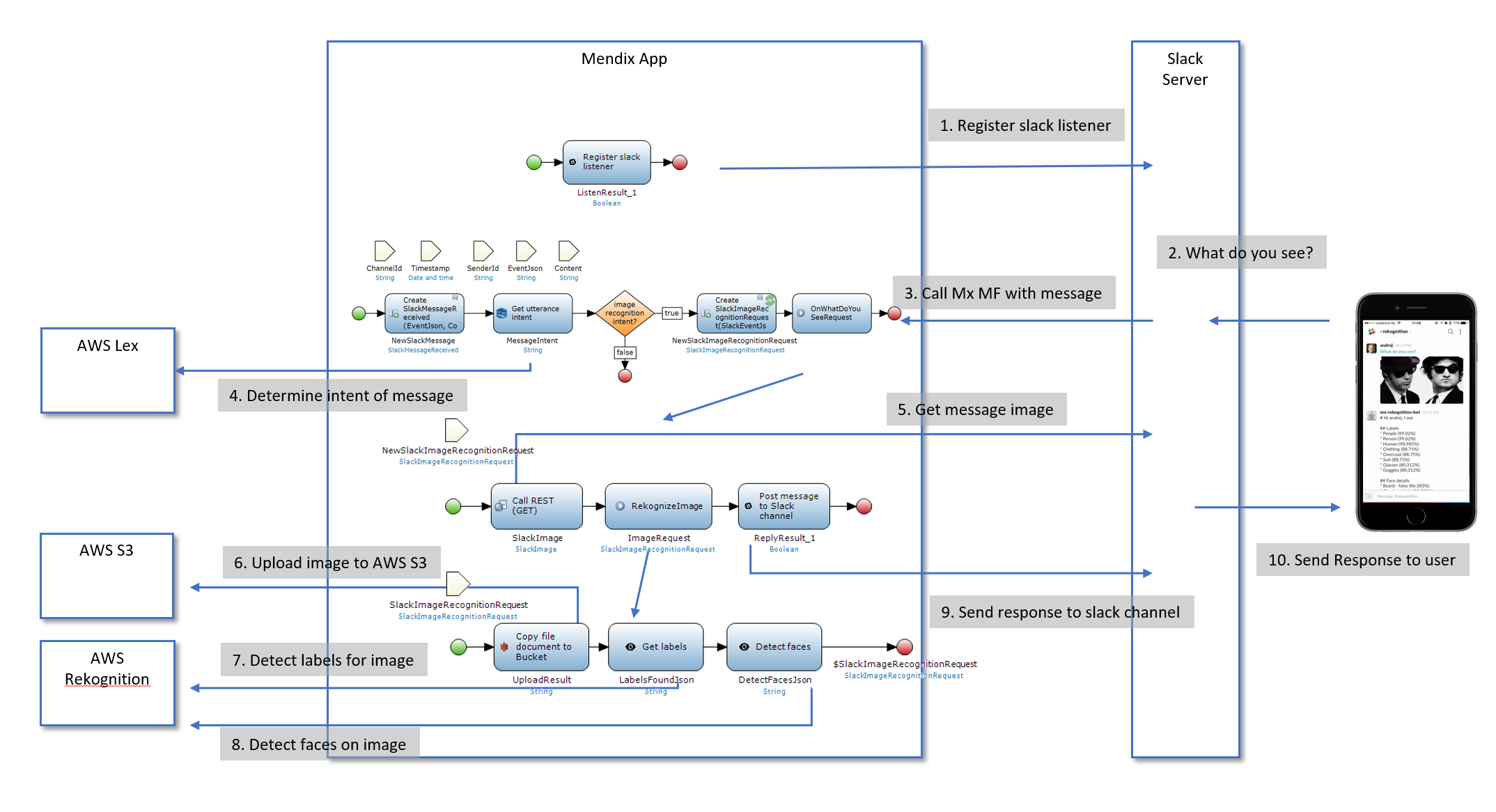
# Building Microflow actions using the Mendix Connector Kit

Last year with Mendix 6.6 we introduced [the Connector Kit](https://www.mendix.com/blog/introducing-mendix-connector-kit/). The goal of this enhancement is to enable Java developers to easily add powerful and robust new Microflow actions to their Mendix toolbox. These Microflow actions can be shared in the Mendix AppStore so anyone can benefit from these Microflow actions without having to know Java.

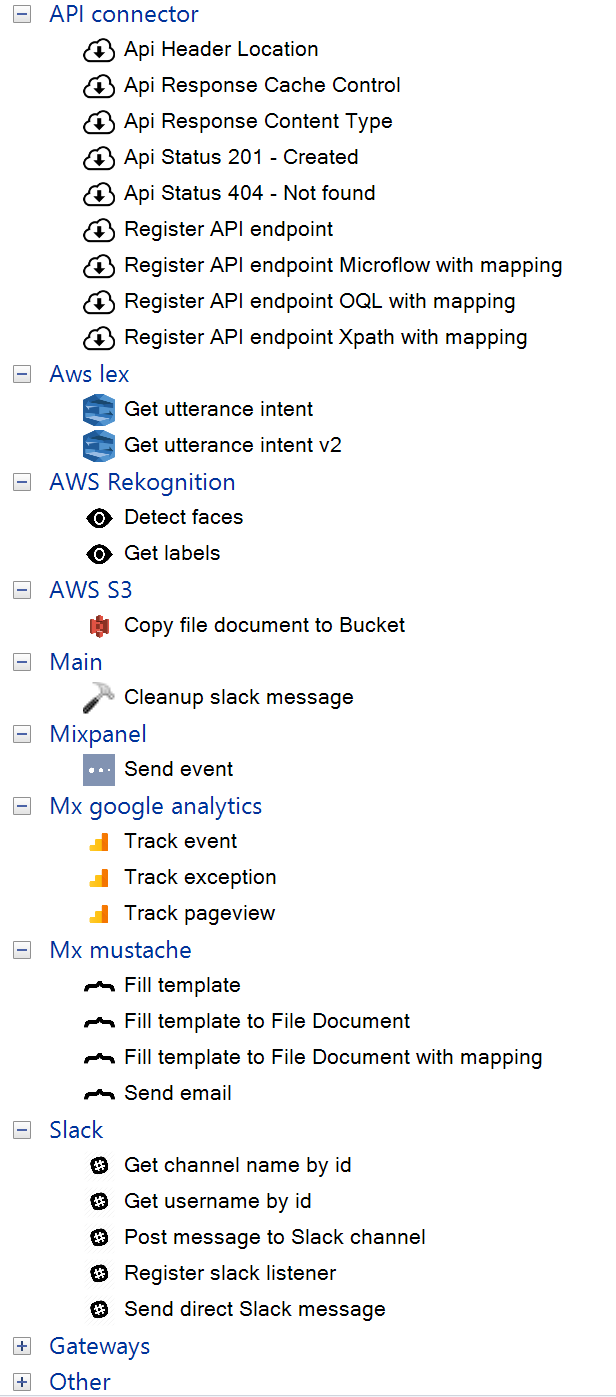
To illustrate the power of the Connector Kit, here's a high-level design diagram for an application we recently built: a Slack bot which enables users to determine things and people in pictures taken with a mobile Slack app.



Slack Rekognition Bot design

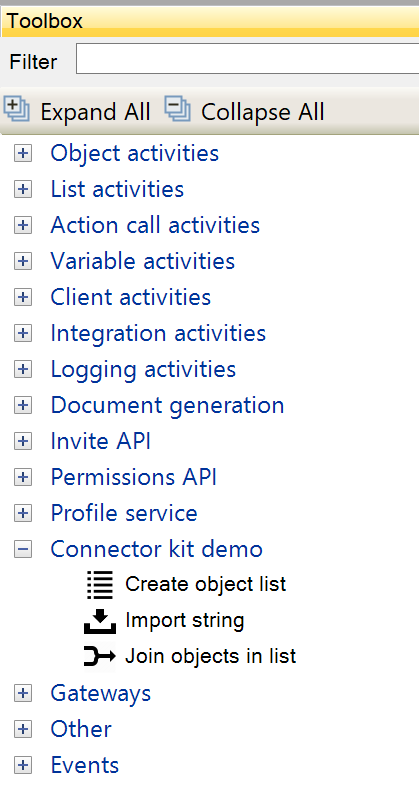
The Mendix application consists of a small number of microflows that use Mendix Microflow actions to offer a conversational user interface using [Slack](https://slack.com/) and different Amazon services: [S3](https://aws.amazon.com/s3/), [Rekognition](https://aws.amazon.com/rekognition/) and [Lex](https://aws.amazon.com/lex/).

The following screenshot shows what the microflow toolbox looks like after including all the modules providing connectors to the services used.



Slack Rekognition bot toolbox

For the basics of building toolbox actions you can read the [Connector Kit introduction blogpost](https://www.mendix.com/blog/introducing-mendix-connector-kit/). In this post I will explain some of the more advanced features that you can use when creating your own Microflow actions. We will also publish a blogpost shortly to discuss the details of the Slack bot created using the Connector Kit.



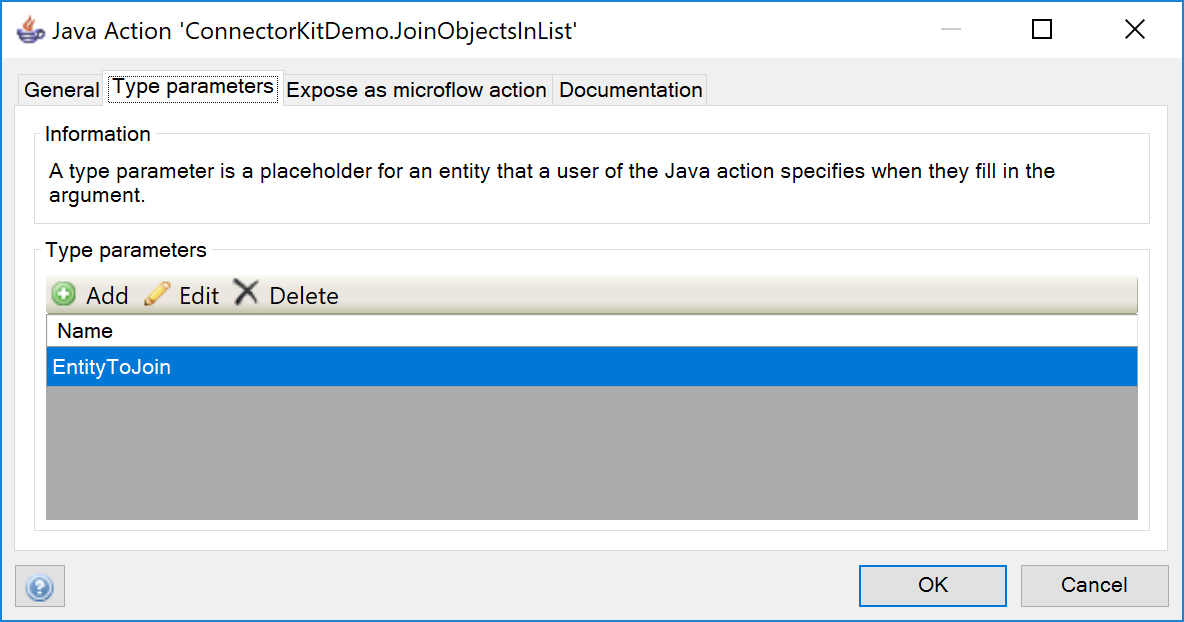
Connectorkit demo toolbox

## Creating generic actions using Type Parameter

Lets start with *Type Parameters*. Mendix 6.6 introduced a new type parameters tab in the java action definition dialog, as illustrated below. You can use a type parameter if you want to ensure that certain parameters of your action share the same entity, but you do not know the name of this entity when defining the actions.

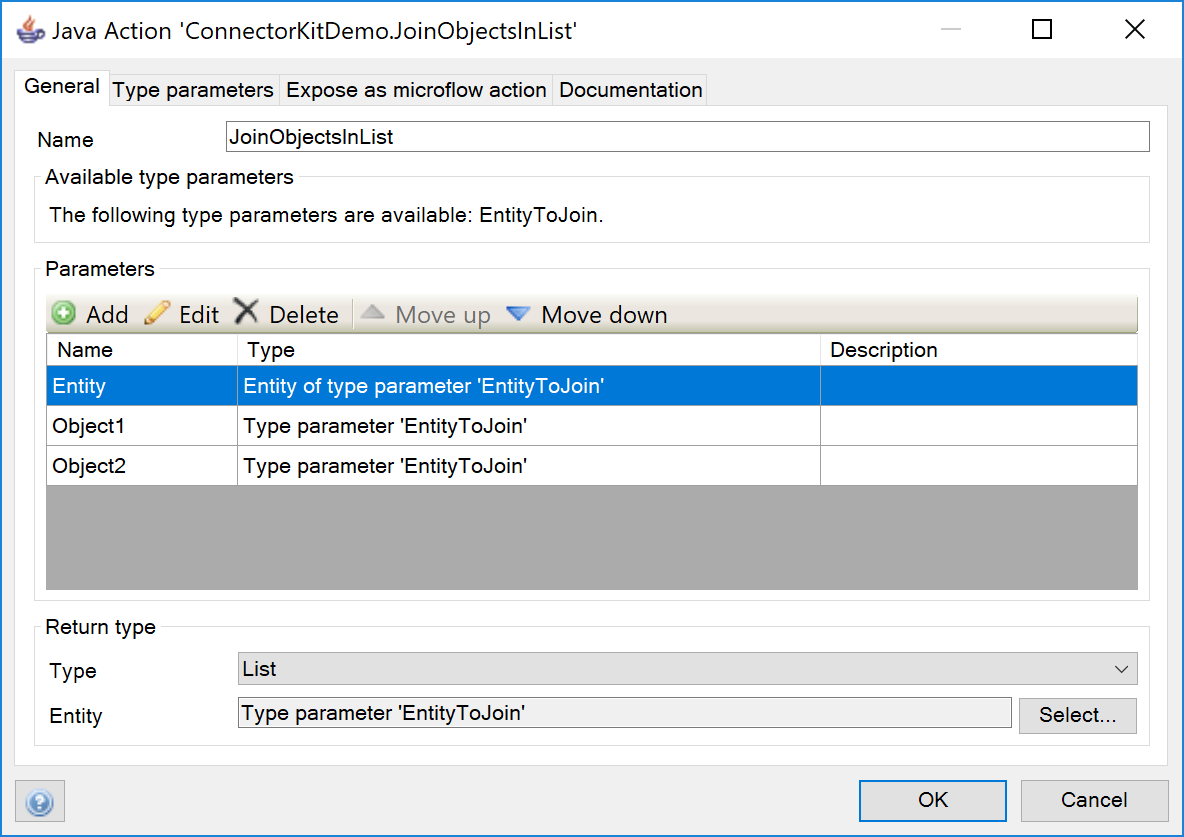
For example: suppose you want to create an action that takes two objects of the same entity and returns a list containing both objects. You can use a type parameter to guarantuee that both the input parameters for specifying the objects, and the resulting list all use the same entity.

First you define the type parameter to hold the entity used by all the parameters.



Type parameter tab

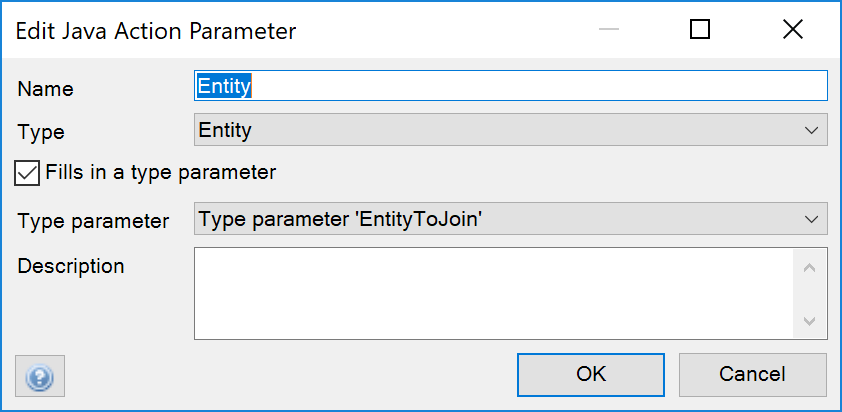
Next you can create parameters using the previously defined type parameter *EntityToJoin*.



Type parameter use

The action needs the following parameters:

* **Entity** - This is used to specify the entity of the objects to join. The entity selected by the user will be stored in the type parameter *EntityToJoin*.

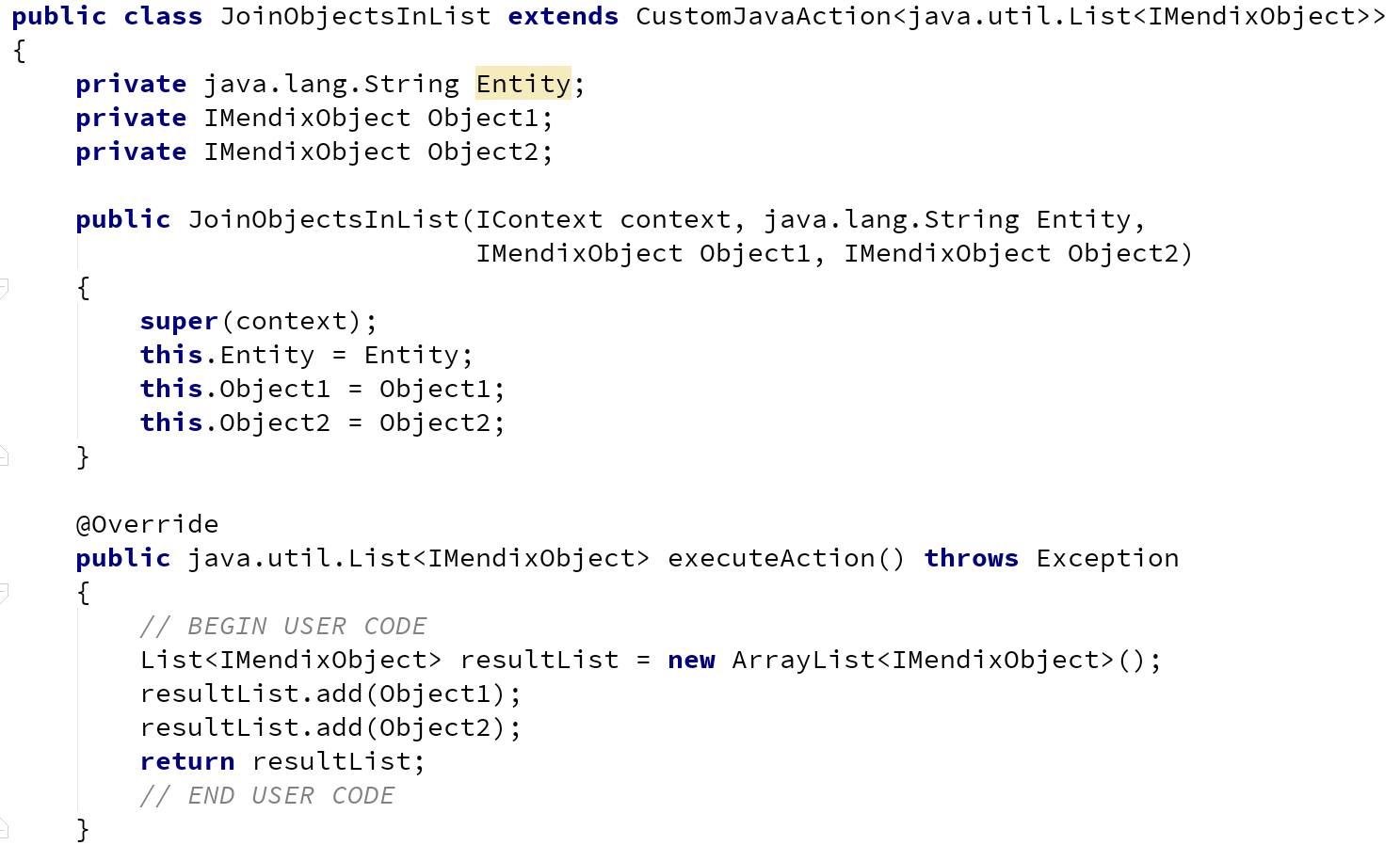


Type parameter use definition

* **Object1** - The first object to be added to the new list. It needs to be an object of entity EntityToJoin
* **Object2** - The second object to be added to the new list.
* **Return type** - the result of the action will be a list of *EntityToJoin* objects.

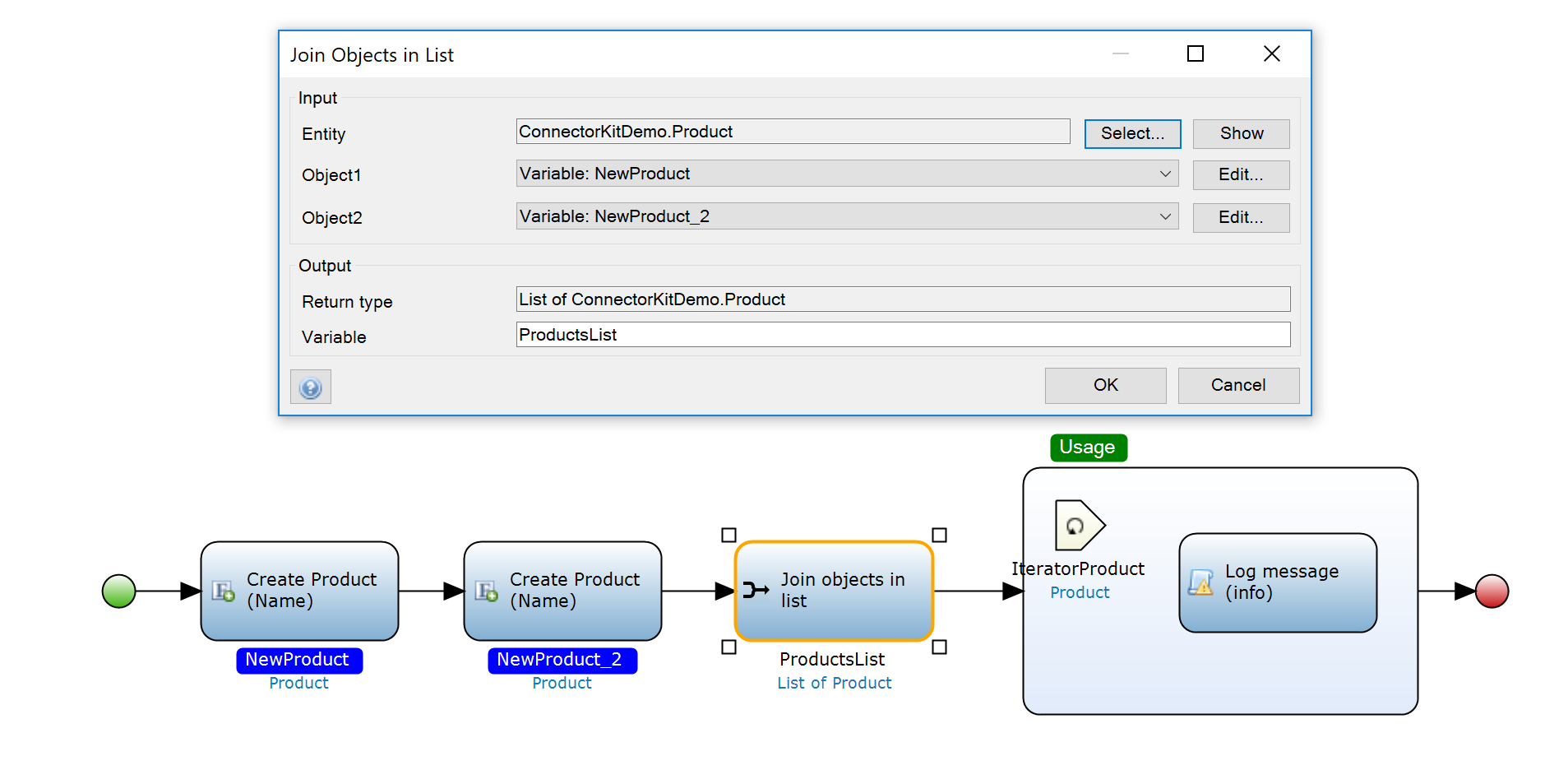
The java implementation still uses strings to specify the name of an entity, which means that you can upgrade your existing java actions to use these new parameter types without having to refactor your existing code.

Finally, here's the actual java implementation of the action defined:



Java implementation join object

You now have a reusable action in your toolbox that will join two objects into a list, as illustrated by the following example.

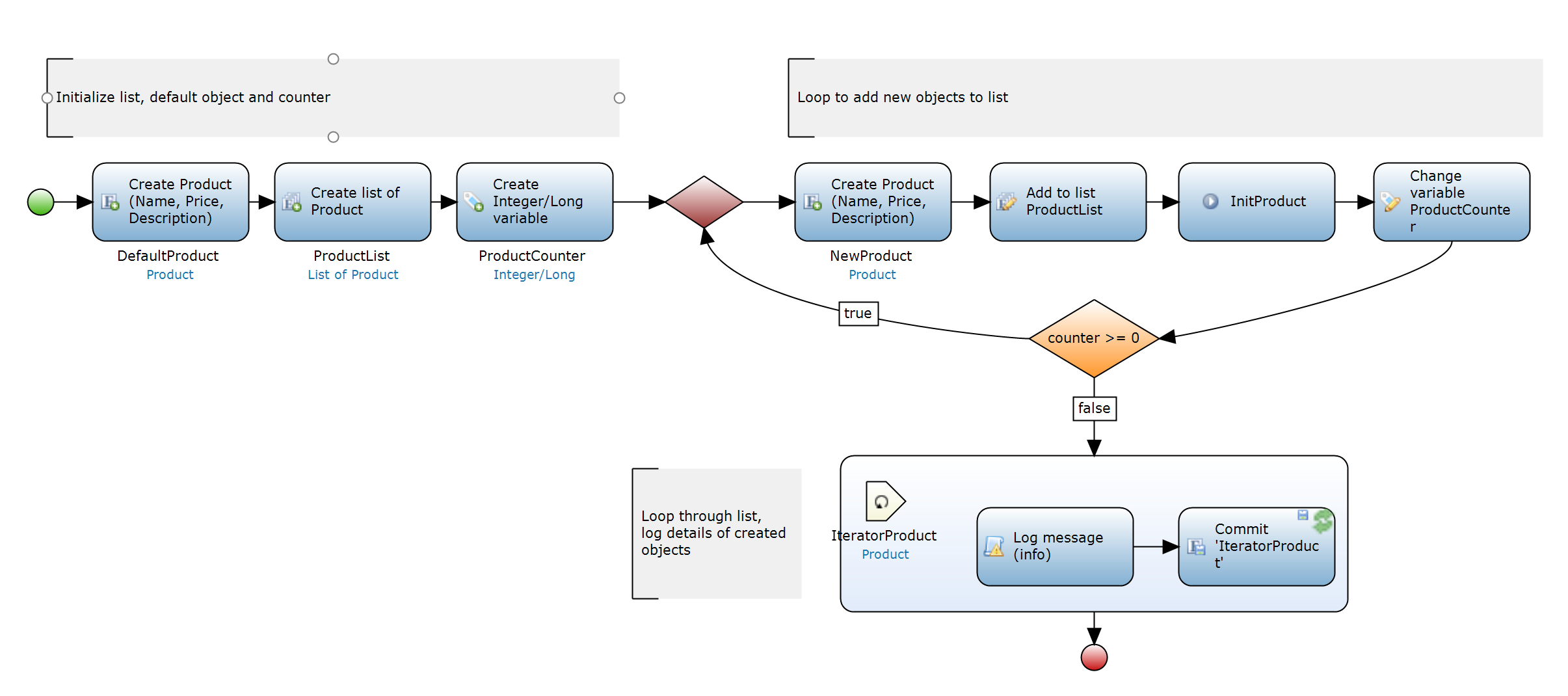


Join objects use

As you can see, *type parameters* enable you to create typesafe generic actions.

## Executing microflows

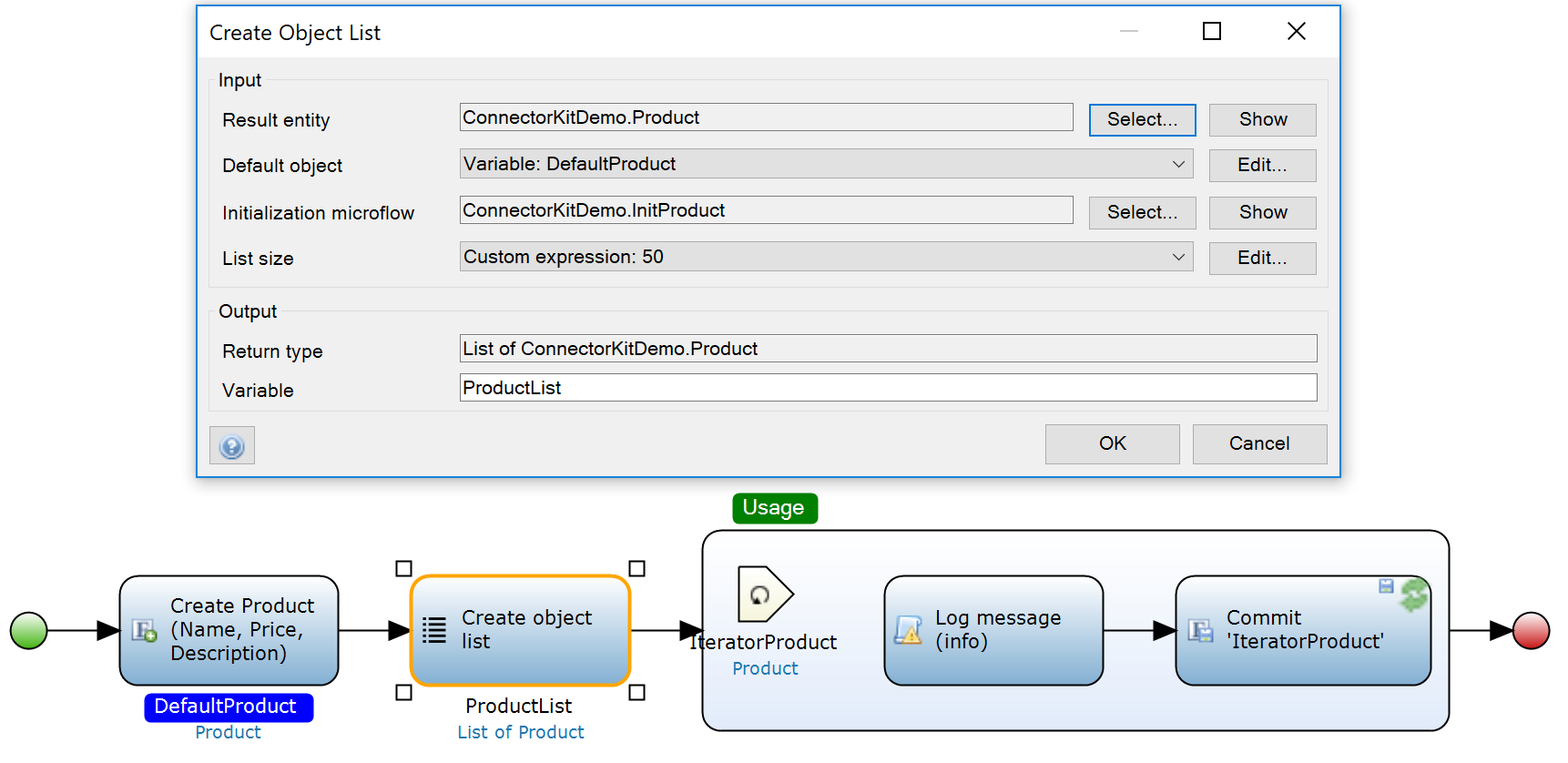
The following example illustrates how you can use microflow parameters. The microflow below creates a list of Product objects and calls a microflow for every project object to initialize it.



Init loop

Lets see if we can create a custom reusable Microflow action to make this Microflow more concise.

Here's a alternative to the Microflow above, but now using a custom java action to replace the loop and instantiation and initialization of the objects with a java action:



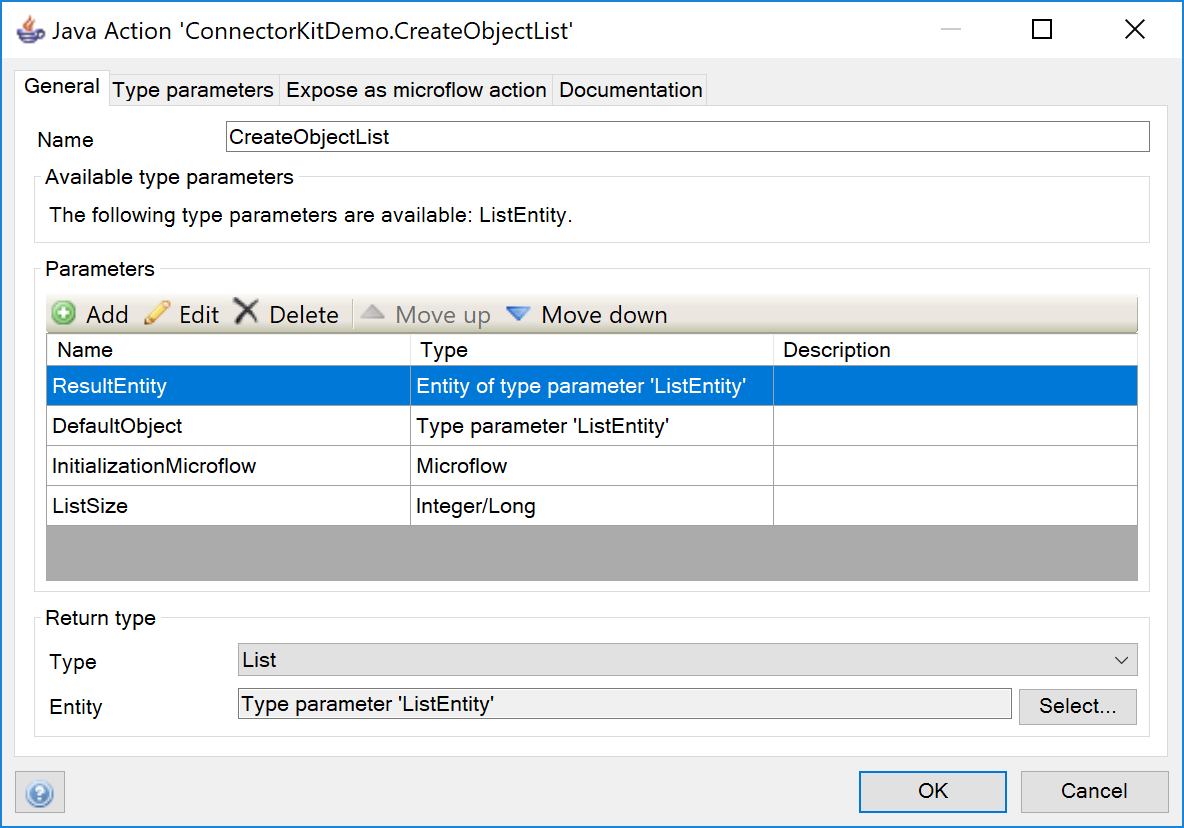
Init list loop with action

The action uses the following parameters:

* **ResultEntity** - entity used for the default object, and the result list
* **DefaultObject** - default value for the objects to be instantiated
* **InitializationMicroflow** - a Microflow that will be called for every new object to initialize it
* **ListSize** - the number of objects to be created in the list

The return type is a list of new initialized objects.

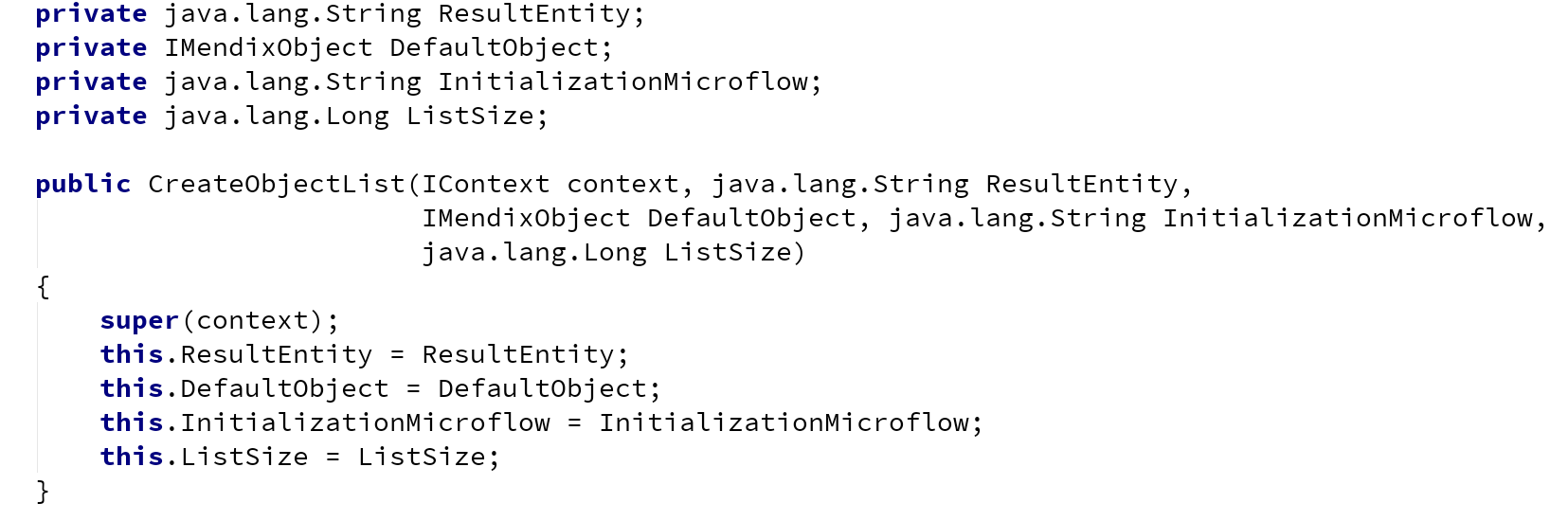
As you can see below, this action uses a new parameter type, Microflow, to indicate that the user needs to specify a Microflow. When using the action, the modeler will show a list of Microflows to make this as easy to use as possible.



Initialize list using microflow action parameters

In the java implementation for this action you'll see the following for the parameters:

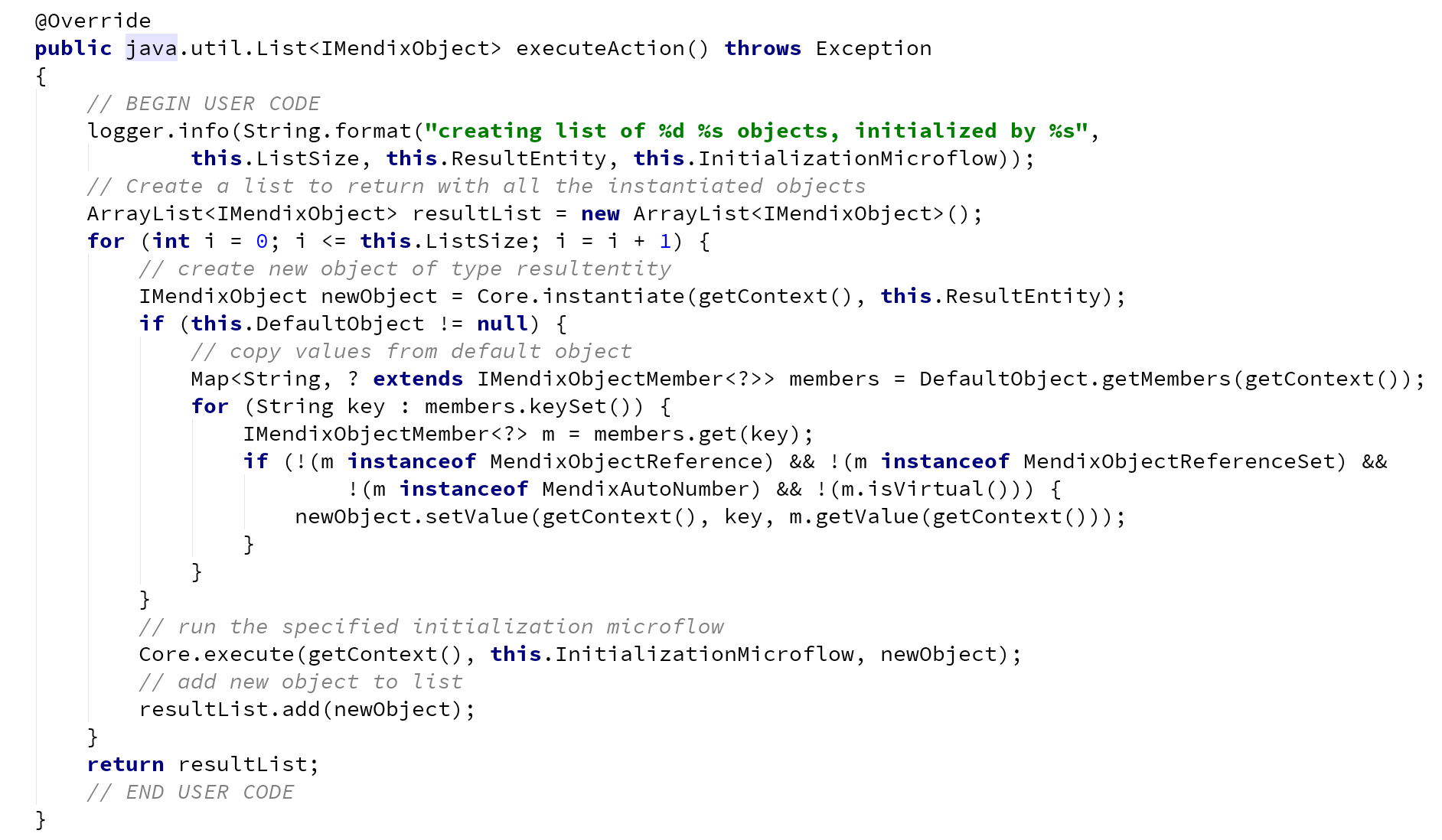
* **ResultEntity** - a string with the entity name used for the default object and the result list
* **DefaultObject** - a IMendixObject instance containing the default object.
* **InitializationMicroflow** - a string containing the name of the initializing microflow
* **ListSize** - a long containing the number of objects desired in the list



Initialize list java implementation 1

The executeAction method is where all the magic happens:

1. It first initializes an ArrayList for the result.
2. Then it has a for loop to create the desired number of objects.
3. The objects are created using *Core.instantiate()*. The entity name specified in the action is used as input to specify what entity to instantiate.
4. Next determine if a default object was specified. If so, copy all attribute values to the new object.
5. Execute the initialization microflow using *Core.execute()*.
6. Add the newly instantiated and initialized object to the result list.
7. Finally return the list of new objects.



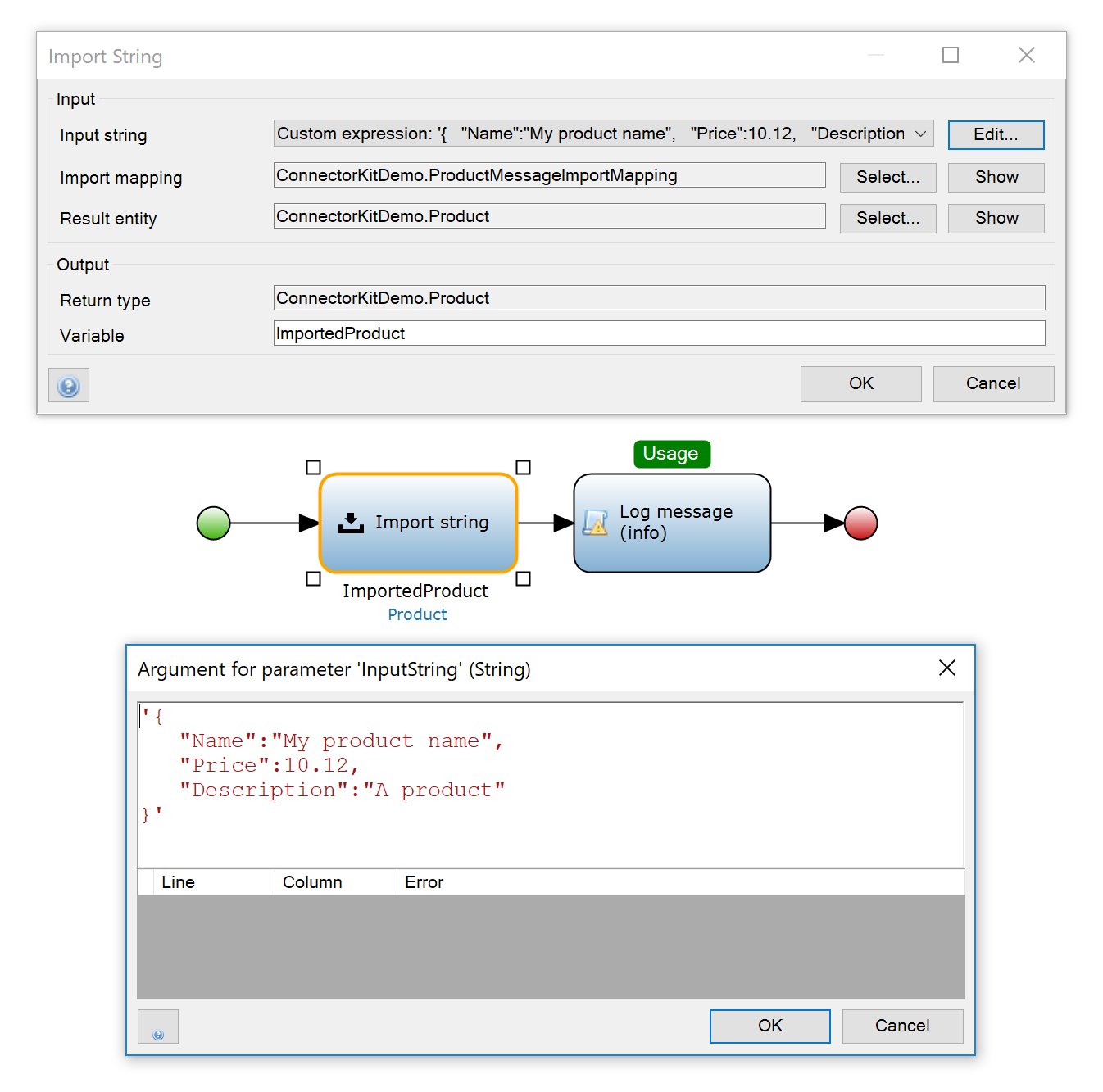
Initialize list java implementation 2

Microflow parameters are especially usefull for handling events. For example, the [MQTT connector](https://appstore.home.mendix.com/link/app/3066/Mendix/MQTT-Client) ([GitHub MQTT Connector project](https://github.com/ako/MqttClient)) will execute a Microflow when receiving an IoT sensor event so it can be handled using a user specified Microflow.

## Using import and export mappings

Finally an example how you can use Mappings in your Java actions. In this example we'll create an action to import a string using an import mapping. This is not particulairly useful, seeing there is a default action in your toolbox already that provides this functionality, *Import with mapping*. However, as an example it illustrates how to use mappings.

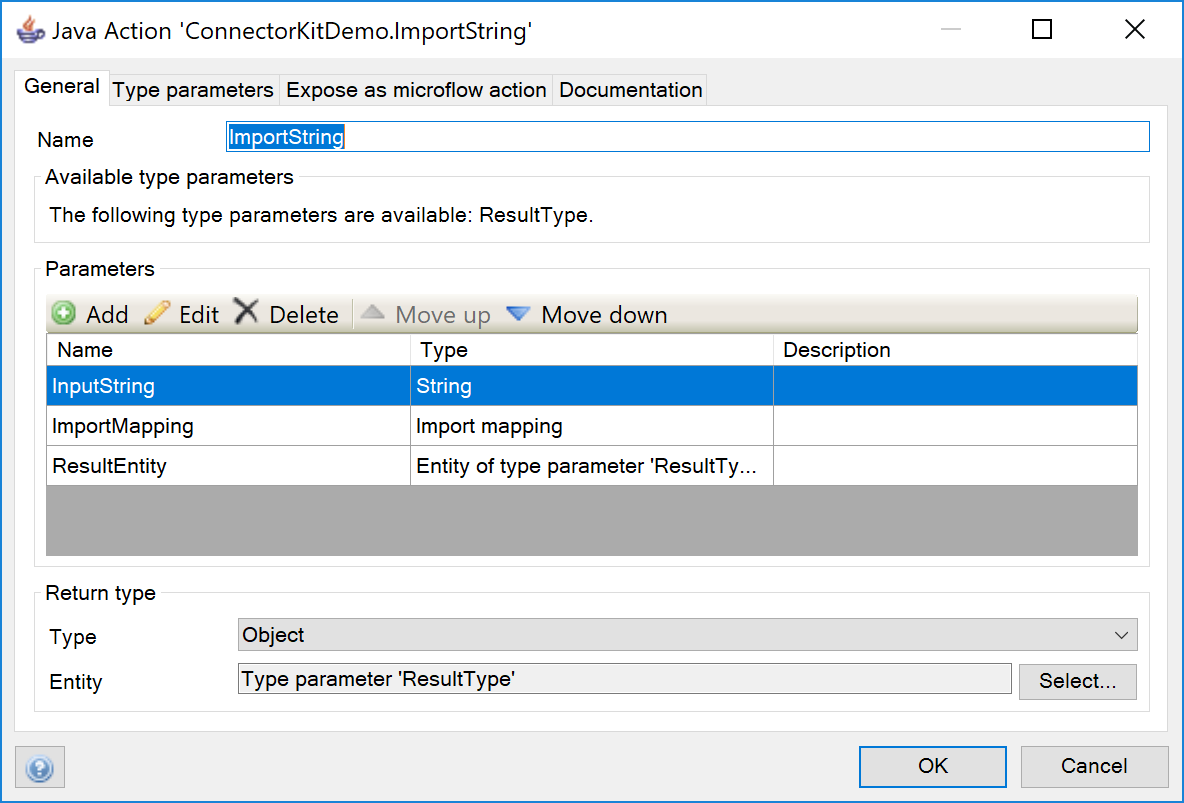
First a screenshot of what we are building: an action to import JSON strings. The action requires the user to provide a string with the JSON to import, select an import mapping, and define the entity of the result. Finally a name needs to be provided for the result of the import mapping.



Example import string use

The action is defined as follows:

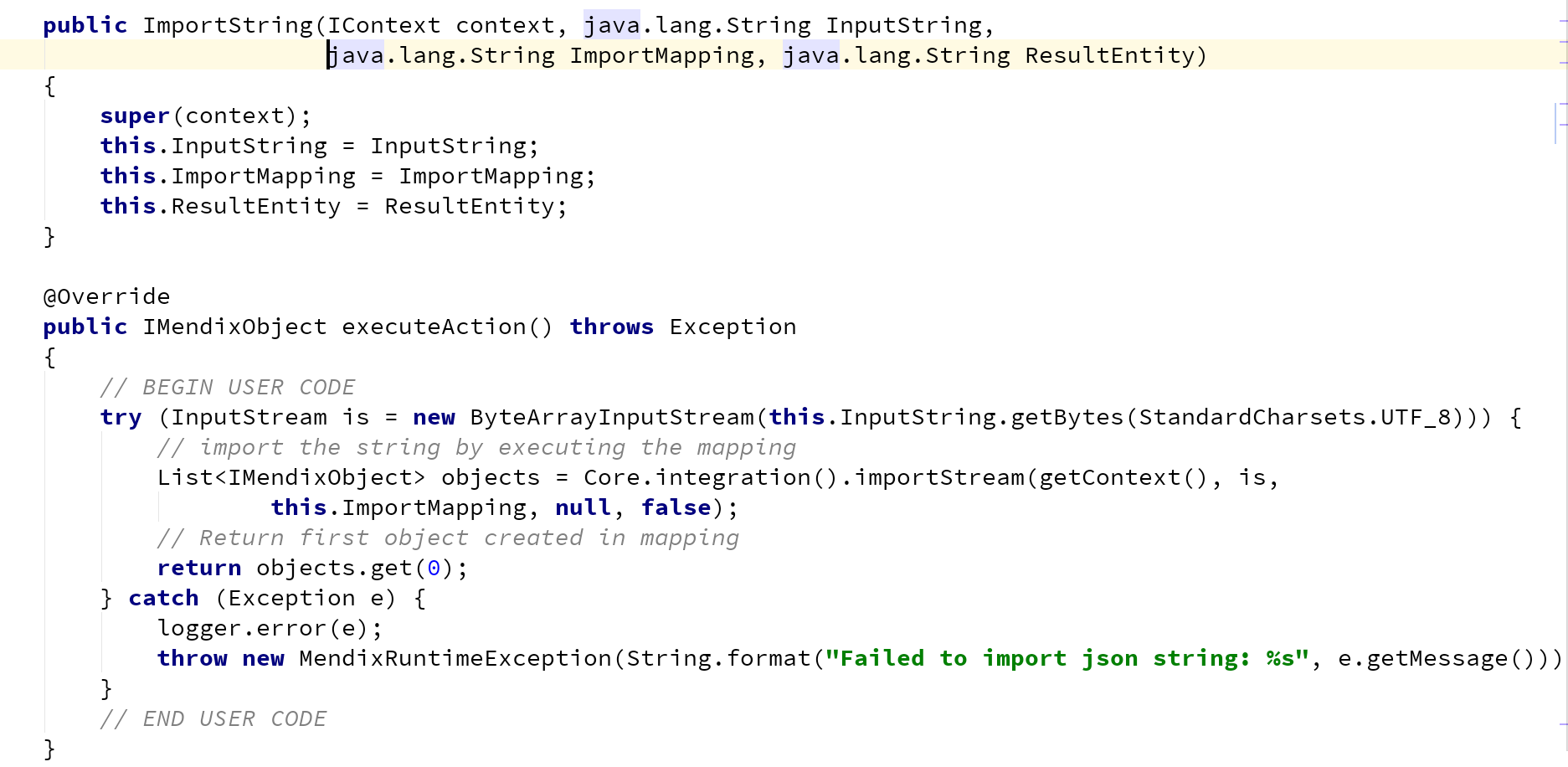
* **InputString** - A string parameter containing the JSON to be imported.
* **ImportMapping** - The name of the mapping to be used for importing the JSON.
* **ResultEntity** - Type of the object that will be the result of the import.
* **Return type** - An object of the type specified with ResultEntity.



Import String with mapping java action parameters

The actions is implemented in java as follows:

1. Create an InputStream from the the JSON input so it can be read by the import mapping.
2. Use *Core.integration().importStream()* to import the JSON with the specified mapping.
3. Return the first object imported.



Import String java action

## Some development tips

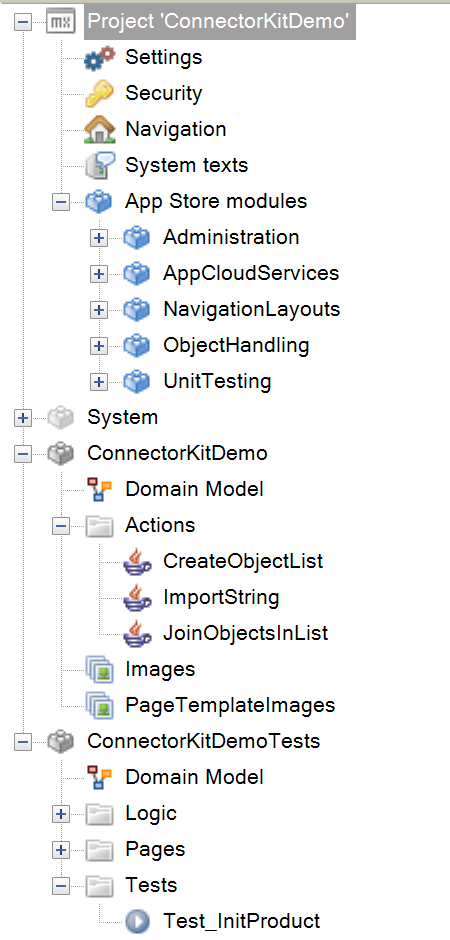
### Unit testing

When developing connector modules you can use the unit test module to test the actions you are implementing.

If you want to publish your module with custom microflow actions to the AppStore for easy reuse it's best to have a module containing only the reusable part. Add another module to your project with all the test Microflows, and anything else you need while developing your application.

The screenshot below illustrates this:

* The *ConnectorKitDemo* module only contains the actions you want to publish to the appstore. You can do this by selecting *export module package...* from the context menu.
* The *ConnectorKitDemoTests* module contains all functionality you need while developing the reusable module: a small domain model with some sample data and some test pages. It also contains one unit test microflow that will be called by the unit test module: *Test\_InitProduct*.



Mendix Connector kit module project with tests

### Managing libraries

When you export the module package for publishing in the AppStore you only want to include the relevant Java libraries. The easiest way to manage this is to use a build tool to specify and download the relevant dependencies.

The [ConnectorKitDemo](https://github.com/ako/ConnectorKitDemo) project on Github contains 2 examples how to do this:

* **Ivy** - [Apache Ivy](http://ant.apache.org/ivy/) is a java library specifically created for managing dependencies. Ivy is small enough that you can include it in your Mendix project. You can create two configurations in an ivy.xml configuration file. One to specify all jars needed to run the project and execute the tests. One to just specify the libraries that should be included when publishing the module for reuse in the AppStore. The demo project contains two configurations called *default* and *export* in the [ivy.xml](https://github.com/ako/ConnectorKitDemo/blob/master/ivy.xml) configuration file.
* **Gradle** - [Gradle](https://gradle.org/) is a full fledged build tool where you can define tasks to build your project. The example in the demo project uses gradle only to manage the dependencies. Again it defines [two build configurations](https://github.com/ako/ConnectorKitDemo/blob/master/build.gradle), one requiring all dependencies, one just the libraries needed for exporting the module (in this case just a library to call Slack).

Before developing custom java actions you can run the build tool to download all dependencies. When you are done, you run the build tool to delete all libraries in your userlibs folder and download only the jars to be included in the published module. Next you export the module and upload it to the AppStore.