Cloudera Data Flow Functions in Azure Functions

Date published: 2021-04-06
Date modified: 2025-09-30

CLOUDERA

Legal Notice

Contents

Creating your first Azure Function App

Azure Functions is a serverless computer service that allows you to write less code, maintain less infrastructure, and save on costs. Instead of deploying and maintaining servers, the cloud infrastructure provides all the up-to-date resources needed to keep your applications running. In Azure Functions, a function app provides the execution context for your individual functions.

About this task

Follow these steps to create an Azure Function App that is able to run Cloudera Data Flow Functions:

Procedure

1. Navigate to the Azure Portal in your cloud account and click the Function App service.
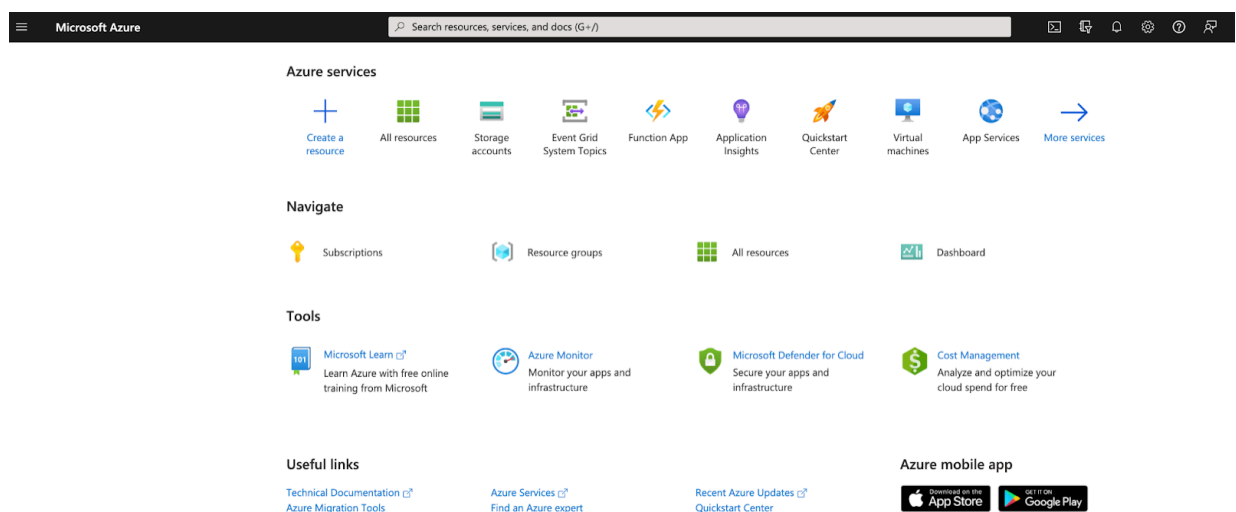
2. Click Create Function App.
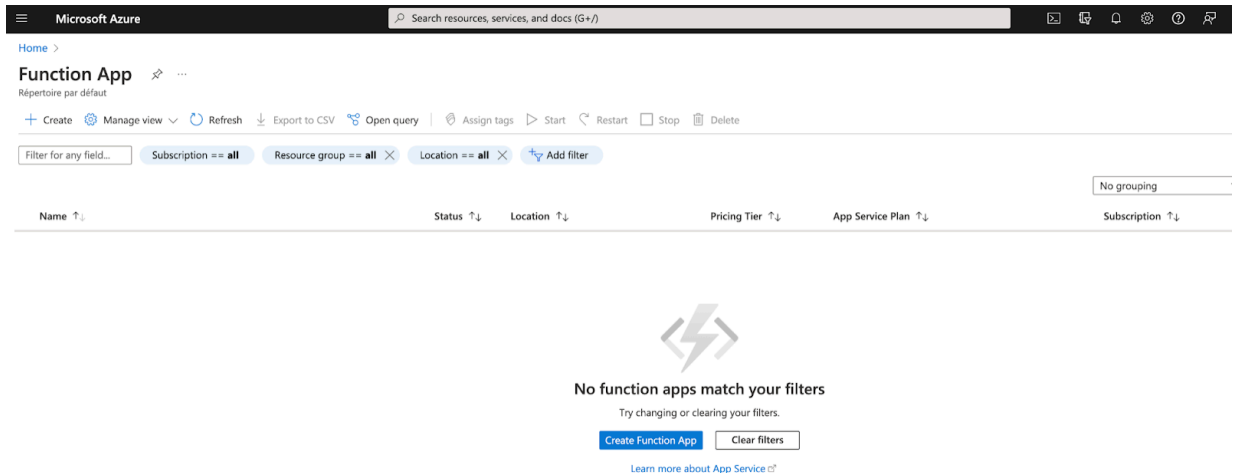
You can now create your Function App:

a) Select the Subscription and the Resource Group that you want to use.

You can also create a new one, dedicated to this Function App.

b) Provide a name for your Function App.

c) Select Java for Runtime stack.

d) Select 11.0 for Version.

e) Select the Region where you want to deploy your Function App.

f) Select the Operating System.

You can keep Windows, or you can choose Linux.

> **Note:** Choosing Linux specifically on a "Consumption" plan is not recommended for Cloudera Data Flow Functions due to a comparative lack of diagnostic, debugging, and logging support.

g) Select the Plan type that you want to use.

Azure has multiple plan types which should be weighed depending on your use case. The default is Consumption, which means that no dedicated servers are provisioned and your Function App is automatically scaled completely based on consumption. There are many differences between these plans, including memory

limits, timeout limits and networking support. For a full comparison of the plans, see Azure Functions hosting options.

After completing the Basics tab, you can either click Next : Hosting > to provide additional details for your function app, or if you want to skip these configurations, click Review + create.

Note: You are advised to go through the additional configuration options as well, as they cannot be changed later.

3. Click Next : Hosting > to continue creating the function app in the wizard.

You can specify an existing an existing or create a new Azure Storage account.

4. Click Next : Networking > to move forward in the wizard.

Depending on the Service Plan you selected, here you may be able to enable network injection.

5. Click Next : Monitoring > to go to the next page.

Application Insights will automatically be integrated unless disabled here.

6. Click Review + create.

At this point, you can safely skip the Tags tab and click Review + create as tags can be set later.

You will get a summary of your Function App details.

7. Click Create to create your Function App, and wait for the deployment to complete.

> **Attention:**
> At this point, some additional steps are required if you selected the App Service Plan Type:

a. Navigate to the storage account selected or created before, and click Access keys.
b. Click Show keys at the top, and copy the key1 "Connection string" value.
c. Go back to the Function App, click Configuration New application setting to add a new setting named WEBSITE_CONTENTAZUREFILECONNECTIONSTRING, and paste the copied value as the setting value.
d. Add another Application setting named WEBSITE_CONTENTSHARE with a value of "-fileshare".
e. Click Continue Save at the top.

This creates a file share to back your Function App that you can easily access from your storage account.

> **Attention:**
> At this point, if you selected a Consumption plan with Linux Operating System, follow these steps to upload the Cloudera Data Flow Functions binary:

a. Follow the instructions in the *Managed Identity* section to create a managed identity for your Function App.
b. Navigate to the new storage account you created, and select the Containers blade.
c. Click + Container on the top left and specify a name for a new Blob Container (e.g., "naaf").
d. Click Create.
e. Click the new container and select the Access Control (IAM) Role assignments .
f. Click Add at the top left, and specify Add role assignment.
g. Select Storage Blob Data Reader and click Next.
h. Change Assign access to to Managed identity.
i. Click the Select members link.
j. Under Managed identity, select Function App and click the name of your Function App.
k. Click Select and Review + assign.
l. Go back to the new Container, click Upload at the top left and upload the NaafAzureFunctions.zip file provided by Cloudera.

For Windows based Function App or non-Consumption plans, follow these steps to upload the Cloudera Data Flow Functions binary to your newly created Function App:

a. Navigate to the storage account selected or created before, and click the File shares blade under Data storage
b. Click the file share name.

   You can see that this share has a 5 TB quota. It will be mapped to /home on the Function App filesystem.

c. Click the data directory (or create it if it does not yet exist), and click Add directory at the top to create a directory named SitePackages.

d. Enter the new directory and upload the NaafAzureFunctions.zip file.

e. Create a file named packagename.txt with the contents of NaafAzureFunctions.zip and upload it as well.

% echo   "NaafAzureFunctions.zip" > packagename.txt

8. The Function app is now created and the binary is in place, but you still need to configure the Application Settings to complete the deployment.

See the *General configuration* section for configuration details.

General configuration

You must configure the function to specify the dataflow to run and add any necessary runtime configuration using environment variables.

Procedure

1. Click Configuration under Settings on the left of your Function App screen.
2. Select the Application Settings tab.
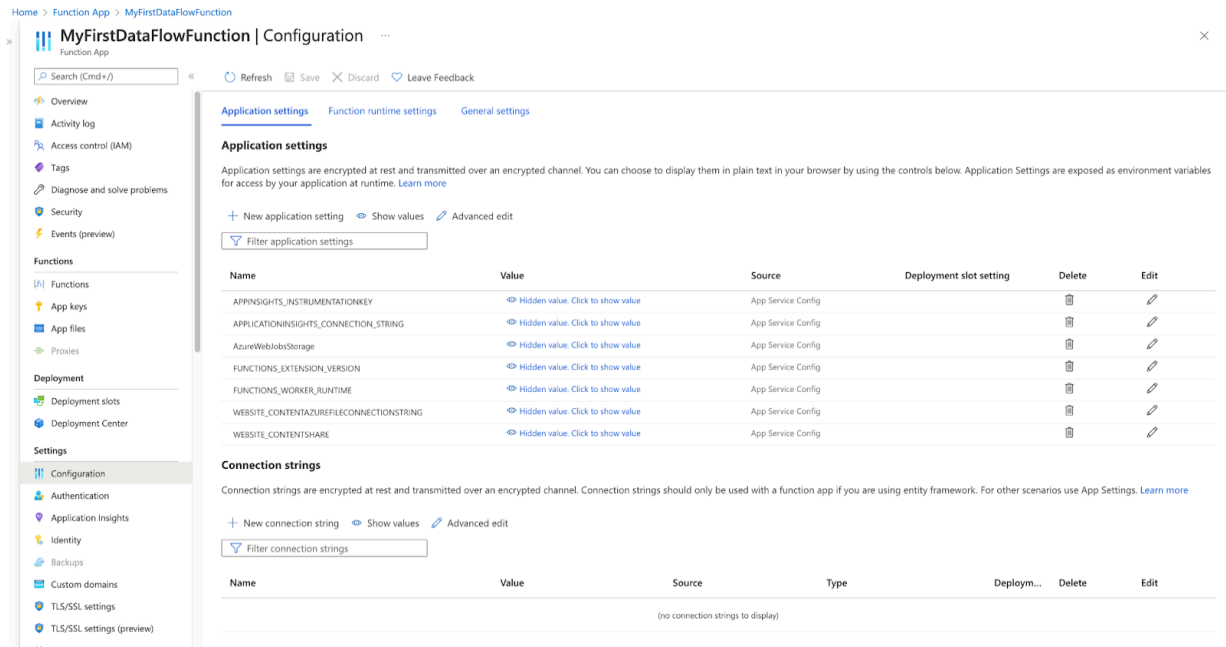
Application settings are exposed as Environment Variables inside the function. Azure pre-populates some of these, which are used by the Function framework itself. You have to add the remaining Application Settings.

3. Click New application setting at the top to add the first setting, which is required for all Cloudera Data Flow Functions deployments.

4. As a name, enter WEBSITE_RUN_FROM_PACKAGE.

The value is different depending on the Operating System:

- For Windows, enter 1. This tells the Function App to use the Zip package you uploaded earlier to the File share.
- For Linux, enter the full path to the NaafAzureFunctions.zip uploaded to the Container.

  a. To get the URL, navigate to the Storage account and click Containers.

  b. Click the container you created before.

  c. Click the NaafAzureFunctions.zip file and click the Copy icon next to the URL.

  d. Back in the Function app, paste the value as the value of the WEBSITE_RUN_FROM_PACKAGE setting.
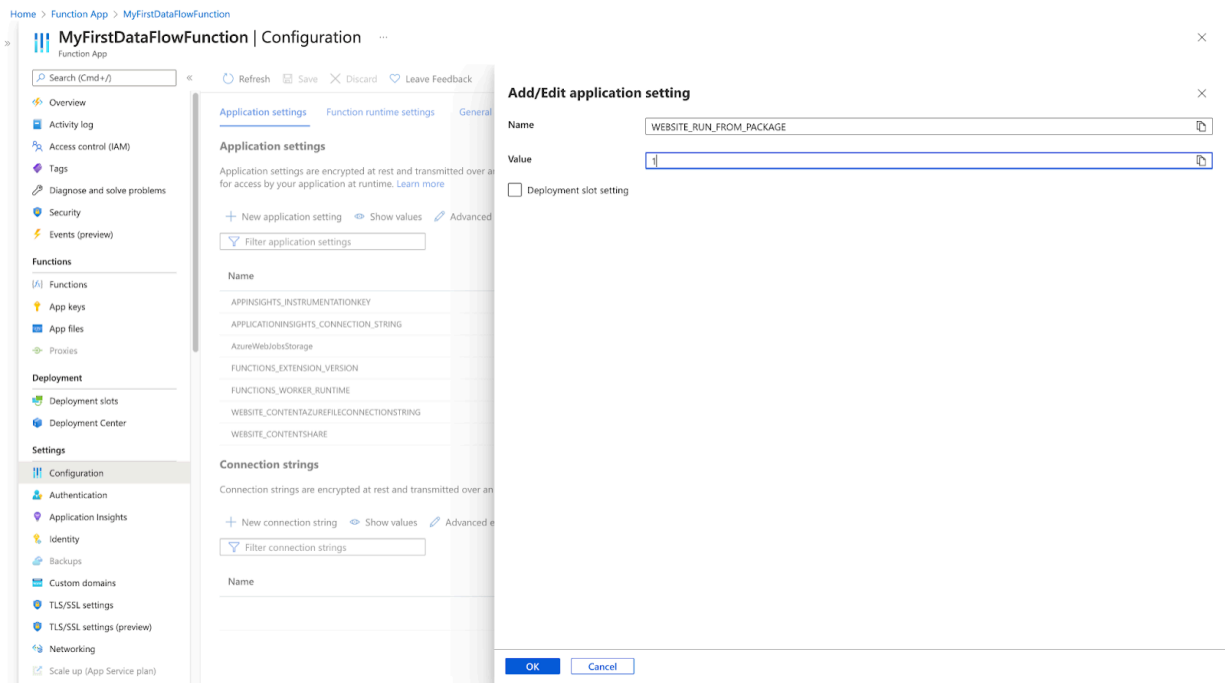
Additionally, the following Application Settings are supported.

Variable Name	Description	Required	Default Value
FUNCTION_NAME	The unique function name for this deployment. It is recommended to simply use the name of the Function App. ⚠️ **Warning:** If more than one Function App has the same value for this variable, data flow state may be corrupted.	true	--
FLOW_CRN	The Cloudera Resource Name (CRN) for the data flow that is to be run. The data flow must be stored in the Cloudera Data Flow Catalog. This CRN should indicate the specific version of the data flow and as such will end with some suffix such as /v.1. For more information, see *Retrieving data flow CRN*.	true	--

Variable Name	Description	Required	Default Value
DF_PRIVATE_KEY	The Private Key for accessing the Cloudera Data Flow service. The Private Key and Access Key are used to authenticate with the Cloudera Data Flow Service and they must provide the necessary authorizations to access the specified data flow. For more information, see *Provisioning Access Key ID and Private Key*. **Note:** It is a best practice to configure the DF_PRIVATE_KEY as a Secret in Key Vault.	true	--
DF_ACCESS_KEY	The Access Key for accessing the Cloudera Data Flow service. The Private Key and Access Key are used to authenticate with the Cloudera Data Flow Service and they must provide the necessary authorizations to access the specified data flow. For more information, see *Provisioning Access Key ID and Private Key*.	true	--
INPUT_PORT	The name of the Input Port to use. If the specified data flow has more than one Input Port at the root group level, this environment variable must be specified, indicating the name of the Input Port to queue up the Cloud Function event. If there is only one Input Port, this variable is not required. If it is specified, it must properly match the name of the Input Port.	false	--
OUTPUT_PORT	The name of the Output Port to retrieve the results from. If no Output Port exists, the variable does not need to be specified and no data will be returned. If at least one Output Port exists in the data flow, this variable can be used to determine the name of the Output Port whose data will be sent along as the output of the function. For more information on how the appropriate Output Port is determined, see *Output ports*.	false	--

Variable Name	Description	Required	Default Value
FAILURE_PORTS	A comma-separated list of Output Ports that exist at the root group level of the data flow. If any FlowFile is sent to one of these Output Ports, the function invocation is considered a failure. For more information, see *Output ports*.	false	--
DF_SERVICE_URL	The Base URL for the Cloudera Dataflow Service.	false	https://api.us-west-1.cdp.cloudera.com/
NEXUS_URL	The Base URL for a Nexus Repository for downloading any NiFi Archives (NARs) needed for running the data flow.	false	https://repository.cloudera.com/artifactory/cloudera-repos/
CONTENT_REPO	The contents of the FlowFiles can be stored either in memory, on the JVM heap, or on disk. If this environment variable is specified, it specifies the path to a directory where the content should be stored. If not specified, the content is held in memory.	false	--
WORKING_DIR	The working directory, where NAR files will be expanded.	false	/tmp/extensions for Linux and C:\local\Temp for Windows
EXTENSIONS_DOWNLOAD_DIR	The directory to which missing extensions / NiFi Archives (NARs) will be downloaded. For more information, see *Providing custom extensions / NARs*.	false	/tmp/extensions for Linux and C:\local\Temp for Windows
EXTENSIONS_DIR_*	Specifies read-only directories that may contain custom extensions / NiFi Archives (NARs). For more information, see the *Providing custom extensions / NARs*.	false	--
STORAGE_ENDPOINT	A Storage account endpoint to look for custom extensions / NiFi Archives (NARs) and resources. For more information, see *Cloud storage*.	false	--
STORAGE_CONTAINER	A Storage account container name in which to look in for custom extensions / NiFi Archives (NARs) and resources. For more information, see *Cloud storage*.	false	--

Variable Name	Description	Required	Default Value
STORAGE_EXTENSIONS_DIRECTORY	The directory in the Storage account Blob container to look in for custom extensions / NiFi Archives (NARs). For more information, see *Cloud storage*.	false	extensions
STORAGE_RESOURCES_DIRECTORY	The directory in the Storage account Blob container to look in for custom resources. For more information, see *Cloud storage*.	false	resources
COSMOSDB_ENDPOINT	The Cosmos DB account URI for the Cosmos DB state provider. It is required if you wish to preserve state in any stateful processors. For more information, see *Data flow state*.	false	--
COSMOSDB_STATE_DATABASE	The Cosmos DB database name for the Cosmos DB state provider. For more information, see *Data flow state*.	false	nifi_state
COSMOSDB_STATE_CONTAINER	The Cosmos DB container name for the Cosmos DB state provider. For more information, see *Data flow state*.	false	nifi_state
WEBSITE_RUN_FROM_PACKAGE	Specifies that the Function App should use the NaaF Zip package as its binary distribution.	true	1
KRB5_FILE	It specifies the filename of the krb5.conf file. This is necessary only if connecting to a Kerberos-protected endpoint. For more information, see *Configuring Kerberos*.	false	/etc/krb5.conf

The following environment variables apply only to HTTP triggers:

Variable Name	Description	Required	Default Value
HEADER_ATTRIBUTE_PATTERN	A Regular Expression capturing all flowfile attributes from the Output Port that should be returned as HTTP headers, if the trigger type is HTTP. For all other trigger types, this variable is ignored.	false	--

Variable Name	Description	Required	Default Value
HTTP_STATUS_CODE_ATTRIBUTE	A flowfile attribute name that will set the response HTTP status code in a successful data flow, if the trigger type is HTTP. For all other trigger types, this variable is ignored. See the section on *HTTP Functions* for more information.	false	--

Related Information

Retrieving data flow CRN

Provisioning Access Key ID and Private Key

Output ports

Providing custom extensions / NARs

Cloud storage

Data flow state

Configuring Kerberos

HTTP Functions

Managed identity

A Function App can use a Managed Identity in order to access other Azure services.

Procedure

1. When configuring your Function App, click the Identity blade on the left.
2. Switch the Status to On and click Save.
3. Click Yes to create a Managed Identity to represent your Function App.

   Your Function App now has an identity in Azure Active Directory, and can be assigned roles and access to specific services like other identities.
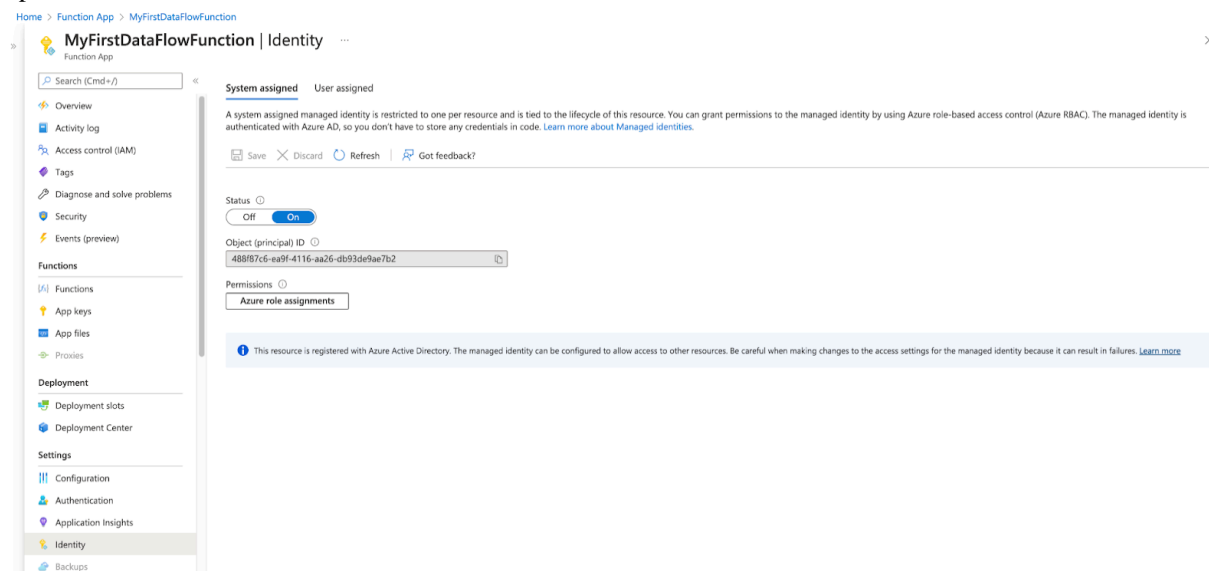
When interacting with Key Vaults, Event Grid, or Storage accounts (besides the File share automatically attached to the Function App), this is the Identity you will use, and it is named after your Function App. You can also grant it permissions to access other Azure services if the data flow itself requires access to them.

Output ports

While all Function trigger types support Failure ports, only HTTP Functions will use the output from non-failure Output Ports. This topic collects different considerations of Output Ports for the different Function trigger types.

Event Grid Function Output Ports

With Event Grid Functions, the Output Port data flow logic is simpler.

If one Output Port is present, and its name is "failure", any data routed to that Output Port will cause the function invocation to fail.

If two Output Ports are present, and one of them is named "failure", that Output Port will behave as the "failure" port above. Any other non-failure Output Port is considered a successful invocation, but the function will do nothing with data routed here.

Any port with the name "failure" (ignoring case) will be considered a Failure Port. Additionally, if the *FAILURE_PORTS* environment variable is specified, any port whose name is in the comma-separated list will be considered a Failure Port.

HTTP Function Output Ports

Since Functions with an HTTP trigger are invoked synchronously, they directly return a response. The Output Port semantics for this case is as follows.

If no Output Port is present in the data flow's root group, no output will be provided from the function invocation, but a default success message will be returned in the HTTP response with a status code of 200.

If one Output Port is present, and its name is "failure", any data routed to that Output Port will cause the function invocation to fail. No output will be provided as the output of the function invocation, and an error message will be returned from the HTTP response, with a status code of 500.

If two Output Ports are present, and one of them is named "failure", that Output Port will behave as the "failure" port above. The other Output Port is considered a successful invocation. When the data flow is invoked, if a single FlowFile is routed to this Port, the contents of the FlowFile will be returned as the HTTP response. If more than one FlowFile is sent to the Output Port, the data flow will be considered a failure, as there is only one HTTP response possible. A MergeContent or MergeRecord Processor may be used in order to assemble multiple FlowFiles into a single output FlowFile if necessary.

Any Port with the name "failure" (ignoring case) will be considered a Failure Port. Additionally, if the *FAILURE_PORTS* environment variable is specified, any port whose name is in the comma-separated list will be considered a Failure Port.

If two or more Output Ports, other than failure ports, are present, the *OUTPUT_PORT* environment variable must be provided. In such a case, this environment variable must match the name of an Output Port at the data flow's root group (case is NOT ignored). Anything that is routed to the specified port is considered the output of the function invocation. Anything routed to any other port is considered a failure.

Note:

In any successful response, the mime.type flowfile attribute is used as the Content-Type of the response.

Parameters

Parameterization is important when building a dataflow that is able to be run outside of the NiFi instance where it was built. NiFi allows users to build a data flow making use of Parameter Contexts. Using Parameter Contexts, you can define Processor and Controller Service properties at runtime instead of at build time. Some parameters are sensitive,

such as a password, while others are not, such as the name of a Kafka topic. The Cloud function allows you to specify these parameters through Application Settings, either directly or mapped from Secrets.

Application Settings

Any parameter can be specified using the Application Settings of the Azure Function App. When configuring the Function App, in the Configuration blade, add an application setting whose name matches the name of a parameter in your Parameter Context.

Key Vault Secret

A more secure mechanism for storing sensitive data flow parameters and other sensitive Application Settings is to use a Key Vault Secret.

1. Navigate to the Key Vaults Azure service and click the Create at the top-left.
2. Select your Function App's resource group, and name the Key Vault. Select the same region as the Function App.
3. Click Next : Access policy, and Add Access Policy.
4. Under Secret permissions, select List and Get.
5. Click the None selected link next to Select principal.
6. Assuming you have created a Managed Identity for your Function App, enter the name of your Function App and select it on the right panel.
7. Click Select Add Review + create Create .
8. Once the deployment is complete, click Go to resource.

   Here you can create any secrets to be referenced.
9. Click the Secrets blade on the left, and the Generate/Import button at the top.
10. Enter the name of the desired Application Setting.

   This does not have to match the actual name of the Application Setting or Parameter Context, and must follow the convention of only alphanumeric characters and dashes.
11. Enter the value of the Application Setting, and click Create.
12. Back in your Function App, create an Application Setting with the appropriate name (the Parameter name, or one of the supported Application Settings from the "Configuration of my Function App" section).
13. Add the following value: @Microsoft.KeyVault(VaultName={vaultName};SecretName={secretName})
14. Replace {vaultName} with the actual Key Vault name, and replace {secretName} with the name of the newly created secret.
15. Click Save.

Azure will update your configuration and then attempt to link this Application Setting to the Key Vault Secret. To see this link, you can click the Refresh button. In the Source column, you should now see Key Vault Reference and a green check mark if the Function App successfully resolved the Secret. If there is a red X, there is likely an issue with either the permissions of the Function App's managed identity or maybe the vault or secret could not be found.

> ⚠️ **Important:** To use this capability for data flow parameters, the Parameter Contexts used in the flow must have names that are compatible with Azure Application Settings names. To ensure that these naming conventions are followed, Parameter Contexts and Parameters should use names that start with a letter and consist only of letters, numbers, and the underscore (_) character. For example, instead of naming a Parameter "Azure Parameter Context" use the name of AZURE_PARAMETER_CONTEXT when building a data flow.

Cold start

Cold start can be defined as the set-up time required to get a serverless application's environment up and running when it is invoked for the first time within a defined period.

What is a cold start?

The concept of cold start is very important when using Function as a Service solutions, especially if you are developing a customer-facing application that needs to operate in real time. A cold start is the first request that a new function instance handles. It happens because your function is not yet running and the cloud provider needs to provision resources and deploy your code before the processing can begin. This usually means that the processing is going to take much longer than expected.

Before execution, the cloud provider needs to:

- Provision the container that will run the code
- Initialize the container / runtime
- Initialize your function

before the trigger information can be forwarded to the function's handler.

For more information about the concept of cold start, see Understanding serverless cold start in Azure Functions.

How to avoid a cold start?

Cloud providers offer the option to always have at least one instance of the function running to completely remove the occurrence of a cold start. You can read more about this in Consumptions plans for Azure Functions.

Cold start with Cloudera Data Flow Functions

When a cold starts happens with Cloudera Data Flow Functions, the following steps are executed before the trigger payload gets processed:

- The function retrieves the flow definition from the Cloudera Data Flow Catalog.
- The NiFi Stateless runtime is initialized.
- The flow definition is parsed to list the required NAR files for executing the flow.
- The required NAR files are downloaded from the specified repository unless the NARs are made available in the local storage attached to the function.
- The NARs are loaded into the JVM.
- The flow is initialized and components are started.

The step that might take a significant amount of time is the download of the NAR files. In order to reduce the time required for this step as much as possible, the best option is to list the NAR files required for the flow definition and make those files available to the function.

For the related instructions, see Cloud storage on page 22.

Cloud storage

Providing custom extensions / NARs
Cloudera Maven Repository

> The binary that is deployed for running the Google Cloud function does not include the full NiFi deployment with all NARs / extensions, as it would result in very long startup times and would require more expensive configurations for the Cloud function. Instead, it packages only the NARs necessary to run Stateless NiFi. If any other extension is needed to run the data flow, it is automatically downloaded from the Cloudera Maven Repository by default when the Function App is initialized on a cold start.

Nexus repository

> You can configure an alternative Nexus repository by setting the *NEXUS_URL* environment variable to the URL of the Nexus Repository where extensions should be downloaded. For example,

to use Google's mirror of the Maven Central repository, set the *NEXUS_URL* environment variable to: https://maven-central.storage-download.googleapis.com/maven2

> **Note:** Any configured URL must either be accessible via http or be served over https with a trusted certificate.

Providing extensions from file share

This process is different depending on the Plan Type and Operating System. For every type except Linux on a Consumption plan, the file share is already created, and you can simply create a directory in the file share containing all NAR files that are needed.

For Linux on a Consumption plan, you need to create a file share and mount it as follows:

1. In the storage account attached to the Function App, click the File shares blade, and then click + File share on the top left.
2. Name the File share, and click Create.
3. Click the Access keys blade of the Storage account, and then click Show keys.
4. Copy the Key from key1. You will need to paste it in the place of [Copied access key].
5. Click the Cloud Shell icon at the top-right, and accept any prompts needed to set up the shell.
6. When the shell is running, run the following command:

```
az webapp config storage-account add       \
    --resource-group [Resource Group Name] \
    --name [Function App Name]              \
    --custom-id [Pick a unique mount ID]    \
    --storage-type AzureFiles               \
    --share-name [File share Name]          \
    --account-name [Storage account Name]   \
    --mount-path /data                      \
    --access-key [Copied access key]
```

The mount path can be different, but should not be /home or /tmp.

Now you can create a directory in the File share, and upload any NAR files to the directory.

Regardless of OS or plan type, you need to tell the Function App where to look for the extensions. You can do this by adding another Application Setting whose name starts with EXTENSIONS_DIR_. You can add multiple of these, such as EXTENSIONS_DIR_1, EXTENSIONS_DIR_2, EXTENSIONS_DIR_MY_CUSTOM_LIBS. The name of the Application Setting is the path where the Function App should look for the extensions.

- In Linux Consumption, the File share is mounted where you specified it in the Cloud Shell. For example, if you upload the NARs to the File share under a directory called extensions/custom, and the File share is mounted under /data, you can add an Application Setting called EXTENSIONS_DIR_CUSTOM with a value of /data/extensions/custom.
- In all other plans, since the File share is mounted at /home, this should always start with /home. For example, if you upload our NARs to the File share under a directory called extensions/custom, you could add an Application Setting called EXTENSIONS_DIR_CUSTOM with a value of /home/extensions/custom.

Adding many extensions will result in the Function App configuration taking a bit longer to load on a cold start, and it takes more memory to load the additional classes. So it is not recommended to upload an extremely large set of files if they are not needed.

Providing extensions / resources with storage containers

If extensions or resources are to be shared among many Function Apps, it is easier to have a single Container in a Storage account with one copy of the extensions or resources. To integrate with an arbitrary Storage Container, specify the Storage account's endpoint in the STORAGE_ENDPOINT Application Setting (e.g., https://<storageAccount>.blob.core.windows.net), and the Container name in the STORAGE_CONTAINER Application Setting.

Your Function App will need permission to access the container. Assuming you have created a Managed Identity for your Function App:

1. Edit your Function App and click the Identity blade on the left.
2. Click the Azure role assignments button under Permissions, and click Add role assignment.
3. Under Scope, select Storage, and then select the desired Storage account under Resource.

   Storage Blob Data Reader should be sufficient to download Blobs.
4. Click Save.

Next, you need to upload any extensions to a directory in the Container and specify the directory name in the STOR AGE_EXTENSIONS_DIRECTORY Application Setting. Upload any data flow resources to another directory in the Container and specify the directory name in the STORAGE_RESOURCES_DIRECTORY Application Setting. Now when your App Function is restarted, it will download any extensions or resources from these locations.

If you are using the Storage Container approach, you may see the following errors in the log at function startup, which can be safely ignored:

```
2022-01-21T18:36:33.976 [Error] [pool-2-thread-4] ERROR com.azure.identity.E
nvironmentCredential -
Azure Identity => ERROR in EnvironmentCredential: Missing required environme
nt variable AZURE_CLIENT_ID
2022-01-21T18:36:34.460 [Error] [pool-2-thread-4] ERROR com.azure.identity.i
mplementation.IntelliJCacheAccessor - IntelliJ Authentication not available.
 Please log in with Azure Tools for IntelliJ plugin in the IDE.
2022-01-21T18:36:34.917 [Error] [parallel-2] ERROR com.azure.identity.Enviro
nmentCredential - EnvironmentCredential authentication unavailable. Environm
ent variables are not fully configured.To mitigate this issue,
please refer to the troubleshooting guidelines here at https://aka.ms/azsdk/
net/identity/environmentcredential/troubleshoot
2022-01-21T18:36:35.779 [Error] [pool-2-thread-4] ERROR com.azure.identity.E
nvironmentCredential - Azure Identity => ERROR in EnvironmentCredential: Mis
sing required environment variable AZURE_CLIENT_ID
2022-01-21T18:36:36.142 [Error] [pool-2-thread-4] ERROR com.azure.identit
y.implementation.IntelliJCacheAccessor - IntelliJ Authentication not availab
le. Please log in with Azure Tools for IntelliJ plugin in the IDE.
2022-01-21T18:36:36.451 [Error] [parallel-2] ERROR com.azure.identity.Env
ironmentCredential - EnvironmentCredential authentication unavailable. Envir
onment variables are not fully configured.To mitigate this issue,
please refer to the troubleshooting guidelines here at https://aka.ms/azsdk/
net/identity/environmentcredential/troubleshoot
```

Providing additional resources

In addition to NiFi extensions, some data flows may also require additional resources in order to function properly. For example, a JDBC driver may be required for establishing a database connection, or a CSV file to provide data enrichment. The recommended approach for providing such resources to your Function App is using the attached Storage account's File share. To set this up, follow the instructions in the File share section.

1. Create a directory for your resources.
2. Upload all resources required for the data flow to run to this directory.

The directory will be mapped under the mapped File share directory for access by the data flow.

Because each deployment of a given data flow may need to reference files in a different location, depending on its environment, it is generally considered a best practice to parameterize any resource that needs to be accessed by the data flow. For example, if the data flow contains a DBCPConnectionPool Controller Service for accessing a database, it is recommended to use a Parameter for the "Database Driver Location(s)" property of the Controller Service. This allows each deployment to specify the location of the JDBC driver independently.

So, for example, you can set the "Database Driver Location(s)" property to a value of #{JDBC_DRIVER_LOCATI ON}. Then, in your Storage File share, you can create a directory called something like resources, which contains

your JDBC driver database-jdbc-driver.jar. And you can add an Application Setting named JDBC_DRIVER_LOCA TION with a value of /home/resources/database-jdbc-driver.jar (or /data/resources/database-jdbc-driver.jar for Linux Consumption plans).

By taking this approach, your data flow is more reusable, as it can be deployed in many different environments.

Disk storage considerations

Azure Function Apps have two basic options for local storage used for downloading and expanding NARs.

1. Temporary storage, under C:\local\Temp - for Consumption plans, the total local disk size is limited to 500 MB, including this Temp directory, making the actual limit somewhat less than 500 MB. This limit can be increased significantly with App Service and Functions Premium plans (for example, the Basic plan starts at 11 GB). This storage is attached directly to the Function App container, and is therefore much faster than the File Share option, discussed next. This storage also requires no additional configuration, but caution must be taken not to fill up the disk. Mitigations for this are discussed below.
2. File Share storage, which is mapped to C:\home. When setting up the initial function, you already attached a File share from your Storage account, so this should already be available to you. The File share has a default limit of 5 TB, but is not locally attached to your Function App container, and so can be significantly slower than writing to the Temp directory.

Although Temporary storage is faster, Cloudera Data Flow Functions has a storage optimization that reuses downloaded and unpacked extension NARs. Therefore, the recommended approach is to use the File share for both the WORKING_DIR and EXTENSIONS_DOWNLOAD_DIR. This will incur a significant time penalty on the very first cold start, but afterwards the downloading and unpacking of NARs can be skipped, greatly reducing the function execution time.

Note: Because of the initial execution time cost, when using the File share for either directory, it is recommended that you perform a single "warming" invocation of your function in order to download and unpack all extension NARs for later reuse.

You can use the File share for storage by doing the following:

1. Navigate to your Storage account and click the File shares blade on the left. Select your file share name on the right.
2. Click the data directory (or create it if it does not yet exist).
3. Click the + Add directory button, name it extensions, and click OK. Repeat this for a directory called working.
4. In your Function App, click the Configuration blade on the left, and add (or set) an Application Setting named EXTENSIONS_DOWNLOAD_DIR with a value of C:\home\data\extensions.
5. In your Function App, click the Configuration blade on the left, and add (or set) an Application Setting named WORKING_DIR with a value of C:\home\data\working.
6. Save the configuration.
7. Optionally (but recommended), perform a single "warming" invocation of your Function App.

Data flow state

By default, if your data flow contains any stateful processors (for example, ListSFTP), this state will automatically be stored in a Cosmos database called nifi_state, though this name can be set using the *COSMOSDB_STATE_DATABASE* environment variable. To integrate with Cosmos DB, some additional steps are required. If you do not perform these steps, the data flow will still run, but the state will not be preserved.

1. In the Azure console, navigate to the Cosmos DB service.
2. Click Create.
3. Click Create under Core (SQL) - Recommended.

4. Select your Resource Group, and enter an account name.

5. All other configurations are optional, and may be different depending on your use case. While provisioned throughput is supported, it is recommended to select Serverless.

6. Click Review + Create and check the function details.

7. Click Create.

8. In your Function App, go to the Identity blade, and note the Object (principal) ID, for the following steps.

9. Click the Cloud Shell icon in the upper right and follow any initial prompts that may set up the web-based cloud shell for you.

10. Run the following commands in the cloud shell:

```
resourceGroup=<your resource group>
accountName=<your CosmosDB account name>
principalId=<your Function App Object principal ID from above>
az cosmosdb sql role assignment create --account-name $accountName \
   --resource-group $resourceGroup --scope "/" \
   --principal-id $principalId --role-definition-id 00000000-0000-0000-0
000-000000000001
az cosmosdb sql role assignment create --account-name $accountName \
   --resource-group $resourceGroup --scope "/" \
   --principal-id $principalId --role-definition-id 00000000-0000-0000-0000
-000000000002
```

11. On the Overview blade, click the Add Container.

12. Specify the database and container IDs.

   If you use nifi_state for both, the COSMOSDB_STATE_DATABASE and COSMOSDB_STATE_CONTAINER Application Settings are not needed.

13. Specify /id for the Partition Key and click OK.

14. In your Function App Configuration, add an Application Setting called COSMOSDB_ENDPOINT with the Cosmos DB URI (from the Overview page).

   If you specified any other database or container name than nifi_state, you will need to specify the COSMOSDB _STATE_DATABASE and COSMOSDB_STATE_CONTAINER Application Settings, respectively.

The Cosmos DB state provider can be disabled even if your data flow contains stateful processors by setting the DISA BLE_STATE_PROVIDER Application Setting to true.

⚠️ **Warning:** This will cause processor state to be lost between function executions.

Configuring Kerberos

Depending on the dataflow that is used, you may need to enable Kerberos authentication in order to interact with some service(s).

To enable Kerberos, you must provide an appropriate krb5.conf file. This file tells the Cloudera Data Flow Function, among other things, where the Key Distribution Center (KDC) is located and the Kerberos Realm to use.

Depending on the data flow that is used, you may need to enable Kerberos authentication in order to interact with some service(s). To enable Kerberos, you must provide an appropriate krb5.conf file. This file tells the Cloudera Data Flow Function, among other things, where the Key Distribution Center (KDC) is located and the Kerberos Realm to use.

To specify this, the krb5.conf must be made available to Cloudera Data Flow Function. You can do this by using an Azure Storage Container. See the corresponding section to learn more about this.

Additionally, you must tell Cloudera Data Flow Function where it can find this krb5.conf file. To do this, add an environment variable to your Lambda function. The environment variable must have the name KRB5_FILE and the value must be the fully qualified filename of the krb5.conf file. For example, it might use /home/resources/krb5.conf.

In order to use Keytabs in your data flow, it will also be important to provide the necessary keytab files to the Cloudera Data Flow Function. This is accomplished in the same manner, using a Lambda Layer or an S3 bucket. While the krb5.conf file is a system configuration, the keytab is not. The keytab will be configured in the data flow on a per-component basis. For example, one Processor may use keytab nifi-keytab-1 while some Controller Service makes use of keytab hive-keytab-4. The location of these files may change from deployment to deployment. For example, for an on-premises deployment, keytabs may be stored in /etc/security/keytabs while in Cloudera Data Flow Function, they may be made available in /home/resources.

So it is better to use Parameter Contexts in order to parameterize the location of the keytab files. This is a good practice for any file-based resource that will need to be made available to a deployment. In this case, because there is a parameter referencing the location, we need to provide Cloudera Data Flow Function with a value for that parameter. For more information on this, see *Parameters*.

Related Information
Parameters

Handling failures

You can use a failure Output Port to notify the Function that an invocation has failed.

You can use a failure Output Port to notify the Function that an invocation has failed. For HTTP triggers, there is no retry. For Event Grid triggers, Azure Event Grid may automatically retry the invocation, if configured in the Event Subscription, for a default of 30 attempts. After that, if it still fails, the Event Subscription will simply drop the event notification. This could result in data loss, as it means that our Function App will not have a chance to process the data again.

Testing your data flow

It is important to test the data flow you have built before trying to deploy it.

Any data flow that is to be used by Azure Function Apps requires an Input Port and most often contains at least one failure Output Port. You can simply move up to the Parent Process Group within NiFi and create a GenerateFlowFile processor. You can then set the "Custom Text" property to any value that you may wish to feed into the data flow. For example, to simulate an Event Grid event indicating that data was added to a Storage Container:

1. Set the "Custom Text" property to the value:

```
{
  "topic": "/subscriptions/73db9d72-863b-45e9-9a1a-890cff0ac0e6/resourceGr
oups/my-rg/providers/Microsoft.Storage/storageAccounts/mystorageaccount",
  "subject": "/blobServices/default/containers/my-container/blobs/input/te
stfile.json",
  "eventType": "Microsoft.Storage.BlobCreated",
  "id": "d4c93b9a-901e-0078-1dd4-d34b8d06bbad",
  "data": {
    "api": "DeleteBlob",
    "requestId": "9d4c93b9-901e-0078-1dd4-d34b8d000000",
    "eTag": "0xD9A1EBFB6C05A58",
    "contentType": "application/json",
    "contentLength": 136,
    "blobType": "BlockBlob",
    "url": "https://mystorageaccount.blob.core.windows.net/my-container/i
nput/testfile.json",
```

```
      "sequencer": "000000000000000000000000000004F50000000003e3cc87",
      "storageDiagnostics": {
        "batchId": "69a23459-7006-003d-00d4-d39e6e000000"
      }
    },
    "dataVersion": "",
    "metadataVersion": "1",
    "eventTime": "2021-11-07T12:41:55.7182885Z"
  }
```

2. Connect this GenerateFlowFile processor to the Input Port of our Process Group that you will run in the Function App.

   The "Run Once" feature of NiFi makes it easy to create this FlowFile and send it to the Process Group.

3. Start the Process Group and ensure that the data processes as expected.

   If there are any issues, you can fix the data flow and trigger the GenerateFlowFile processor again, until you have properly handled the data.

When the test is successful, you are ready to download the flow from NiFi and upload the flow definition to the Cloudera Data Flow Catalog.

Testing your Azure Function App

Once you have built and verified your dataflow and uploaded it to the Cloudera Data Flow Catalog, you can create your Azure Function App.

After deploying the function, you should test if all settings have been properly configured.

1. Click the Functions blade of your Function App.
2. Select the Function you want to test.
3. Click the Code + Test blade on the left, and the Test/Run button at the top.

   This brings up a side panel to input sample data and have the data flow run. If there are problems, you can inspect the log messages to understand the issue. If the logs do not provide enough details, you may need to adjust the log levels to gain more debugging information, and view the complete Log Stream. For more information, see Monitoring and logs.

   Note: At this time, the Azure Console cannot submit valid requests in the "Code + Test" view specifically for Event Grid trigger functions.

As a workaround, you can test your Function without generating an actual Event Grid event as follows:

1. Click the Get function URL button at the top of the Code + Test view, and click the Copy button next to the URL.
2. Create a file, for example, named event-grid.json with the contents of the Event Grid schema JSON you want to test.
3. Paste the URL in the place of the URL in the following curl command:

```
curl https://<app-function-name>.azurewebsites.net/runtime/webhooks/Even
tGrid?functionName=StatelessNiFiEventGridTriggerFunction&code=<code> \
    -H 'Content-Type: application/json'\
    -H 'aeg-event-type: Notification'
    --data-binary @event-grid.json
```

For more details on submitting this request, see Azure Event Grid trigger for Azure Functions.

The amount of time the cloud function takes to run depends heavily on the data flow and how many extensions it needs. Because the Cloud function may have to download some extensions from Nexus and perform initialization, a cold start may take many seconds. Even 20-30 seconds is not uncommon for a data flow with several extensions, while other data flows may complete in 10 seconds. After you run the function successfully using the Test button,

it may be helpful to run several additional iterations in order to understand how the function will perform when a cold start is not necessary. Again, this depends heavily upon the configured data flow and other services that it may interact with. A simple data flow may complete in as little as 5 milliseconds for HTTP triggers or 100 milliseconds for Event Grid triggers, while a data flow that must perform complex transformations and reach out to one or more web services or databases for enrichment may take several seconds to complete.

It is important to adjust the function's timeout configuration if the cold start takes longer than the amount of time allocated.

Related Information

Monitoring and logs

Securing your Azure Function App

This section guides you on securing the HTTP endpoint for your App.

Enforce HTTPS

1. In your Function App, click the TLS/SSL Settings blade on the left.
2. Next to HTTPS Only, click On.
3. Ensure that "Minimum TLS Version" is the latest version.
4. You can also configure the server certificate on this page.

Add an Identity Provider

By default, your HTTP Function endpoint is completely public. In order to restrict access to only authorized users, you can add an Identity Provider:

1. In the Authentication blade of your Function App, click the Add Identity Provider button.

    Several options are available. This example uses Microsoft.
2. Leave the App registration type as Create new app registration.
3. Enter the name of your Function App as the registration name.
4. Select the desired Supported account types (for example, "Any Azure AD directory - Multi-tenant")
5. Ensure Restrict access is set to Require authentication.
6. Select the desired Unauthenticated requests value.

    This example uses "HTTP 401 Unauthorized: recommended for APIs".
7. Ensure Token store is selected.
8. Click the Add.

The identity provider appears in the Authentication blade. You can follow the Quickstart link for details on how to use it to authenticate for your HTTP Function. The example presented here shows a common use case.

To access your HTTP Function endpoint programmatically, you can create a new App Registration representing the calling client:

1. In the App Registrations service, click New registration.
2. Give the registration a name representing your client (e.g., myfunction-client).
3. Select the desired Supported account types option, and click Register.
4. In the Certificates & secrets blade, select Client secrets New client secret .
5. Specify a description, and click Add.
6. Copy the secret Value, because it will not be shown again.
7. In the Overview blade, add the Directory (tenant) ID and Application (client) ID for the next step.
8. Add the Application (client) ID of your Function App's App Registration.

9. Request an access token similar to the following:

```
# <tenant-id> is the tenant ID copied above
# <client-id> is the client App Registration's "Application (client) ID"
 copied above
# <client-secret> is the client App Registration's client secret copied ab
ove
# <application-id> is your Function App's App Registration "Application
(client) ID" copied above
curl -X POST https://login.microsoftonline.com/<tenant-id>/oauth2/token -H
  "Content-Type: application/x-www-form-urlencoded" -d \
     'grant_type=client_credentials&client_id=<client-id>&client_secret=<
client-secret>&resource=<application-id>' -X POST
```

This will give you an OAuth 2 response.

10. Copy the value of the "access_token" to use below.

11. Submit a request to your HTTP function now as follows:

```
TOKEN=<access_token> # copied from above
curl https://<your-function-app>.azurewebsites.net/api/StatelessNiFiHtt
pTextTriggerFunction \
     -H "Authorization: Bearer $TOKEN" \
     ... etc.
```

Monitoring and logs

Azure Functions offers an integration with Azure Monitor Logs to monitor functions.

You can view Function logs either directly in the Log Stream blade at the Function App level, or under the Monitoring blade of the individual Function. Function output log files are also available in the Storage account File share under LogFiles/Application/Functions/Host.

Adjusting logs levels

Cloudera Data Flow Functions makes use of the SLF4J Simple Logger for logging purposes. You can update the log levels by adjusting the JVM's system properties.

To do so, you need to set the *JAVA_TOOL_OPTIONS* application setting.

• To set a log level for a given logger, you can set the JVM system property named org.slf4j.simpleLogger.log.< logger name> to the desired log level.

  For example, if you want to enable DEBUG logging for the StandardStatelessNiFiFunction to view the actual data flow JSON being used, you would set the *JAVA_TOOL_OPTIONS* environment variable to -Dorg.slf4j.simpleLogger.log.com.cloudera.naaf.StandardStatelessNiFiFunction=DEBUG

• You can also set additional logger levels by adding multiple -D options separated by spaces. For example, to enable DEBUG logs on both the Cloudera Data Flow Functions framework and the Stateless Bootstrap class, you would set *JAVA_TOOL_OPTIONS* to a value of:

```
-Dorg.slf4j.simpleLogger.log.com.cloudera.naaf.StandardStatelessNiFiFunc
tion=DEBUG
         -Dorg.slf4j.simpleLogger.log.org.apache.nifi.stateless.bootstra
p.StatelessBootstrap=DEBUG
```

Azure Function triggers

After you fully deploy your Function App, you can go to the Functions blade to view the different Functions that are available in Cloudera Data Flow Functions.

Overview

**StatelessNiFiEventGridTriggerFunction**

> Triggers based on an Azure EventGrid event. This allows a wide range of possible events to trigger your Function App. For more information configuring Event Grid to interact with your Function App, see the section on *Event Grid Functions*.

StatelessNiFiHttpBinaryTriggerFunction

> Triggers in response to an HTTP request with a binary payload (Content-Type: application/octet-stream). See the section on *HTTP Functions* for more information on this trigger type.

StatelessNiFiHttpTextTriggerFunction

> Identical to the StatelessNiFiHttpBinaryTriggerFunction, but supports text-based payloads such as JSON (Content-Type: application/json), XML (Content-Type: application/xml), and so on.
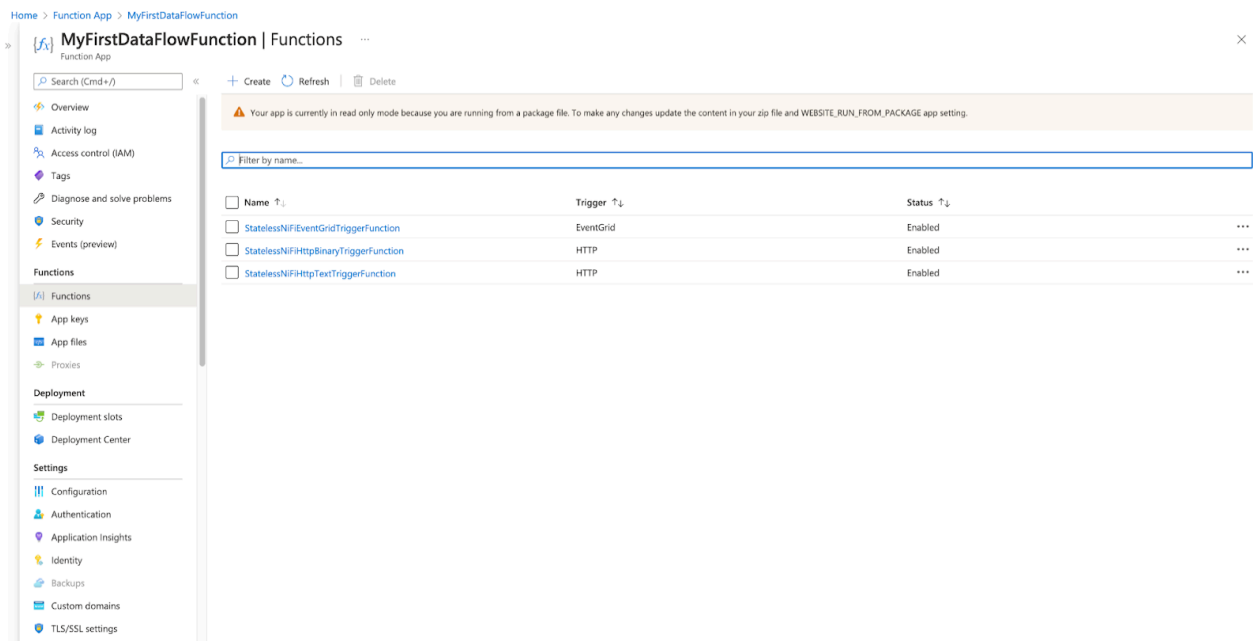
Note:

Your data flow will most likely be written differently between the two basic Function trigger types of Event Grid and HTTP, but all three Functions are enabled by default.

To disable the other functions, click the ... button at the right of the Functions table and then click Disable.

Event Grid Functions

The StatelessNiFiEventGridTriggerFunction can be used to trigger based on an Event Grid event, responding to a variety of Azure signals.

To set up an Event Grid Subscription for your Function App, follow these steps:

1. Navigate to the Event Grid System Topics service, and click Create in the top-left.

2. Select the Topic Type based on what service you want to trigger your Function App.

   For example, if you want your Function to trigger every time a file is dropped into a Storage Container, select Storage Accounts (Blob & GPv2). After that select the Subscription of the Resource you want to target, and select the Resource Group and Resource accordingly.

3. Enter a name for your System Topic, and check the Enable system assigned identity box.

4. Click Review + create, and then Create.

5. When the deployment completes, click Go to resource.

   Although you could have created an Event Grid Subscription directly, creating a System Topic to house the Subscription allows additional diagnostics to be configured, such as routing information about DeliveryFailures to a Storage account, which is particularly useful in troubleshooting. Check the Diagnostic settings blade for more information on configuring this setting.

6. On the Event Grid System Topic Overview blade, click + Event Subscription in the top-left.

7. Enter a name for the subscription.

8. Select the Event schema, which specifies the format of metadata sent as input to your NiFi data flow.

   For details on event grid properties and schema, see Azure Event Grid event schema. For details on the Cloud Event Schema, see CloudEvents v1.0 schema with Azure Event Grid.

9. Optionally, select the Event Types (for example, "Blob Created" for Storage accounts).

10. For Endpoint Details, select Azure Function, and click the Select an endpoint link.

   Your function will likely be automatically selected, but if not, you can select the appropriate Function from the right panel, and then click Confirm Selection.

11. Click Create.

You can now monitor the subscription from the System Topic, and should see any activity in the Overview blade.

HTTP Functions

With HTTP functions, the Function is invoked by submitting an HTTP request, and the Function output from the data flow is returned as the HTTP response, with specific Output Port semantics. For more information, see *HTTP Function Output Ports*.

This is an example of invoking the StatelessNiFiHttpTextTriggerFunction using curl:

```
curl https://<function-app-name>.azurewebsites.net/api/StatelessNiFiHttpText
TriggerFunction \
     -H 'Content-Type: application/json' \
     --data-binary @my-file.json
```

The StatelessNiFiHttpBinaryTriggerFunction invocation is very similar, but allows binary data to be submitted:

```
curl https://<function-app-name>.azurewebsites.net/api/StatelessNiFiHttpBina
ryTriggerFunction \
     -H 'Content-Type: application/octet-stream' \
     --data-binary @my-file.pdf
```

Note: The HTTP request body for Azure HTTP Function triggers is read completely into memory on the Heap, so it is recommended to limit the size of the data sent in the HTTP request. Microsoft's recommendation for working with large data objects is to store the objects in a Storage account, and only pass around pointers to the objects in the HTTP request body. NiFi data flows can make use of this recommendation by using the FetchAzureBlobStorage processor to retrieve the object if needed.

In any successful response, the mime.type flowfile attribute will be used as the Content-Type of the response. Conversely, the HTTP request Content-Type will be added as the mime.type attribute of the input flowfile.

There are two Application Settings that can be configured to customize the HTTP response:

HEADER_ATTRIBUTE_PATTERN

can provide a regular expression matching any flowfile attributes to be included as HTTP response headers. This means that any attributes to be included as headers should be named as the headers themselves.

HTTP_STATUS_CODE_ATTRIBUTE

specifies a flowfile attribute that sets the HTTP status code in the response. This allows a code other than 200 for 'success' or 500 for 'failure' to be returned. If this environment variable is specified but the attribute is not set, the status code in the HTTP response will default to 200.

In the "failure" scenario, the HTTP_STATUS_CODE_ATTRIBUTE is not used. Therefore, if a non-500 status code is required, it must be provided in a successful data flow path. For example, If a 409 Conflict is desired, it could be achieved as follows:

1. In the data flow, include an UpdateAttribute processor that sets an arbitrary attribute, for example http.status.code, to the value 409.
2. Optionally, add a ReplaceText processor that sets the contents of the flowfile using Always Replace to whatever HTTP response body is desired.
3. Route this part of the flow to the main Output Port used for the success case.
4. Set HTTP_STATUS_CODE_ATTRIBUTE to http.status.code.

The same method may also be used to set a non-200 success code.

All HTTP headers sent in the HTTP request will be imported as Flowfile attributes in the data flow, with an attribute name like azure.http.header.<lowercaseHeaderName>. For example, submitting a request with a header X-My-Test-Header: test will result in the flowfile having an attribute named azure.http.header.x-my-test-header with a value of test.

Creating an Azure function using CLI

Follow these steps to deploy a Cloudera Data Flow Functions using the Azure CLI.

Before you begin

- Your NiFi flow definition is stored in the Cloudera Data Flow Catalog and you have the CRN corresponding to the flow version you want to execute as a function
- You have created a Machine User with the proper role and you have its Access Key and Private Key credentials
- You have installed and configured the Azure CLI on your local machine

About this task
These steps provide an example for deploying a Function App that listens for HTTPS requests with a publicly exposed endpoint. Cloudera recommends going through the documented steps in the Azure Console UI to generate the appropriate templates and use those templates for CLI-based deployments.

Procedure

1. Create a Resource Group that you will use for the Function App. You can skip this step if you want to reuse an existing Resource Group.

```
az group create -l francecentral -n MyDFFResourceGroup
```

2. Download the template.json file available here.

```
az deployment group create --resource-group MyDFFResourceGroup \
  --template-file <path to template.json> \
  --parameters \
  subscriptionId=<existing subscription ID to use>  \
  name=<name of your FunctionApp> \
```

```
    location=<region where to deploy your Function App> \
    hostingPlanName=<name of the plan to be created> \
    resourceGroup=<name of the resource group previously created> \
    storageAccountName=<name of the storage account to be created> \
    FLOW_CRN=... \
    DF_ACCESS_KEY=... \
    DF_PRIVATE_KEY=...
```

If you have additional parameters, you can add those in the appSettings section of the template.json file.

Note: For sensitive parameters, Cloudera recommends to use Azure Key Vault as explained in the Parameters documentation. This is not covered in the provided template.

Example:

```
az deployment group create --resource-group MyDFFResourceGroup \
  --template-file /tmp/template.json \
  --parameters \
      subscriptionId=00000000-0000-0000-000000000000 \
      name=DFFResizeImage \
      location=francecentral \
      hostingPlanName=MyHostingPlanDFF \
      resourceGroup=MyDFFResourceGroup \
      storageAccountName=mydffstorageaccountname \
      FLOW_CRN=crn:cdp:df:us-west-1:00000000-0000-0000-0000-000000000000:
flow:my-flow/v.1 \
      DF_ACCESS_KEY=00000000-0000-0000-0000-000000000000 \
      DF_PRIVATE_KEY=00000000000000000000000000000000
```

3. Retrieve the File Share name that has been created during the previous operation by executing the below command:

```
az storage share list --account-name mydffstorageaccountname
```

You can automate the retrieval with jq using the below command:

```
FILESHARE=`az storage share list --account-name mydffstorageaccountname |
  jq -r '.[0].name'`
```

4. Create the directories in the File Share:

```
az storage directory create --account-name mydffstorageaccountname --sha
re-name dffresizeimageadcb --name data
az storage directory create --account-name mydffstorageaccountname --shar
e-name dffresizeimageadcb --name data/SitePackages
```

5. Upload the binary to the File Share and create the package file:

```
az storage file upload -s dffresizeimageadcb --source /tmp/NaafAzureFunc
tions.zip --path data/SitePackages/ --account-name mydffstorageaccountname

echo "NaafAzureFunctions.zip" > /tmp/packagename.txt && az storage file
upload -s dffresizeimageadcb --source /tmp/packagename.txt --path data/S
itePackages/ --account-name mydffstorageaccountname && rm /tmp/packagena
me.txt
```

Results

Your function is now ready to be used with HTTPS triggers. See the documentation for more details. If another type of trigger is required, you need to create an Event Grid subscription as explained in Azure Function triggers.