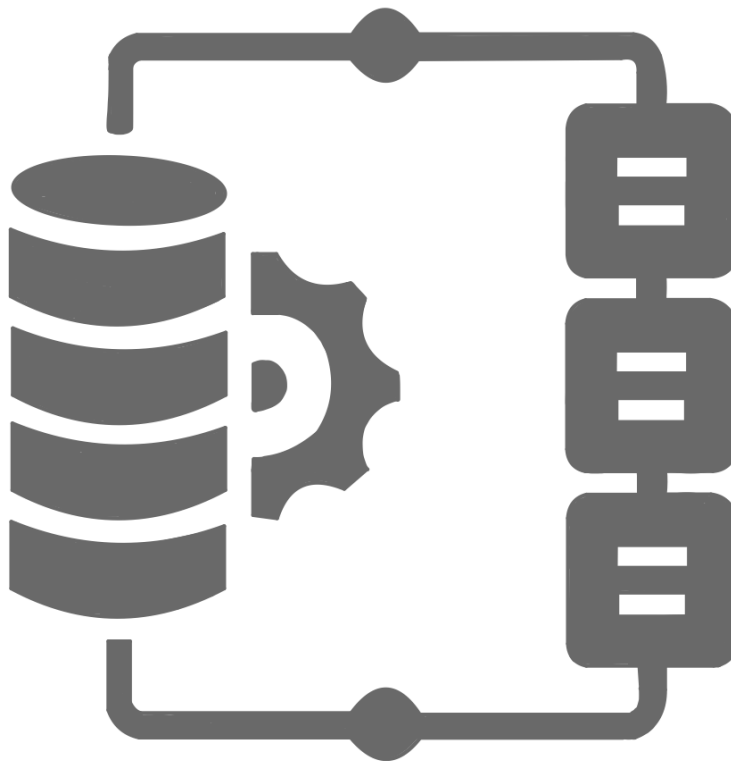


FLD Integration Solutions:

Azure Logic Apps & Azure Data Factory Documentation

Natalie Danner
Nicholas Gonzalez
Jessica Silang



Introduction/ Getting Started	3
Azure Logic Apps	4
Figure 1: Whole Workflow	4
Prerequisites:	4
Setting up the File System Trigger:	5
Figure 2: File System Trigger	5
Figure 3: File System Connection Settings	6
Initializing the Variables	7
Extracting, Transforming, and Loading the Data	7
Figure 4: ETL Process	7
Step 1: Extracting the File Content From Each File	7
Step 2: Setting the variables to correspond to the current file's properties	8
Table 1: Setting Variables Explanation	8
Step 3: Conditioning (Differentiate between Player vs. Game vs. Retailer)	8
Figure 5: Branching Logic	8
Figure 6: XML → JSON Conversion	9
Figure 7: Parsing the JSON	10
Loading the Entry Into a SQL Server Database	10
Figure 8: Insert Row into Database	11
Figure 9: Error Handling	12
Executing the Stored Procedure	12
Azure Data Factory	13
Basic Concepts:	13
Azure Data Factory Prototype Description	15
Logic App Trigger	15
Azure Storage Account	15
Configuring Data Factory Pipeline	18
Creating/Connecting to SQL Server and Azure SQL Database	30
Creating Stored Procedure in SQL:	30
Our Final Recommendation	32
Additional Information	33
Potential Areas of Improvement	33

Introduction/ Getting Started

The UF Senior Design Team was tasked with automating the Florida Lottery System's current file storage process. The scope of this project included retrieving a file (Player, Retailer, or Game Focused) and having the creation of the file in a storage trigger a workflow to insert JSON objects into a SQL server. Our team explored two different methods to achieve this goal- Azure Logic Apps and Azure Data Factory. We noted the pros and cons of each platform and documented how to recreate these workflows.

Each of these solutions require access to the Microsoft Azure platform. For the purposes of this project, we were able to utilize the 12-month free trial to implement our solutions. Beyond that, there are certain "premium" features that you can choose to pay for. In terms of annual cost, the cost of Azure is measured by usage. You pay for services based on how much they are utilized and at what capacity. A plus side is the advanced features to monitor and track spending and usage.

The documentation below will showcase our implemented prototypes and the following information:

1. Prerequisites to implementing
2. Step-by-step tutorial for implementation
3. Things to look out for or errors we ran into

After the documentation portion comes the team's final recommendation for which platform we encourage the Florida Lottery Department Team to utilize. Lastly, our team documented research regarding features that we would have liked to implement given a longer timeline.

Azure Logic Apps

Azure Logic Apps utilized built in connectors such as on-premise gateways, liquid templates, and an Azure SQL Server database. Our Logic App prototype creates a file system trigger that is fired once a file is entered into a local storage. From there, we copy the file content and file name, transform the format from XML to JSON, parse the data, and insert the entity into a SQL Server database. The whole work can be seen below in Figure 1.

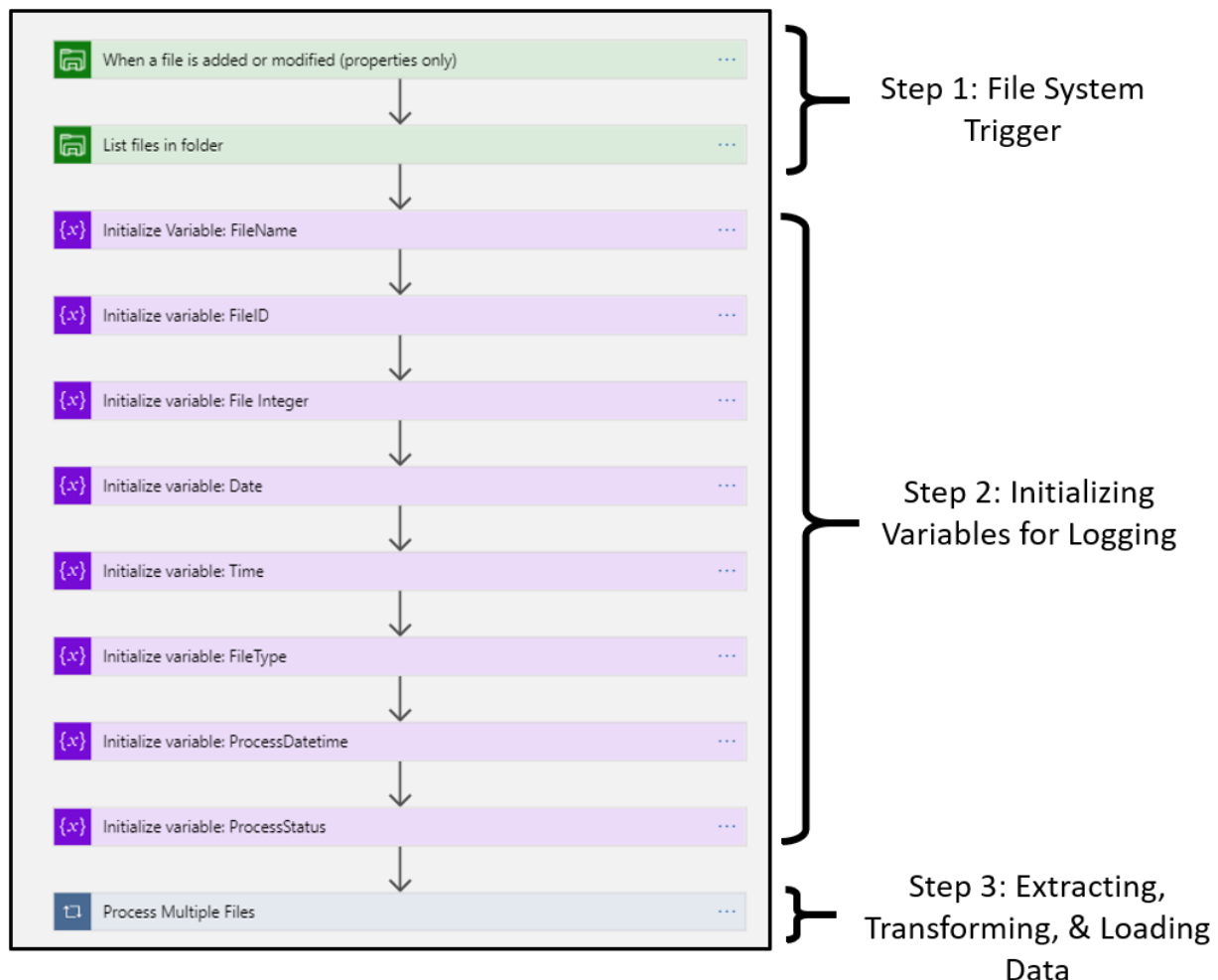


Figure 1: Whole Workflow

Prerequisites:

- Download and install on-premises data gateway

- Configure Azure SQL Server Database
- Secure access to text editor (Visual Studios Code, etc.) to write .liquid templates

Step 1: Download and Configure On-Premises Data Gateway

An on-premises data gateway must be downloaded on your device in order to utilize the on-premises data gateway connection on Logic Apps. An on-premises data gateway allows you to bridge the gap between your on-premise device and your configured Logic App. A step by step walkthrough on the installation of this on-premise data gateway is found here: [Install on-premises data gateway - Azure Logic Apps](#)

Setting up the File System Trigger:

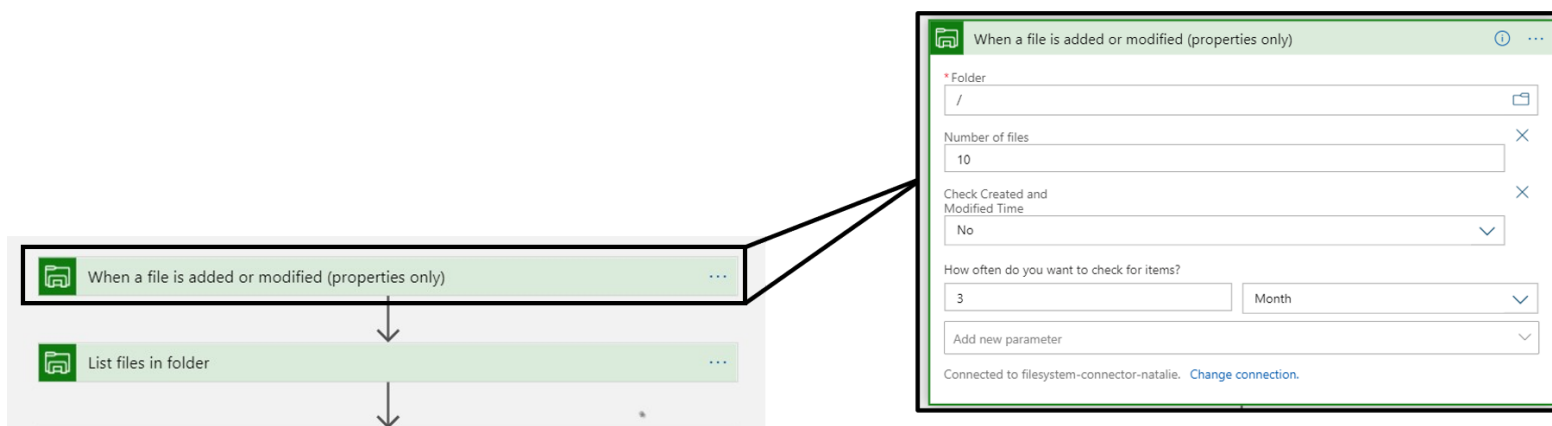


Figure 2: File System Trigger

Once your on-premise data gateway is configured, you can select the file system connector within Logic Apps to fire your workflow. The specific connector we used is titled "When a file is added or modified (properties only)." If no connections are set up, Azure will prompt you to create a new one. The required inputs are shown in Figure 3.

Figure 3: File System Connection Settings

- *Connection Name*: This is user specified- name it as you wish
- *Root folder*: The root path to the folder acting as your local storage
 - C:\Users\Jessica\Desktop\Spring 2021\Senior Design\Storage
- *Authentication Type*: Windows is the **only option**
- *Username*: the username of your desktop-- found in System Information
 - Ex: DESKTOP-__________
- *Password*: the password to your laptop

The next step titled “List files in folder” is what enabled us to implement processing multiple files. This step will allow the workflow to iterate through a list of files in the storage.

Initializing the Variables

The purpose of this step is to define and instantiate the variables needed throughout the workflow and for the logging process.

Extracting, Transforming, and Loading the Data

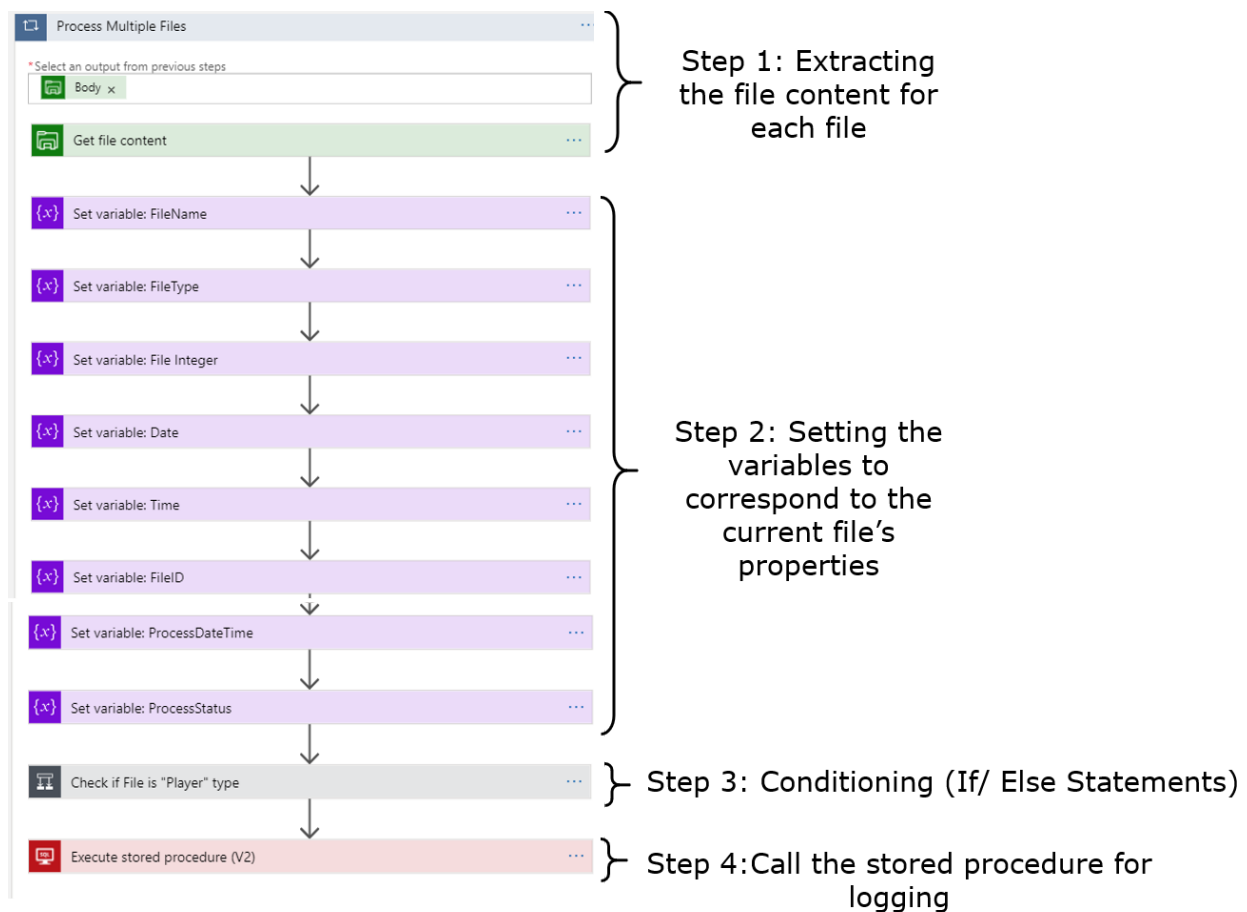


Figure 4: ETL Process

Step 1: Extracting the File Content From Each File

This prototype utilized a "For Each" loop to handle processing multiple files. The for each loop titled "Process Multiple Files" takes the body of each file listed in a folder and extracts the file content. The content will then go through the steps necessary to transform and load the data into the SQL Server Database.

Step 2: Setting the variables to correspond to the current file's properties

As mentioned above, variables were initialized in order for the logging process to be implemented into this workflow. This step is where the variables are assigned specifically to the properties of the current file within the loop. Table 1 outlines the variables used, the functions used to set them, and a sample output.

Ex: game_04072021_114300		
File Name	Pull Name From Dynamic Content	game_04072021_114300
File ID	concat(file_integer,date,time)	
File Integer	if(equals(variables('file_type'), 'player'),1,if(equals(variables('file_type'), 'game'),2,3))	2
Date	substring(variables('fileName'), add(indexOf(variables('fileName'),'_'),1), 8)	04072021
Time	substring(variables('fileName'), add(1, lastIndexOf(variables('fileName'),'_')), 6)	114300
FileType	substring(variables('FileName'),0,indexOf(variables('FileName'),'_'))	game
ProcessDateTime	utcNow()	2021-04-07T12:35:10.4677961Z
ProcessStatus	Success	Success

Table 1: Setting Variables Explanation

Step 3: Conditioning (Differentiate between Player vs. Game vs. Retailer)

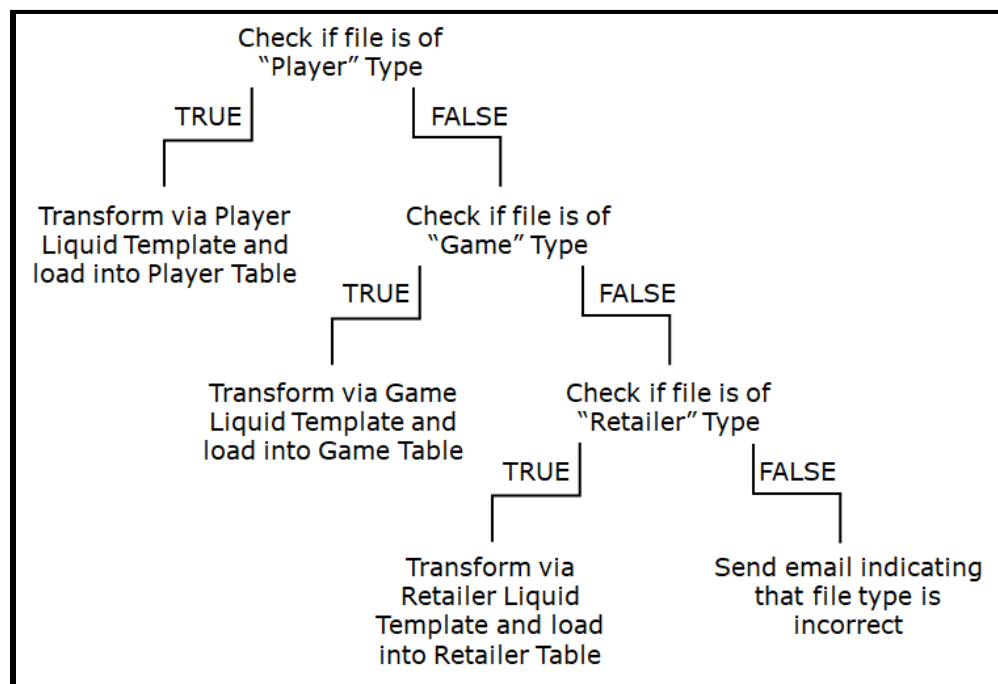


Figure 5: Branching Logic

Figure 5 outlines the logic in which the file type is determined between Player, Game, and Retailer. This is essential in order to make sure that the XML

content is transformed into JSON using the right liquid template. The method of checking the file type is by creating an If statement that verifies whether or not the file name contains the word “player”, “retailer”, or “game”.

Once the file type is determined, the XML content is transformed via mapping the content through a Liquid Template. In Azure, we utilized the Liquid connector that has an action of transforming XML content to JSON. We created the map ourselves using Visual Studios Code. The syntax for Liquid is quite simple and all maps follow the same structure with different text based on different attributes. Figure 6 shows the liquid template file that was used to transform a file of type “player” to JSON.

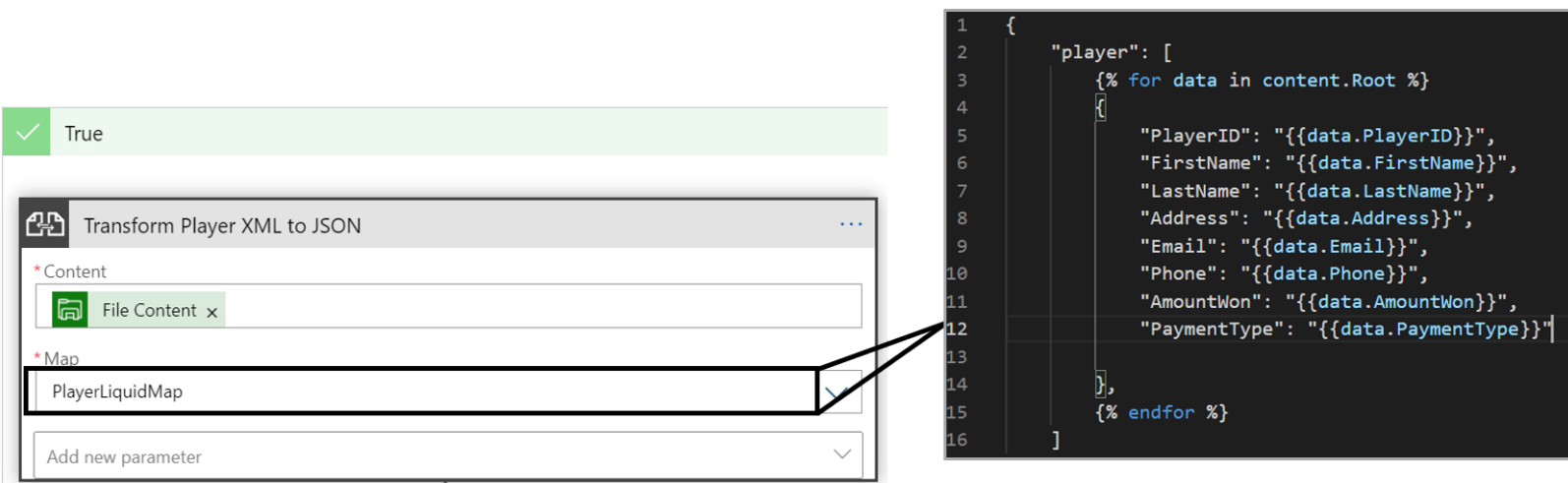


Figure 6: XML → JSON Conversion

The Liquid file is relatively simple and essentially loops through the content of the XML file and grabs the different attributes from the file.

Note: In order to use a map in the Transform XML to JSON step, you must create an integration account within your resource group, upload the liquid file, and configure your workflow settings to link the integration account to the workflow. A tutorial for creating the integration account and configuring the workflow settings can be found at [this website](#).

Now that the content is formatted in JSON, the next step is to parse the JSON using the Parse JSON function provided by the Data Operations connector in Logic Apps.

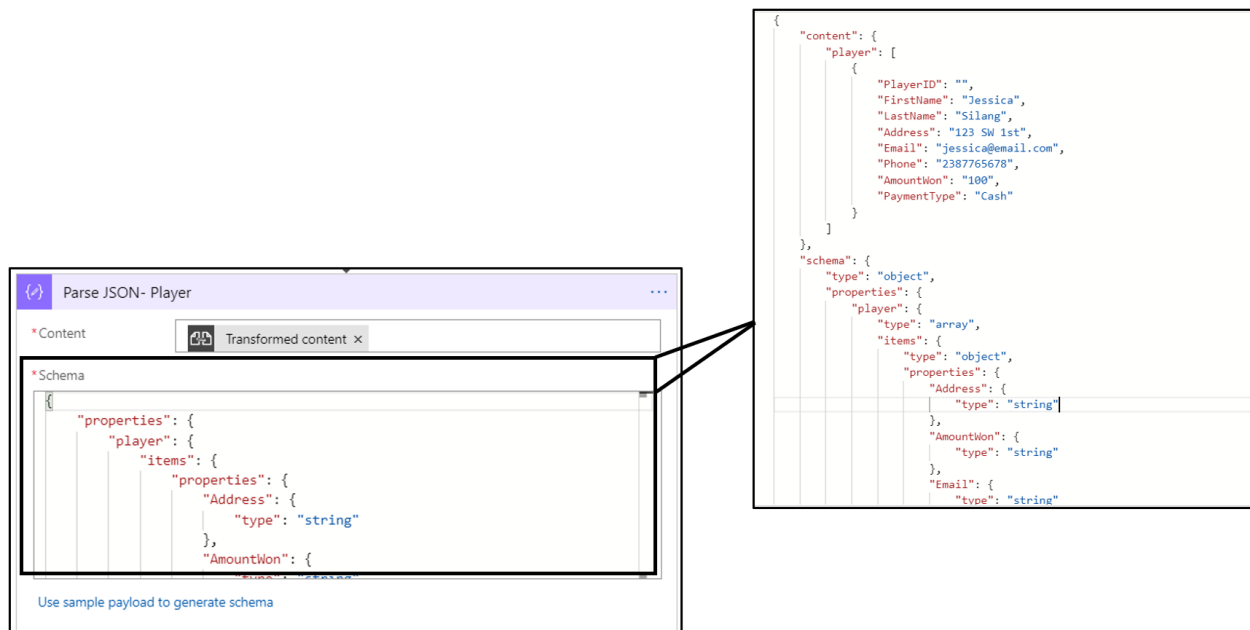


Figure 7: Parsing the JSON

This connector allows you to select the Transformed content from the previous output and parse the JSON using a JSON schema that is provided by the user. We used Visual Studio Code to create a schema outlining the properties, attributes, and data types- which can be seen in Figure 7.

[How to setup a SQL Server Database](#)

Loading the Entry Into a SQL Server Database

In order to insert every attribute into the database, you must insert a for each loop that inserts each parsed JSON object into the database. In figure 8, you can see the setup of the loop and how it takes every player object and inserts each attribute into the database.

To load into the database, we used a connection titled "Insert Row (V2)". This action allows you to select a server, database, and specific table within a SQL server to add entries into via a connection. For each file type, the insert row step inserts entries into each respective database. Be sure to click "Add New Parameters" and match each parsed JSON object to the correct column name.

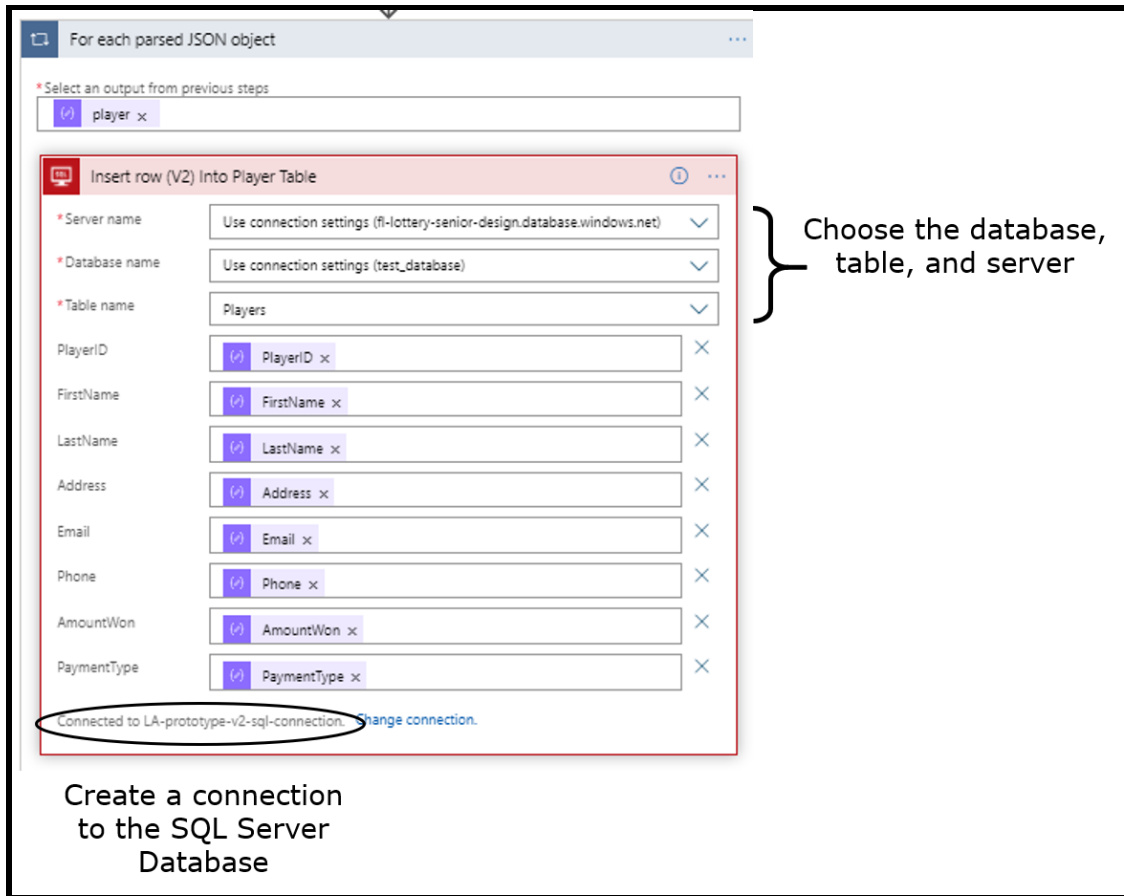


Figure 8: Insert Row into Database

After all of the content is inserted into the database, we created an action that deletes the file from the local storage so that it is not duplicated the next time the workflow is triggered.

This process repeats for file types "Game" and "Retailer." However, if the file type is outside of these three types, the workflow is configured to send an email to a specified list of users to indicate that an error has occurred. In addition, the logging table will indicate that a process failed due to the fact that we set the Process Status to "Failed." The process is outlined in Figure 9.

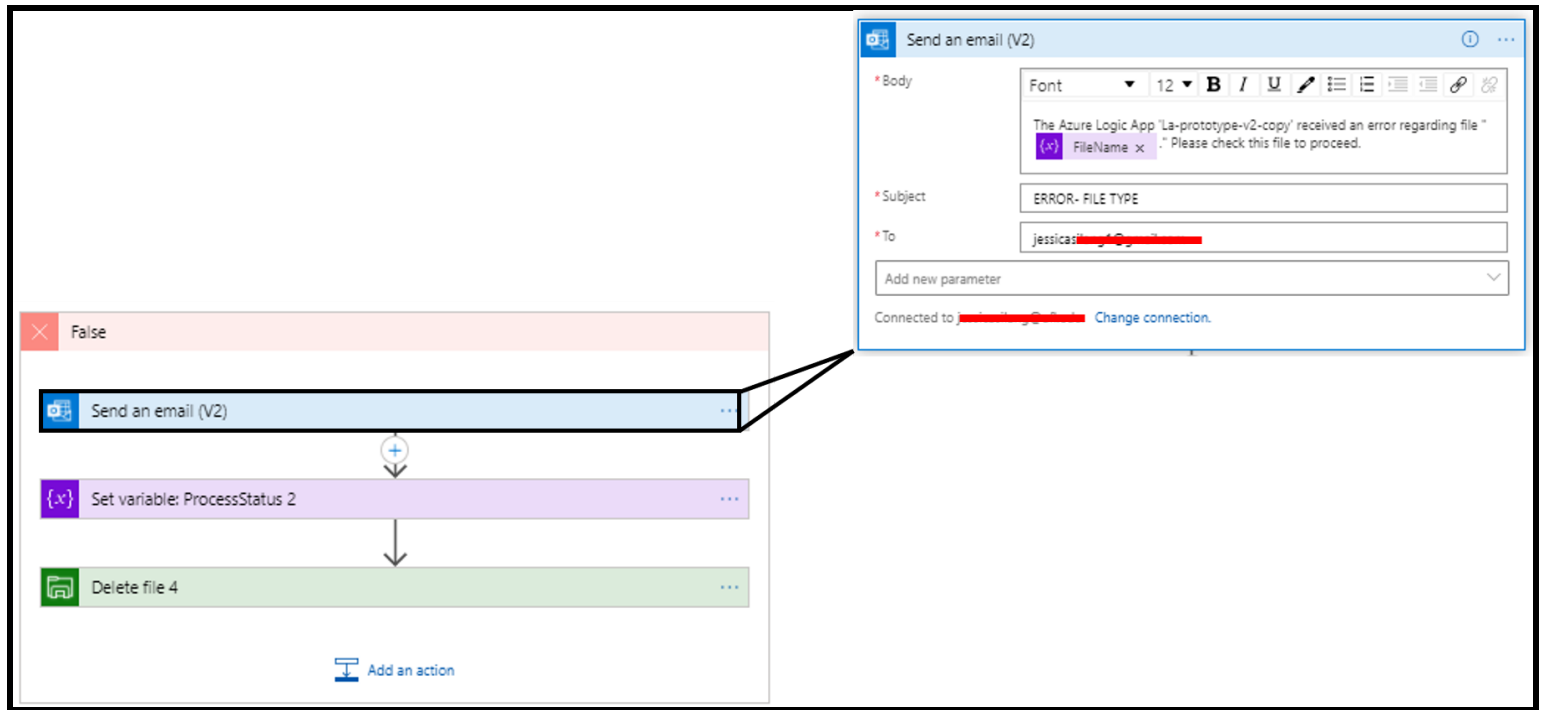


Figure 9: Error Handling

Executing the Stored Procedure

The instructions for completing this step can be found [here](#).

Azure Data Factory

Basic Concepts:

The basic structure of a project in Azure Data Factory involves securing connections to all sources and applications that the data must move between, followed by consolidating the data in a centralized storage container. From this storage area, the data can easily be transformed and analyzed using Azure's various tools. For this project's Azure Data Factory prototype, Azure Blob Storage is used as the centralized storage container, and Data Factory tools provide a dynamic movement of the data from Blob Storage to our SQL Database. An overview of the Azure Blob Storage Account is provided [here](#).

The primary elements of the Data Factory prototype are Linked Services, Integration Runtimes, Pipelines, Datasets, and Activities:

1. Integration Runtime: The IR is the data integration compute infrastructure used by ADF. This is the environment that an activity or compute service gets executed on or dispatched from, respectively. The prototype uses the builtin Azure Auto Resolve Integration Runtime, which supports all activities between cloud-based services. However, it is important to note that if an on-premises data source is to be connected, the user must use a self-hosted IR. Even though the self-hosted IR wasn't used in the final prototype, there is an example of one provided in this documentation.

Source and additional information on Integration Runtime:

<https://docs.microsoft.com/en-us/azure/data-factory/concepts-integration-runtime>

2. Linked Services: In order to establish a connection to a data store, the user must create a linked service, which defines the connection properties as well as the integration runtime used to make the connection. A linked service is the Data Factory equivalent to the Connectors used in Logic Apps.

Source and more information on Linked Services:

<https://docs.microsoft.com/en-us/azure/data-factory/concepts-linked-services>

3. Datasets: Datasets go hand and hand with linked services, as they represent the actual data structure that is connected via the linked service. The dataset is used to reference the data itself in the data movement/transformation activities in Data Factory.

Source and more information on datasets:

<https://docs.microsoft.com/en-us/azure/data-factory/concepts-datasets-linked-services>

4. Activities: Data Factory abstracts their various data manipulation tools into what is called an activity. An activity defines an action to perform on data, and there are three categories of activities to choose from:
 - a. Data movement: In our ADF prototype, the “Copy Data” activity is used to move data from one dataset to another.
 - b. Data transformation: For example, the prototype uses a “Execute Stored Procedure” activity to run a custom stored procedure we created.
 - c. Control Flow Activities: Allow for the utilization of variables and logic such as loops, wait steps, and conditional statements

Source and additional information:

<https://docs.microsoft.com/en-us/azure/data-factory/concepts-pipelines-activities#control-flow-activities>

5. Pipelines: A pipeline is a logical grouping of individual activities that is meant to achieve a certain task. There may be more than one pipeline per Data Factory, so it is up to the user how they organize the activities and pipelines. The activities may be structured in series in the pipeline, with the option of piping output from one activity to the next one’s input. Alternatively, the activities may execute in parallel so that they are independent from each other.

Source and additional information:

<https://docs.microsoft.com/en-us/azure/data-factory/concepts-pipelines-activities>

Azure Data Factory Prototype Description

Logic App Trigger

When it comes to event-based triggers, Azure Data Factory is quite limited since it only offers two types of event based triggers. The first trigger is a *storage event trigger* which manages events in an Azure Blob Storage account such as a file being uploaded or deleted to a blob. The other trigger is a *custom event trigger* which requires utilization of Azure's Event Grid. Since the scope of this project entailed retrieving files from a local directory rather than an online storage account, it was determined to pursue the On-Premises Data Gateway and implement the solution using their trigger from within Logic Apps. After being triggered, the Data Factory pipeline would then be called from within the Logic Apps workflow. The following steps detail how to configure the Logic Apps portion of this prototype that eventually leads to the calling of the Data Factory pipeline.

Azure Storage Account

Blob storage on an online Azure storage account was chosen as the primary storage solution due to its optimal capabilities when storing unstructured data files. A single container was created which consisted of three separate folders, each defining the different data entities (view figure 10).

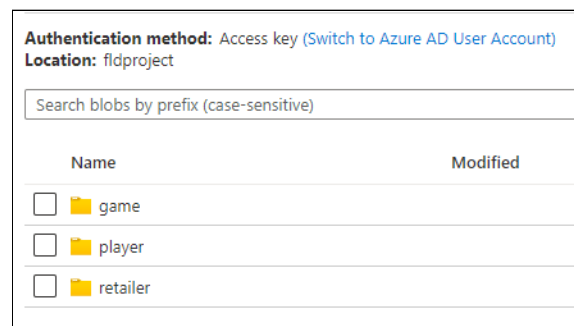


Figure 10: Azure Storage Account

Each of these folders contained another set of three folders, which represents all possible locations for a file to be stored, as seen on figure 11.





Authentication method: Access key (Switch to Azure AD User Account)	
Location: fldproject / game	
Search blobs by prefix (case-sensitive)	
Name	Modified
<input type="checkbox"/>  [.]	
<input type="checkbox"/>  error	
<input type="checkbox"/>  history	
<input type="checkbox"/>  processing	

Figure 11: File Locations Within a Data Entity Folder

When files are first transferred from their local directories to the Azure storage account, they're placed into their respective data entity's 'processing' folder. Here, these files will wait for the data factory pipeline to be executed before they are moved again. Upon successful completion of copying the data file from the 'processing' folder to the SQL database, data files are placed into the 'history' folder where all other successfully transferred files are stored. However, in case of an error while trying to perform the copy activity, files will be transferred to the 'error' folder where they will be further examined by an administrator.

Azure Data Factory Pipeline Configuration

Step 1: Download and Configure On-Premises Data Gateway

Reference [Azure Logic App section](#)

Step 2: Initialize Variables

Directly following the file system trigger and before the data factory pipeline is executed, variables must be initialized and set with information regarding the files that were added to the local directory (view figure 12).

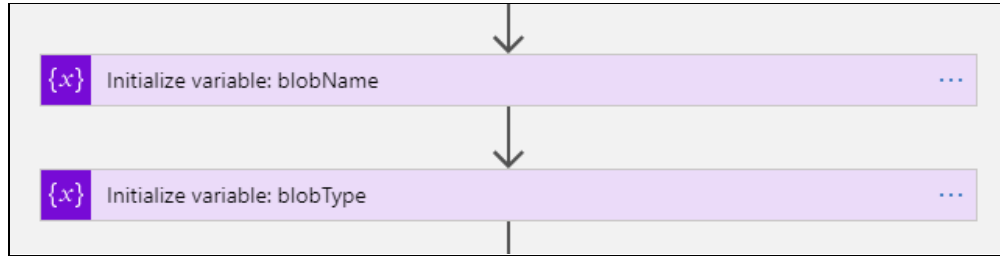


Figure 12: Variable Initialization

The “Initialize variable: blobName” activity dynamically stores the name of the data file, while the “Initialize variable: blobType” uses a custom expression to extract the entity type from the data file name, as seen in figure 13.

```
substring(variables('blobName'), 0, indexOf(variables('blobName'), '_'))
```

Figure 13: Code Expression to Determine Blob Type

These two variables will be passed into the data factory pipeline and help locate where the data is stored within the online storage account.

Step 3: Create Blob

After recording information about the data file, a new blob is created within the Azure storage account. As seen in figure 14, the folder path within the container is dynamically specified so that file can be placed in its appropriate entity's folder. The other fields, blob name and blob content, are also dynamically inputted using outputs from previous logic app actions. This is the last step before the Azure data factory pipeline is called.

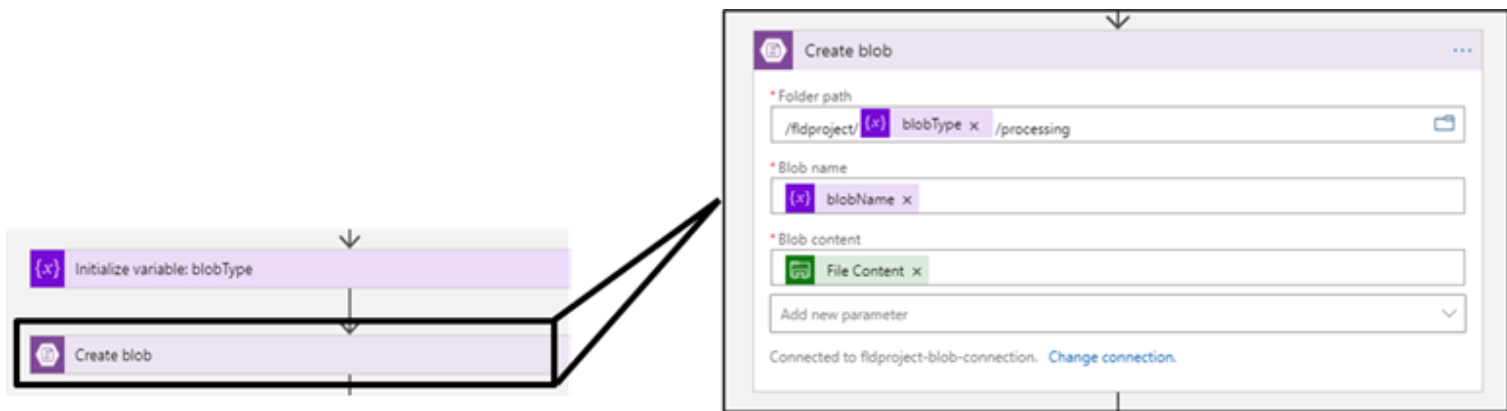


Figure 14: Create Blob Activity

Step 4: Create a Pipeline Run

Once the blob has been successfully added to the storage account, the next step would be to initiate a pipeline run via Logic Apps. Figure 15 shows the required parameters needed to call upon a requested pipeline. In order to pass information about the newly created blobs into the pipeline, the “parameters” field needs to be utilized by inputting a JSON array corresponding to the exact variables in the pipeline. In other words, the string variables “blobName” and “blobType” must exactly match the parameter names (and data types) created in the “final_pipeline_FLDProject” pipeline, as seen in figure 15.

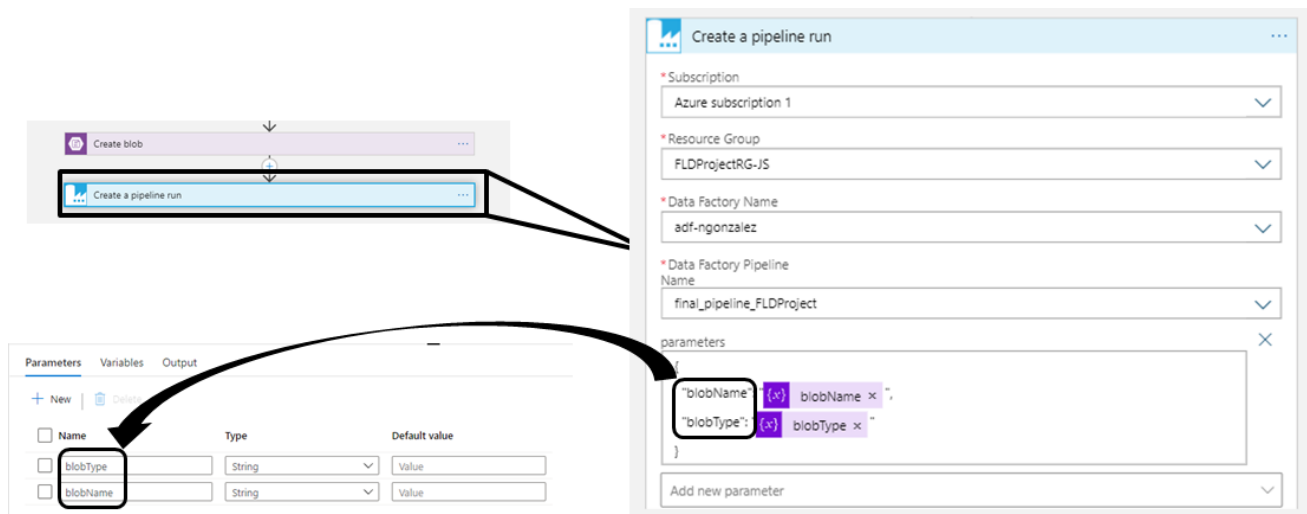


Figure 15: Create a Pipeline Activity

Configuring Data Factory Pipeline

Step 1: Create Data Factory resource following Microsoft Documentation

<https://docs.microsoft.com/en-us/azure/data-factory/quickstart-create-data-factory-portal#create-a-data-factory>

Be sure to configure the resource group, region, and name of resource to project specific needs.

Step 2: Select “Create Pipeline” from home tab

To create a new pipeline, select the plus symbol on the ‘Factory Resources’ task pane towards the left of the Azure Data Factory workspace, and then select ‘Pipeline’ (view figure 16). A ‘Properties’ task pane will appear on the right prompting the user to input the pipeline name.

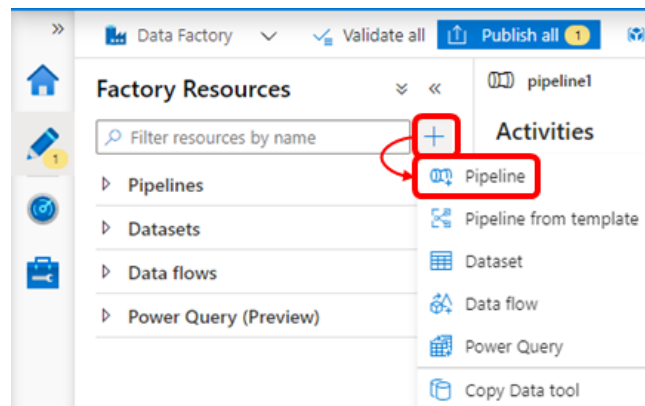


Figure 16: Creating New Pipeline

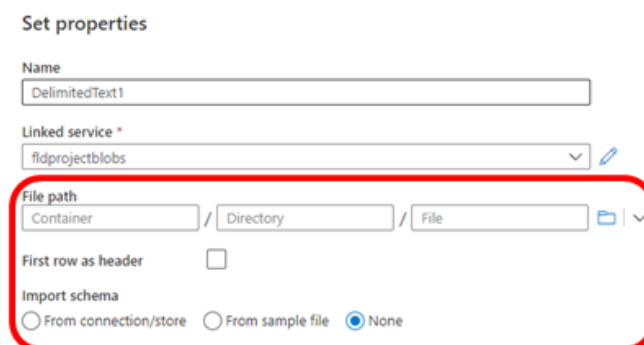
Step 3: Create linked service to storage account

Navigate to the Manage tab in the Data Factory UI to manage connections. Follow these [directions](#) from Microsoft documentation to create a linked service for a blob storage container.

Step 4: Create Dataset for Azure Blob Storage

To create a new dataset, select the plus symbol on the ‘Factory Resources’ task pane towards the left of the Azure Data Factory workspace, and then select ‘Dataset’. Select ‘Azure Blob Storage’ as the data store, followed by the desired format type of data (in the case of this project, it’s delimited text). The user will then

be prompted to name the dataset and connect it to the appropriate storage account. As seen in figure 17, leave the following fields blank.



Set properties

Name
DelimitedText1

Linked service *
fldprojectblobs

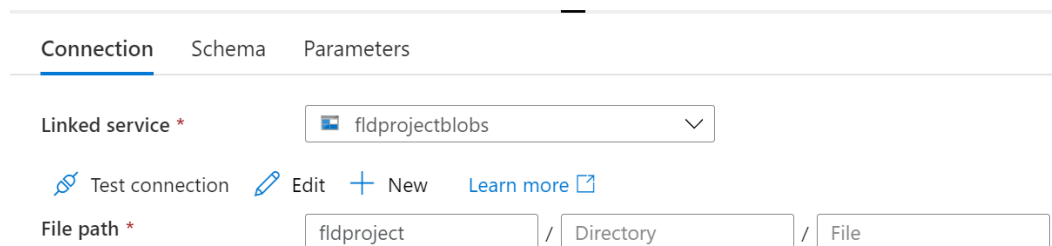
File path
Container / Directory / File

First row as header ☐

Import schema
☐ From connection/store ☐ From sample file ☒ None

Figure 17: Leaving File Path Blank for Dynamic Capabilities

File path only specifies the container name 'fldproject' which contains the three directories for 'game' 'player' and 'retailer'. By leaving the rest of the file path input blank, we can dynamically reference the dataset to one of the three entities when used in a pipeline activity (view figure 18).



Connection Schema Parameters

Linked service *
fldprojectblobs

Test connection Edit New Learn more

File path *
fldproject / Directory / File

Figure 18: File Path Specification

Step 5: Create linked service to Azure SQL Database

Follow similar directions as in step 3 to create a new linked service, but this time select Azure SQL Database as the type. Use the built in integration runtime along with the authentication information for the SQL database.

Step 6: Set variable activities for process log table

Each pipeline execution represents the processing of one file; information about each run is to be stored in a 'process log' table that keeps track of the pipeline's history. Each record in this 'process log' table will be uniquely identified by a

sequence of numbers that is derived from information about a specific file. This 'file ID' has three different components and is constructed at the beginning of the workflow by using several 'Set variable' activities. The three components are file entity type (game, player, retailer), file date creation, and file time creation which are all extracted from the file name, as seen in figure 19.

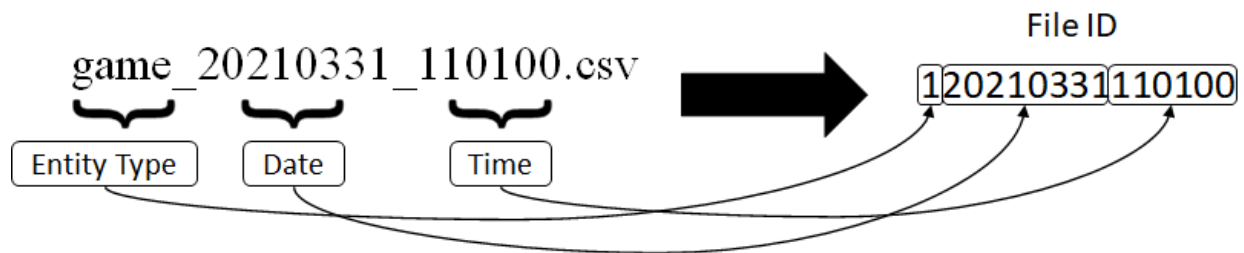


Figure 19: File ID Derivation

Since the file name was passed into the workflow as a parameter from Azure Logic Apps, and is stored in the pipeline parameter 'blobName'. Figure 20 shows how all components required to construct the File ID are retrieved by utilizing Azure Data Factory's dynamic content capabilities along with advanced string manipulation, where the pipeline parameters are highlighted to show they are utilized.

Time component

```
@substring(pipeline().parameters.blobName,add(lastindexof(pipeline().parameters.blobName,'_'), 1) , 6)
```

Date component

```
@substring(pipeline().parameters.blobName,add(indexOf(pipeline().parameters.blobName,'_'), 1), 8)
```

Overall File ID

```
@concat(if(startswith(pipeline().parameters.blobType,'player'),1,
if(startswith(pipeline().parameters.blobType,'game'),2, 3)), variables('date'),variables('time'))
```

Figure 20: Expressions for File ID Construction

Step 7: Configure Copy Data Activity to copy data from blob storage to SQL database

In the Activities Menu, expand Move & Transform to drag and drop the "Copy Data" activity to the pipeline (view figure 21).

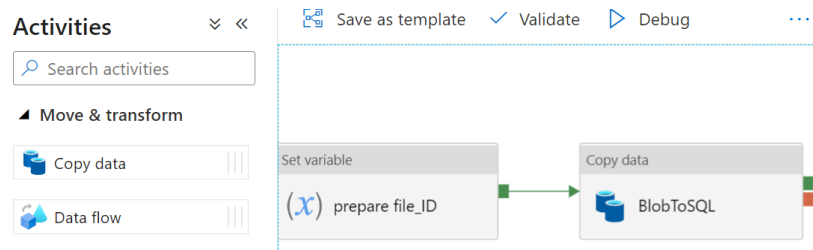


Figure 21: Copy Data Activity

This copy activity will directly follow the last Set variable activity from the previous step. Click on the Copy data activity to begin configuration (view figure 22).

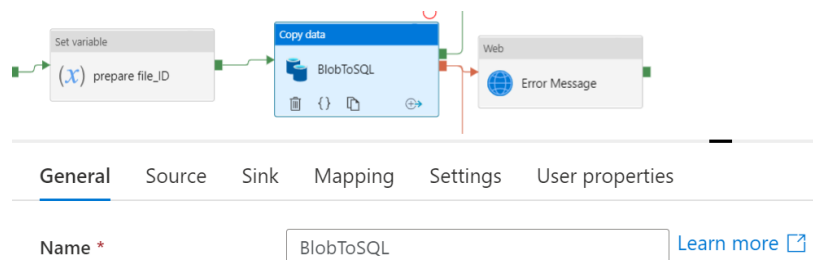


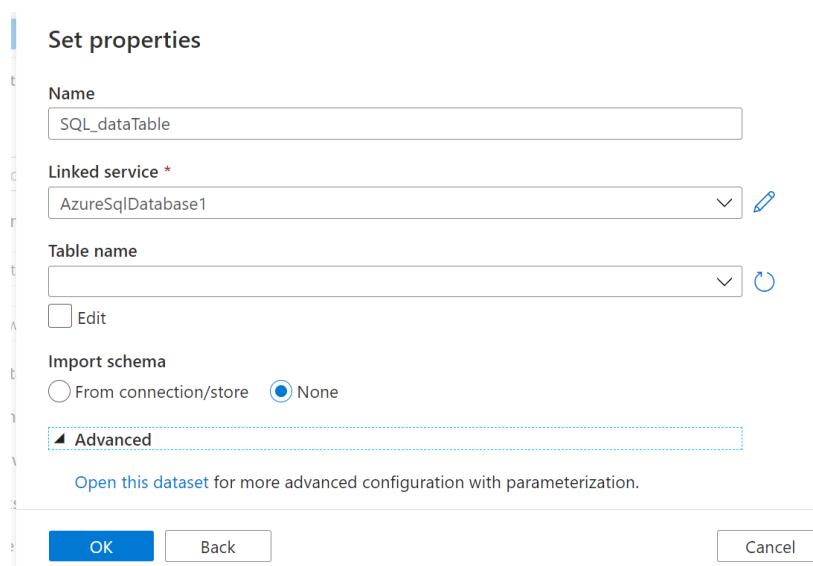
Figure 22: Copy Data Activity (continued)

In the 'General' tab, specify the name of the activity. Under the 'Source' tab, set the source dataset to the blob storage dataset created in step 4. Choose 'Wildcard file path' as the 'File path type'. This is how we will utilize the pipeline parameters to dynamically reference the proper blob storage folder that corresponds to the file type being processed. Fill in the path as done in Figure 23.

Figure 23: Using Parameters to Specify File Path Location on Storage Account

Recalling from the Logic App portion of this prototype, the blobType variable is equivalent to the entity type (game, player, or retailer), and the blobName is the complete name of the file that should be located in the /processing folder at the time of the activity.

Next is the 'Sink' settings. Since we want the copy activity to copy the file content into our Azure SQL Database, the sink must be the dataset used to represent the SQL database table. We must create this dataset by choosing '+ New' next to the Sink dataset selection. Select the Azure SQL Database in the window that opens up, and it will take you to a window to set the properties of the dataset (view figure 24).



The screenshot shows a 'Set properties' dialog box for configuring a dataset. The 'Name' field is set to 'SQL_dataTable'. The 'Linked service' dropdown is set to 'AzureSqlDatabase1'. The 'Table name' field is empty, with an 'Edit' checkbox below it. Under 'Import schema', the 'None' radio button is selected. An 'Advanced' section is expanded, showing a link to 'Open this dataset' for more configuration. At the bottom are 'OK', 'Back', and 'Cancel' buttons.

Figure 24: Setting up SQL Table Sink

Specify a name for the dataset and select the Linked Service created in step 3 to connect to the database. The specific table name must be dynamic since it depends on the file type being processed. To do this, we must parameterize the dataset similar to how we parameterized the pipeline because a dataset does not have access to the pipeline parameters directly. Before selecting a Table name, click the 'Open this dataset' link under Advanced settings in order to parameterize the table name. Navigate to the Parameters tab and create a new parameter as shown in Figure 25.

Name	Type	Default value
tableName	String	tableName

Figure 25: Parameterizing Table Name

Now that the parameter exists, we can reference the parameter under the Connection settings as the value for table name as shown in Figure 26.

Linked service * AzureSqlDatabase1

Test connection Edit + New Learn more

Table dbo . @dataset().tableName

Edit

Figure 26: Referencing Dynamic Table Name

The dataset is fully configured. Now redirect to the copy activity sink settings and the dataset parameter should be visible under the dataset name. Use dynamic content to set the value of 'tableName' corresponding to the pipeline parameter 'blobType'. This is shown in Figure 27.

General Source Sink Mapping Settings User properties

Sink dataset * DestinationDataset_tw0 Open + New Learn more

Dataset properties

Name	Value	Type
tableName	@if(startswith(pipeline().parameters.blobType, 'player'),'Players', if(startswith(pipeline().parameters.blobType, 'game'),'Games', 'Retailers'))	string

Add dynamic content

```
@if(startswith(pipeline().parameters.blobType, 'player'),'Players',
if(startswith(pipeline().parameters.blobType, 'game'),'Games', 'Retailers'))
```

Figure 27: TableName Parameter Set to Dynamic Content

For this prototype, the default method for Mapping is used, which maps the data from the source to the sink based on column names being equivalent. This is impractical for many systems, so Data Factory offers options for explicit mapping

and parameterized mapping which more information can be found here:
<https://docs.microsoft.com/en-us/azure/data-factory/copy-activity-schema-and-type-mapping#explicit-mapping>

Step 8: Configure Copy Activity to move file from 'processing' to 'history'

After the data from the blob container fldproject/blobType/processing/blobName (i.e. fldproject/game/processing/game_YYYYMMDD_HHMMSS) is successfully copied to the SQL database table in the previous copy data activity, the file must be moved to an archive folder called 'history'. To do this, another copy data activity must be created that will be set up just like the BlobtoSQL activity, except the sink dataset will represent the blob container 'history' instead of the SQL database table. As seen in Figure 28, the copy data activity MoveToHistory is executed after successful completion of the BlobToSQL activity.

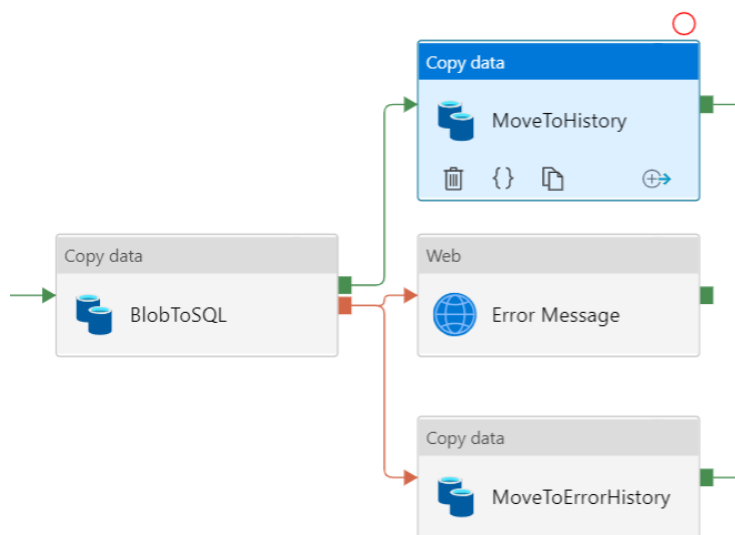



Figure 28: Copy Data Activity, Moving Files to History Folders

The configuration of the 'Source' settings for this activity is exactly the same as in Step 7 since the data is coming from the /processing/ folder of the azure blob storage container. The 'Sink' settings requires a new dataset to be created, which we called 'blob_history_dataset'. This dataset must contain a parameter to allow dynamic content to be used as the file path input. The configuration of the dataset settings 'Parameters' and 'Connection' can be seen in Figure 29.



DelimitedText
blob_history_dataset

Connection
Schema
Parameters

+ New
Delete

Name	Type	Default value
<input type="checkbox"/> folder_directory	String	Value

Connection
Schema
Parameters

Linked service *
fldprojectblobs
Test connection
Edit
New
Learn more

File path *
fldproject
/
@dataset().folder_directory
/
File
Browse

Compression type
None

Column delimiter ⓘ
Comma (,)
Edit

Row delimiter ⓘ
Default (\r,\n, or \r\n)
Edit

Encoding
Default(UTF-8)

Escape character
Backslash (\)
Edit

Quote character
Double quote (")
Edit

First row as header
☒

Figure 29: Dataset Configuration for Blob History

Finally, set the new dataset as the 'Sink dataset' and specify the value of the parameter using dynamic content (@concat(pipeline().parameters.blobType,'/history')) as shown in Figure 30.

General
Source
Sink
Mapping
Settings
User properties

Sink dataset *
blob_history_dataset
Open
New
Learn more

Dataset properties ⓘ

Name	Value	Type
folder_directory	@concat(pipeline().parameters.blobType,'/')	string

Figure 30: folder_directory Parameter Initialization

Step 9: Configure Copy Activity to move file from 'processing' to 'error'

Repeat the steps for configuring the 'MoveToHistory' activity to create a 'MoveToErrorHistory' activity that will execute if the 'BlobToSQL' activity fails. Use the same dataset, 'blob_history_dataset', as the sink and change the 'folder_directory' value to "@concat(pipeline().parameters.blobType,'/error/')" so the file will be moved to its corresponding /errors/ folder instead of /history/.

Step 10: Delete file from 'processing'

After a file has been moved either to its respective entity's 'history' or 'error' folder, it needs to be deleted from the 'processing' folder where it was originally placed. This step can be achieved from within the Azure Data Factory pipeline by using a delete activity. After adding the activity to the workflow, click the Source tab from the expanded pane, and select the Azure storage account dataset.

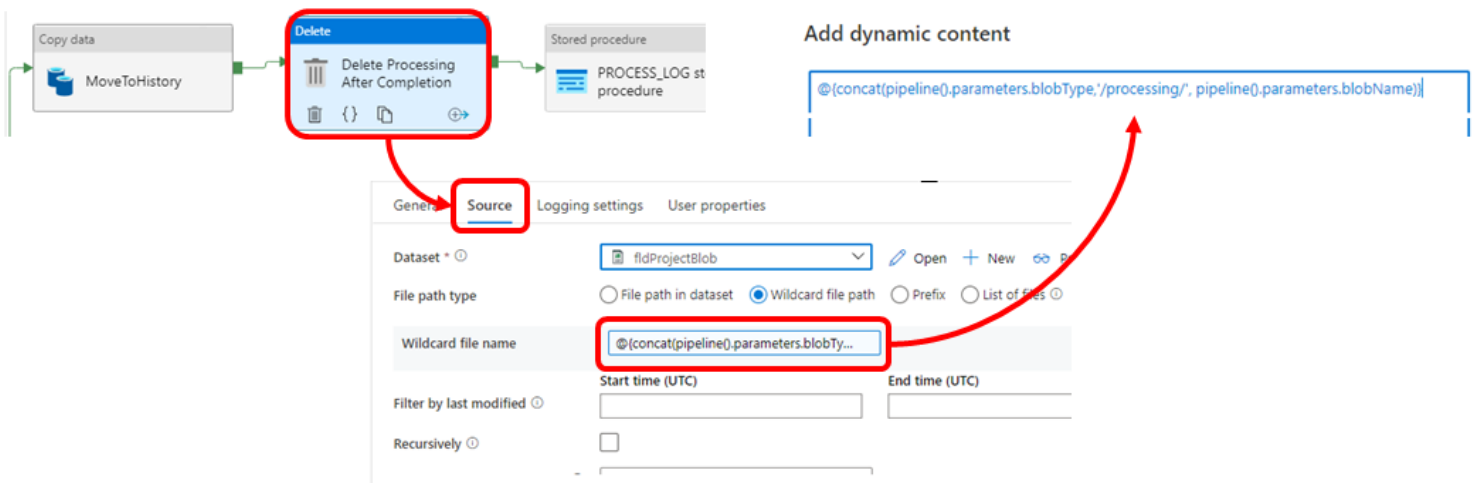


Figure 31: Delete Activity Configuration

As figure 31 shows, 'Wildcard file path' must be selected as the file path type so that the file name can be inputted dynamically. Then, using the 'concat' string method from the 'Add dynamic content' section, the file name can be appropriately constructed by combining the passed in pipeline parameters from Azure Logic Apps.

Step 11: Execute stored procedure

As previously mentioned in step 6, details about each pipeline execution are to be collected and stored in a SQL table.

Under the activities menu in the data factory UI, under the 'General' drop down list, there's an activity titled 'Stored Procedure' which will execute an existing stored procedure that it's connected to via the SQL database linked service. Drag and drop the activity to follow the successful completion of the 'Delete' Activities created in the previous step as shown in figure 32.

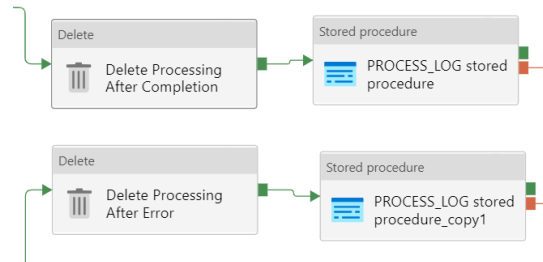


Figure 32: Stored Procedure Activities

See [Creating Stored Procedure in SQL Database](#) for instructions on how to create the stored procedure. The parameters declared in the stored procedure will show up in the settings of the activity, and must be filled in with fixed or dynamic values. Configuration of the parameters in the pipeline is shown in figure 33 below.

There are five details to be stored in the logging table: the file ID which uniquely represents each pipeline run, the file name and type (represented by the pipeline parameters 'blobName' and 'blobType'), and the process datetime and status. The process status can take on either two values, 'success' or 'failure' depending whether the pipeline was able to successfully transfer the file contents to the SQL database.

General Settings User properties			
<input type="checkbox"/>	Name	Type	Value
<input type="checkbox"/>	file_ID	String	@variables("file_ID")
<input type="checkbox"/>	file_name	String	@pipeline().parameters.blobName
<input type="checkbox"/>	file_type	String	@pipeline().parameters.blobType
<input type="checkbox"/>	process_datetime	String	@utcnow()
<input type="checkbox"/>	process_status	String	'Success'

Figure 33: Inputting Parameters for Stored Procedure

Previous activities cannot lead into the same activity, since doing so creates an and-type relationship where both previous activities need to occur before the next activity, which is not possible in the pipeline. Therefore, there are two stored procedure activities, one after a successful pipeline run and the other after a failed run.

Step 12: Create Error Messages

Another functionality that was added to the workflow was the ability to send email notifications to the administrators when particular errors occur. Currently, the pipeline includes error alerts to be sent at two different locations in the workflow: after attempting to transfer the file contents to the SQL database and after attempting to add the pipeline execution details to the logging table.

When one of these activities fail, an HTTP POST is made thus triggering a separate Azure Logic Apps workflow, as seen in figure 35. Information about the particular failure in the data factory pipeline is listed in the body of the POST method, which is formatted as a JSON array. In order to have the body message configured this way, enter 'Content-Type' for the name field and 'application/json' for the value field under the 'Headers' section, as seen in figure 34.



Figure 34: JSON Array With Pipeline Error Information

When the HTTP POST method is received, a simple two-step workflow in Logic Apps is triggered (view Figure 35). The initial trigger action processes the HTTP body with a previously generated JSON schema. Afterwards, an email is sent to a specified address containing the information passed in from the HTTP POST method, where the email's body is now formatted appropriately using a predefined template.

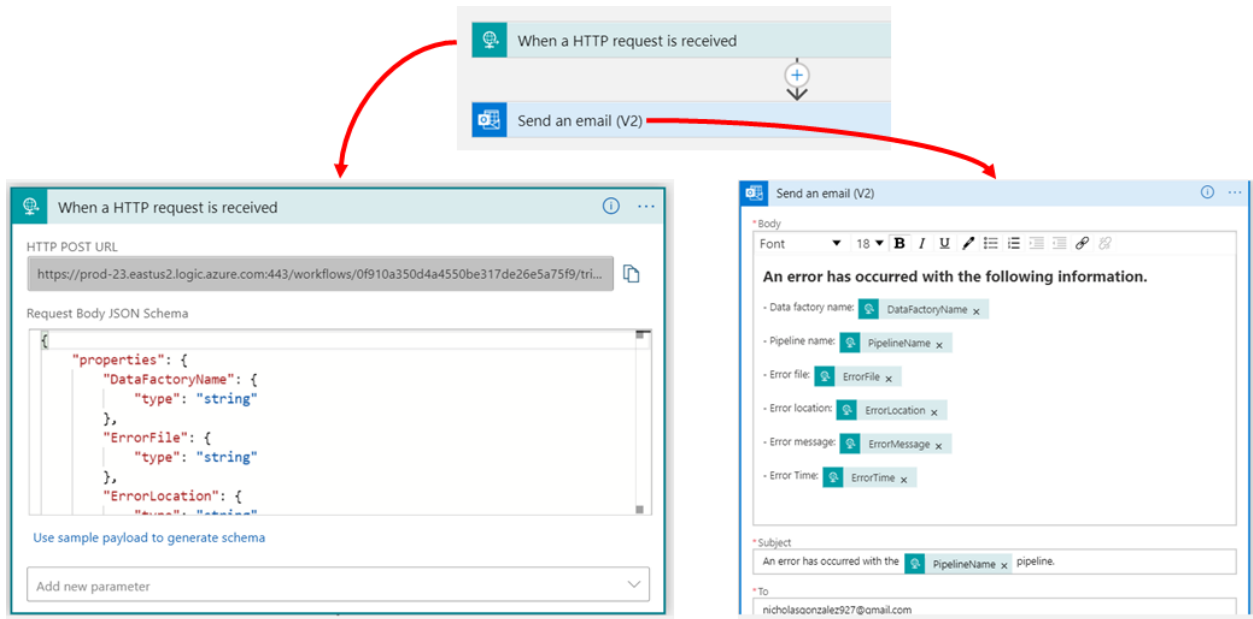


Figure 35: Logic App Workflow for Email Notification

Creating/Connecting to SQL Server and Azure SQL Database

For this project, we utilized Azure SQL Database as our data store since it was easily configured through Microsoft applications and could easily be connected to in our workflows. To create the database, which required creation of a SQL Server since we did not have an existing one, and then connect to SSMS for editing, follow this [tutorial](#):

For the prototypes, we created 4 tables, 'dbo.FILE_PROCESS_LOG', 'dbo.Games', 'dbo.Players', and 'dbo.Retailers'. The values were all of type nvarchar(50). Examples of the final table design and example entries are shown in Figure 36-39.

FILE_PROCESS_LOG:

	file_ID	file_type	file_name	process_datetime	process_status
1	120210406111600	player	player_20210406_111600.csv	2021-04-06T15:18:32.4913566Z	'Success'
2	120210416111111	player	player_20210416_111111.xml	2021-04-17T00:33:57.9303940Z	'Success'
3	120210416222222	player	player_20210416_222222.xml	2021-04-17T00:27:58.0284890Z	'Success'

Figure 36: File Process Log Table

Games:

	GameID	Title	Region	Address	TotalGamesPlayed	GamesWon	GamesLost	AmountWon	Revenue
1	200	ScratchOff	East	000 Main St N	100	3	97	\$25.00	\$100.00
2		Powerball	West	123 55th In	4	2	2	250	100

Figure 37: Game Table

Players

	PlayerID	FirstName	LastName	Address	Email	Phone	AmountWon	PaymentType
1		Jessica	Silang	123 SW 1st	jessica@email.com	2387765678	100	Cash

Figure 38: Player Table

Retailers

	RetailerID	RetailerName	Region	Address	Email	Phone	CurrentInventory	AmountSold	Backordered	BalanceOwed
1	1	Store	East	000 Main St N	fakeretailer@gmail.com	555-555-5555	1000	100	0	\$2,000

Figure 39: Retailers Table

Creating Stored Procedure in SQL:

For more details on this process, refer to Microsofts' [tutorial](#):

Our team utilized Microsoft SQL Server Management(SSMS) studio to edit, view, and query our Azure SQL Database. From the UI of SSMS, there is a simple way to quickstart creating a stored procedure query. In the Object Explorer, there is a dropdown menu under <server-name> -> <database name> -> <Programmability> -> <Stored Procedures>. By right clicking on <stored Procedures>, there is an option to create a new procedure, which will open a new query with template code as shown in Figure 40.

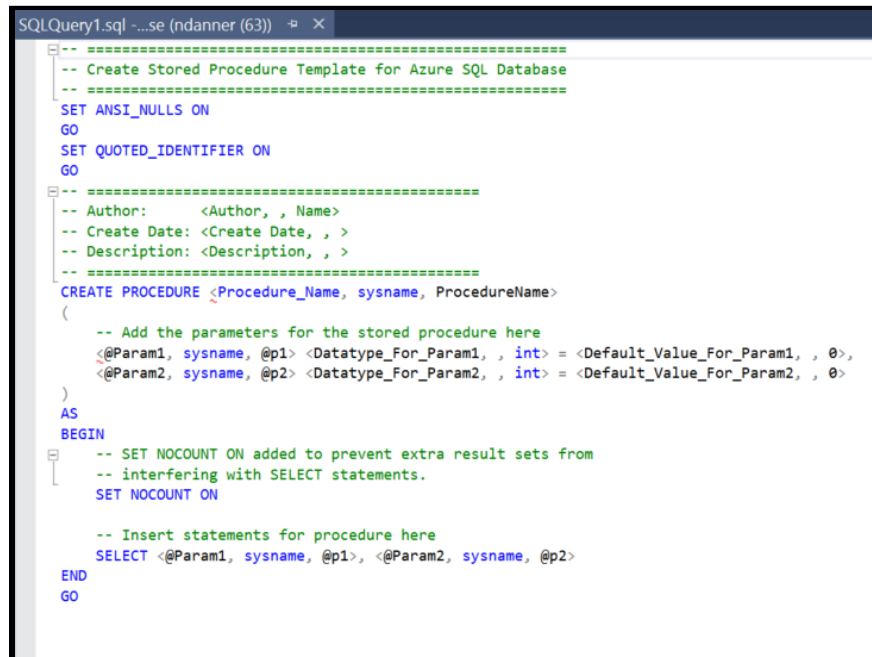
The image is a screenshot of a SQL query editor window titled 'SQLQuery1.sql - ...se (ndanner (63))'. The query is a template for creating a stored procedure. It starts with 'SET ANSI_NULLS ON' and 'SET QUOTED_IDENTIFIER ON', followed by 'GO'. Then, it has a section for metadata: 'Author: <Author, , Name>', 'Create Date: <Create Date, , >', and 'Description: <Description, , >'. The main body starts with 'CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>' followed by an opening parenthesis. Inside the parenthesis, there are two parameter definitions: '<@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>' and '<@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>'. This is followed by 'AS', 'BEGIN', and a comment 'SET NOCOUNT ON added to prevent extra result sets from interfering with SELECT statements.' followed by 'SET NOCOUNT ON'. Then, there is a comment 'Insert statements for procedure here' followed by a 'SELECT' statement: 'SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>'. The query ends with 'END' and 'GO'.

Figure 40: Template Code Query

You may easily fill in custom values for the parameters given in the template by navigating the 'Query' menu and selecting 'Specify Values for Template Parameters'. For this project, the parameters related to the columns in our FILE_PROCESS_LOG Table, so we could execute entering the file metadata into the table in one easy step in both the Azure Logic Apps workflow, and the ADF pipeline. The query used to modify the procedure if needed is shown in Figure 41.


```

SQLQuery2.sql -...se (ndanner (71))  SQLQuery1.sql -...se (ndanner (63))
/***** Object: StoredProcedure [dbo].[insert_FileProcessLog_procedure]    Script Date: 4/18/2021 9:4
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- Author:      Natalie Danner
-- Create Date:
-- Description: Insert adf pipeline parameters into FILE_PROCESS_LOG table
-- ****
ALTER PROCEDURE [dbo].[insert_FileProcessLog_procedure]
(
    -- Add the parameters for the stored procedure here
    @file_ID nvarchar(50) = NULL,
    @file_type nvarchar(50) = NULL,
    @file_name nvarchar(50) = NULL,
    @process_datetime nvarchar(50) = NULL,
    @process_status nvarchar(50) = 'Fail'
)
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON

    -- Insert statements for procedure here
    INSERT INTO FILE_PROCESS_LOG (file_ID, file_type, file_name, process_datetime, process_status)
    VALUES (@file_ID, @file_type, @file_name, @process_datetime, @process_status)
END

```

Figure 41: SQL Query

Our Final Recommendation

In order to provide a qualitative analysis of the two softwares- Logic Apps and Data Factory, we created a binary comparison matrix as well as a decision matrix. The combination of these two matrices allowed us to outline what categories were covered or supported by each platform and then form general categories, selection weights, and total scores for both.

The softwares were given a score between 1 and 5 abiding by these standards:

- **5=** Excellent
- **4=** Very Good
- **3=** Good
- **2=** Fair
- **1=** Poor

Note: The selection weights and scores were determined by the developers based on their knowledge and experience with the two platforms.

The following criteria the platforms were scored on include:

- *User Friendliness*

- Definition: User-friendly describes a hardware device or software interface that is easy to use. It is "friendly" to the user, meaning it is not difficult to learn or understand (per [TechTerms](#)).
- Reasoning: Coming into this project, none of the team's developers had previous experience working in the Azure portal. The purpose of this category is to showcase specific features that we thought made implementation much easier.
- *Connectivity*
 - Definition: The ability to connect systems or application programs (per [IBM](#)). The purpose of doing so is to develop an application or enhance the functionality of an existing one (per [PCmag](#)).
 - Reasoning: The data used throughout this project needed to be retrieved, processed, and stored across a multitude of different applications. The purpose of this category is to highlight a platform's flexibility when it comes to connecting to other services and applications.
- *Supportability*
 - Definition: The inherent characteristics of the system and the enabling system elements that allow effective and efficient sustainment (including maintenance and other support functions) throughout the system's life cycle (per [DAU](#)).
 - Reasoning: We often found that when trying to examine how a workflow performed or where specifically it failed, it was necessary to have platform capabilities for viewing previous runs and triggers. The purpose of this category is to note how these platforms offer support after the workflow is finished being developed.
- *Functionality*
 - Definition: In information technology, functionality is the sum or any aspect of what a product, such as a software application or computing device, can do for a user. A product's functionality is used by marketers to identify product features and enables a user to have a set of capabilities (per [TechTarget](#)).

- Reasoning: Functionality was the leading factor in determining how effective a platform was going to be throughout the project. The purpose of this category is to highlight some of the key features utilized while developing both workflows.

The finalized matrices are shown in Figures 42 and 43. Figure 42 can be used to determine which platform supports certain functionalities and can help developers decide which service to use for a particular problem. In Figure 43, the final scores are shown. The two scores are fairly comparable with Azure Logic Apps scoring a 4 and Azure Data Factory scoring a 4.3. With this knowledge, the team's final recommendation is to integrate the two softwares to utilize them for their respective strengths.

Figure 42: Comparison Matrix

Criteria	Weight	Azure Logic Apps	Azure Data Factory
User Friendliness	20%	4	3
Connectivity	30%	5	4
Supportability	10%	5	5
Functionality	40%	3	5
Total Score		4	4.3

Features	Azure Logic Apps	Azure Data Factory
Simplified User Interface (e.g. accessible task panes, run buttons, code views)	✓	✓
Drag and Drop Configuration	✓	✓
Intuitiveness for Beginner Programmers	✓	□
Efficient Modification of Workflow Sequence	□	✓
Extensive Documentation available	✓	✓
Post Execution Analysis and Debugging	✓	✓
Windows Compatible	✓	✓
Ability to Connect to Microsoft Outlook	✓	□
Ability to Connect to SQL Server	✓	✓
Wide access to third-party applications	✓	✓
Access Azure Storage Accounts	✓	✓
Ability to generate a structured PDF	✓	✓
View previous runs and triggers	✓	✓
Detailed dashboards of log and metric data through Azure Monitor	✓	✓
Alert and notification monitoring through Azure Monitor	✓	✓
File System Trigger Capability	✓	□
Stored Procedure Execution	✓	✓
Multiple File Processing	✓	□
Dynamic Content Usage	✓	✓
Capacity to handle a large amount of data in single activity	□	✓
Copy data directly from flat file to structured data store	□	✓
Data Element Mapping (Explicit Mapping)	□	✓
Efficiently copies data in one step	□	✓
User Defined and System Variables	✓	✓
Parameterization	✓	✓
Facilitated CSV to JSON Transformation	□	✓

Figure 43: Weighted Decision Matrix

Additional Information

With more time, our team would have attempted to explore potential improvements to these foundational prototypes we have provided. Instead, we have compiled some relevant resources that can help the implementation process in the future.

Potential Areas of Improvement

1. CSV to JSON Conversion

If the timeline for this project had been longer, our team would have looked into the capabilities of implementing a CSV→JSON conversion for the Logic Apps prototype. The route we took involved the file starting as an XML format and then being converted to JSON via the liquid templates. In order to do the CSV to JSON conversion, we found that we would have to use a third party application. Some roadblocks with this solution include cost, heavy

coding, and a shortage of time. Here are some links that go over the potential process of implementation.

- [Other Ways to Convert CSV to JSON](#)
- [Creating a CSV to JSON Converter From Scratch](#)

Another possible method that our team researched for converting from CSV included utilizing BizTalk Server, an integrated development environment that possesses tools for facilitating pipeline creations. More specifically, BizTalk Server offers the 'BizTalk Flat File Schema Wizard' application which automatically generates flat file schemas used to transform CSV files to other formats such as XML or JSON. Here are some links that explain how to get started with BizTalk Server.

- [Installing BizTalk Server in Visual Studio](#)
- [Using Flat File Schema Wizard to generate a Schema](#)
- [Using Flat File Schema in Logic Apps to convert CSV to XML](#)

2. Multiple File Processing For Azure Data Factory

As previously discussed, Azure Data Factory's capabilities for retrieving files from local directories was very limited. For this reason, when a new file is added to a local directory, a simplified Azure Logic Apps workflow is triggered. This Logic Apps workflow begins to process the file and eventually calls the Data Factory pipeline, passing in several details about the file. Furthermore, if multiple files are added at the same time, then multiple triggers are set off where each trigger is counted as a separate execution of the Data Factory pipeline. However, for a larger scaled implementation of this workflow, event-based triggers such as adding a file to local directory might not be the most efficient method. Below is a link to a Microsoft article which highlights how utilizing a scheduled-based trigger could potentially be more beneficial in comparison to an event-based trigger.

- [Schedule-based triggers in Azure Logic Apps](#)

3. Use of Explicit Mapping in Azure Data Factory

Currently, our Azure Data Factory prototype maps the CSV data to JSON using the column names. This works under the assumption that all files will be formatted uniformly- which is not always going to be a guarantee in reality. To provide a more foolproof method of mapping the raw data, we wanted to look more into explicit mapping and provide some foundational research in order for the Florida Lottery Department team to potentially implement this improvement in the future.

- [Schema and Data Type Mapping In Copy Activities](#)

4. FTP/ Security Checks

This suggestion came later on in our timeline. It ended up being out of our project scope, but we gathered a couple of sources that could be helpful in implementing this feature in the future.

- [Deploy your app to Azure App Service using FTP/S](#)
- [How to Deploy a Secure FTP on Microsoft Azure](#)

5. Generating a PDF within the WorkFlow

The ability to generate a structured PDF as a step in the workflows was another advanced task that we did not get to conquer, but we were able to find out different possibilities to integrate this feature.

In Logic Apps:

Given that Logic Apps is more diverse with the connectors they provide, there are a couple of connectors related to PDFs such as PDF4me and Adobe PDF tools that can help generate the PDF.

- [PDF4me Connector](#)

In Data Factory:

Completing this action in Data Factory is more complex in comparison to Logic Apps. It requires coding using the app Tabula, Python, and Azure Functions. This source outlines some potential solutions:

- [PDF Table Extraction Using Tabula](#)

6. Importing/Exporting and Git Repository

Azure resources have the ability to export and import as ARM templates. The team's prototypes were transferred to the Florida Lottery in that form. More help on how to import resources as templates can be found [here](#).

Azure data factory has the ability to connect to a git repository, whether that be through Github or Azure Devops. Additional information can be found [here](#).

7. Azure Monitor for building Logic Apps/Data Factory dashboards

Our Azure Data Factory workflow contains functionality for storing details about each pipeline execution within a separate logging table. However, Azure offers a more versatile platform for analyzing a workflow's performance called Azure Monitor. Azure Monitor provides an array of capabilities such as creating dashboards that visualize the current state of a workflow, sending smart alerts when errors arise in a process, and diagnosing potential issues across different applications. An overview of Azure Monitor is linked below in addition to articles on integrating it within a Logic Apps or Data Factory solution.

- [Azure Monitor Overview](#)
- [Azure Monitor for Azure Logic Apps](#)
- [Azure Monitor for Azure Data Factory](#)