k2view

Alpinist

LU Schema Functions

WELCOME

Agenda

- Project Functions
- Types of Project Functions
- Creating Functions on the Cloud
- Decision Function
- Trigger Function
- Enrichment Functions
- Event Function
- LUDB Function
- Reading the GET Logs

Project Functions

- Fabric Project functions are user-defined Java functions that are added to the project implementation.
- Functions can be created in Logical Units, References, Web Services, Shared Objects or from existing Table Populations.
- A function can be defined as a Shared Object and can then be used in any object in a project.
  If a function is defined in an LU, Reference or Web Service, it is accessible only within that specific object.

Types of Project Functions

Fabric supports six types of Java functions, each distinguished by its purpose, signature, usage, and context:

Function type	Purpose	Context	Signature *In blue - mandatory

Decision Function	Determines whether a table population will be executed during the sync process	Sync process	@type(**DecisionFunction**)
			@out(name = "decision", type = Boolean.class, desc = "")
			public static **Boolean** descisionFunc**()** throws Exception {
			}

Trigger Function	Executed at the record level when a record undergoes a change (INSERT, UPDATE, or DELETE)	Sync process	@type(**TriggerFunction**)
			public static **void** triggerFunc(**TableDataChange tableDataChange**)
			throws Exception {
			}

Enrichment Function	Executed during the sync process, once all the LU populations have been executed	Sync process	public static **void** enrichmentFunc**()** throws Exception { }

Event Function	Executed as part of the GET process, after the sync process is completed.	GET process	@type(**EventFunction**)
			public static **void** eventFunc(**EventDataContext eventDataContext**)
			throws Exception {
			}

LUDB Function	Executed from within an SQL statement	SQL Statement	@type(**LudbFunction**)
			@out(name = "result", type = String.class, desc = "")
			public static **String** ludbFunc(@desc("") **String param1**) throws Exception {
			}

Java Function	Regular Java function	Any Fabric object	@out(name = "", type = String.class, desc = "")
			public static void regularFunc(@desc("") String param1) throws Exception {
			}

Types of Project Functions

Studio integration with functions:

- The @type annotation in the function declaration determines where the studio will display this function, such as under the Event Functions list, Trigger Functions list, or Enrichment Functions list.
- The function declaration must comply with the function type rules

BW integration with functions:

The LuFunction Actor can be utilized in BW flow to invoke a Java function. The functionName input parameter is used to determine which Java function to call. Once set, Fabric will automatically perform the following actions:

- Set the actor's input parameters corresponding to the names specified in the function's input, for example:
  @out(name = "result", type = String.class, desc = "")
  public static void functionName(@desc("") String **param1**) throws Exception {
  }

- Add the actor's output parameter corresponding to the name defined in the @out annotation:
  @out(name = "**result**", type = String.class, desc = "")
  public static void functionName(@desc("") String param1) throws Exception {
  }

Types of Project Functions

Functions context

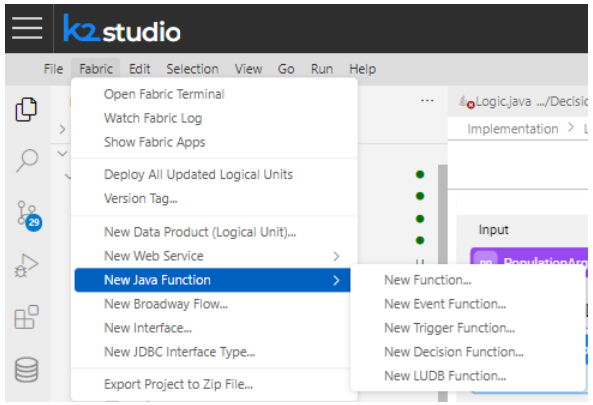
If a function is within the sync context, it can utilize sync process context functions, such as:

- getInstanceID()
- getLastSyncTime()
- getTableName
- and others.

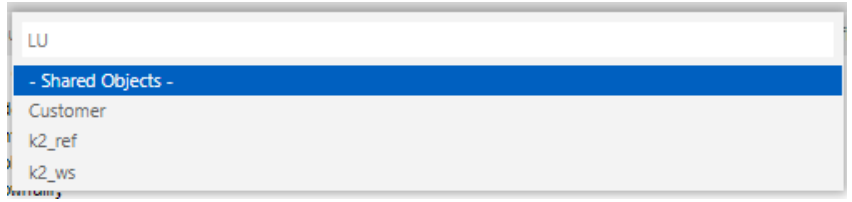
If a function is not running within the sync context, it cannot use these functions. However, if it’s part of the GET process, the LUI is still attached to the session and therefore can be queried.

Steps to Create Project Function in Cloud Studio

1. Main Menu → Fabric → New Java Function → Select the desired function type



2. Choose LU type name or Shared Objects (if to be shared across all project LUTs)



3. Choose Category

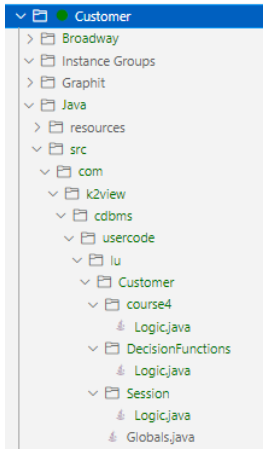
Steps to Create Project Function in Cloud Studio

Each Category defines a new Package, containing Logic.java file.

- Each package creates a folder with Lofic.java file, under workspace/project/Implementation/LogicalUnits/Customer/Java/src/com/k2view/cdbms/usercode/lu/Customer

```
fabric@dev-fabric-deployment-848b7d9f89-jc668:~/workspace/project/Implementation/LogicalUnits/Customer/Java/src/com/k2view/cdbms/usercode/lu/Customer$ ll
total 24
drwxrwsr-x 5 fabric fabric 4096 Aug 18 13:35 ./
drwxrwsr-x 3 fabric fabric 4096 Jun 26 08:41 ../
drwxrwsr-x 2 fabric fabric 4096 Aug 14 16:42 DecisionFunctions/
-rw-rw-r-- 1 fabric fabric  376 Jun 26 08:41 Globals.java
drwxrwsr-x 2 fabric fabric 4096 Jul  7 11:11 Session/
drwxrwsr-x 2 fabric fabric 4096 Aug 18 13:35 course4/
```

- Each package creates a folder in the project:

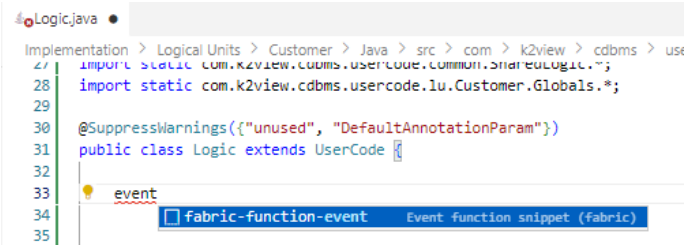


Steps to Create Project Function in Cloud Studio

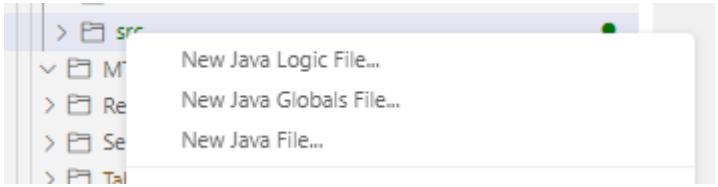
4. The Logic.java file serves as the main container for all functions within the category.
   It extends the UserCode.

   Note: You can view a list of UserCode functions in the main menu under the “Documentation” section on the web page.

5. Once the Logic.java file is created, you can add functions by typing the desired function type, and Fabric will automatically generate the function signature.



Alternatively, you can right click “src” and open the below menu:



- **New Java Logic** file option will create a new category
- **New Java file** option can be used if you want to call Java functions from within your Logic files. However, functions written in "New Java file" won't be detected by Fabric Studio.

Steps to Create Project Function in Cloud Studio

Note:

- When functions with the same name exist in multiple packages ("Category"), Fabric does not raise an error. Instead, the server will execute one of the functions at random.
- To use a function from a one package ("packageName1") within another package, you can:
  - Add the following import statement: import static com.k2view.cdbms.usercode.lu.Customer.packageName1.Logic.*;
  - Alternatively, use the full package name when calling the function.

Decision Function

What Is Decision Function?

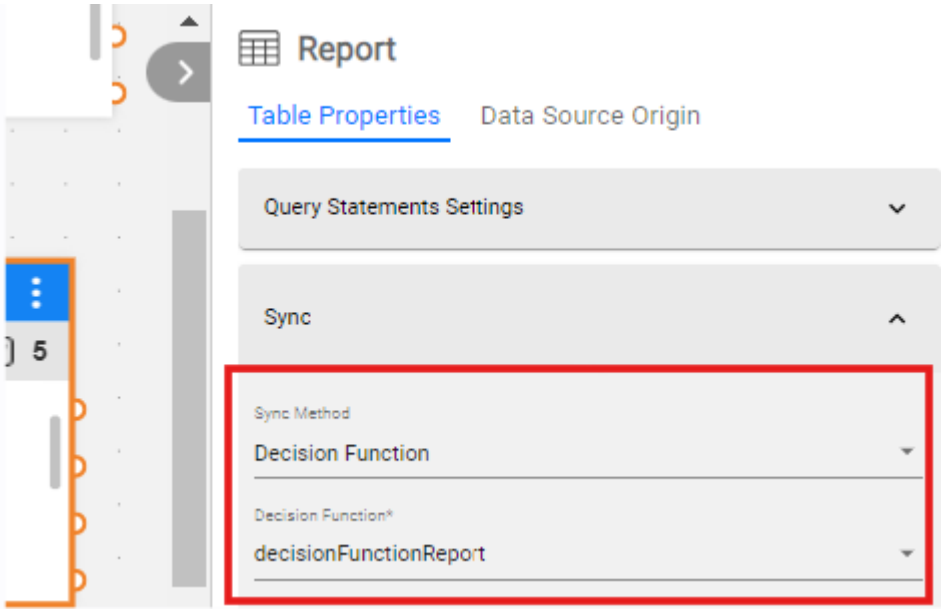
Decision functions are used to control whether a table population should be executed during an LUI sync process.

These Java-based functions return a Boolean value:

- **True**: the population will be executed.
- **False**: the population will NOT be executed.

Decision functions can be defined at various levels:

- **LU Schema:** Will be used by all the table that are in 'Inherit' sync method.
- **LU Table:** Will be used by all the table populations that are in 'Inherit' sync method
- **Table population:**
  - **.Net Studio:** Will be used by the specific population
  - **Cloud Studio:** Not applicable



Decision Function

Key characteristics of Decision function

- Decision function is running in the context of the sync process. Therefore, it can use the sync process context functions, such as: getInstanceID(), getLastSyncTime(), getTableName, etc.

- Decision function take precedence over any Sync Mode (expect for sync OFF): Sync ON, Sync FORCE, first sync – all those scenarios will activate the decision function, and accordingly the population will run or not. Make sure your decision function return true for isFirstSync and if the sync mode is FORCE.

- In the event of a schema upgrade, the decision function will take precedence over the schema change. This means that if the decision function returns false, the populations will not run, even though the schema upgrade has occurred. Consequently, the LUI will not go through this schema upgrade again.
  To ensure proper handling, every decision function must account for schema changes by using the isStructureChanged() built-in function.

Decision Function

When to use Decision Function?

Use decision function when syncing a table is bound to rules or pre-checks. The logic if to execute the population is implemented in the decision function's code.

For example:

Run sync only during off-peak hours.
A Decision function can check the current date and time.

- If the current date and time = off-peak, return True to Sync the LUI.
- If the current date and time = peak, return False to skip the Sync.

In this example, it is recommended to use the skipSync() method in the Decision function to perform a one-time execution of the Decision function per LUI (in case all tables in the LU inherit the same logic).

Decision Function

Function signature & Best Practices

```
@type(DecisionFunction)
@out(name = "decision", type = Boolean.class, desc = "")
public static Boolean descisionFunc() throws Exception {
        return true;
}
```

Best Practices:

- If the decision function returns the same result for each population, it's advisable to set it on the Root Table's population and invoke skipSync(). This approach allows Fabric to execute the Decision function once per LUI, rather than for each population individually.

- Since decision functions impact the overall sync process time, it's important to avoid overloading them with complex processing logic.

- If the LU Schema is changing, the decision function will still execute. To prevent blocking the populations from running, use isStructureChanged() in your implementation, to return true.
  Same for logic should be handled for first sync or sync mode OFF:

```
Boolean toRun = (isFirstSync() || isStructureChanged() || getSyncMode().equals(SyncMode.FORCE.name()));
```

- A failure in a decision function will cause the entire LUI sync to fail.

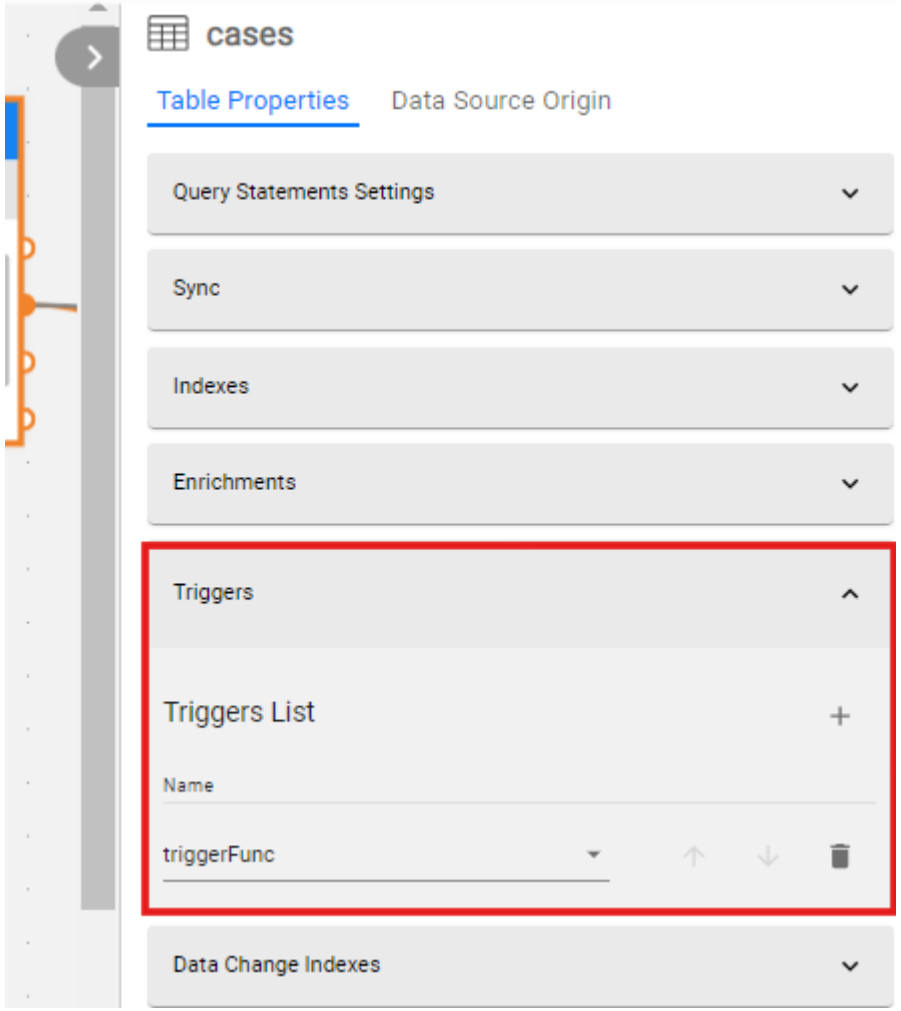
Trigger Function

What Is Trigger Function?

A Trigger function is defined on LU table, and triggered once a record in the table is changing/added/deleted.

The trigger function receives an input called **TableDataChange**, which provides details about the change, including:

- Table name
- Type of event (Insert/Update/Delete)
- Old values of the record (empty on insert)
- New values of the record (empty on delete)



Trigger Function

Key characteristics of Trigger function

- Trigger function does not have context of the sync process. Therefore, it cannot use the sync process context functions, such as: getInstanceID(), getLastSyncTime(), getTableName, etc.

  Still, the LUI is attached and can be queried. For example:

```
String customer_id = ""+fabric().fetch("select customer_id from customer").firstValue();
```

- Triggers are defined at the table level; you cannot specify trigger for individual fields.
- Trigger functions are executed for every insert, update, or delete operation:
  - The trigger will activate even for an update command that doesn't change any value in the record.
  - If a record is updated multiple times, the trigger function will execute each time.
- A Trigger function runs immediately after the transaction is applied, without waiting for the sync to finish successfully or for a commit. As a result, the functionality will execute regardless of the sync process outcome (success or failure).

Trigger Function

Key characteristics of Trigger function

- Trigger functions are added to the LUI during each GET request, right before the table population is executed, and then removed after the SQLite is committed (and Event functions are executed).
  If trigger is defined on a table that its population is not being executed during the GET, the trigger will not be created on the LUI.
  *Therefore, during sync OFF triggers are not being created

Trigger Function

When to use Trigger Function?

Trigger Functions are used to perform an action when a specific set of data or value is inserted, updated or deleted.

For example:

Track order status and executing special logic when order status changes from Terminated to Activated.

Trigger Function

Function signature & Coding

```
@type(TriggerFunction)
public static void triggerFunc(TableDataChange tableDataChange) throws Exception {
}
```

The Trigger function receives TableDataChange input, which provides detailed information about the changes that occurred in the record:

- **getTable()** - Returns the name of the table where the change occurred.
- **changedFields()** - Returns a hashmap containing all fields whose values were changed.
  Example: `if (tableDataChange.changedFields().get("FIELD_NAME") == ...)`
- **getType()** - Returns the type of transaction (INSERT/UPDATE/DELETE).
  Example: `if (tableDataChange.getType().equals(DataChangeType.INSERT))`
- **oldValues()** - Returns a hashmap of fields with their old values.
- **newValues()** - Returns a hashmap of fields with their new values.
  Example:
  `if (tableDataChange.newValues().get("ID").equals(tableDataChange.oldValues().get("ID")))`

Trigger Function

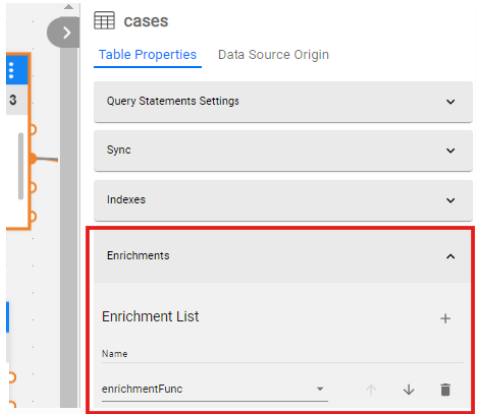
Best Practice

- Trigger functions are created and dropped on each GET. Be mindful of this potential overhead when considering the addition of multiple trigger functions.
- When a Trigger function is defined to a table, it is activated for any change in any field. If your logic depends on a specific field, make sure to check the old and new values of that field before taking any action.
- When comparing the type of data change, use the DataChangeType enum for accuracy.
  - Instead of: tableDataChange.getType().equals("DELETE")
  - Use: tableDataChange.getType().equals(DataChangeType.DELETE)
- If a primary key (PK) is not set on the trigger's table, Fabric will insert the same record multiple times instead of updating it.
- When a population is running, Fabric will, by default, delete all records in the table and re-insert them from the source. This triggers SQLite to activate triggers for both deletions and insertions for each record that existed in the LUI and is now being fetched from the source. To avoid this, set the delete mode to "Non-updated" and the Sync Method to "upsert."
- When using “Insert or replace” command, in case the record exists, SQLite deletes the record and inserts it back again. Using “Insert...on conflict update” command to avoid this behavior

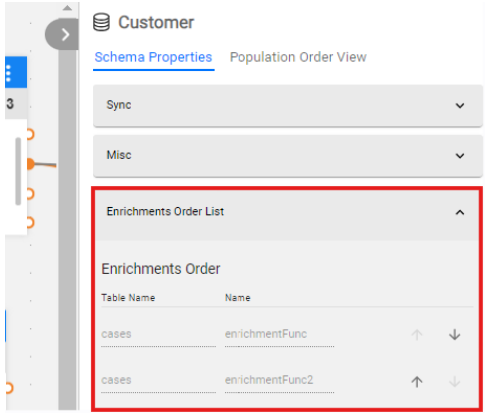
Enrichment Function

What Is Enrichment Function?

An Enrichment function is designed to execute specific functionality after all populations have been completed during the sync process.



Enrichment function is set on a table level, and the execution order of all Enrichment functions is set on the LU Schema properties.



Enrichment Function

Key characteristics of Enrichment function

- Enrichment function is linked to an LU table and will be executed only if at least one of the populations were executed during the Sync process.
- The function is triggered only after ALL populations within the LU schema have been executed.
- The execution order of all enrichment functions is determined in the Schema properties.
- When defining multiple Enrichment functions they will run in sequence.
- The Enrichment function runs within the sync process context, allowing it to access and utilize sync process context functions.

Enrichment Function

When to use Enrichment Function?

As the name implies, an enrichment function enhances the functionality of the LU.

For example:

- Populating an LU table with calculated data derived from other LU tables, such as calculating the total amount of a customer's payments and updating this value in the CUSTOMER LU table.

Enrichment Function

Function signature & Best Practices

```
public static void enrichmentFunc() throws Exception {
}
```

Best Practices:

- Enrichment function is running in the context of the sync process. Therefore, it can use the sync process context functions, such as: getInstanceID(), getLastSyncTime(), getTableName, etc.
- While enrichment functions are very beneficial, it's recommended to use population flows for better control and visibility.
- Enrichment functions contribute to the overall sync process time, so avoid overloading them with heavy processing logic.
- Since enrichment functions don't receive input, thread globals are typically used as flags or for sharing information.
- If the enrichment function is used for updating data in a table within the LUI, there's no need to perform a commit; it will automatically be done by Fabric when the sync is completed.
- A failure in an enrichment function will cause the entire LUI sync to fail.

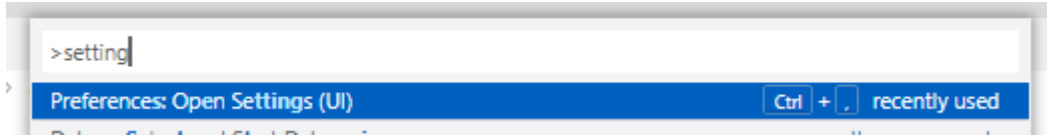
Enrichment Function

Cloud Studio

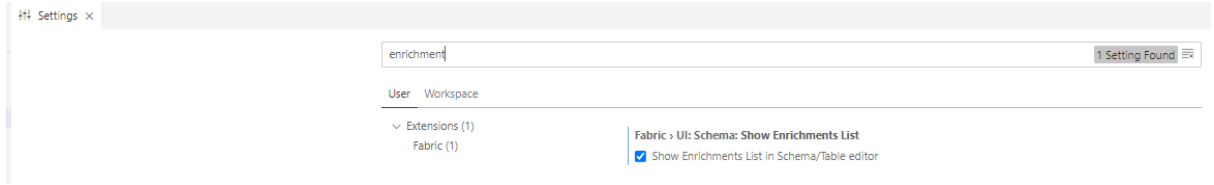
Enrichment functions are not supported in the Cloud studio by default.

To enable enrichment functions, do the following:

1. Main menu → View → Command Pallet
2. Type “setting” and select “Reference: Open Settings (UI)”



3. Type “enrichment” in the tab that is opened and check “Show Enrichment List in Schema/table editor”



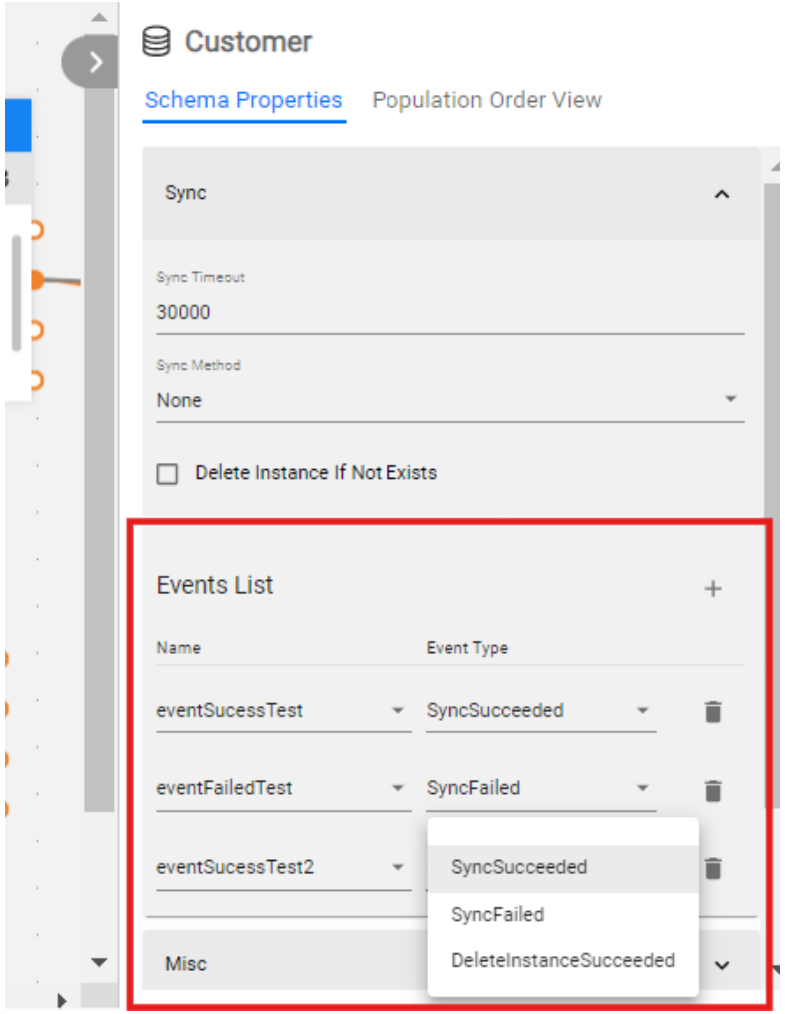
4. ‘Enrichment Order List’ is added to the Schema properties
5. ‘Enrichments’ is added to the table properties.

Event Function

What Is Event Function?

An Event function is triggered at the final step of the GET process, after the sync process is completed. Users can configure three types of events:

1. On Sync Success (“SyncSucceeded”)
2. On Sync Failure (“SyncFailed”)
3. On Successful Instance Deletion (“DeleteInstanceSucceeded”)



Event Function

Key characteristics of Event function

- Event functions operate outside the sync process, meaning they do not have access to the sync context.
    - For success or failure GET events, the LUI is attached to the session, allowing it to be queried.
    - When deleting the LUI, use the EventDataContext parameter to retrieve the necessary information.

Event Function

When to use Event Function?

An Event function is the first step executed after the instance sync process is completed, allowing you to act based on the outcome - whether the sync was successful, failed, or the instance was deleted.

For example:

- The “SyncSucceeded” event type can be used to update stats table or send notifications to a third-party system.
- The “SyncFailed” event type can be used to handle specific exceptions, allowing you to take actions such as logging the issue as a statistic, sending a notification, or raising an alert.
- The “DeleteInstanceSucceeded” event type is useful when customer data (or any other LUI data) needs to be cleaned from Cassandra lookup tables once the LUI is deleted from Fabric.

Event Function

Function signature

```
@type(EventFunction)
public static void eventFunc(EventDataContext
eventDataContext) throws Exception {
}
```

The **EventDataContext** data type exposes a set of methods which allow getting additional information about the change such as:

- Instance ID - eventDataContext.getInstanceId()
- LU Type name - eventDataContext.getLuTypeName()
- Exception - eventDataContext.getLastException()

Event Function

Best Practices

- Event functions are executed as part of the GET process and run synchronously, which can extend the duration of the GET operation.

- If multiple event functions of the same type are defined, they will be executed sequentially, with each function waiting for the previous one to complete before starting. if you want to run some activities in parallel, do it under the same event function.

- If your event function involves heavy processing, consider implementing asynchronous code:
  - Using Java
  - By executing your code as a BW Job

  Note: When activating a BW flow from an event function, even if the BW runs its functionality using innerFlowAsync, the GET process will not complete until the innerFlowAsync operation is finished.

- If an exception occurs in the event function, the GET process will fail. Since the event function runs only after the sync process is completed, the sync itself may have finished successfully, with the final changes already committed to storage.

- If "skip sync" is used during the sync process, event functions will not be triggered.

- If "reject instance" is used during the sync process, only DeleteInstanceSucceeded will be triggered.

- When setting thread globals during the sync, ensure they are cleared afterward in both "Sync Succeeded" and "Sync Failed" functions.

Ludb Function

What Is Ludb Function?

An LUDB (Logical Unit Data Base) function is a Project function invoked from an SQL query to perform more complex operations on an LU or reference data than those performed using standard SQL statements.

- LUDB functions are invoked from an SQL statement.
- LUDB functions must have at least one output value.

The LUDB function is created on the SQLite DB during the GET process, and can then be used by SQL statement.

For example:

```
String sql = "select  case_date, case_type, fnludbFunc(cases.status) as new_status from case"
try (Db.Rows rows = ludb().fetch(sql,input)😉 {
    for (Db.Row row:rows){
        yield(row.cells());
    }
}
```

Ludb Function

When to use Ludb Function?

Use an LUDB function when the desired logic cannot be achieved through a SQL statement, either due to its complexity or the need for Java's access to additional resources and libraries.

Ludb Function

Function signature & Best Practices

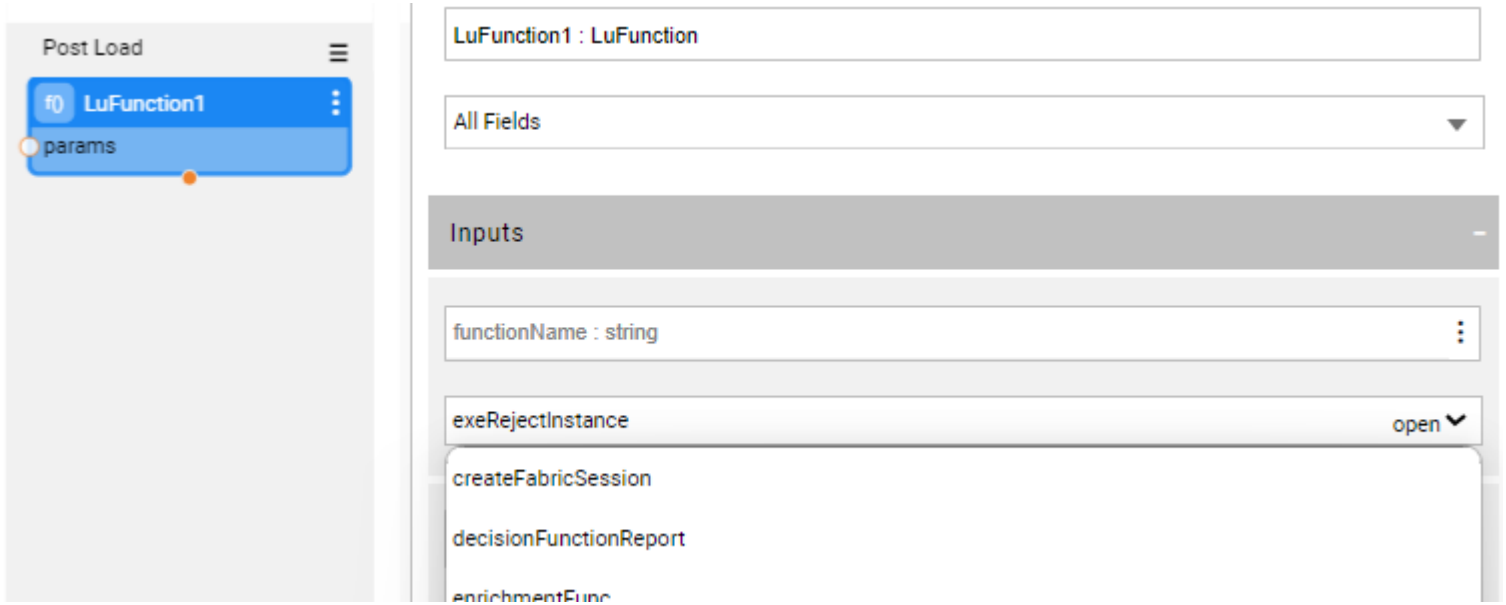
```
@type(LudbFunction)
@out(name = "result", type = String.class, desc = "")
public static String ludbFunc(@desc("") String param1)
throws Exception {
}
```

Best Practices:

LUDB functions should accept input and return output. An LUDB function that does not have or use input parameters will be called multiple times and will return the same result for all rows.

Regular Java Function

To execute a Regular Java function from BW Flow, use LuFunction Actor



GET process logs

```
INFO  2024-08-31 21:48:38,424 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.FabricSession - START - ATTACH Customer.215
INFO  2024-08-31 21:48:38,426 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.FabricSession - Access to [Customer.215] by user shani.alpinist@k2view.com is authorized.
INFO  2024-08-31 21:48:38,426 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.FabricSession - local get request
INFO  2024-08-31 21:48:38,430 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.FabricSession - START - sync Customer.215
INFO  2024-08-31 21:48:38,432 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.s.l.SyncExecution - Start operation 'Sync Customer.215'
INFO  2024-08-31 21:48:38,443 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------Trigger function for table CASES
INFO  2024-08-31 21:48:38,449 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------Trigger function for table CASES
INFO  2024-08-31 21:48:38,458 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------Decision function for table REPORT
INFO  2024-08-31 21:48:38,461 [LID240400000000007a] [TerminalWebSocket] c.k.b.a.b.Logger - ----------Report is running
INFO  2024-08-31 21:48:38,463 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.s.l.SyncExecution - End operation 'Sync Customer.215' successfully.  [31ms]
INFO  2024-08-31 21:48:38,464 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------enrichment function
INFO  2024-08-31 21:48:38,465 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------another enrichment function
INFO  2024-08-31 21:48:38,466 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.s.l.SyncExecution - Customer:215 was synced from source
INFO  2024-08-31 21:48:38,470 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------Event function success
INFO  2024-08-31 21:48:38,470 [LID240400000000007a] [TerminalWebSocket] c.k.f.e.Events - Event method SyncSucceeded.eventSucessTest was executed
INFO  2024-08-31 21:48:38,470 [LID240400000000007a] [TerminalWebSocket] c.k.c.s.u.UserCode - ----------Another Event function success
INFO  2024-08-31 21:48:38,471 [LID240400000000007a] [TerminalWebSocket] c.k.f.e.Events - Event method SyncSucceeded.eventSucessTest2 was executed
INFO  2024-08-31 21:48:38,471 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.FabricSession - FINISHED - sync Customer.215 (UPDATE)
INFO  2024-08-31 21:48:38,471 [LID240400000000007a] [TerminalWebSocket] c.k.f.s.FabricSession - FINISHED - ATTACH Customer.215 (UPDATE)
```

Sync process:
Populations
Triggers functions
Decision functions
Enrichment functions
Commit the LUI changes

Event functions

Regular Java Function

1. `c.k.f.s.FabricSession - START – ATTACH Customer.215`
   a. Validate LU type.
   b. Check LUI authentication permissions for the Fabric User.
2. `c.k.f.s.FabricSession - Access to [Customer.215] by user shani.alpinist@k2view,com is authorized.`
   a. Verify if there is any MDB already attached to the session that cannot be released due to an ongoing transaction (from the same LU type). If so, an error will occur: "Attached LU can't be detached while in transaction."
3. `c.k.f.s.FabricSession - local get request`
   a. Extract the LUI from storage.
   b. Decompress the LUI.
   c. Decrypt the LUI if needed.
   d. Check for schema upgrades by comparing to the LU type definition.
4. `c.k.f.s.FabricSession - START - sync Customer.215`
   a. Perform ElevatedPermissions Check
   b. Attach file- MDB file locking.
5. `c.k.f.s.s.l.SyncExecution - Start operation 'Sync Customer.215'`
   a. Execute Decision functions, create **Triggers** and run all populations
   b. Execute Trigger functions
   c. Update k2_objects_info after each population
6. `c.k.f.s.s.l.SyncExecution - End operation 'Sync Customer.215' successfully. [60078ms]`
   a. Execute all enrichment functions
7. `c.k.f.s.s.l.SyncExecution - Customer.215 was synced from source`
   a. Perform SQLite commit
   b. Execute Event functions
   c. Remove triggers
   d. Compress the LUI.
   e. Encrypt the LUI if needed.
   f. Save the LUI back to storage.
   g. Clean up resources.
8. `c.k.f.s.FabricSession - FINISHED - sync Customer.215 (UPDATE)`
9. `c.k.f.s.FabricSession - FINISHED - ATTACH Customer.215 (UPDATE)`