.0.0). You can provide any value for this element. Use this value to document significant changes in your template. When deploying resources using the template, this value can be used to make sure that the right template is being used.

2.4.2.2.3. parameters

2.4.2.2.3.1. Optional. Values that are provided when deployment is executed to customize resource deployment.

2.4.2.2.4. variables

2.4.2.2.4.1. Optional. Values that are used as JSON fragments in the template to simplify template language expressions.

2.4.2.2.5. resources

2.4.2.2.5.1. Mandatory. Resource types that are deployed or updated in a resource group or subscription.

2.4.3. In ADF, deploy a new empty data factory and then click ARM template menu and choose the import option

2.4.3.1. This kicks off a custom deployment

2.4.3.2. Click hotlink "Build your own template in the editor"

2.4.3.2.1. Click Load file, point to your ARM template file and then Save

2.4.3.2.1.1. Fill in the values for the parameters and then create deployment

2.4.3.2.1.1.1. Note: the parameters marked with * are required and sensitive, so the values are obfuscated in the UI when entered

2.4.3.2.1.1.2. I copied and pasted from the code { } representations of the data factory components in my working upstream (Dev) data factory, which seemed to work in that each parameter value passed validation

2.4.3.2.1.1.3. I got a deployment error regarding my self hosted integration runtime, which apparently needs separate registration and permission sharing from one data factory to the next

2.4.3.2.1.1.3.1. This highlights that it's not so straightforward to deploy data factories via ARM templates without considering the linked services and integration runtimes that will be used by the downstream data factory

2.5. You can create ARM templates from scratch as well (i.e. not based on Azure development + export as ARM template)

2.5.1. A great starting point for this is Azure Quickstart Templates

Link: https://azure.microsoft.com/en-gb/resources/templates/

Link: https://azure.microsoft.com/en-gb/resources/templates/

2.5.1.1. If you search "data factory", you'll see a bunch of ARM templates for various data factory deployment scenarios

2.5.2. ARM templates can be developed in VS Code or Visual Studio

3. Copy Data activity

3.1. SQL Database auto table creation

3.1.1. Copy activity allows you to have ADF auto-create a table in SQL Database sink

3.1.1.1. This means you can have delimited file data load into a SQL staging table without needing to predefine schema on either the source or sink of the Copy data activity

3.1.1.2. When the pipeline runs again and the SQL table already exists, it will simply append data to the table (assuming you don't add a separate stored procedure activity to truncate it)

3.1.1.3. It won't cope with schema drift though

3.1.1.3.1. If the source file gains a schema change, this feature will not add, modify or drop columns in the SQL Database table, so you'll get an error

3.1.1.3.2. However, if the load pattern was source->stage->load repeated in that sequence, and the auto-created table was for staging, you could add a stored procedure activity to drop the table if exists before the copy activity, and that would work as an automated schema drift solution of sorts

3.1.2. Example: Load files on wildcard pattern match from ADLS Gen2 to SQL Database

3.1.2.1. Create a new dataset for the source based on Azure Data Lake Storage Gen2 and Delimited File

3.1.2.1.1. When setting the properties of the dataset, specify only the file path, not the actual file

3.1.2.1.1.1. Remember also to check the First row as header option

3.1.2.2. Create a new pipeline with a copy activity

3.1.2.2.1. When setting the Source properties, pick the dataset previously created and configure a wildcard pattern

3.1.2.2.1.1. File path type = Wildcard file path

3.1.2.2.1.2. Wildcard paths = "test-files/../*employee*

3.1.2.2.1.3. Recursively = off

3.1.2.2.1.3.1. When on, this will apply the wildcard file pattern to recursive to every sub-directory found in the container/directory specified

3.1.2.2.1.3.2. In this example we don't need recursive, so we can turn it off

3.1.2.2.2. When creating a new dataset for the sink, you can edit the table name manually and defer the import schema to allow ADF to create it on the fly

3.1.2.2.3. When setting the Sink properties, select Table option = Auto create table

3.1.2.3. Ensure the source file is uploaded to ADLS (see attached sample of file)

3.1.2.4. Debug the pipeline

3.1.2.4.1. If successful, you should see data loaded into newly created SQL Database table

3.1.2.4.1.1. Note how the text file source results in a cautious SQL table schema with all columns set to nvarchar(max)

4. Dynamic content

4.1. Key to more advanced usage of ADF

4.1.1. Parameters are fundamental

4.1.1.1. You can parameterise linked services, datasets, pipelines and data flows

4.1.1.2. Parameter values are passed in at run time (i.e. when pipeline is triggered)

4.1.2. Expressions use parameters and variables to create flexible and dynamic configuration

4.1.2.1. Within the expression editor, you have access to functions, parameters, variables (system and user), arrays (list objects) and metadata (e.g. capturing error messages, rows read, etc.)

4.1.3. Variables are different to parameters but also fundamental for enabling dynamic configuration

4.1.3.1. Unlike parameters that are set externally and passed in at run time and do not change throughout that run time, variables exist within the context of the run time and can change throughout it

4.2. Dataset parameterisation

4.2.1. Wherever you see the "Add dynamic content" prompt in a dataset property, this is a candidate for setting via a parameter

4.2.2. We can add a parameter to hold the file name of a file-based dataset for example

4.2.2.1. The default value for the parameter can be set at run time, which will typically take its value from a separate pipeline parameter

4.2.3. Once we have dataset parameters, we can use them to set dataset properties where dynamic content is supported

4.2.3.1. Once set, the dataset property holds a value that represents the dataset parameter, based on the notation of the ADF expression language

4.3. Dynamic content helps us to implement design patterns in a flexible, re-usable way

4.3.1. Example design pattern: If File Exists

4.3.1.1. Based on the Until activity, which runs a sequence of sub activities (Get Metadata, Set variable and Wait) until a variable (set by the sub activity) changes from false to true

4.3.1.2. 1. Create new pipeline and add Boolean variable for tracking file existence, defaulting it to false

4.3.1.2.1. In this case we named the variable fileexists

4.3.1.3. 2. Add Until activity and under Settings, set the Expression to the fileexists pipeline variable

4.3.1.3.1. The expression is set for you - note that you could type it in directly using the expression language syntax but the dynamic content editor makes building expressions easier

4.3.1.3.2. By design, the Until activity will run its activities (not yet defined) in a loop until the expression evaluates to true

4.3.1.3.2.1. By defaulting the fileexists expression to false, we are assured of running the Until activities at least once

4.3.1.3.2.2. The Until activities must include something that will change the value of the variable to true under some condition(s)

4.3.1.4. 3. Edit the Until activities

4.3.1.4.1. 3.1 Add Get Metadata activity and configure the Dataset properties

4.3.1.4.1.1. Pick the dataset that represents the file (typically a wildcard pattern) that we want to check for

4.3.1.4.1.1.1. We can remind ourselves that the filename of the dataset is set using a parameter with a wildcard value

4.3.1.4.1.1.1.1. The Connection file path property includes the dynamic expression, referencing the dataset parameter with the syntax @dataset.<ParameterName>

4.3.1.4.1.2. Add new item to field list and select the Exists argument

4.3.1.4.1.2.1. The Exists argument holds Boolean values and is added as an output of the Get Metadata activity

4.3.1.4.2. 3.2 Add Set variable activity, connect the output of Get Metadata to the Set variable input and select the pipeline variable (fileexists)

4.3.1.4.2.1. In the expression builder, you select the Get Metadata activity output and append the specific argument we configured previously

4.3.1.4.2.1.1. Selecting the Get Metadata activity output gives you an incomplete expression in the following format:

4.3.1.4.2.1.1.1. @activity('<Name of activity').output

4.3.1.4.2.1.2. You must manually append the Get Metadata argument that the activity was configured to output using dot notation

4.3.1.4.2.1.2.1. In this case we must append .exists

4.3.1.4.2.1.2.2. The final expression is therefore:

4.3.1.4.2.1.2.2.1. @activity('<Name of activity').output.exists

4.3.1.4.3. 3.3 Add Wait activity, connect the output of Set variable to the Wait input and set the Wait time in seconds

4.3.1.4.3.1. Whilst we can hard code the time in seconds to wait, I opted here to use the expression builder, leveraging the if() function

Link: https://docs.microsoft.com/en-us/azure/data-factory/control-flow-expression-language-functions#if

Link: https://docs.microsoft.com/en-us/azure/data-factory/control-flow-expression-language-functions#if

4.3.1.4.3.1.1. The syntax is @if(<boolean expression>,<value if true>,<value if false>)

4.3.1.4.3.1.2. As our fileexists variable is typed as Boolean, we can simply reference that for the expression, then set value to 0 for true to indicate not to wait, otherwise wait 10 seconds

4.3.1.4.3.1.2.1. The final expression is therefore:

4.3.1.4.3.1.2.1.1. @if(variables('fileexists'),1,10)

4.3.1.4.3.2. Screenshot shows I set wait time to 0 seconds but I discovered that minimum value is 1, and having value of 0 causes an error

4.3.1.5. Debug the pipeline

4.3.1.5.1. Initially you should see the pipeline iterating through the child activities of the Until activity, and you can check the output of the Set variable task for fileexists

4.3.1.5.1.1. The output shows that fileexists is false

4.3.1.5.2. Part way into the debug session, we upload a file with a name that the pipeline is configured to wait for

4.3.1.5.3. The pipeline now completes successfully and when we check the output of the final Set variable activity, we see that fileexists was set to true

4.3.1.5.4. The Get Metadata activity does not support filename wildcards

4.3.1.5.4.1. Using wildcards (as I initially did when first trying this) does not cause an error but it results in the Until activity repeating endlessly even when you upload a single file that matches the wildcard pattern

4.3.1.5.4.2. To fix this I had to edit the Get Metadata activity and explicitly set the name of the file

4.3.1.5.4.2.1. Further investigation needed on how to check for file existence in scenarios where you don't know the exact filename (e.g. when file name includes timestamps and/or batch sequence numbers)

4.3.1.5.4.2.1.1. Maybe using a ForEach activity that nests the Until activity might be an option

4.3.2. Example: count Until activity loop iterations and exit loop when counter is X

4.3.2.1. This technique is based on variables but requires a hack due to two limitations:

4.3.2.1.1. Integer variable types are not supported: only string, boolean and array

4.3.2.1.2. Expression language does not support variable self referencing, so it's not possible to have a variable value set via direct reference to itself (i.e. x += 1 pattern is not allowed)

4.3.2.1.3. To work around these limitations, we need to use two string variables, each set using two separate Set variable activities, and then we must leverage type conversion functions in the expressions

4.3.2.2. Add two variables to the pipeline containing the Until activity and set the default values to 0

4.3.2.2.1. In this example, I set the names to retries and retriesTemp, the types to String (because the only other options are Boolean and Array) and a default value of 0 for both

4.3.2.2.2. Having two variables instead of one is due to the limitation of the ADF expression language not allowing retries to be updated via a self reference (i.e. retries = retries + 1)

4.3.2.2.3. Modify the Until expression to use the or() function and add the expression for retries, which includes use of the greaterOrEquals() function and the int() function

4.3.2.2.3.1. The expression is:

4.3.2.2.3.1.1. @or(variables('fileexists'),greaterOrEquals(int(variables('retries')),5))

4.3.2.2.3.2. In this example, I set the exit point for the Until loop at retries >= 5

4.3.2.2.3.3. Edit the Until child activities, add a Set variable activity for retriesTemp and set its value via expression leveraging the add(), int() and string() functions

4.3.2.2.3.3.1. The expression is:

4.3.2.2.3.3.1.1. @string(add(int(variables('retries')),1))

4.3.2.2.3.3.2. Note how the retries variable is converted to an integer by the int() function, then we increment its value by 1 using the add() function and finally convert back to a string

4.3.2.2.3.3.3. Add a second Set variable activity for the retries variable, connecting its input to the previous activity's output, and set the value simply as the value of the retriesTemp variable

4.3.2.2.3.3.3.1. The expression is:

4.3.2.2.3.3.3.1.1. @variables('retriesTemp')

4.3.2.2.3.3.3.2. Ensure there is no file existing (so that the variables('fileexists') part of the condition is not met and then debug the pipeline, noting how it increments the retries variable and completes after reaching the configured value (5)

4.4. Parent Child design pattern

4.4.1. The idea here is to have a parent pipeline that looks up metadata from a SQL Database source in the form of a list of target tables to be loaded and uses a ForEach activity to iterate the list, calling a common dynamic pipeline to handle the processing for each table

4.4.1.1. Parameters form the basis of this design

4.4.1.2. We want a parent pipeline that uses a ForEach loop to iterate over a table name list retrieved from a SQL Database

4.4.1.2.1. The child pipeline has parameters that will take different values on each iteration of parent pipeline ForEach loop activity

4.4.1.2.2. The dataset has parameters that will take their values from the child pipeline

4.4.2. This design involves a number SQL Database metadata objects, which are generic enough to re-use in many scenarios

4.4.2.1. Tables

4.4.2.1.1. dbo.ADFErrorLog

4.4.2.1.1.1. For capturing details of errors arising from ADF pipeline executions

4.4.2.1.2. dbo.PipelineExecutionHistory

4.4.2.1.2.1. For capturing details of ADF pipeline runs, including metrics like number of files processed, number of rows read and written, duration of copy activity, etc.

4.4.2.1.3. dbo.TaskQueue

4.4.2.1.3.1. For holding a list of tables to be processed as part of a batch load, tracking the processed date and status

4.4.2.2. Stored procs

4.4.2.2.1. dbo.usp_ErrorLog

4.4.2.2.1.1. Inserts a record into dbo.ADFErrorLog using input parameters

4.4.2.2.2. dbo.usp_GetTablesToProcess

4.4.2.2.2.1. Returns a list of tables (schema name + table name as two separate columns) from dbo.TaskQueue that are not yet processed

4.4.2.2.3. dbo.usp_PipelineExecutionHistory

4.4.2.2.3.1. Inserts a record into dbo.PipelineExecutionHistory using input parameters

4.4.2.2.4. dbo.UpdateTaskQueue

4.4.2.2.4.1. Updates a record in dbo.TaskQueue identified via input parameters (for schema and table name), setting the processed indicator and date

4.4.3. Example: load multiple files dynamically from ADLS Gen2 to SQL Database tables

4.4.3.1. To keep things simple, I've prepared two very basic test files, both flat text files with a pipe delimiter

4.4.3.1.1. argento_employees.txt

4.4.3.1.2. orders_20201121.txt

4.4.3.1.3. The files are uploaded to ADLS Gen2 using Azure Storage Explorer

4.4.3.2. Create child pipeline, add Copy activity and add 3 parameters to the child pipeline

4.4.3.2.1. SourceFilenamePattern

4.4.3.2.1.1. String value that will be yielded from the SQL procedure, dbo.usp_GetTablesToProcess and will be used to pair a source file dynamically with a SQL table

4.4.3.2.2. Schema

4.4.3.2.2.1. String value that will be yielded from the SQL procedure, dbo.usp_GetTablesToProcess and will be used to dynamically identify the target SQL Database schema

4.4.3.2.3. TableName

4.4.3.2.3.1. String value that will be yielded from the SQL procedure, dbo.usp_GetTablesToProcess and will be used to dynamically identify the target SQL Database table

4.4.3.3. Create dynamic dataset for the source file

4.4.3.3.1. Set only a file path, not a file name, check the first row as header option and set Import schema = None

4.4.3.3.2. Add string parameter to the dataset, SourceFilenamePattern

4.4.3.3.2.1. No need to set any values for these as they are intended to be populated by the child pipeline

4.4.3.3.3. Under Connection, set the column delimiter to pipe and set the file name property via dynamic expression that pulls in the SourceFilenamePattern parameter value (i.e. the dataset parameter)

4.4.3.4. Create dynamic dataset for the target table

4.4.3.4.1. When initially creating the new dataset, don't specify a table and ensure Import schema = None

4.4.3.4.2. Add two string parameters to the dataset: Schema and TableName

4.4.3.4.2.1. No need to set any values for these as they are intended to be populated by the child pipeline

4.4.3.4.3. Under Connection, check the Edit option for the table name and set schema and table name using dynamic expressions that reference the two dataset parameters: Schema and TableName

4.4.3.5. Create parent pipeline

4.4.3.5.1. Add a Lookup activity and set it to return its content via the stored procedure, dbo.usp_GetTablesToProcess

4.4.3.5.1.1. Note that the dataset does needs to arbitrarily point to any table in the SQL Database that hosts the proc, but I think it is most logical to point it to the dbo.TaskQueue table

4.4.3.5.1.2. Ensure the First row only option is unchecked

4.4.3.5.2. Add a ForEach activity and set the Items via dynamic expression, selecting the lookup activity output and appending ".Value"

4.4.3.5.2.1. The final expression for the ForEach Items is:

4.4.3.5.2.1.1. @activity('01_Lookup_Task_Queue').output.Value

4.4.3.5.3. Add an Execute Pipeline child activity to the ForEach activity, point it to the child pipeline and configure the parameters using dynamic expressions

4.4.3.5.3.1. Each iteration of the ForEach activity yields an array of 3 string values, the names of which are determined by the dbo.usp_GetTablesToProcess stored procedure in this case

4.4.3.5.3.1.1. Because the ForEach activity is yielding an array of "items", we must encapsulate the item() function in a pair of curly braces { } , where these braces denote an array

4.4.3.5.3.1.2. For each item() call within the array, we must specify the field name yielded by the proc (which forms the Lookup activity output into the ForEach activity, and each iteration of ForEach yields 1 row (i.e. a single dimension array of 3 string values)

4.4.3.5.3.2. The Wait on completion option, when checked, causes the triggered pipeline to run synchronously and block execution of subsequent tasks until its finished

4.4.3.5.3.2.1. In this context, with the Execute Pipeline activity being a child activity of the ForeEach activity, enabling this means we are specifying a sequential processing of each task in the task queue (i.e. one file load at a time)

4.4.3.5.3.2.1.1. In the real world, we will probably uncheck this option and let ADF run the ForEach tasks in parallel as this will deliver a potentially significant performance boost (assuming that the tasks in the queue have no dependencies on one another)

4.4.3.5.3.2.1.2. When running a pipeline via debug, the Wait on completion setting is disregarded and the ForEach child activities execute synchronously in serial fashion

4.4.3.6. Complete child pipeline configuration

4.4.3.6.1. Set the Source dataset to point to the dynamic source file dataset and set the SourceFilenamePattern value via dynamic expression that references the child pipeline parameter of same name

4.4.3.6.2. Set the Sink dataset to point to the dynamic target table dataset and set the Schema and TableName values via dynamic expressions that reference the child pipeline parameters of same name

4.4.3.6.2.1. Note: set Table option = Auto create table to have ADF auto create the target table if it doesn't already exist

4.4.3.7. Set up dbo.TaskQueue (i.e. the metadata for driving our parent-child pipeline)

4.4.3.7.1. But take care over the SourceFilenamePattern values!

4.4.3.8. Debug the parent pipeline

4.4.3.8.1. If all is well, you should see the parent pipeline successfully invoke the child pipeline twice

4.4.3.8.1.1. Gotcha! If you specify wildcard values for SourceFilenamePattern in dbo.TaskQueue, you will get a "no such file exists" error produced by the Copy activity

4.4.3.8.1.1.1. I remedied this by aligning the SourceFilenamePattern values exactly with the source file names (i.e. updated dbo.TaskQueue table)

4.4.3.8.1.1.2. In the real world, where source filenames often include a datetime component, I think we will need to use the Get metadata activity to help capture the filenames and build the content of dbo.TaskQueue more dynamically

4.4.3.8.2. We can verify that the source file content made it into the target tables in SQL Database

4.5. Source database to data lake with dynamic directory and file name

4.5.1. Imagine using data factory to extract data from a source system database and land these into a data lake where the directory structure is "raw/argento/yyyy/mm/dd"

4.5.2. We start by making our dynamic dataset for the ADLS Gen2 test delimited file even more dynamic

4.5.2.1. 1. Ensure the dynamic dataset has 3 parameters: ContainerName, DirectoryName and FileName

4.5.2.2. 2. Parameterise the container, directory and filename in the Connection file path settings of the dataset

4.5.3. Create a pipeline and add a Copy activity

4.5.3.1. 1. Add a pair of parameters to the pipeline for Schema and TableName

4.5.3.1.1. In this case we will demonstrate the pipeline via a single execution that relies on the parameter default values, which is just of simplicity because the focus of this technique is on the dynamic sink, but we can easily imagine this pipeline being invoked repeatedly as the child in the parent child design pattern

4.5.3.2. 2. Use the dynamic SQL Database dataset for the source

4.5.3.2.1. We set the Schema and TableName using the pipeline parameters

4.5.3.3. 3. Use the dynamic ADLS Gen2 delimited text file for the sink and set the 3 parameters using dynamic expressions

4.5.3.3.1. For FileName we use a combination of 3 functions: concat(), utcnow() and formatDate(), plus the TableName pipeline parameter

4.5.3.3.1.1. Expression is:

4.5.3.3.1.1.1. @concat(pipeline().parameters.TableName, '_', formatDateTime(utcnow(),'yyyy'), formatDateTime(utcnow(),'MM'), formatDateTime(utcnow(),'dd'), '.txt')

4.5.3.3.1.2. Expected result when TableName = 'orders' and current date is 23/11/2020:

4.5.3.3.1.2.1. orders_20201123.txt

4.5.3.3.2. For ContainerName I chose to set this statically to "raw"

4.5.3.3.3. For DirectoryName we use a combination of 3 functions: concat(), utcnow() and formatDate()

4.5.3.3.3.1. Expression is:

4.5.3.3.3.1.1. @concat(formatDateTime(utcnow(),'yyyy'), '/', formatDateTime(utcnow(),'MM'), '/', formatDateTime(utcnow(),'dd'))

4.5.3.3.3.2. Expected result when current date is 23/11/2020:

4.5.3.3.3.2.1. 2020/11/23

4.5.4. Debug pipeline

4.5.4.1. If all is well, you should see a successful result

4.5.4.2. Using Storage Explorer we can confirm that the requested date-based directory path has been created and the file is named after the table with a date suffix

4.6. Restartability

4.6.1. A key idea for restartability is the concept of the task queue, which can be implemented as a table in SQL Database

4.6.1.1. To demonstrate this concept, we'll build on the parent child concept by adding a pair of stored procedure activities to follow the child pipeline copy acitivity

4.6.1.1.1. As a first step, we'll tweak the dbo.usp_UpdateTaskQueue stored procedure to include @IsProcessed as an input parameter and no longer use ProcessedDate Is Null as a filter

4.6.1.1.1.1. The idea here is that IsProcessed will take on 3 possible values: 0 (unprocessed), 1 (processed successfully), 2 (processed unsuccessfully)

4.6.1.1.1.1.1. Downstream pipelines and associated task queues can be made dependent on an upstream task queue being fully cleared (i.e. IsProcessed = 1 across the board)

4.6.1.1.1.1.2. Task queue items flagged IsProcessed = 2 can be used as the source of alerts, prompting manual intervention and once resolved, the task queue item can be updated to IsProcessed = 0 and the pipeline restarted

4.6.1.1.2. We will also add an intentionally dodgy item into the dbo.TaskQueue table

4.6.1.1.2.1. Note that the table name, FactInternetSales has been mis-spelled in out task queue, which will cause an error in the child pipeline copy activity

4.6.1.1.3. Add a stored procedure task to child pipeline and connect to success output from the copy activity, then set to call the dbo.usp_UpdateTaskQueue proc, import the parameters and set the values

4.6.1.1.3.1. We set IsProcessed to 1 and TableName is set via dynamic expression that references the child pipeline parameter, as is TableSchema

4.6.1.1.4. Add a stored procedure task to child pipeline and connect to failure output from the copy activity, then set to call the dbo.usp_UpdateTaskQueue proc, import the parameters and set the values

4.6.1.1.4.1. We set IsProcessed to 2 and TableName is set via dynamic expression that references the child pipeline parameter, as is TableSchema

4.6.1.1.5. Debug parent pipeline

4.6.1.1.5.1. As expected, the execution of the child pipeline fails on the 3rd task queue item

4.6.1.1.5.2. When we check the dbo.TaskQueue table, we see that the first two items in the task queue are flagged as processed successfully and the third item unsuccessful

4.6.1.1.5.3. If we "restart" the parent pipeline, we see that it succeeds due to there no longer being any unprocessed items in the task queue

4.7. Custom logging

4.7.1. This solution is based on a SQL Database table for a custom log table (dbo.PipelineExecutionHistory) and another for logging errors (dbo.ADFErrorLog)

4.7.1.1. To demonstrate this concept, we'll build on the same parent child pipelines we used for restartability, by adding a couple of stored procedures to the child pipeline for updating our custom logging tables, leveraging metadata available from a combination of system variables and the outputs from the Copy activity

  4.7.1.1.1. As a first step we'll reset the dbo.TaskQueue items so that IsProcessed = 0 for all of them

    4.7.1.1.1.1. Remember that the 3rd item in the queue is dodgy and will trigger an error, which will allow us to test the dbo.usp_ErrorLog stored procedure

  4.7.1.1.2. Add a stored procedure task to child pipeline and connect to success output from the copy activity, then set to call the dbo.usp_PipelineExecutionHistory proc, import the parameters and set the values

    4.7.1.1.2.1. Set the following input parameters using system variables:

      4.7.1.1.2.1.1. DataFactory

        4.7.1.1.2.1.1.1. @pipeline().DataFactory

      4.7.1.1.2.1.2. Pipeline

        4.7.1.1.2.1.2.1. @pipeline().Pipeline

      4.7.1.1.2.1.3. PipelineRunID

        4.7.1.1.2.1.3.1. @pipeline().RunId

      4.7.1.1.2.1.4. TriggerType

        4.7.1.1.2.1.4.1. @pipeline().TriggerType

    4.7.1.1.2.2. Set the following input parameters using child pipeline parameters

      4.7.1.1.2.2.1. Destination

        4.7.1.1.2.2.1.1. @pipeline().parameters.SourceFilenamePattern

          4.7.1.1.2.2.1.1.1. Note: this is just poor naming of the pipeline parameter on my part - it would be better to be something like Filename

      4.7.1.1.2.2.2. Source

        4.7.1.1.2.2.2.1. @concat(pipeline().parameters.Schema,'.',pipeline().parameters.TableName)

    4.7.1.1.2.3. Set the following input parameters using Copy activity outputs

      4.7.1.1.2.3.1. Firstly, here's a nice tip for verifying the Copy activity output keys for mapping to your stored proc input parameters:

        4.7.1.1.2.3.1.1. Put a breakpoint on the Copy activity (which disables all downstream activities), click Debug, manually set the pipeline parameters values when prompted, and then view the output of the activity

- 4.7.1.1.2.3.2. CopyDuration
    - 4.7.1.1.2.3.2.1. @activity('01_Dynamic_Load').output.copyDuration
- 4.7.1.1.2.3.3. FilesRead
    - 4.7.1.1.2.3.3.1. @activity('01_Dynamic_Load').output.filesRead
- 4.7.1.1.2.3.4. RowsRead
    - 4.7.1.1.2.3.4.1. @activity('01_Dynamic_Load').output.rowsRead
- 4.7.1.1.2.3.5. RowsWritten
    - 4.7.1.1.2.3.5.1. @activity('01_Dynamic_Load').output.rowsCopied

4.7.1.1.2.4. Set the following input parameters to be treated as Null

- 4.7.1.1.2.4.1. This is because the error output will not exist when the Copy activity succeeds
- 4.7.1.1.2.4.2. ErrorCode
- 4.7.1.1.2.4.3. ErrorMessage

4.7.1.1.3. Add a stored procedure task to child pipeline and connect to failure output from the copy activity, then set to call the dbo.usp_ErrorLog proc, import the parameters and set the values

4.7.1.1.3.1. Set the following input parameters using system variables:

- 4.7.1.1.3.1.1. DataFactory
    - 4.7.1.1.3.1.1.1. @pipeline().DataFactory
- 4.7.1.1.3.1.2. Pipeline
    - 4.7.1.1.3.1.2.1. @pipeline().Pipeline
- 4.7.1.1.3.1.3. PipelineRunID
    - 4.7.1.1.3.1.3.1. @pipeline().RunId
- 4.7.1.1.3.1.4. TriggerType
    - 4.7.1.1.3.1.4.1. @pipeline().TriggerType

4.7.1.1.3.2. Set the following input parameters using child pipeline parameters

- 4.7.1.1.3.2.1. Destination
    - 4.7.1.1.3.2.1.1. @pipeline().parameters.SourceFilenamePattern
        - 4.7.1.1.3.2.1.1.1. Note: this is just poor naming of the pipeline parameter on my part - it would be better to be something like Filename
- 4.7.1.1.3.2.2. Source

4.7.1.1.3.2.2.1. @concat(pipeline().parameters.Schema,'.',pipeline().parameters.TableName)

4.7.1.1.3.3. Set the following input parameters using Copy activity outputs

4.7.1.1.3.3.1. Again, we can use a breakpoint to run the Copy activity in isolation, supplying a pair of parameter values that will trigger an error so we can check the details of the output relating to errors

4.7.1.1.3.3.1.1. Here we can see that that the errors value is an array and the two keys of interest are named Code and Message

4.7.1.1.3.3.2. ErrorCode

4.7.1.1.3.3.2.1. @first(activity('01_Dynamic_Load').output.errors).Code

4.7.1.1.3.3.2.1.1. Note: an alternative to using the first() function (one of several that works on array input) is to use [n] syntax:

4.7.1.1.3.3.2.1.1.1. @activity('01_Dynamic_Load').output.errors[0].Code

4.7.1.1.3.3.3. ErrorMessage

4.7.1.1.3.3.3.1. @first(activity('01_Dynamic_Load').output.errors).Message

4.7.1.1.4. Debug parent pipeline

4.7.1.1.4.1. As expected, the execution of the child pipeline fails on the 3rd task queue item

4.7.1.1.4.2. We see the two successful child pipeline executions logged in dbo.PipelineExecutionHistory

4.7.1.1.4.3. We see the unsuccessful child pipeline logged in dbo.ADFErrorLog

5. Wrangling data flows

5.1. Provides Power Query UI for building out your transformations, based on the M language

5.1.1. Same experience as you get in Power BI

5.2. Example: use a wrangling data flow to transform a csv file in ADLS Gen2, sinking the transformed file back into data lake

5.2.1. Upload raw csv file to data lake using Storage Explorer

5.2.1.1. In this example the file is called BMI.csv

5.2.2. In ADF, create a new data flow and choose Wrangling Data Flow for the type

5.2.2.1. Set source properties, pointing to the raw csv file in the data lake

5.2.2.1.1. Although screenshot shows data lake connection, I later changed to a Blob storage account due to an issue with my ADLS Gen2 linked service using managed identity authentication

5.2.2.2. Set sink properties, pointing to the target folder "transformed" in the data lake, but with no schema import (because the file does not exist yet)

5.2.2.2.1. Screenshot shows a Blob storage account, not data lake, which is due to my data lake linked service using managed identity authentication (unsupported at time of writing)

5.2.2.3. Click OK to create new wrangling data flow, and you will see Power Query editor (familiar UI for Power BI developers)

5.2.2.3.1. Gotcha! If your data lake linked service uses Managed Identity for authentication, you will get an error when creating the wrangling data flow, as at time of writing (30-Nov-2020) it only supports access key or service principal

5.2.2.4. Make some transformations and save data flow

5.2.2.4.1. In this example, I did the following:

5.2.2.4.1.1. Promoted top row to header

5.2.2.4.1.2. Selected first column and unpivoted other columns

5.2.2.4.1.3. Renamed columns (Country, Year, BMI)

5.2.2.4.1.4. Applied Trim to Country

5.2.2.4.1.5. Set Year as integer

5.2.2.4.1.6. Set BMI as decimal

5.2.2.5. Create a pipeline and add Data Flow activity

5.2.2.6. Debug pipeline

5.2.2.6.1. Gotcha! Many unsupported transformations, despite being available in the editor

Link: https://docs.microsoft.com/en-us/azure/data-factory/wrangling-data-flow-functions

Link: https://docs.microsoft.com/en-us/azure/data-factory/wrangling-data-flow-functions

5.2.2.6.1.1. But this will presumably reduce and resolve over time as Microsoft release more functionality and take this feature out of public preview

5.2.2.6.1.2. Promote top row to header unsupported

5.2.2.6.1.2.1. Workaround is to set the "First row as header" option in the source data set

5.2.2.6.1.3. Pivot and unpivot transformations unsupported

5.2.2.6.1.3.1. Don't know of a workaround for this so just lived with pivoted set and reduced the amount of transformations

5.2.2.6.2. Verify that pipeline debug run completes successfully

5.2.2.6.3. Verify transformations in Azure storage account

5.2.2.6.3.1. Note that transformations much more limited than I originally developed due to the gotcha of unsupported transformations

5.2.2.6.3.2. This would work better in a data lake where I can drop the sink output from the wrangling data flow into a named folder, rather than everything going into the same container top level as the source file

6. Log Analytics integration

6.1. Azure Log Analytics provides a means of centralising all of your Azure resource logging in one place and monitoring system health via interactive dashboard functionality

6.1.1. It is most suited to enterprise environments where you will likely have multiple different data factories, and keeping track of all of them is a challenge

6.2. At a high level, integrating ADF with Log Analytics is a 3-step process:

6.2.1. 1. Provision an Azure Log Analytics workspace

6.2.2. 2. Configure your data factory instance diagnostic settings to log all metrics and send to your Log Analytics workspace

6.2.3. 3. Provision an Azure Data Factory Analytics workspace and bind this to your Log Analytics workspace

6.3. Provision Azure Log Analytics workspace

6.3.1. In Azure Portal, create a new resource and choose Log Analytics

6.3.1.1. Set the subscription, resource group, name (of new Log Analytics workspace) and region

6.3.1.1.1. Set the pricing tier (e.g. Pay-as-you-go)

6.4. Configure Data Factory to send diagnostic log data to Log Analytics

6.4.1. Go to your data factory and under Diagnostic settings, click the "Add diagnostic setting" option

6.4.1.1. Log ActivityRuns, PipelineRuns and TriggerRuns for AllMetrics to your Log Analytics workspace

6.4.1.1.1. Note other possible destinations for your diagnostics logs are Azure storage account or event hub

6.4.1.1.2. Destination table should be Resource specific

6.5. Provision Azure Data Factory Analytics workspace

6.5.1. In Azure Portal, create a new resource and choose Data Factory Analytics

6.5.1.1. Bind your new Data Factory Analytics workspace to your Log Analytics workspace and click Create

6.6. At this point, you will need to either initiate (or wait for) some pipeline runs, ideally including a mic of success and failure

6.7. Build your Log Analytics dashboard for ADF

6.7.1. Go to your ADF Analytics solution

6.7.1.1. This is the Azure Data Analytics workspace, not the Log Analytics workspace

6.7.2. Under Workbooks, click the built in AzureDataFactoryAnalytics

6.7.3. Click the pin option and create a new dashboard for the items to be pinned to (e.g. ADF Dashboard)

6.7.3.1. The new dashboard will pin the overall workbook, which means the info we are interested in is effectively hidden one level down, so to remedy this we can edit the workbook, click on the pin options and selectively pin specific elements we are interested in

6.7.4. Once we've pinned the individual items we want and re-arranged the dashboard, it might look something like the attached

6.7.4.1. The experience of pinning and arranging the tiles is somewhat awkward at time of writing

6.7.4.2. Whilst we can see a pipeline failure on the dashboard, the only thing clickable is a link to the data factory instance itself, so you would need to go rummaging within the ADF monitoring to get to the actual error details

6.7.4.3. One notable feature is that some tiles have a button (highlighted in the previous screenshot) that when clicked, take you directly into Log Analytics

6.7.4.3.1. Log Analytics has its own query language and you can customise the dashboard tile by editing the underlying Log Analytics query

7. Pipeline execution

7.1. Debugging

7.1.1. One more advanced option you can use is to provision your own IR for data flow debugging

7.1.1.1. To save costs, you can provision a new IR that is compute-optimized

7.1.1.2. To boost debug performance, you can provision a new IR that is memory-optimized

7.1.1.3. After you've provisioned a dedicated IR for data flow debugging, you get the option to set this when you start data flow debugging

7.2. Event based triggers using Logic Apps

7.2.1. The built-in event-based trigger for data factory supports only two events: Blob creation and Blob deletion, but by leveraging Logic Apps we can hugely expand on this

7.2.1.1. In this advanced scenario, we can use a Logic App to trigger our data factory pipeline, and we'll use the creation of a new row in a SQL Database table as the trigger

7.2.2. Demo setup details:

7.2.2.1. We have a new file named argento_employees.txt that has arrived in our data lake

7.2.2.1.1. We will be adding this to our TaskQueue table in order to trigger processing of the file

7.2.2.2. We are going to use our TaskQueue table in SQL Database as a trigger for our Logic App

7.2.2.2.1. We can imagine another process (perhaps another Logic App) automatically adding a row to the TaskQueue table as part of (or in response to) the file being landed in our data lake location, but for our demo, we'll do the insert into TaskQueue manually

7.2.2.3. We already have a dynamic parent pipeline for processing the task queue, which we named Pipeline_Load_Dynamic_Parent

7.2.2.3.1. This is what we'll have our Logic App call after receiving the trigger of a new row added to the TaskQueue table

7.2.2.4. Our Logic App will need to authenticate to our SQL Database, so I decided to create a SQL credential for this

7.2.2.4.1. In SSMS, create login for argento-sql-logic-app against the master database

7.2.2.4.2. In SSMS, create user for argento-sql-logic-app against the main database (ArgentoTraining in this case)

7.2.3. Demo

7.2.3.1. Provision a new Logic App

7.2.3.2. Search for "sql" and choose the SQL Server trigger "When an item is created"

7.2.3.2.1. Despite the "SQL Server" reference being suggestive of on-prem, this trigger works for Azure SQL Database and Managed Instance too

7.2.3.3. Create a new API connection to SQL Database by selecting SQL Server Authentication and entering the previously created user credentials

7.2.3.4. Configure the SQL Server trigger step

7.2.3.4.1. For server name and database name we use the values from the API connections settings previously created

7.2.3.4.2. For table name, we use a custom value to specify the table of interest to us

7.2.3.4.2.1. Actually no! Disregard the idea of entering a custom value - you should ensure table meets criteria for the logic app trigger in order for it to be displayed in the drop down list

7.2.3.4.3. Set the the trigger frequency (e.g. every 1 minute)

7.2.3.5. Add a second step and search for "data factory", then select the "Create a pipeline run" action

7.2.3.5.1. Click the Sign in button

7.2.3.6. Configure the ADF "Create a pipeline run" pipeline and click Save to complete the logic app

7.2.3.7. Insert a new row into the SQL table to cause the logic app trigger to fire

7.2.3.8. Gotcha! The SQL Database table must feature two columns with the following characteristics:

7.2.3.8.1. An identity column

7.2.3.8.2. A column typed ROWVERSION

7.2.3.8.3. In order to get this demo to work, I had to recreate the dbo.TaskQueue table as per the attached screenshot

7.2.3.9. Check the Logic App run history to verify that a successful run of the logic app got triggered by the SQL table insert

7.2.3.9.1. If you drill into the trigger history, you should see the successful trigger event corresponding to the time you inserted the new table row in SQL Database

7.2.3.10. Check the ADF pipeline runs to verify that the expected pipeline got triggered at the same time the Logic App trigger fired

7.2.3.10.1. Note: in this example, the pipeline in question failed, but this is incidental to the main purpose of the demo, which is to prove that a Logic App can trigger an ADF pipeline based on the event of adding a new row into a SQL Database table

7.3. Call child pipeline in child data factory from a separate parent data factory

7.3.1. This idea illustrates how you can implement the idea of a "parent" data factory coordinating the work of many pipelines being run by separate "child" data factories

7.3.1.1. It is based on a technique described in a Microsoft blog

Link: https://cloudblogs.microsoft.com/industry-blog/en-gb/technetuk/2020/03/19/enterprise-wide-orchestration-using-multiple-data-factories/

Link: https://cloudblogs.microsoft.com/industry-blog/en-gb/technetuk/2020/03/19/enterprise-wide-orchestration-using-multiple-data-factories/

7.3.1.2. The main elements of the solution are as follows:

7.3.1.2.1. Azure Managed Identities for data factory

7.3.1.2.1.1. Using the fact that data factory is one of the Azure services that supports managed identity, we assign the "parent" data factory membership of the "Data Factory Contributor" RBAC role for the "child" data factory

7.3.1.2.2. Web activity or Webhook activity

7.3.1.2.2.1. The pipeline in the "child" data factory will be triggered by a pipeline in the "parent" data factory via a Web activity or a Webhook activity

7.3.1.2.2.2. Choosing between Web or Webhook depends on whether or not you want the "parent" pipeline to wait for the "child" pipeline to execute and take account of its outcome

7.3.1.2.2.2.1. The Web activity will trigger the child pipeline and complete immediately upon that trigger operation

7.3.1.2.2.2.2. The Webhook activity will trigger the child pipeline and wait for the child execution to complete, with the option to implement conditional processing based on the child pipeline's execution outcome

- 7.3.2. Demo: call pipeline in child data factory from parent data factory using the Web activity
  - 7.3.2.1. Provision a separate data factory to act as the parent
    - 7.3.2.1.1. In this example, I named my data factory argento-parent-datafactory
  - 7.3.2.2. Grant the parent data factory membership of the "Data Factory Contributor" RBAC role for the child data factory
    - 7.3.2.2.1. In this example, argento-datafactory is my child data factory - you use the Access Control to add a role assignment and then search for + select the parent DF before clicking Save
  - 7.3.2.3. Create a child pipeline in the child data factory
    - 7.3.2.3.1. In this example, I just created a very simple test pipeline that just runs a basic Wait activity
  - 7.3.2.4. Create a parent pipeline in the parent data factory and add a Web activity to it
    - 7.3.2.4.1. Settings:
      - 7.3.2.4.1.1. URL:
        - 7.3.2.4.1.1.1. See URL at top of attached file as template
        - 7.3.2.4.1.1.2. 4 elements in URL template to be replaced by references to child data factory:
          - 7.3.2.4.1.1.2.1. <child-factory-subscription-id>
          - 7.3.2.4.1.1.2.2. <child-resource-group>
          - 7.3.2.4.1.1.2.3. <child-factory-name>
          - 7.3.2.4.1.1.2.4. <child-pipeline-name>
      - 7.3.2.4.1.2. Method = POST
      - 7.3.2.4.1.3. New Header
        - 7.3.2.4.1.3.1. Name = Content-Type
        - 7.3.2.4.1.3.2. Value = application/json
      - 7.3.2.4.1.4. Body:
        - 7.3.2.4.1.4.1. {"type":"object","properties":{"inputPath":{"type":"string"},"outputPath":{"type":"string"}}}
      - 7.3.2.4.1.5. Advanced | Authentication = MSI
        - 7.3.2.4.1.5.1. Resource:
          - 7.3.2.4.1.5.1.1. https://management.core.windows.net/
  - 7.3.2.5. Publish all changes in both child and parent data factories

7.3.2.6. Trigger the parent pipeline in the parent data factory

7.3.2.6.1. Verify successful run of parent pipeline in parent data factory

7.3.2.6.2. Verify successful run of child pipeline in child data factory

8. Alerts

8.1. This was not part of my Pragmatic Works course but I wanted to test out the built in alert feature to see if I can receive alert messages whenever a pipeline failure occurs in my data factory

8.2. Go the ADF Author & Monitor

8.2.1. Under Monitor | Alerts & metrics, click option to create a new alert rule

8.2.1.1. Give the alert rule a name such as PipelineFailure, set a severity level and then click the Add criteria option

8.2.1.2. Select Failed pipeline run metrics

8.2.1.3. Set options for the Failed pipeline metrics threshold criteria

8.2.1.3.1. Under Dimension, I just selected all pipelines (under Name) and all failure types

8.2.1.3.1.1. Note: if more pipelines get added after the alert rule is set up, this might require revisiting the alert rule

8.2.1.3.2. For the threshold, I went with the default Greater than Total of 0 (i.e. alert promptly if any type of failure occurs in any pipeline)

8.2.1.4. Configure notification by assigning an action group

8.2.1.4.1. I already had an action group set up, which is configured with an email address, but you can create a new one if required

8.2.1.5. Click the Create alert rule button once the configuration of the new rule is completed

8.2.1.5.1. This will create and immediately enable the new alert rule

8.3. Test out the new alert by setting up a particular pipeline for a failure and triggering that pipeline

8.3.1. You should receive an email alert notification within 5 minutes of a triggered pipeline failure

8.3.2. Click the option to "View the alert in Azure Monitor"

8.3.2.1. In order to actually drill into the details of what failed and what the error message was, you will need to click the link to the Data Factory instance

8.3.2.1.1. Under the ADF pipeline runs you will see the failures and the error message details are available

8.3.3. Interestingly, you will get an automatically generated email to say the alert has been resolved within a few minutes, which is probably due to the alert criteria looking over the period of last 1 minute

8.3.3.1. So, because there have been no failures noted within the last 1 minute, you get a second alert notification generated to say the alert condition is resolved, which is a bit misleading

9. Mapping data flows

9.1. One of the nice features of ADF mapping data flows is the ability to automatically handle schema drift and automatic inference of data types

9.2. Key things to remember when you want to leverage the features for allowing schema drift

9.2.1. Datasets for data flow source and sink should not have the schema imported

9.3. Example: load a csv file into a SQL Database table using options to allow schema drift and infer drifted column types

9.3.1. See attached how the source file looks; a csv file named Sales.csv

9.3.1.1. I uploaded it into my data lake to a container named data-flow-no-schema-files and a folder named "in"

9.3.2. For the source dataset, I will re-use an existing dynamic one for csv files, which is identified by parameters

9.3.2.1. The schema for this dataset is NOT imported

9.3.3. Create a new data flow and set the source settings

9.3.3.1. We point to the dynamic, schema-less dataset and enable options to allow schema drift and infer drifted column types

9.3.4. Add the sink and enable schema drift

9.3.4.1. Initially we will skip the transformation step, as we'll be using debug to see the outcome of auto-mapping

9.3.4.2. Under Settings, we'll set Table action to Recreate table

9.3.4.2.1. Under Mapping, we enable auto-mapping

9.3.5. Enable data flow debug and then go to the Data preview for the source, where we are prompted to enter values for the parameters relating to the source and sink

9.3.5.1. When we refresh the preview, we can see the data for our source file

9.3.5.1.1. Note how Customer column has been inferred as an integer type and the SoldDate inferred as a datetime type

9.3.5.2. Gotcha! My attempt to use an expression, toLower(''), for the Directory part of the source file path failed because expressions are not supported in this context and it treats the expression as a string literal in the path

9.3.5.2.1. I overcame this problem by deleting and recreating the data flow, and changing the default values for the parameters in my dynamic dataset (see attached)

9.3.6. Add a Derived Column transformation

9.3.6.1. In this example, we set the State column to transform always to upper case

9.3.6.1.1. The challenge with schema-less is that the column name is not directly addressable, but as you can see from attached screenshot, we can overcome this by using the byName() function

            - 9.3.6.2. Data preview shows that our State column is transformed to uppercase
        - 9.3.7. Create a new pipeline and add parameters for the data flow
            - 9.3.7.1. 6 parameters added, 4 for the data flow source and 2 for the sink
                - 9.3.7.1.1. ContainerName, DirectoryName, FileName, ColumnDelimiter, Schema, TableName
            - 9.3.7.2. Add a data flow and under Settings specify that the values for the data flow source and sink shall come from the corresponding pipeline parameter
                - 9.3.7.2.1. For example, the data flow source file name is set via the following expression:
                    - 9.3.7.2.1.1. @pipeline().parameters.FileName
        - 9.3.8. Debug the pipeline
            - 9.3.8.1. Enter the pipeline parameter values to test your pipeline against
            - 9.3.8.2. Verify that pipeline debug completes successfully
            - 9.3.8.3. Using SSMS, verify that dbo.RetailSales table is successfully created and loaded
    - 9.4. Rule-based mapping
        - 9.4.1. With rule-based mapping, we leverage the flexibility of data flows to automatically handle change, including the following changes that can occur in both source and sink
            - 9.4.1.1. Source changes
                - 9.4.1.1.1. Column names changing
                - 9.4.1.1.2. Data types changing
                - 9.4.1.1.3. New columns being added
                - 9.4.1.1.4. Columns being removed
            - 9.4.1.2. Sink changes
                - 9.4.1.2.1. Columns added
                - 9.4.1.2.2. Columns removed
        - 9.4.2. Rule-based mappings are a specific feature of the Select transformation, and involve turning off the auto-mapping in that transformation and replacing it with expression-based mappings
        - 9.4.3. Example: we have two sales files from two different vendors, both in csv format but with different names for some of the columns and one of the files with an extra column not present in the other
            - 9.4.3.1. Here's what the two source files look like:
                - 9.4.3.1.1. Sales.csv
                    - 9.4.3.1.1.1. Let's imagine this file comes from source vendor A

9.4.3.1.1.2. Column order is different to equivalent file from source vendor B and the 4 columns highlighted in yellow are named differently despite being semantically the same

9.4.3.1.2. ConferenceSales.csv

9.4.3.1.2.1. Let's imagine this file comes from source vendor B

9.4.3.1.2.2. Column order is different to equivalent file from source vendor A and the 4 columns highlighted in yellow are named differently despite being semantically the same

9.4.3.1.2.3. In addition, this file has an extra column (NewColumn), highlighted in green, which is specific to the vendor B file

9.4.3.2. This builds on the previous example

9.4.3.2.1. Add a Select transformation after the source, and before the Derived column transformation

9.4.3.2.1.1. Under Select settings, disable auto mapping, and add a new mapping

9.4.3.2.1.1.1. Because the source is schema-less, what we get here is a rule-based mapping instead of a fixed mapping

9.4.3.2.1.1.2. We can recognise this as a rule-based mapping by the red box highlight and the double inverted chevron symbol

9.4.3.2.1.1.3. Clicking inside either red box for the mapping will take you into the expression builder

9.4.3.2.1.2. Enter the rule expression for the incoming column name, using the locate() function

9.4.3.2.1.2.1. In this example, we tackle the "First" vs "FirstName" column name dilemma

9.4.3.2.1.2.2. The locate() function takes a substring for its first argument and searches the string of the name, returning the start position of the substring within the (column) name

9.4.3.2.1.2.2.1. name represents the inbound column name in the data flow

9.4.3.2.1.2.2.2. When the result is 0, this means the substring does not exist

9.4.3.2.1.2.3. The expression we use to match "First" and "FirstName" to the same common column name in our data flow is:

9.4.3.2.1.2.3.1. locate('first', lower(name)) != 0

9.4.3.2.1.3. Under the output column name expression, add the common column name you want to use as a string literal, then Save and Finish

9.4.3.2.1.3.1. In this case we use 'FirstName'

9.4.3.2.1.4. Repeat process to add rule-based mappings to handle each of the 4 columns with the varying names

9.4.3.2.1.4.1. At this point we have handled our differing column names, but we've lost our auto-mapping for common column names that already matched

9.4.3.2.1.5. Add a final catch-all rule-based mappings to handle the common column names that would otherwise auto-map

9.4.3.2.1.5.1. This is bit of a workaround trick - this expression will be evaluated after all the preceding ones we added

9.4.3.2.1.5.1.1. Our expression is:

9.4.3.2.1.5.1.1.1. name != 'Known Column'

9.4.3.2.1.5.1.1.1.1. We are choosing a column name value that we are confident will not exist in the file and which will be a mutually exclusive condition from the preceding rule-based mappings based on the locate() function

9.4.3.2.1.5.1.2. Our output column is:

9.4.3.2.1.5.1.2.1. $$

9.4.3.2.1.5.1.2.1.1. $$ represents "this column name" without having to specify anything about that name

9.4.3.2.1.5.2. Here's the final Select settings

9.4.3.2.1.6. Using Storage Explorer, we make sure we have our two different sales files in place

9.4.3.2.1.7. Debug the pipeline and set the parameters for loading the Sales.csv file

9.4.3.2.1.7.1. After a successful run, use SSMS to verify that the Sales.csv has dynamically re-populated the dbo.RetailSales table

9.4.3.2.1.8. Debug the pipeline and set the parameters for loading the ConferenceSales.csv file

9.4.3.2.1.8.1. After a successful run, use SSMS to verify that the ConferenceSales.csv has dynamically re-populated the dbo.RetailSales table

10. Databricks integration

10.1. ADF data flows are built on top of Azure Databricks

10.2. ADF data flows provide an easy-to-use GUI for developing ETL pipelines, whilst Databricks provides notebooks that you can use to develop ETL pipelines using code

10.3. You can call Azure Databricks notebooks from ADF pipelines to perform your ETL (i.e. the transformations)

10.4. The advantage of using Databricks instead of ADF data flows is sheer flexibility due to the code-based approach

10.4.1. Any time you can't get the job done using ADF data flows, you can almost certainly get it done using Databricks

10.5. Example: Databricks notebook to load multiple csv files (with same schema) into Spark dataframe and then save dataframe into JSON format, targeting directory with name specified via Databricks input parameter, and demonstrate calling this notebook from ADF pipeline, setting the input parameter in the process

- 10.5.1. In Databricks, we create a notebook
  - 10.5.1.1. Cell 1 uses dbutils.widgets to create a string input parameter named Table and then set a local variable named table_name to that parameter value
  - 10.5.1.2. Cell 2 sets up some Spark config so that the notebook can connect to ADLS Gen2 storage account using the access key (blacked out in screenshot)
    - 10.5.1.2.1. More secure practice is to use secrets for the access key (i.e. Azure Key Vault or Databricks secret scope)
    - 10.5.1.2.2. We also test out our connection to ADLS Gen2 by issuing a dbutils.fs.ls command
  - 10.5.1.3. Cell 3 calls spark.read with some options to load the csv files in from the data lake, storing it in a variable, df, and then displaying the resultant dataframe
  - 10.5.1.4. Cell 4 invokes the write.json method of the dataframe, appending the table_name variable to the output path
- 10.5.2. Test run the notebook in Databricks
  - 10.5.2.1. Note that we set the Table parameter to "aaaa" for this test run
- 10.5.3. Check result in ADLS Gen2 using Storage Explorer
  - 10.5.3.1. In this example, it ingested 21 csv files into a Spark dataframe and exported that dataframe as 4 JSON files, grouped logically in a Table folder labelled "aaaa"
- 10.5.4. Create a new Databricks access token
  - 10.5.4.1. Copy the access token when it displays - if you close the window, you'll have to re-generate the access token
- 10.5.5. Create a new ADF pipeline and add a parameter for Table
  - 10.5.5.1. In this example, we set the default value to "InternetSales_JSON"
- 10.5.6. Add a new pipeline, and drag on a Databricks Notebook activity
  - 10.5.6.1. Create a new Databricks linked service and paste in the access key
    - 10.5.6.1.1. I chose "Existing interactive cluster" here for speed but normally you would choose "New job cluster"
    - 10.5.6.1.2. Pasting in the access key to the linked service config is not best practice - better to use Azure Key Vault
  - 10.5.6.2. Under Settings, browse to your Databricks notebook and then add a parameter that you name Table and map via the pipeline parameter of same name
  - 10.5.6.3. Debug the pipeline and verify that it completes successfully

10.5.6.3.1. Using Storage Explorer, verify that the "InternetSales_JSON" input parameter was used to capture the JSON files produced by the Databricks notebook

11. Copy + Delete activities to archive or move files

11.1. This technique uses two datasets, one for the original file and another for the target archive location

11.1.1. One issue with this approach is that the number of datasets can get out of hand quite quickly

11.1.2. Leveraging parameters and dynamic expressions is a more advanced technique that can keep your data factory components more compact (i.e. less of them) and more flexible

11.2. Example: pipeline to archive "employee" source files after they've been processed (by another pipeline)

11.2.1. Create a new pipeline with the Copy and Delete activities connected in sequence

11.2.2. Configure Copy activity Source properties

11.2.2.1. Dataset from earlier pipeline (i.e. the one that loads the file into a database) is re-used in this example

11.2.2.2. As the dataset references only the source file directory and not the filename, we must specify a wildcard path so the copy activity knows what files need to be copied

11.2.3. Create dataset for the file archive target

11.2.3.1. We can specify a new container for the archive file that doesn't yet exist, as the Copy activity will create it for us if required

11.2.3.2. I forgot to change the default delimiter character, which had the interesting effect of transforming the file on archive from pipe delimited to comma delimited

11.2.4. Configure Copy activity Sink properties

11.2.4.1. We only need to point it to the dataset - no need for further mapping or setting config

11.2.5. Configure Delete activity Source properties

11.2.5.1. We only need to point this to the source dataset of the Copy activity

11.2.5.2. As the source dataset references simply a directory, we should be aware that it will delete all files from this directory

11.2.5.2.1. Because I used a specific wildcard pattern on the Copy activity Source properties, I verified that this results in deleting a completely unrelated file from test-files (e.g. test3.json)

11.2.6. Configure Delete activity Logging settings

11.2.6.1. We specify a storage account and a directory for the logging

11.2.6.1.1. The Delete activity will create the directory for us if it doesn't already exist

11.2.6.2. The logs are stored in subfolders named with the Delete activity run ID (a unique GUID)

11.2.6.2.1. The log file itself is a simple csv that can be opened in Excel - it tells us what file(s) got deleted and whether any errors occurred

11.2.7. Debug the archive pipeline to verify success

11.3. An alternative technique for archiving files is to use Azure Logic Apps

12. Leveraging ADF Managed Identity to simplify access to other Azure resources

12.1. Managed Identity key features and benefits:

12.1.1. Data Factory automatically registers a managed identity in Azure Active Directory at the time it is provisioned

12.1.2. Allows you to keep passwords, credentials and/or certificates out of code

12.1.2.1. Good security practice!

12.1.3. No additional cost for managed identities

12.1.4. Requires object to be held in same AAD tenant

12.1.4.1. So you cannot use managed identity held in one AAD to grant access to resources held in a separate Azure tenant (i.e. different Azure account)

12.2. Example: grant data factory instance access to Azure storage account using RBAC and the data factory managed identity

12.2.1. Go to storage account Access Control (IAM) and add a new role to the specific data factory

12.2.1.1. The role will typically be one of the built in ones that begin "Storage Blob"

12.2.1.2. When you type in part of the data factory name in the Select field, you will see listed the name of the data factory you are interested in and you click to move it to Selected members

12.2.1.3. Click Save to complete

12.2.2. Note that the managed identity is distinguishable from regular service principals in the list of role assignments for the storage account

12.2.2.1. It bears the data factory icon rather than the generic icon for registered apps (service principals)

12.2.2.2. The type is given as "Data Factory" instead of "App"

12.2.3. In ADF, configure the storage account linked service to use managed identity

12.2.3.1. Authentication method = Managed Identity

12.2.3.2. Account selection method = From Azure subscription

12.2.4. You can make a simple test again using a pipeline with a Get Metadata task that's bound to the linked service for the storage account

12.2.5. This example illustrates that Managed Identity offers a simpler but equally secure method for application authentication than Azure Key Vault

12.3. As an aside, managed identities are represented as an Enterprise Application

12.3.1. Enterprise applications are contained by Azure Active Directory, and you can browse many of these applications using AAD, but it seems like managed identities are hidden

13. Integrating ADF with Azure Key Vault

13.1. You must not store sensitive information within the components of your data factory, such as API keys, passwords, certificates, etc. - all of these are secrets that should be stored in Azure Key Vault and securely retrieved by ADF at runtime

13.2. Secrets are added to Key Vault but in order for any person or application to access those secrets you must create an appropriate access policy

13.3. Recommended best practice is to have different key vaults for each environment (i.e. one for Dev, one for QA, one for PRod, etc.)

13.4. Two options for using key vault from data factory:

13.4.1. ADF linked service to key vault

13.4.2. DevOps release pipeline that pulls from key vault

13.5. Example: ADF with Key Vault to manage secrets for secure connections to Azure storage account + Azure SQL Database

13.5.1. Go to Azure Storage Account | Access Keys and copy the access key

13.5.2. Add new secret to your key vault

13.5.2.1. Name the secret (e.g. DataLakeAccessKey) and paste the storage account access key into the secret value before clicking Create

13.5.2.1.1. My original choice of secret name, DevDataLakeConnStr, was poor as the idea is not to name secrets with environment references (i.e. "Dev" in this case) but to maintain secrets with identical names across separate key vaults for each environment, and secret values differing per environment

13.5.2.1.1.1. Also, I made a mistake (misled by the Pragmatic Works training - probably due to a change in way ADF/key vault integration works) of initially choosing the storage account connection string for the secret instead of key, which will lead to a connection error in the ADF linked service for the storage account

13.5.3. Add a second secret to hold your SQL Database admin user password

13.5.3.1. In the real world, you'd probably go for a less privileged SQL account than the admin one set up at time of provisioning the resource

13.5.4. Go to ADF and add a new linked service for Azure Key Vault

13.5.4.1. Configure linked service to connect to the key vault instance that holds your secret storage account connection string

13.5.4.1.1. Note that the dialog warns and prompts you to ensure an access policy is set up on the key vault based on the managed identity of your data factory instance

13.5.4.1.1.1. If you click this link it will open up a new browser tab that navigates to the key vault instance and the access policies page

13.5.5. In Key Vault, add a new access policy

13.5.5.1. Configure the data factory managed identity for secret management and click Add to create new access policy

13.5.5.1.1. Note that you can retrieve the managed identity ID from ADF Properties

13.5.5.1.2. Don't forget to click Save to confirm the new access policy

13.5.6. In ADF, configure your linked service for the Azure Storage Account to use key vault instead of connection string

13.5.6.1. If the key vault secret is correctly set up and referenced and the key vault policy is in place, you should see a successful connection test

13.5.6.2. If you did not set up the key vault access policy for your data factory instance, you will get a connection error:

13.5.6.2.1. Caller was not found on any access policy in this key vault

13.5.6.3. If you used the wrong value for the key vault secret that is supposed to hold your storage account access key, you will get a connection error:

13.5.6.3.1. The specified account key is invalid. Check your ADF configuration.

13.5.7. In ADF, configure your linked service for the Azure SQL Database to use key vault instead of password

13.5.7.1. If your Azure SQL Server firewall does not permit access from Azure services, you will get a connection error

13.5.7.1.1. Cannot connect to SQL Database... Client with IP address... is not allowed to access the server

13.5.8. In ADF, you can set up a quick and simple connectivity test for the linked services that use key vault integration to retrieve secrets

13.5.8.1. Create a new pipeline with two Get Metadata tasks, one pointed to a flat file dataset hosted in your Azure storage account and the other pointed to a SQL Database table

13.5.8.1.1. After publishing and running debug on the pipeline, you should get a successful result

13.6. As an aside, you can retrieve your secrets manually from Azure Key Vault using the portal, as long as the key vault access policy allows it for your user account

13.6.1. Just navigate to the key vault instance and down to the particular secret version, and there you will see an option to show secret value and/or copy it

14. Pipeline templates

14.1. Rather than creating ADF pipelines from scratch, they can be created from templates

14.2. In order to create your own templates, you must have your data factory Git integrated

14.3. Creating a pipeline template

14.3.1. Every pipeline has the option to save as a template

14.3.1.1. You can set the template name and description

14.3.1.1.1. Note the Git location (non editable), which will commit the template to the linked repo in the templates directory

14.3.1.1.2. You can also check the various services used by the ADF pipeline template, add tags and a hyperlink to custom documentation if you have that set up

14.3.1.1.3. As an alternative to saving the template in your Git repo, you also have the option to export the template to your local drive

14.4. Create new pipeline from template

14.4.1. When creating a new pipeline, choose Pipeline from template

14.4.1.1. You are presented with the template gallery and can choose one of your own templates or someone else's templates

14.4.1.1.1. Note that there are a bunch of built in templates supplied by Microsoft, which will be interesting to explore

14.4.1.1.2. Complete user inputs for template and click the Use this template button