Magic xpi Connector Builder

Introduction

The Magic xpi Connector Builder is a new feature that lets developers write professional grade connectors with all of the capabilities used by the Magic xpi built-in connectors.

Connectors developed with the new builder can have the following capabilities:

- A step with custom configurations and built-in data mappers (flat file, JSON and XML).
- Methods interface.
- Custom Resources and Services, a Validate button and three additional action buttons.
- A trigger with either custom or static configuration screens.
- Polling, External and Endpoint triggers are supported.
- External and Endpoint triggers can have synchronous or asynchronous behavior.
- Built-in expression editor, variable selection and error/message dialog boxes for the configuration UI in a single line of code.
- .NET, Java and Magic xpa are supported for the runtime technology.
- .NET technology for configuration UI implementation (WinForms or WPF).

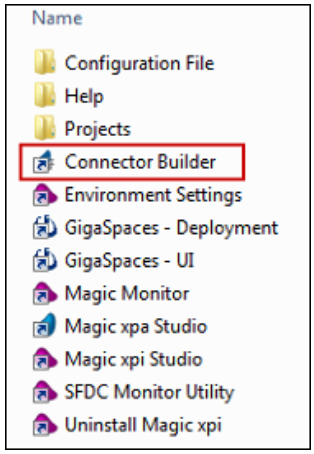
The following document covers the creation of a new dynamic connector using the Magic xpi Connector Builder. In addition, triggers will be discussed.

Writing a Dynamic Step

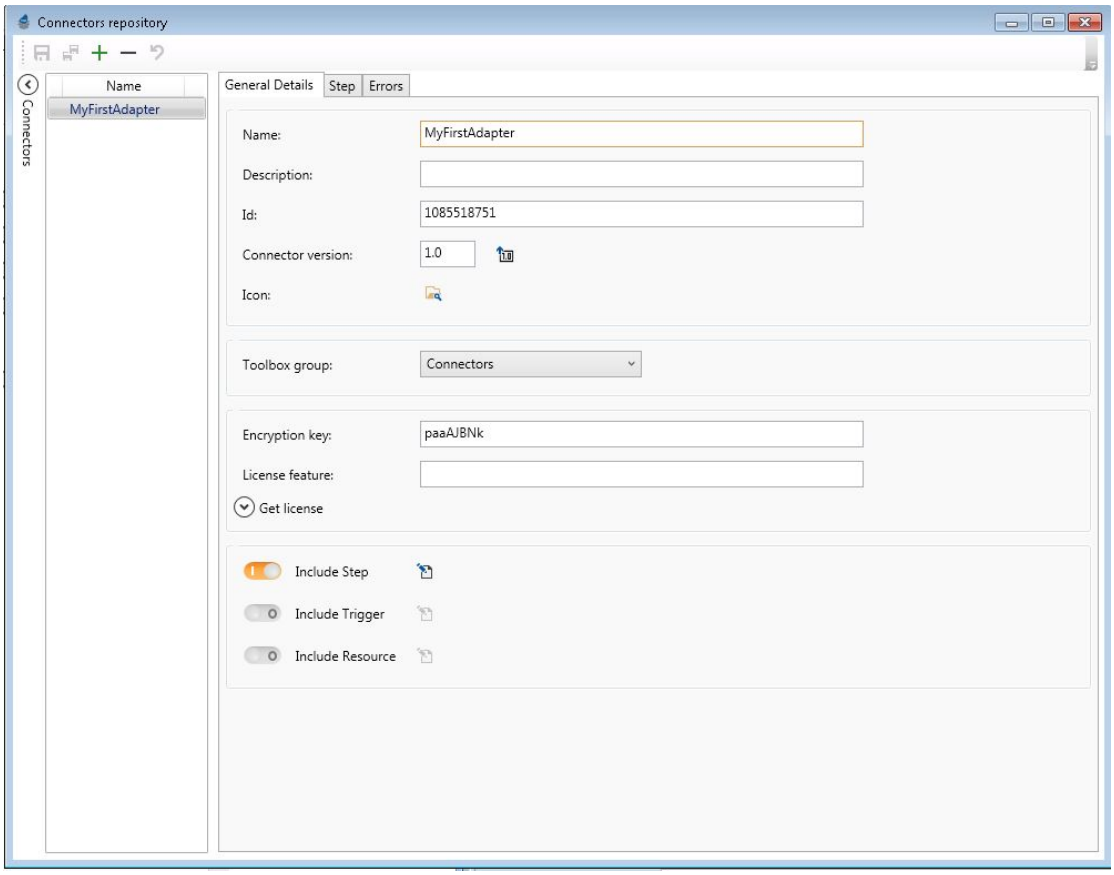
The first step in creating a connector is to define it using the Connector Builder utility. This utility combines all of the different parts of the connector together, such as the resource, UI code, runtime code, errors, and icon etc.

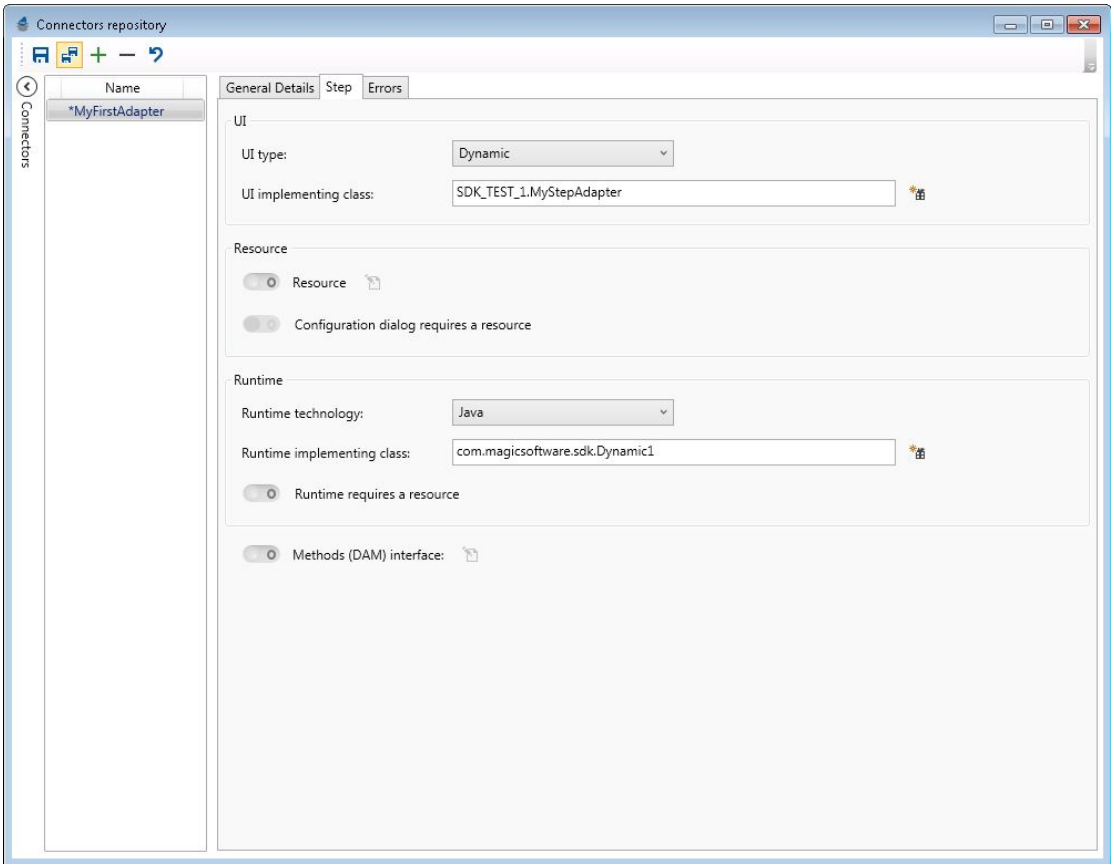
Defining the Connector Using the Connector Builder

1. Open the Connector Builder utility from the Magic xpi start menu or desktop folder.
2. Define a new connector using the **Create new connector** button at the top left corner of the window.



3. Define the connector name and optional icon and description.
4. Open the Step tab and define the **UI type** as **Dynamic**.
5. Provide the **UI implementing class** using the following syntax: <namespace>.<class name>
    - In the given example we used: **SDK_TEST_1.MyStepAdapter**
6. To make the connector Local Agent compatible, turn on the Include Resource toogle button and then on Resource tab turn on the Local Agent compatibility toogle button. By default this option is off.
7. Select Java or .NET or Magic xpa as the runtime technology.
8. Define the Runtime implementation class.
    - For Java use: com.magicsoftware.sdk.Dynamic1
    - For .NET use: SDK_TEST_1.MyStepAdapterRT
    - For Magic xpa use: MyConnectorName.ecf
9. Click Save all connectors changes button and close the builder.





This configuration will create the connector folder in the following path: **<Magic xpi root>\Runtime\addon_connectors**.

The connector folder will have the same name as the one you gave to the connector.

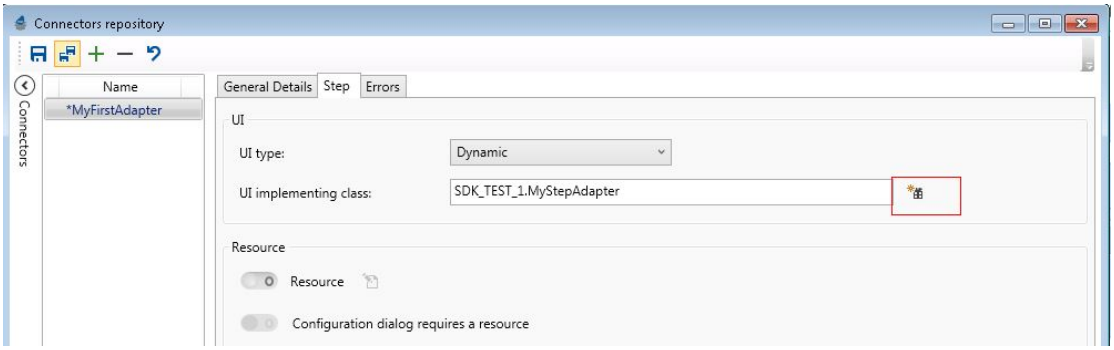
The connector folder contains the following important locations:

- **ui\lib** – This folder will contain the UI dlls.
- **runtime\java\lib** – This folder will contain the runtime jar files that you create for the runtime part.
- **runtime\dotnet \lib** – This folder will contain the runtime dlls that you create for the runtime part.
- **Runtime\magic xpa\lib** – This folder will contain the runtime ecfs that you create for the runtime part.

Creating the Configuration UI for the Studio

The easiest way to start developing the UI part of the connector is to generate a pre-defined template.

Use the **Generate UI Project** utility found next to the UI class name. This utility will generate a preconfigured sample project with all references, implemented interfaces and some usage examples.



In the utility you will be asked for the .NET project name and its location. The namespace for this project will be taken from the **UI implementing class** property that you filled in previously.

*** Appendix A contains the manual steps for setting the UI project in the IDE.

Once the project is built, the project dlls need to be copied to the **<connector>\ui\lib** folder.

IUserComponent Interface Explained

The step adaptor class for the UI part of the connector contains several methods that have to be implemented:

CreateDataObject() method

This method is expected to return an object with a set of properties that you intend to use for your connector configuration. This object is your class instance that holds a set of Magic xpi properties. (You can find an example of this type of class in the generated template.)

The properties can be Magic primitives, such as Alpha or Numeric, but can also represent a **Variable** or **Expression**.

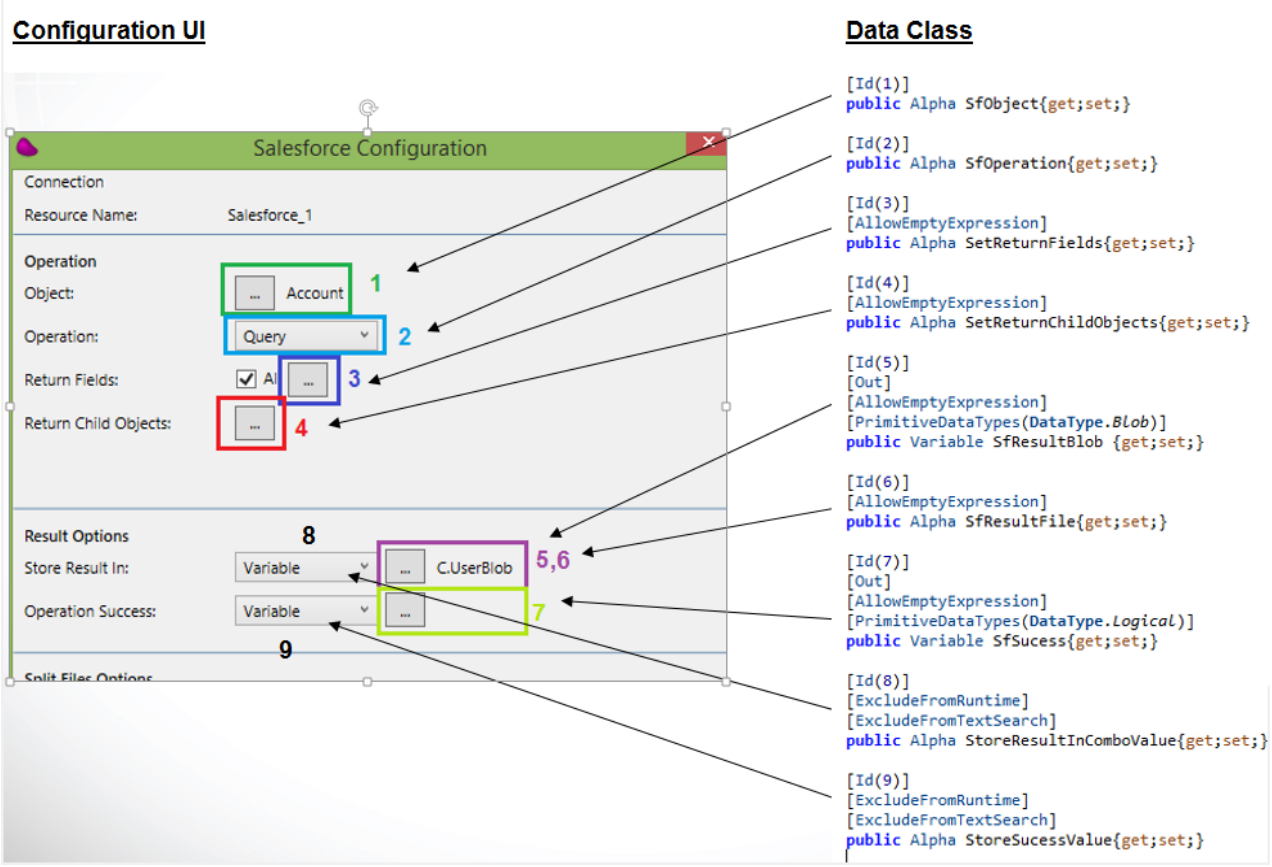
For each property, you can define a set of annotations that control its behavior. For example, for a property of type Variable you can define the direction. This means that at runtime, the variable mapped to this property can only pass its value to the connector **[In]**, only update a new value after the connector execution **[Out]** or both **[InOut]**. Another example is the definition of Encoding set for an Alpha property.

In a step, primitive Magic types (Alpha, Logical, Numeric, Date, and Time) and the Expression type can only pass values to the step. This means that their direction is always **IN**. There is no need to define the direction annotation for these types since **IN** is the default direction.

For the Variable Magic type, since a variable can be used both as the input data to the step and as the one holding the step result, it is possible to define that a Variable type will be:

- **[In]** – The variable data will be passed to the step at runtime. If, for example, the selected variable is C.UserString, the data in C.UserString will become available to the runtime code.
- **[Out]** – The step's runtime logic can update some data into this variable. If, for example, the selected variable is C.UserString, once the step ends, the C.UserString may hold some data updated from the step's logic.
- **[InOut]** – The Variable data will be passed to the step at runtime and the step's logic can update some data into this variable when the step ends.

As an example, we will take the Salesforce connector and map its configuration properties to the expected data class:



- Object (1) – Clicking the button will open a connection to Salesforce, request the list of all objects, list them in a dialog box and let the user select the required object. The field will hold a static Alpha value of the selected object name. This value is all that is needed for runtime from the **Object** property.
- Operation (2) – This field will use part of the metadata retrieved when selecting the object to list the object's supported operations. The field will hold a static Alpha value or a static numeric representing the enumeration of the operation (1,2,3..) This value is all that is needed for runtime from the **Operation** property.
- Return fields (3) – Clicking the button will open a connection to Salesforce and retrieve the object fields list. The user can multi-select the required fields. This field will hold a static Alpha value with the list of object fields. This value is all that is needed for runtime from the **Return fields** property. Since this property is not required, we will use the annotation **[AllowEmptyExpression]** to prevent the checker from raising an error if this property does not hold a value.

- Return child objects (4) – Clicking the button will open a connection to Salesforce and retrieve the list of child objects. The user can multi-select objects. This field will hold a static Alpha value with the list of child objects' names. This value is all that is needed for runtime from the **Return child objects** property. We will use the annotation **[AllowEmptyExpression]** to prevent the checker from raising an error if this property does not have a value.
- Store result in > Variable selection (5) – This property should hold a name of a variable that will be updated with the step result at runtime. We will add an annotation of **[Out]** to indicate the direction. Since this is a variable, we must define its type for both the checker and runtime. We will do that by adding another annotation **[PrimitiveDataTypes(DataType)]**. We also see the annotation **[AllowEmptyExpression]** on this property since we will either have a file or a variable, so neither of these options can be set as mandatory.
- Store result in > File selection (6) – This property should hold the file path that will be updated with the step result at runtime. Since this is a file path, the connector must get this value at runtime. Therefore, the direction of this property is IN. The actual connector code at runtime will create the file. We also see the annotation **[AllowEmptyExpression]** on this property since we will either have a file or a variable, so none can be set as mandatory. Making sure that one of them will have a value can be done either at the UI level or with the **check()** method that adds additional logic to the checker process.
- Operation Success > Variable selection (7) – This property should hold a name of a variable with the direction OUT. The definition is similar to **Store result in** except for the variable type is Logical instead of BLOB.
- **Store result in combo box (8) and Operation Success combo box (9)** – The data class allows the developer not only to store properties related to the runtime but also to store properties related to design time. For the Salesforce configuration screen we would like to store the combo box value, so when the connector is reconfigured, the same values that the user selected previously will appear in the combo box. There are two important annotations in this example:

  1. **[ExcludeFromRuntime]** – This annotation indicates that this property value should not be passed to the runtime code.

  2. **[ExcludeFromTextSearch]** – This annotation indicates that this property should be excluded from the Studio text search. We do not want a result from the combo box value to appear in the text search results pane.

configure() method

The **configure()** method is the main entry point for the configuration UI of the connector. From this method developers should:

- Open their custom configuration UI.
- Run their custom configuration logic, such as connecting to Salesforce to retrieve the list of objects.
- Get the Resource properties, such as the credentials to connect to a server.
- Update the data object with values configured by the user.
- Define which schema should be opened (JSON, XML or flat file).
- Define if the configuration has changed and needs to be saved (dirty indication in the Studio).
- Identify which property should be focused on as a result of a Move-to request from one of the built-in utilities (Text Search, Checker, or Cross Reference).

configure() – return bool value behavior

- When **True** is returned:
  - It means that the configuration was completed with OK.
  - A call to **GetSchema()** will be performed and the Data Mapper will be opened with the provided schema.
  - If the **configure()** method returned True and **configurationChanged** is set to **True**, the Data Mapper will open in a dirty state to indicate that it needs to be saved.
- When **False** is returned:
  - It means that the configuration was completed with **Cancel**.
  - The method will exit without opening the Data Mapper and without saving any changes made to the data class.

configurationChanged behavior

As mentioned above, this property controls the dirty indication of the connector. If **configurationChanged** is set to True it means that something was changed.

In order to get to a dirty state, the **configure()** method must return **True**.

GetSchema() method

This method is called after the **configure()** method returns **True.** This method should return a **SchemaInfo** class object. There are three derived implementations for the **SchemaInfo** class:

- **XMLSchemaInfo** – Represents an XML Data Mapper destination and has dedicated properties to define the XSD, encoding, root element, and so on.
- **FlatFileSchemaInfo** – Represents a Flat File Data Mapper destination and has dedicated properties to define the delimiter, the encoding and the actual flat file structure.
- **JSonSchemaInfo** – Represents a JSON Data Mapper destination and has dedicated properties to define the JSON schema, encoding, and so on.

Resource-related methods

A Dynamic step can be configured with a Resource. The Resources are defined in the Connector Builder utility using a dedicated builder.

In addition to the standard Resource fields, it is possible to define a validation button and three action buttons.

The validation and action buttons are implemented using the following methods:

- **public bool ValidateResource**(IReadOnlyResourceConfiguration resouceData, out string errorMsg) – This method is called when the Validate button is clicked in the Resource. The method has a read-only copy of the Resource properties' object and should return True for valid or False with some error message for not valid.
- **public void InvokeResourceHelper**(string helperID, IResourceConfiguration resouceData) – This method is called when one of the action buttons is clicked in the Resource. The method receives the name of the clicked button and an updatable Resource properties' object. From this method the developer will usually open additional dialog boxes and update some of the Resource values. An example of how to update the resource can be found in the UI templates.

Check() method

This method is called by the Studio checker and allows the developer to add custom checker results to the connector. For example, to make sure that the user either selected a variable to hold the step result or provided a file path.

Responding to the Debug/Design Studio State

The Magic xpi Studio has two main states:

- Design state – This is the initial Studio state allowing development.
- Debug state – This is the Studio state when debugging the project. In this state, the dialog boxes may have a different behavior blocking the user from making changes to the sources of the running project.

When building the UI part of the Connector Builder, it is good practice to comply with the standard Studio dialog boxes and prevent the user from making any changes. It is usually enough to disable the OK button while still allowing the user to view the configuration.

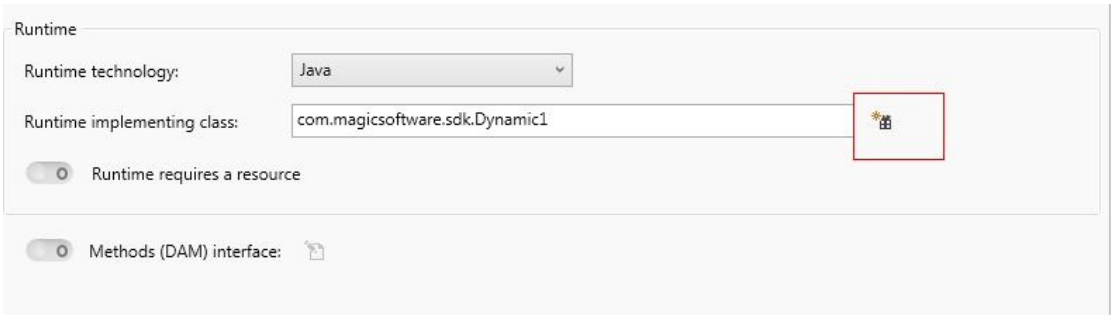
The state of the Studio is available in the **config()** method as a property of the **Utils System Properties** collection called **IsStudioInDesignMode**. This property returns a string with a value of True or False.

Accessing this property is done by adding the following line of code:

```
utils.GetSystemProperty("IsStudioInDesignMode");
```

Creating the Runtime Implementation

The easiest way to start the development of the runtime part of the connector is to generate a template project from the Connector Builder utility.



Use the **Generate Runtime Project** button next to your defined runtime class to open the utility. In the utility you will be asked for the .NET or Java or Magic xpa project name and its location. The namespace or package name for this project will be taken from the **Runtime implementing class** property that you filled in previously.

In this example we will use a **Java** implementation.

1. In Eclipse, create a new Java project.
2. Give the project a name.
3. Select the **Create project from existing source** radio button.

4. Select the path to the project that you generated using the **Generate runtime Project** utility.
5. Click the **Finish** button to finish creating the project.
6. Navigate to your project in Eclipse and open your class (the name you provided in the Connector Builder).

While compiling the jar file, make sure to use Java 1.7 compiler which is compatible with Magic xpi.

Your class implements the IStep interface and contains a single method called **invoke()**.

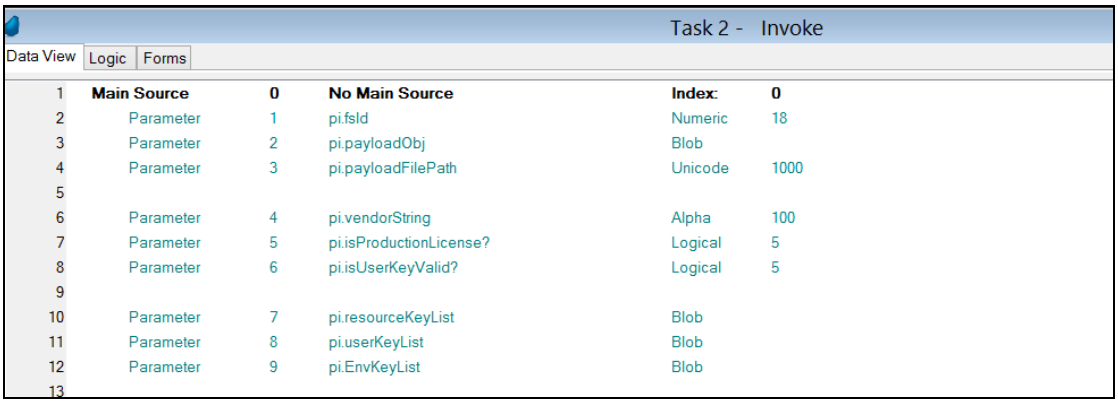
The object of type **StepGeneralParams** provided in the **invoke()** method contains the following:

a. The Data Mapper payload: Use **getPayloadOBject()** or **getPayloadFile()** to get a byte array of the payload. The location of the payload (File or BLOB) is controlled from the schema UI configurations' .NET code: xmlSchemaInfoLocal.DataDestinationType = 0/1 (0=Blob and 1=file)
b. The properties defined in the UI (Data Class) using **getUserProperties().get("property name");**
c. Once you get a specific user property you can:
    i. Get its value using: **getValue().toString()**
    ii. Set a new value (if it was defined as Variable Out type in the data class):
        - setAlpha(String value)
        - setBlob(Byte [] args)
        - setDate(Date d)
        - setLogical(Boolean b)
        - setNumeric(Double d)
        - setTime(String t)
d. Get the Resource properties: Using **getResourceObject()**
e. Get the license details.
f. Get an indication of license validity if the connector is licensed.
g. Get the environment settings, such as the encoding defined in the Magic.ini file, project location, installation location, connector location and so on.

In this example we will use an **xpa** implementation.

Open the xpa studio. In that open the xpa solution which is generated by the Connector Builder.

There will be a program named Invoke with the following parameters.



Task 2 - Invoke

Data View | Logic | Forms

1	**Main Source**	**0**	**No Main Source**	**Index:**	**0**
2	Parameter	1	pi.fsId	Numeric	18
3	Parameter	2	pi.payloadObj	Blob	
4	Parameter	3	pi.payloadFilePath	Unicode	1000
5					
6	Parameter	4	pi.vendorString	Alpha	100
7	Parameter	5	pi.isProductionLicense?	Logical	5
8	Parameter	6	pi.isUserKeyValid?	Logical	5
9					
10	Parameter	7	pi.resourceKeyList	Blob	
11	Parameter	8	pi.userKeyList	Blob	
12	Parameter	9	pi.EnvKeyList	Blob	
13					

The second parameter named as **pi.payloadObj** contains the Data Mapper payload and the third parameter named as **pi.payloadFilePath** contains the location of the payload.

The following functions can be used to set and get the user properties.

- **getUserPropertyValueByKey(PropertyName)**
  This function returns the value of a user property.
- **setUserPropValueA(PropertyName,Value)**
  This function sets the value for the user property of the Alpha type.
- **setUserPropValueN(PropertyName,Value)**
  This function sets the value for the user property of the Numeric type.
- **setUserPropValueD(PropertyName,Value)**
  This function sets the value for the user property of the Date type.
- **setUserPropValueT(PropertyName,Value)**
  This function sets the value for the user property of the Time type.
- **setUserPropValueL(PropertyName,Value)**
  This function sets the value for the user property of the Logical type.
- **setUserPropValueB(PropertyName,Value)**
  This function sets the value for the user property of the Blob type.

The following function can be used to get the Resource Property values:

getResourceValueByKey(ResourceParameterName)

The **pi.vendorString**, **pi.isProductionLicense?** and **pi.isuserkeyValid?** parameters provide the information on the license details.

The following function can be used to get the environment settings:

getEnvSettings(EnvPorperty)

The **pi.resourceKeyList** parameter provides the resource parameter names. The **pi.userKeyList** parameter providers the user properties. The **pi.envKeyList** parameter provides the environment details. Each of the parameter has a property value delimited by a comma.

Writing a Trigger

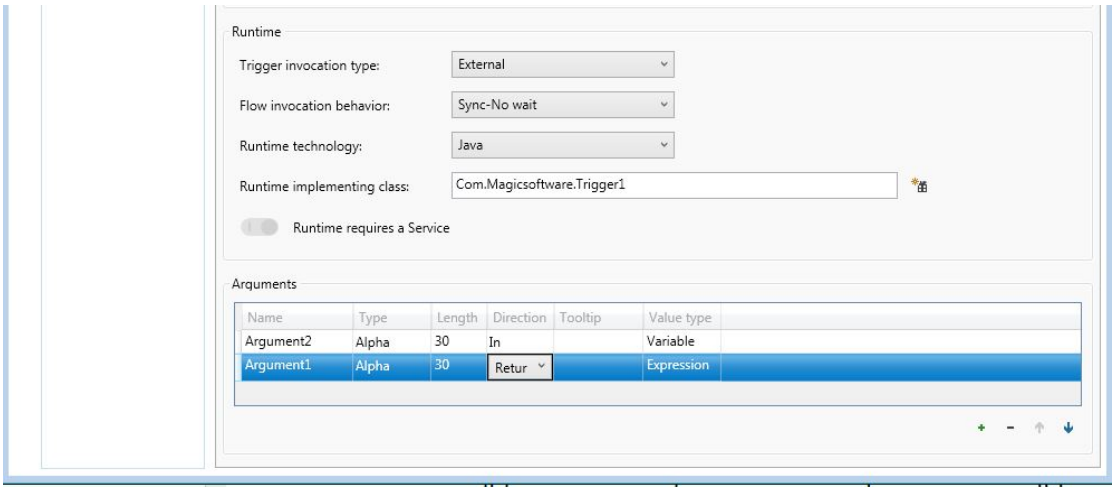
Writing an External Trigger with a Static UI

In this example we will write an external trigger with a Static UI.

External trigger means that the flow invocation is controlled by the trigger itself and not by any polling mechanism. An external trigger is suitable for implementations that contain a callback. For example, a messaging infrastructure that once it receives a message will invoke a callback method and the flow will be invoked from this method.

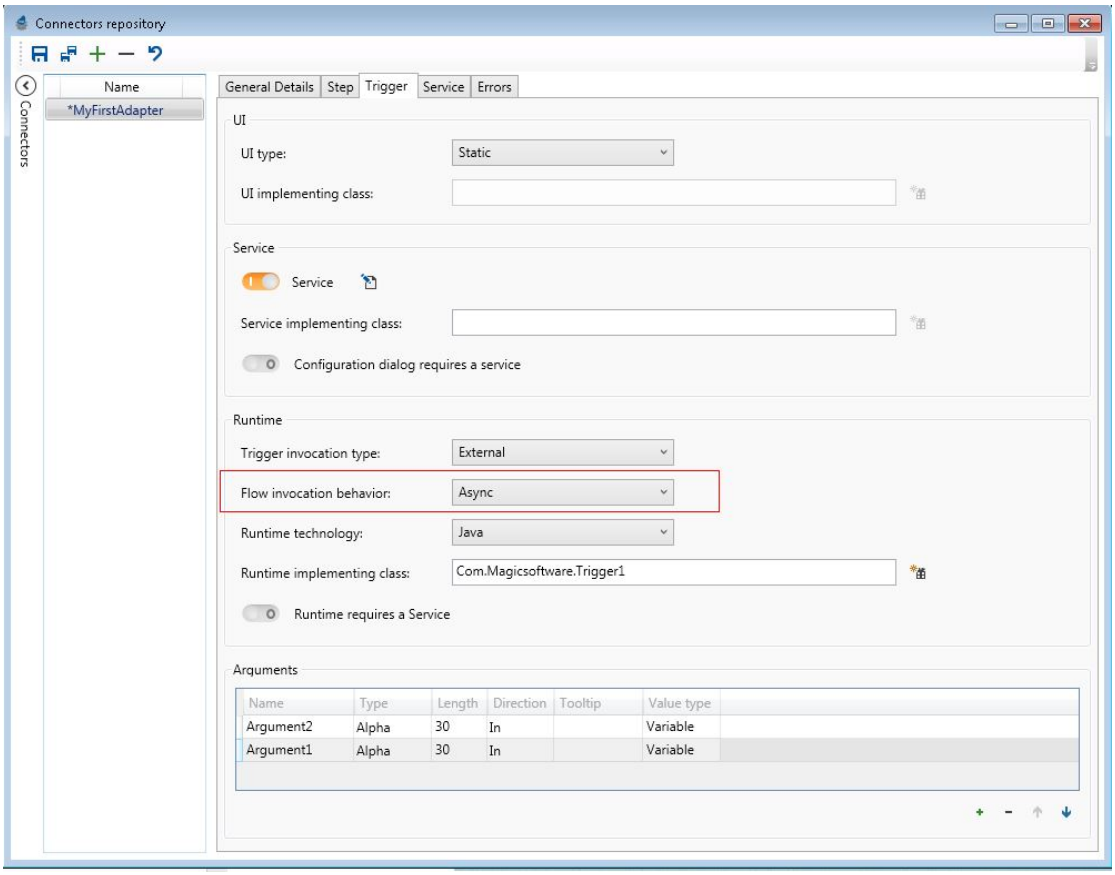
Static UI means that the UI is a simple table configured from the Connector Builder with the following possible directions:

1. **In** – Represents properties that are passed from the trigger runtime code to the flow. These properties are defined as a picklist type since they must be mapped to variables in the invoked flow.
2. **Return** – Represents the value returning from the flow and passed back to the trigger. Only a single return is possible when using the static UI and its type is either a variable or an expression.



Later on when we discuss the dynamic UI we will see that it is possible to pass another type of property in the data class used solely for configuration. This type of variable will have a special annotation and at runtime will be evaluated once when the **load()** method is called and will never be passed to the flow.

Trigger Settings



Note the **Flow invocation behavior** combo box. For an external trigger the following invocation types are available:

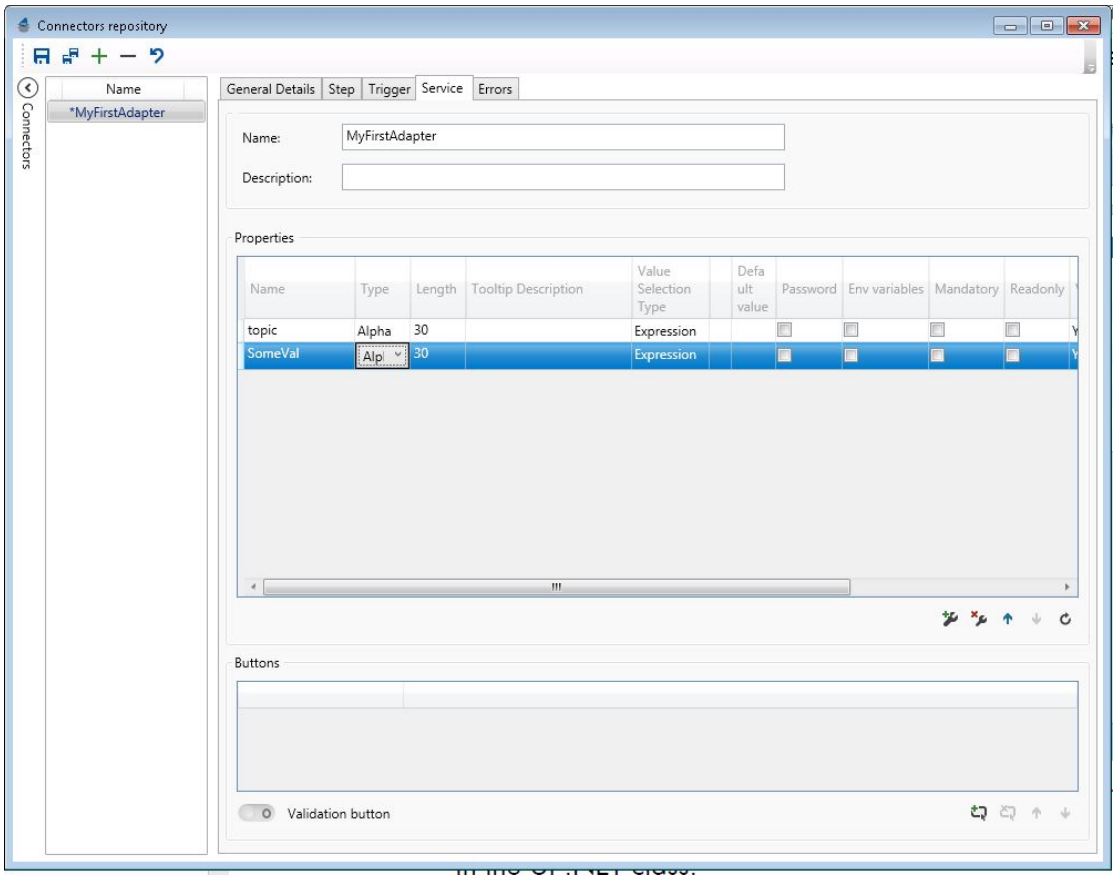
- **Sync-wait** – The flow invocation is synchronous and the message will wait if the flow is within its defined Max instance value.
- **Sync-no wait** – The flow invocation is synchronous but the message will not wait if the flow is within its defined Max instance and will return immediately with an error.
- **Async** – The flow invocation is asynchronous. The invoke method will return once the message is in the space and will not wait for the flow to return a result.

Sync invocation is suitable for triggers with a return value, such as TCP triggers.

Async invocation is suitable for triggers with no return value, such as messaging triggers with callback methods.

Service Settings

A trigger can be configured with a Service. The Service definitions are made in the Connector Builder.



In addition to the standard Service fields, it is possible to define a validation button and three action buttons. The implementation of the validation and the action buttons is in the UI .NET class.

For a trigger with a Dynamic interface, the implementation methods are already part of the **IUserTriggerComponent** interface.

For a trigger with a Static interface, the user will have to implement a dedicated interface containing only the Service methods. The class name implementing this interface is defined in the Connector Builder's **Service implementing class** property.

The methods are:

- public bool ValidateService(IReadOnlyServiceConfiguration serviceData, out string errorMsg) – This method will be called once the Validate button is clicked in the Service. The method has a read-only copy of the Service properties' object and can only return True for valid or False with some error message for not valid.

- public void InvokeServiceHelper(string helperID, IServiceConfiguration serviceData) – This method will be called once one of the action buttons is clicked in the Service. The method receives the name of the clicked button and an updatable Service properties' object. From this method the developer will usually open additional dialog boxes and update some Service values.

External Trigger – Dynamic UI

Implementing the UI code is very similar to the dynamic step with the differences listed below.

Data Class

[TriggerIn] – Use this annotation to represent values going from the trigger runtime logic and into the flow. The property type must be a Variable.

[TriggerReturn] – Use this annotation to represent a value returning from the flow back to the trigger. This type of property can be either a Variable or an Expression.

[UseForConfiguration] – Use this annotation to indicate that a specific property is used only for configuration. These properties are passed to the trigger at load time (to the **load()** method) and will not be passed to the flow itself.

Adaptor Class

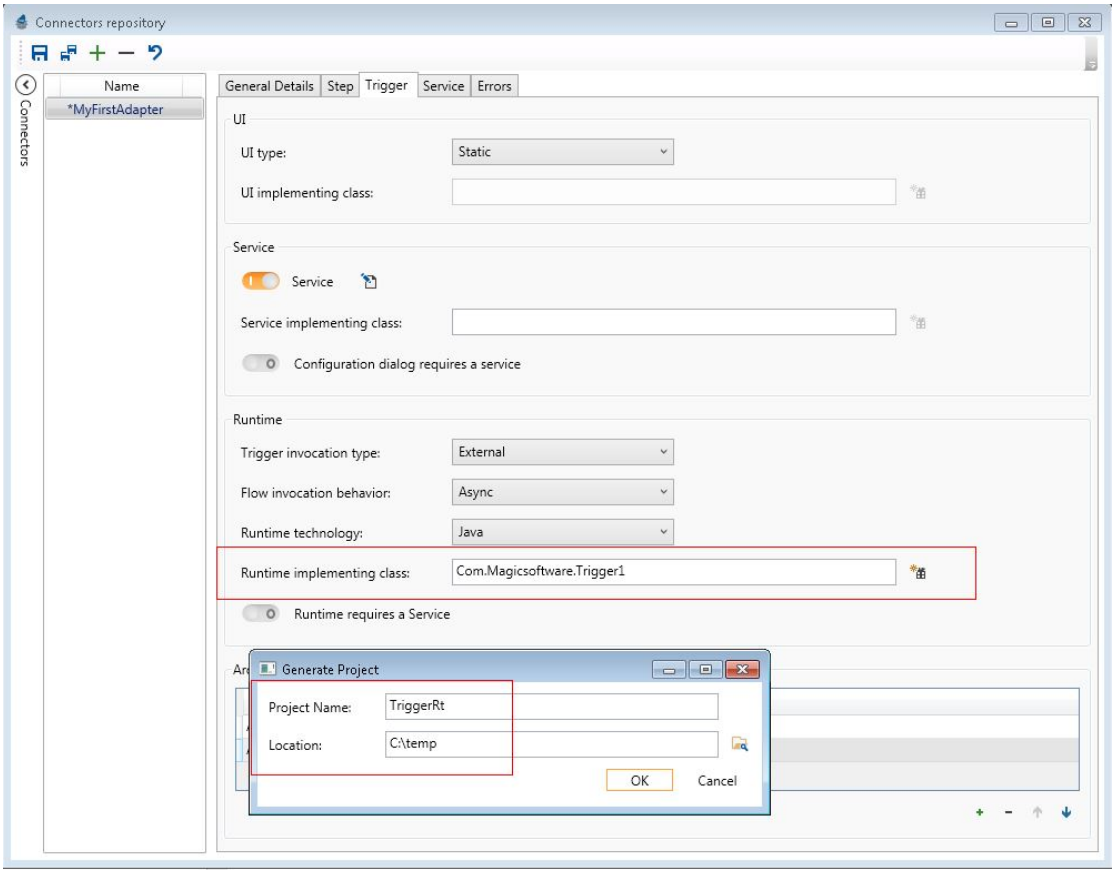
1. The class should implement and export the **IUserTriggerComponent** interface.
2. The Configure method is very similar to the Dynamic step but contains the Service object instead of the Resource object.
3. There are no schema-related methods, since triggers do not use the Data Mapper.

Example implementation

```
using System;
using System.Windows.Forms;
using MagicSoftware.Integration.UserComponents.Interfaces;
using MagicSoftware.Integration.UserComponents;
using System.ComponentModel.Composition;
using DynamicTrigger1;

namespace DynamicTrigger1
{
    [Export(typeof(IUserTriggerComponent))]
     class TriggerAdapter  : IUserTriggerComponent
    {
        public TriggerAdapter()
        {
        }

        #region IUserTriggerComponent implementation
        public object CreateDataObject()
        {
            return new TriggerDataClass();
        }
        public bool? Configure(ref object dataObject, ISDKStudioUtils utils, IReadOnlyServiceConfiguration
serviceData, object NavigateTo, out bool configurationChanged)
        {
            configurationChanged=true;
             TriggerDataClass triggerData = new TriggerDataClass();
            if (dataObject is TriggerDataClass)
                triggerData = (dataObject as TriggerDataClass);
            else
                dataObject = triggerData; // Set the reference to a new instance of the data class
            trigger2 dialog = new trigger2(triggerData,utils); // Open UI form
            DialogResult dr= dialog.ShowDialog();

            if(dr.Equals(DialogResult.OK))
            {
                configurationChanged=dialog.configHasChanged();
                return true;
            }

            return false;

        }
        public ICheckerResult Check(ref object data, IReadOnlyServiceConfiguration serviceData)
        {
            return null;
        }
        public bool ValidateService(IReadOnlyServiceConfiguration serviceData, out string errorMsg)
        {
            errorMsg="";
            return true;
        }
        public void InvokeServiceHelper(string helperID, IServiceConfiguration serviceData)
        {

        }
        #endregion
    }
}
```

External Trigger – Runtime

The easiest way to start the development of the runtime part of the trigger is to generate a template project from the Connector Builder.



Use the **Generate runtime Project** button next to your defined runtime class to open the utility. In the utility you will be asked for the .NET or Java or Magic xpa project name and its location. The namespace or package name for this project will be taken from the **Runtime implementing class** property that you filled in previously.

In this example we will use a **Java** implementation.

1. In Eclipse, create a new Java project.
2. Give the project a name.
3. Select the **Create project from existing source** radio button.
4. Select the path to the project that you generated using the **Generate runtime Project** utility.
5. Click **Finish** to finish creating the project.

6. Navigate to your project in Eclipse and open your class (the name you provided in the Connector Builder).
7. Your class implements the **IExternalTrigger** interface and contains the following methods:

    - **load(TriggerGeneralParams triggergeneralparams, FlowLauncher flowlauncher)** – This method is the main entry point to the trigger. The method is called only one when loading the trigger. The **FlowLauncher** object holds the logic that allows the developer to invoke the flow. The **TriggerGeneralParams** object holds the Service, Resource and other settings passed to the trigger. From that point on, users must implement their own logic in a new thread, so that Magic xpi server can call the disable() and enable() methods when needed.
    - **disable()** – Called when the flow is disabled. It is the developer's responsibility to implement the trigger's disable logic.
    - **enable()** – Called when the flow is enabled. It is the developer's responsibility to implement the trigger's enable logic.
    - **Unload()** –Called when the engine is shutting down.

*** When invoking the flow using the FlowLauncher object, the invocation timeout is set by default to 90 seconds. It is possible to get/set the timeout by using the following methods of the **FlowLauncher** class:

public void setTimeOut(int timeoutSec)

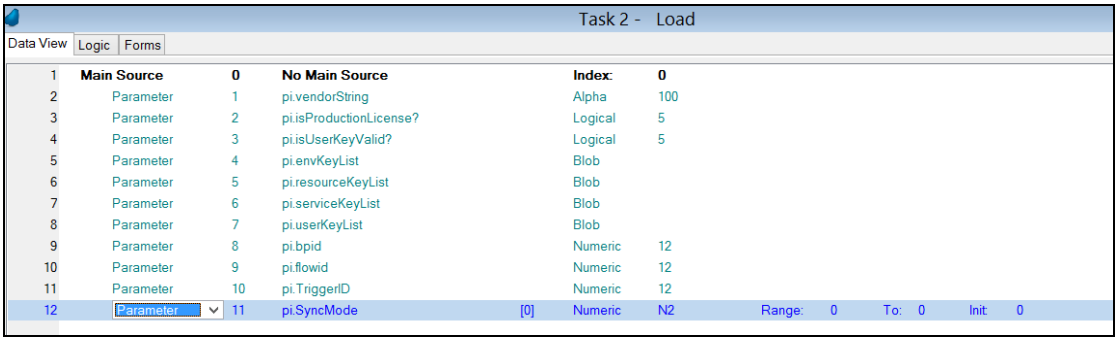
public int getTimeOut()

Setting the timeout to 0 means no timeout.

In this example we will use an **xpa** implementation.

Open the xpa studio. In that open the xpa solution which is generated by the Connector Builder.

There will be a program named Load. This program is the main entry point for the trigger. It is called only once when loading the trigger and provides the following parameters.



The following functions can be used to get and set the user and the resource properties:

- **getUserPropertyValueByKey(PropertyName)**
- **setUserPropValueA(PropertyName,Value)**
- **setUserPropValueN(PropertyName,Value)**
- **setUserPropValueD(PropertyName,Value)**
- **setUserPropValueT(PropertyName,Value)**
- **setUserPropValueL(PropertyName,Value)**
- **setUserPropValueB(PropertyName,Value)**
- **getResourceValueByKey(ResourceParameterName)**

The **pi.envKeyList** parameter provides the environment details. The **pi.resourceKeyList** parameter provides the resource parameter names. The **pi.serviceKeyList** parameter provides the service parameter names. The **pi.userKeyList** parameter provides the user properties. Each of the parameter has a property value delimited by a comma.

From here onwards, the users can implement their custom logic.

To map the incoming arguments to the mapped variables, user must invoke the event – **'MapInArguments'**.

To invoke the Flow, user must use the event **'TriggerInvokeFlow'**.

To map the return argument, user must invoke the event – **'MapReturnArgument'**.

For more information, refer the Call Program sample project.

Polling Trigger

A polling trigger is a trigger that explicitly polls external systems for data in a predefined interval and if data is found, invokes the flow. A few examples of polling triggers are an Email trigger that checks for new email every few seconds and the Directory Scanner trigger that scans a folder every interval.

UI

The UI part of the trigger is very similar to the one in the External trigger. For a dynamic interface, the same **IUserTriggerComponent** interface is used. The only difference is the type of properties that a trigger may use. A polling trigger is always asynchronous. There is no client waiting for a flow to return a result. Therefore, there are only two types of properties that can be defined for a Dynamic interface in the data class:

[TriggerIn] – Values passed from the trigger code and into the flow.

[UseForConfiguration] – Values passed to the load() method and are never passed to the flow.

For the static UI, only a picklist type can be selected since the value passed to the flow must be mapped to a variable.

Runtime

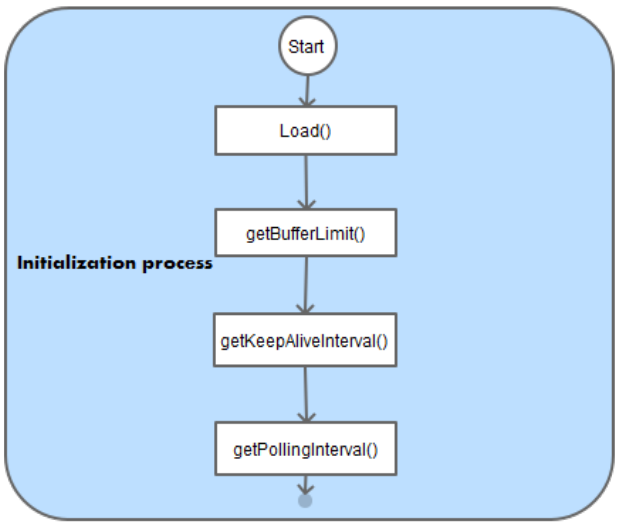
The runtime part of a polling trigger is a bit more complicated compared to an external trigger. In a polling trigger, the Magic xpi engine is responsible for the trigger invocation in a predefined interval, for enforcing the buffer size and for the keep alive interval.

The following are the features available for the polling trigger:

1. Control the polling interval of the trigger – The server will call the trigger in this interval.
2. Control the keep alive interval of the trigger – The trigger will be restarted if it did not respond.
3. Control the buffer – The server will not call the trigger if the amount of messages created by the trigger exceeded the buffer size.
4. Allows to split a payload to multiple flow invocations – to cope with very large payloads.

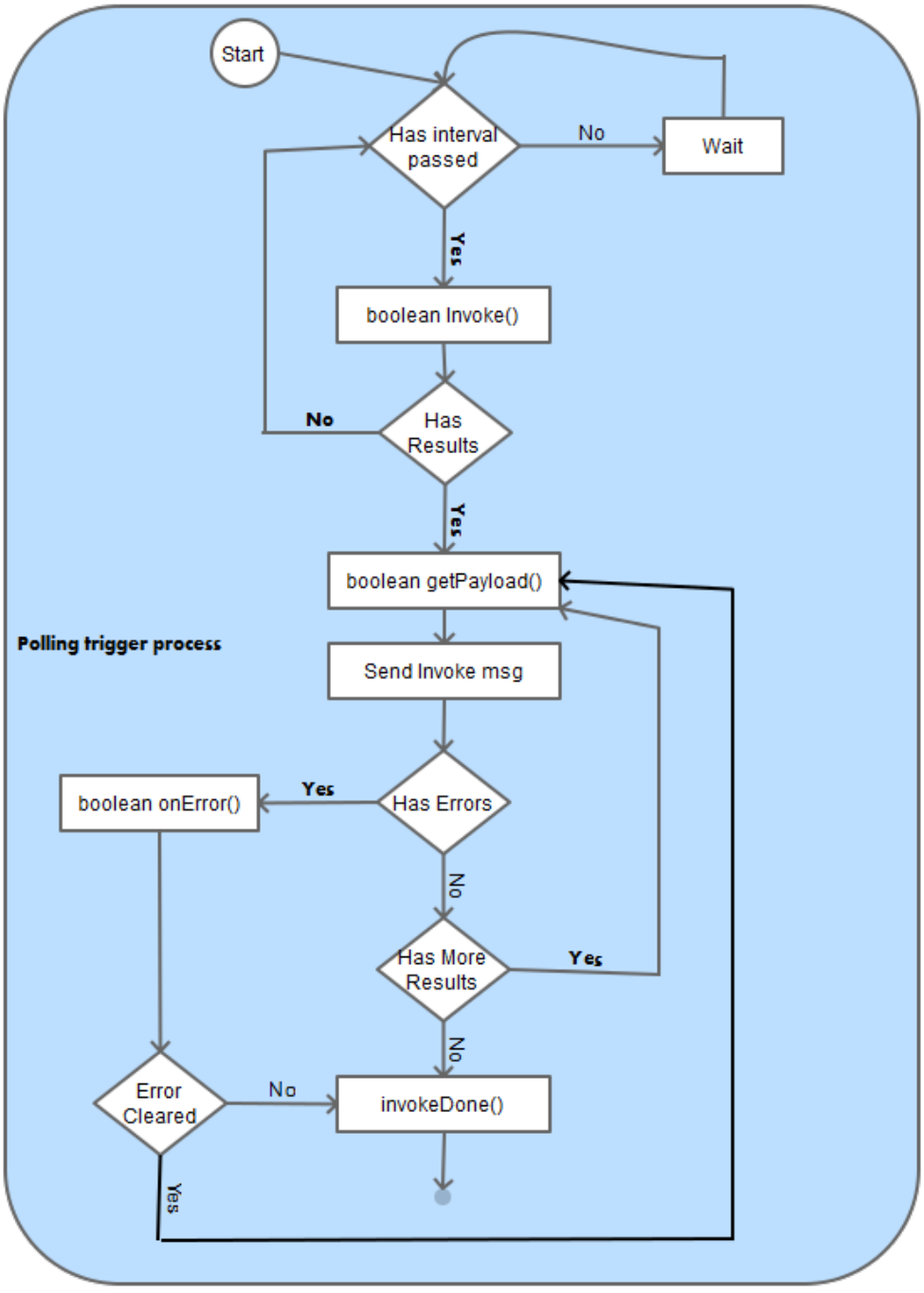
Initialization

This part is responsible for calling the **Load()** method with a configuration object and in sequence call the **getBufferLimit()**, **getKeepAliveInterval()** and **getPollingInterval()**. The developer must provide implementation for those methods.



Polling Sequence

In this part, the invoke() method will be called in each interval. The flow is explained in the following chart:



boolean Invoke() – From this method the developer should access the target system and poll for results. If this method returns **True** it indicates that the invocation ended with results (for example, new emails were found). If the method returns **False** it indicates that there are no results (for example, no new emails were found).

boolean getPayload(HashMap<String, Object> payload) – This method will be called once the Invoke() method returns **True**. The **payload** object is passed by ref and the developer must populate it with the arguments that have to be passed to the flow.

The **getPayload()** methods allows the developer to implement a split of the payload.

If the method return **False**, a flow request message will be sent to the flow and this cycle of invocation will end. At the end of the cycle, the server will call the **invokeDone()** method to allow the developer to release the cached results.

If the method returns **True** it indicates that there are more results and the payload was split. The server will then send a flow request message and call the **getPayload()** again to get the next split.

If there is an error from the **getPayload()** method, the **onError()** method will be called by the server. If the result of this method is **True,** it means that the error will be ignored and the **getPayload()** method will be called again to get the next split. If the result is **False** the server will call the **invokeDone()** method.

Endpoint Trigger

An Endpoint trigger is a new type of trigger introduced in Magic xpi 4.6. The Endpoint Trigger is unique across all the projects and is unique in a single space. This type of trigger has its lifecycle managed external to Magic xpi.

As an example for such a trigger we can take the HTTP trigger. This trigger is managed by the IIS and when a request arrives, the prgname, appname trigger name and endpoint name makes the unique connection between the arriving request and the specific project/flow/trigger combination that should handle it.

For this new type of trigger, Magic xpi provides an API client library that allows the developer of the Endpoint trigger to:

1. Check the space for an available license feature.
2. Invoke a flow by providing the unique endpoint ID value and name/value pairs (Invocation can be sync or async).

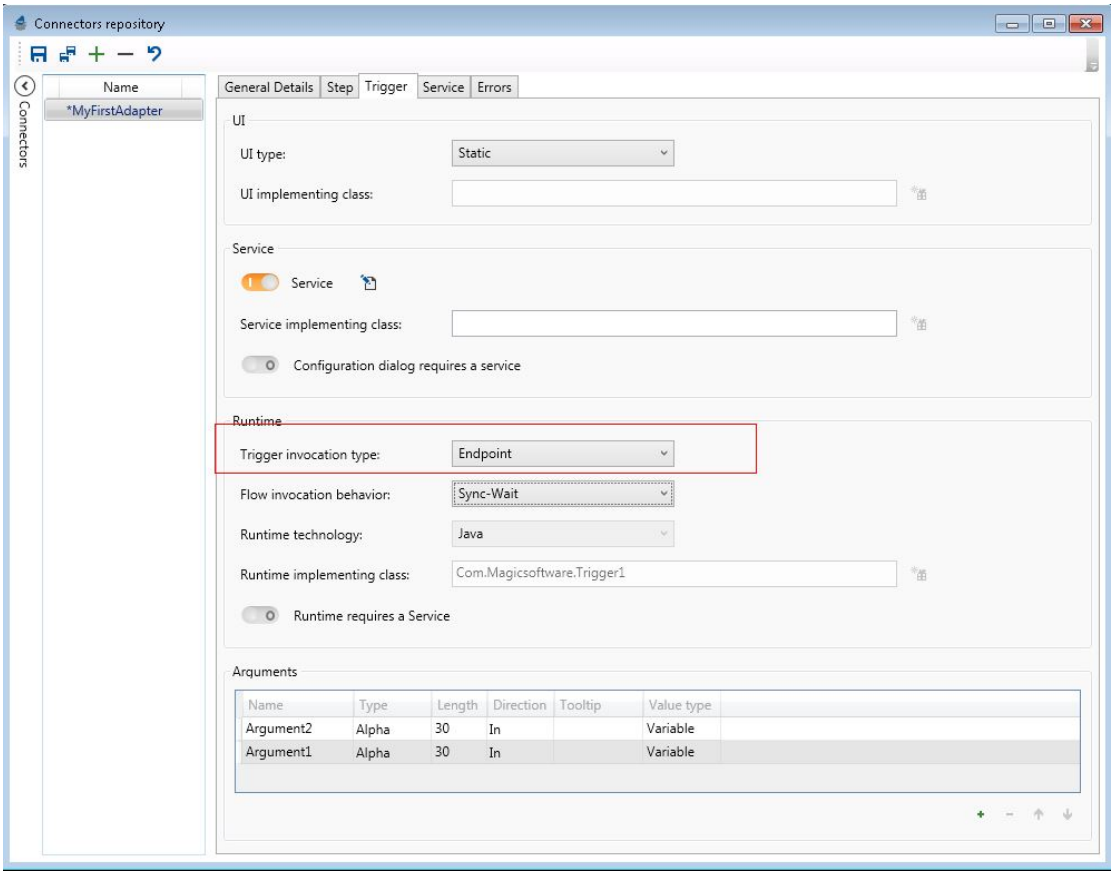
When using the Endpoint trigger, a unique ID should be used to match the request to the specific project/flow/trigger combination. This unique ID is called the Endpoint.

Connector Builder Setting

From the Connector Builder's **Triggers** tab you can set the **Trigger invocation type** to **Endpoint**.

This type of trigger supports both a Dynamic UI and a Static UI.

An Endpoint trigger has no runtime code that is managed by Magic xpi and as a result there is no option to provide a Runtime implementing class.



Static UI

Static UI for the Endpoint trigger is similar to the static UI of the External trigger except for the additional endpoint field. When dragging the trigger to the flow's trigger area, the developer must provide a unique endpoint that will identify this trigger instance at runtime. The same endpoint must be sent when invoking the flow from the endpoint trigger.

Dynamic UI

For the dynamic UI, the code is similar to the code of the External or Polling trigger. The only difference is the addition of the endpoint that must be defined in the DataClass.

First, in the class level an annotation should be used to declare which property is holding the endpoint value:

```
i
[EndPointProperty("MyTriggerEndpoint")]
public class MyData
{
```

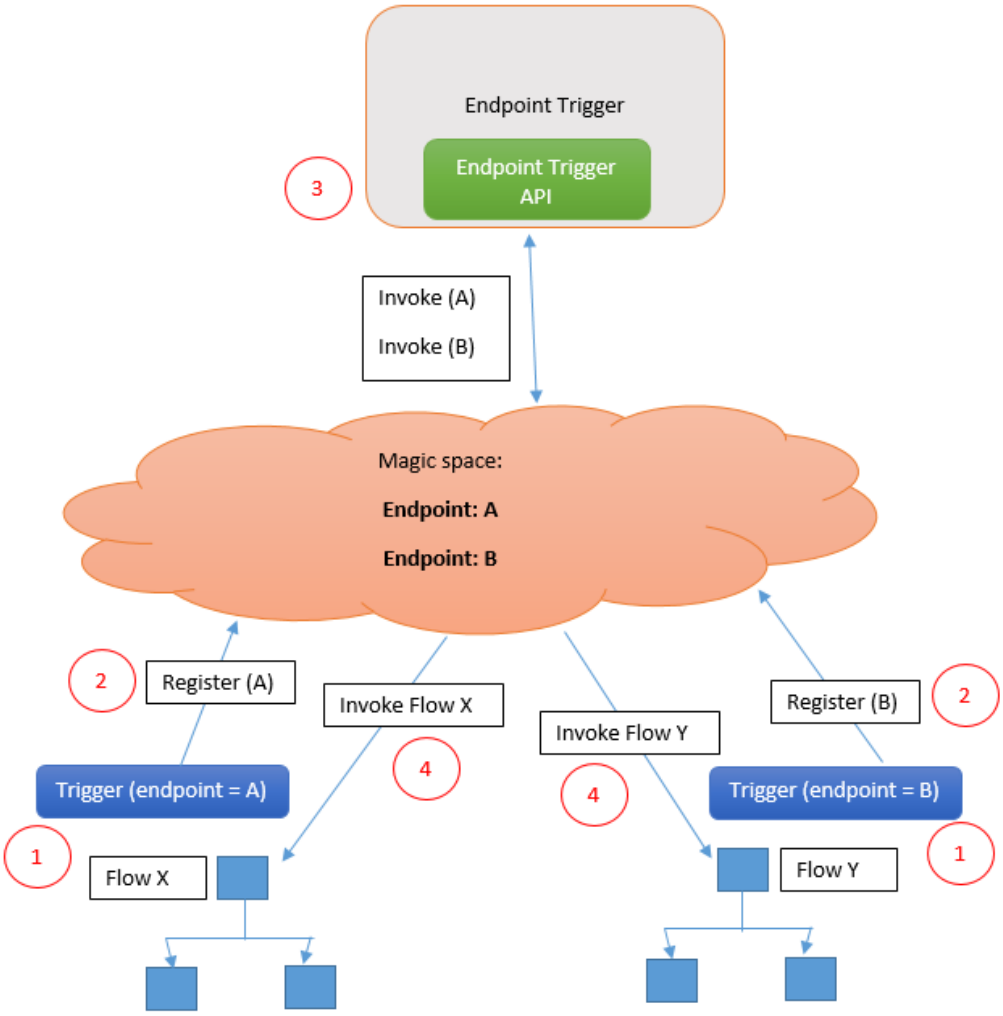
Next, this property must be defined as one of the class members. The property should be an IN property of type Alpha:

```
//This TriggerEndpoint property is only relevant for Endpoint triggers. The property must hold the unique endpoint of the trigger
 [Id(4)]
 public Alpha MyTriggerEndpoint { get; set; }
```

This endpoint property must have a value when the user configures the trigger in the studio. A checker error will be raised if it does not.

The template that can be generated from the Connector Builder utility already contains the required properties for the Endpoint trigger.

Operation Diagram



1. Design time – A trigger is dragged to the flow and a unique endpoint is defined.

2. Runtime – When the project starts, all Endpoint triggers are registered in the space with their unique endpoint value and the addressing of the project/BP/Flow/TriggerID.

3. Runtime – The trigger calls the invoke method and passes the unique endpoint ID and any arguments that the trigger expects. The result of this operation is an endpoint trigger object in the space.

4. Runtime – A processing unit (PU) in the space attempts to match between the endpoint defined in the new object to the registered endpoints. If a match is found, the PU creates a Flow Request message with the addressing of the specific project/BP/Flow/Trigger ID and with the properties sent by the trigger. A worker belonging to this project takes the Flow Request message and runs the flow.

Using the Magic xpi Endpoint Trigger API

Magic xpi provides a client library to be used by the Endpoint trigger implementation to invoke the trigger instances.

The **Standalone Invoker.zip** file containing all of the necessary files is available in the **Runtime\Support** folder.

When using the API, the folder structure must be kept to allow the API to locate all references.

The **mgreq.ini** file should be configured with the space's **LookupLocators** and **LookupGroup**.

Trigger code example

The following sample Java code shows an invocation of the **Endpoint_Unique_Id** endpoint. In addition, the code shows how the trigger can check if a specific license was loaded (feature MYTRG) and if the server license is a production license.

When configuring the IDE, all jars from the **Java** and **lib** folder available in the **Standalone Invoker.zip** file should be added to the classpath. In addition, a **log4j** jar file should be added. The **log4j** jar file is not provided in the zip.

```
package com.magicsoftware.pm.TriggerXpi;

import java.io.IOException;
import java.util.HashMap;
import com.magicsoftware.xpi.sdk.standalone.MagicxpiProxy;
import com.magicsoftware.xpi.sdk.standalone.MagicxpiProxy.ConnectionStatus;
import com.magicsoftware.xpi.sdk.standalone.license.ComponentLicense;
import com.magicsoftware.xpi.server.messages.StandaloneResponse;

public class Trigger1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		try {

			// create the proxy instance with connection properties from the
			// provided mgreq.ini
			MagicxpiProxy magicxpiproxy = new MagicxpiProxy(
					"C:/temp/mgreq.ini");

			// connect the proxy to the space
			ConnectionStatus connectionstatus = magicxpiproxy.Connect();
			if (connectionstatus.equals(ConnectionStatus.CONNECTED)) {
				// check if Magic xpi was loaded with the provided license
				// feature
				ComponentLicense componentlicense = magicxpiproxy
						.getLicense("MYTRG");
				if (componentlicense != null
						&& componentlicense.isValidLicense()) {
					System.out.println("Vendor String = "
							+ componentlicense.getVendorString());
					System.out.println("is production license = "
							+ componentlicense.isProduction());
					System.out.println("Is Valid license = "
							+ componentlicense.isValidLicense());
				}
				// building the payload that will be sent when triggering the
				// flow
				// The keys and values should match the properties defined for
				// the Endpoint trigger
				HashMap args = new HashMap();
				args.put("inputvalue", "My Input value");
				// double d = 2D;
				// args.put("_InNumeric", Double.valueOf(d));

				// Calling the trigger with the trigger's unique endpoint and
				// the trigger payload
				// the invoke operation timeout is taken from the mgreq.ini
				StandaloneResponse standaloneresponse = magicxpiproxy.invoke(
						"Endpoint_Unique_Id", args);
				if (standaloneresponse != null) {
					// Getting the variables returning from the flow - only
					// relevant for sync invocation
					HashMap responseVars = standaloneresponse.getVariables();
					String result1 = new String((byte[]) responseVars
							.get("resultvalue"));
					System.out.println(result1);
				}

			} else {
				// connection has failed
			}
		} catch (IOException ioexception) {
			ioexception.printStackTrace();
		} catch (Exception exception) {
			exception.printStackTrace();
		}
	}
}
```

Appendix A – Manually Creating the Step UI Project

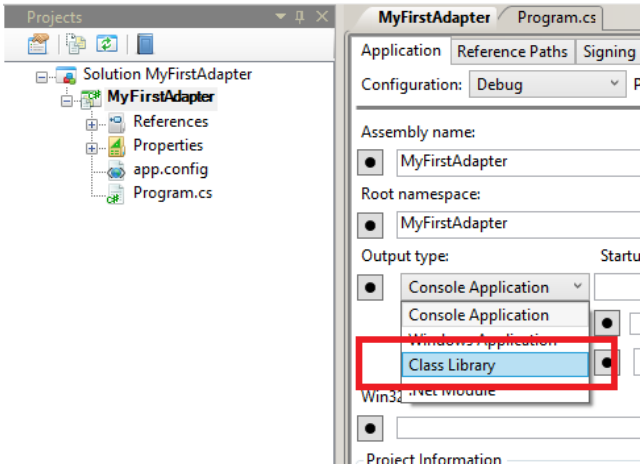
The instructions below are the manual steps needed to create and configure the IDE for the C# project with all of its references. An alternative and recommended approach is to use the **Generate UI project** utility found next to the UI class name.

Manual Steps

We will use SharpDevelop as our C# IDE. You can download it from here:

http://www.icsharpcode.net/OpenSource/SD/Download/

1. Open SharpDevelop and create a new solution named **SDK_TEST_1**.
2. Select **Console Application** as the template and click **OK**.
3. When the project is created, open the project properties and change the **Output type** to **Class Library**.



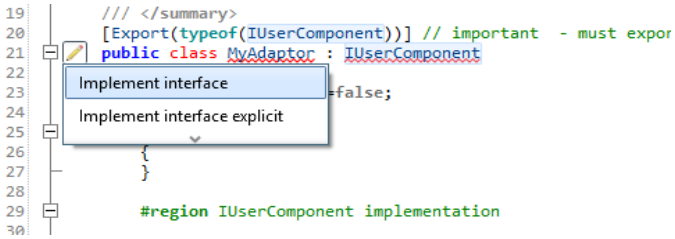
4. Delete the existing **program.cs**.
5. Add a new class named **MyStepAdapter**.

Now you will add references:

6. From the **GAC** tab add:
   a. System.Data.DataSetExtensions
   b. System.ComponentModel.Composition
7. From the **.NET Assembly Browser** tab:
   a. Browse for **<Magic xpi root>\Studio\Extensions\Application**.
   b. Select **MagicSoftware.Integration.UserComponents.dll**.

8. Implement the IUserComponent interface (public class SDK_TEST_1 : IUserComponent

9. Export the class: [Export(typeof(IUserComponent))].
10. Add the following imports:
    a. using MagicSoftware.Integration.UserComponents.Interfaces;
    b. using MagicSoftware.Integration.UserComponents;
    c. using System.ComponentModel.Composition;
11. Implement the interface methods. (In the IDE, there is a shortcut to create an empty skeleton of all the required methods.)



12. Make sure that you do not leave the un-implemented exception call created automatically in the various methods. Leaving this call behind will result in studio exceptions.

About Magic Software Enterprises

Magic Software Enterprises (NASDAQ: MGIC) empowers customers and partners around the globe with smarter technology that provides a multi-channel user experience of enterprise logic and data.

We draw on 30 years of experience, millions of installations worldwide, and strategic alliances with global IT leaders, including IBM, Microsoft, Oracle, Salesforce.com, and SAP, to enable our customers to seamlessly adopt new technologies and maximize business opportunities.

For more information, visit www.magicsoftware.com.