Developing behavior extensions for InfoSphere MDM

Dmitry Drinfeld
Stephanie J. Hazlewood

September 12, 2014

One of the most fundamental extension mechanisms of InfoSphere® Master Data Management (MDM) allows for the modification of service behavior. These extensions are commonly referred to as behavior extensions, and the incredible flexibility they provide allows for organizations to implement their own "secret sauce" to the 700+ business services provided out of the box with InfoSphere MDM. The purpose of this tutorial is to introduce you to behavior extensions and guide you through the implementation, testing, packaging, and deployment of these extensions. You will be introduced to the Open Service Gateway initiative (OSGi)-based extension approach in InfoSphere MDM Workbench Version 11.

Many established organizations end up having unmanaged master data. It may be the result of mergers and acquisitions or due to the independent maintenance of information repositories siloed by line-of-business (LOB) information. In either situation, the result is the same: Useful information that could be shared and consistently maintained is not. Unmanaged master data leads to data inconsistency and inaccuracy. IBM Master Data Management (MDM), specifically the physical MDM set of capabilities, allows the enterprise to create a single trusted record for a party. Similar capabilities exist for mastering product and account information. It can be integrated with content management systems and can also support a co-existence style of master data management — sometimes referred to as *hybrid MDM capabilities,* where linkages among matched records mastered and maintained in various source systems are created in virtual MDM and then persisted in physical MDM.

One of the most fundamental extension mechanisms of InfoSphere® Master Data Management (MDM) allows for the modification of service behavior. These extensions are commonly referred to as behavior extensions, and the incredible flexibility they provide allows for organizations to implement their own "secret sauce" to the 700+ business services provided out of the box with InfoSphere MDM. The purpose of this tutorial is to introduce you to behavior extensions and guide you through the implementation, testing, packaging, and deployment of these extensions. You will be introduced to the Open Service Gateway initiative (OSGi)-based extension approach in InfoSphere MDM Workbench Version 11.

What we will cover

With the release of InfoSphere MDM 11, we adopt the OSGi specification, which allows, among many other things, extensions to be deployed in a more flexible and modular way. This tutorial

will describe a real client behavior-extension scenario and step you through the following required steps:

1. Extension scenario outline
2. Creation of the extension project
3. Development of the extension code
4. Deployment of the extension onto the MDM server
5. Testing deployed code using remote debugging

Extension scenario

It is often necessary to customize an MDM implementation in order to meet your solution requirements. One of the extension capabilities InfoSphere MDM provides is the ability to implement additional business rules or logic to a particular out-of-the-box service. These types of extensions are referred to as behavior extensions because they ultimately change the behavior of a service. In this tutorial, we will create a behavior extension to the searchPerson transaction.

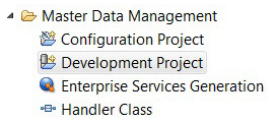
The searchPerson transaction is used to retrieve information about a person when provided with a set of search criteria. You can filter out the result set by active, inactive, or all records that get retrieved by these criteria. It is important to note that this search transaction uses exact match and wildcard characters to retrieve the result set. There are separate APIs available for probabilistic searching; this service is not one of them.

Sometimes, the searchPerson transaction response may contain duplicate parties. For example, if a party contains identical legal and business names, and searchPerson transaction uses last name as criteria, the parent object will be returned twice in the response, since it will be matched by both of the names. While this behavior is acceptable in some circumstances, some cases might need more filtering. So we will create a behavior extension, which will be responsible for processing transaction output and removing any duplicate records in the result set. The InfoSphere MDM Workbench provides us with exactly the right tools to quickly create and deploy such an extension.

Creating an extension project

First, create the extension project structure using the wizards provided by MDM Workbench. Go to **File > New > Other** and search for **Development Project** wizard.

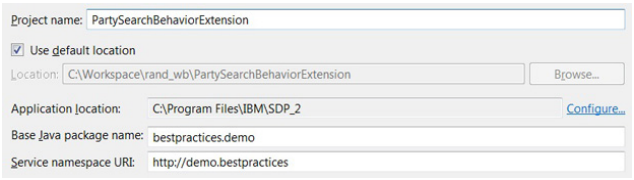
Figure 1. Creating a new Development Project



If you cannot find Development Project wizard within the list, chances are the Workbench has not been installed, verify that you are using IBM Installation Manager.

When creating your project, specify a unique project and package names to avoid conflict with the existing ones.

Figure 2. Specifying project name, package and namespace



Choose the correct server runtime for your projects, as well as unique name for the CBA project.

Figure 3. Specifying server runtime and CBA to contain the behavior extension

Note: You are allowed to choose from the existing CBAs. A single CBA can contain multiple development project bundles.

Click **Finish** and wait for the wizard to generate the required assets.

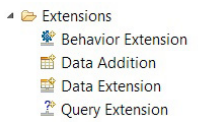
At this point, what we have is a skeletal InfoSphere MDM Development project that contains all of the basic facilities to help us create the desired extension. The next step is to create the extension assets and there are two ways of doing so: by using the behavior extension wizard, or by using the model editor.

Creating a behavior extension using the extension wizard

You can create an extension using a wizard in the MDM Workbench, much like the one used to create a development project:

1. Open Behavior Extension wizard by going to **File > New > Other > Behavior Extension**, located under **Master Data Management > Extension folders**.

   ### Figure 4. Selecting a behavior extension wizard

2. Once in the wizard, select the development project to place the extension under.

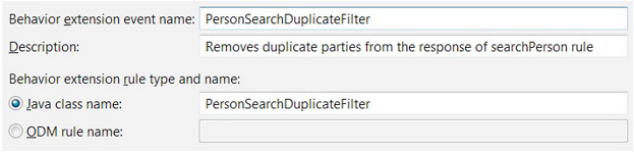
   ### Figure 5. Selecting a project under which to create the behavior extension

   Note: A development project can contain multiple extensions of various types underneath it. You might choose to use development projects to logically group extensions having a similar purpose, type, or to facilitate parallel development activities.
3. In the next window, choose a name and a description for your behavior extension. Choose a Java™ class name for your extension. This is the class that we will be populating with custom logic to achieve desired behavior. Alternatively, if you require to use an IBM Operational

Decision Manager (ODM), previously known as ILOG®, rule–specify this associated parameter. ILOG/ODM rule creation is not covered as a part of this tutorial since we will implement the extension in as a Java extension.

Figure 6. Selecting name, description and Java class for behavior extension

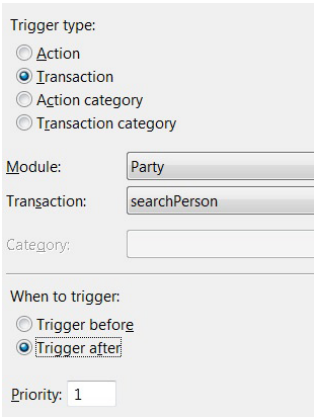


4. In the **Specify details of the trigger** pane, specify the following parameters:
   a. Trigger type:
      i. **Action** will cause the behavior extension to trigger whenever the chosen transaction is run by itself or as a part of another transaction. Actions are executed at the component level.
      ii. On the other hand, if you are looking to trigger extension only on a specific standalone transaction event (otherwise known as controller-level transaction) select **Transaction** for trigger type.
      iii. **Action Category** trigger type executes behavior extension on various data actions, such as add, update, view, or all for extensions to be executed at the component level.
      iv. **Transaction Category** trigger type will kick off behavior extension when a transaction of specified type is executed — namely inquiry, persistence or all.
   b. When to trigger:
      i. **Trigger before** will cause the behavior extension to fire of before the work of the transaction is carried out. Sometimes you will hear this referred to as a *preExecute extension.* It is typically used when some sort of preparation procedure has to be executed before the rest of the transaction is carried out. An example of such a scenario would be preparing data within the business object being persisted.
      ii. **Trigger after** will cause the behavior extension to run after the transaction work has carried out. Sometimes you will hear this referred to as a *postExecute extension.* It is typically used in the scenarios where logic implemented in the behavior extension depends on the result of the transaction. Normally, any sort of asynchronous notification would be placed in a post-behavior extension because there would be no way to roll it back in case of transaction failure, if it is sent before the transaction is executed.
   c. **Priority** parameter indicates the order in which this behavior extension will be triggered. The lower the priority number, the higher the priority. That is to say that a behavior extension with priority 1 would execute first followed by behavior extension with priority 2, 3, or 4 — in that order.

In our scenario, we are looking to filter the response of a specific transaction — namely searchPerson. Therefore, we set the trigger type to be **Transaction** with value of searchPerson. Since we are filtering the response of the transaction, we have to trigger our

behavior extension after the transaction has gone through and response became available. Lastly, in our example, priority does not play a special role, so we will leave it at default of **1**.

Figure 7. Configuring the behavior of the behavior extension



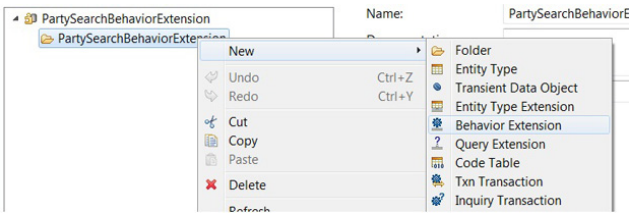
5. After the above configuration is done, click **Next** and review the chosen parameters. Note that there is a checkbox at the top of the dialog, allowing you to generate the code based on the specified parameters immediately. For the purposes of this tutorial, leave it checked and click **Finish**. The workbench will generate the required assets for you.

Creating a behavior extension using the model editor

If you have used the wizard approach above to create the behavior extension already, please feel free to skip ahead to the section titled "Review generated assets" that follows. This section describes how to generate a behavior extension using the model editor. The following steps will guide you through this process:

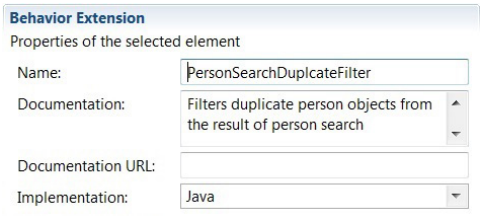
1. Go to the development project you created earlier and open the **module.mdmxmi** file under the root folder of the project. Select the model tab within the opened view.
2. Right click PartySearchBehaviorExtension folder, then select **New > Behavior Extension**

Figure 8. Creating behavior extension in the model editor



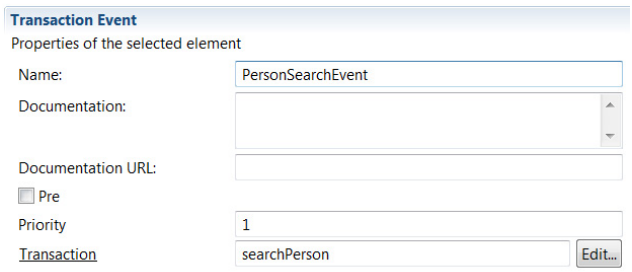
3. Name the behavior extension that has been created PartySearchDuplicateFilter and provide appropriate documentation:

Figure 9. Selecting properties in the behavior extension pane



4. Now we will create a transaction event definition under behavior extension. Right-click the behavior extension, then select **New > Transaction Event**.
5. Once the transaction even has been created, specify the appropriate properties
    a. Because this event is triggered on the personSearch transaction, PersonSearchEvent is appropriate. Recall that sometimes the trigger-before behavior is referred to as preExecute extension.
    b. Because **Pre** checkbox stands for preExecute, (more specifically, the behavior extension gets executed before the rest of the transaction), leave it unchecked. Similar to the wizard configuration, leave priority as 1, since priority of execution does not affect this behavior extension.
    c. Select searchPerson as the transaction of choice by clicking **Edit > Party > CoreParty > searchPerson**.

Figure 10. Specifying behavior properties of behavior extension



After the above configurations are done and reviewed, click **Generate Code** under the **Model Actions** section of the view, telling workbench to generate configured assets.

Review your generated extension code

Once either of the above methods is used, let us review the generated assets:

- Under bestpractices.demo.behavior, we find the PersonSearchDuplicateFilter, which is the actual behavior extension Java class that, once configured, will be executed at runtime. We will be providing the implementation that will filter out duplicate person objects from the response of the searchPerson transaction here shortly.
- In the OSGI-INF/blueprint/blueprint-generated.xml, we can see the OSGi service definition, listing the bestpractices.demo.behaviour.PersonSearchDuplicateFilter class as the extension service.

- Under the resources/sql/<db_type>/PartySearchBehaviorExtesion_MetaData_DB2.sql file, we find the configurations we've specified regarding behavior extension execution:
  - EXTENSIONSET table record defines the behavior extension, its associated class **bestpractices.demo.behaviour.PersonSearchDuplicateFilter** and its priority of 1.

    Figure 11. Behavior extension database record

  - CDCONDITIONVALTP defines a new condition of transaction name being equal to searchPerson.
  - EXTSETCONDVAL connects the above CDCONDITIONVALTP record to the behavior extension record from EXTENSIONSET. Additionally, another EXTSETCONDVAL record connects CDCONDITIONVALTP with ID of 9, which stands for execution of behavior transaction after transaction.

Let us now move on to developing the extension code required to filter out duplicate person records from the result set returned by the searchPerson transaction.

Develop the extension code

The behavior extension skeleton and supporting configuration assets have now been generated. You add your custom logic, or behavior change, in the execute method of PersonSearchDuplicateFilter class. The objective is simple. We need to go through all of the partySearchResult objects returned by the service, and if a same-person object repeats multiple times throughout the result vector, we want to remove it. The following code will achieve just that.

Listing 1. Behavior extension logic

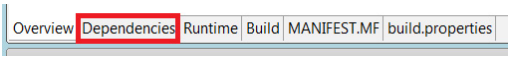
```
public void execute(ExtensionParameters params)
{
    // Only work with vectors in the response
    if(params.getWorkingObjectHierarchy() instanceof Vector)
    {
        // Get the response object hierarchy
        Vector partySearchResultList =
           (Vector)params.getWorkingObjectHierarchy();

        // Iterate through the party search result
        // objects to find duplicates
        Iterator listIterator =
           partySearchResultList.iterator();

        // We will keep the party ids of objects we've already
        // processed to identify the duplicates
        Vector partyIdList = new Vector();
        while(listIterator.hasNext())
        {
            Object o = listIterator.next();
            if(o instanceof TCRMPersonSearchResultBObj)
            {
                TCRMPersonSearchResultBObj personSearchResultBObj =
                      (TCRMPersonSearchResultBObj)o;
                String partyId = personSearchResultBObj.getPartyId();

                // If the party id has not been seen yet, this person
                // object is not a duplicate, otherwise - remove it from
                // the response
                if(partyIdList.contains(partyId))
```

```
                    listIterator.remove();
                else
                    partyIdList.add(partyId);
            }
        }
    }
    System.out.println("PartySearchBehaviorExtension has finished executing.");
}
```

Note: The above implementation is not pagination-friendly and pagination will not be covered as a part of this tutorial.

Once you have compiled the code above, you will notice that some of the classes are not found and have to be imported. You cannot simply import TCRMPersonSearchResultBObj because the package containing this business object is not imported for the bundle we're working with. To fix that, go to **BundleContent > META-INF > MANIFEST.MF**, dependencies tab.

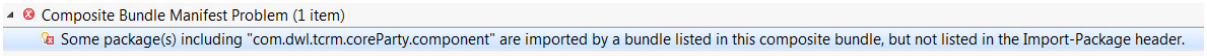
Figure 12. Dependencies tab within the MANIFEST.MF



Add com.dwl.tcrm.coreParty.component to the list of imported packages. Now you should be able to import TCRMPersonSearchResultBObj in our behavior extension class to resolve the dependency error.

After recompiling the projects again, you will notice that the PartySearchBehaviorExtensionCBA now contains an error.

Figure 13. Error on the PartySearchBehaviorExtensionCBA project



This error is occurring because the composite bundle that contains PartySearchBehaviorExtension bundle does not import com.dwl.tcrm.coreParty.component. To resolve the error, go to the manifest file of the CBA, which should be located under root folder of the project, and add com.dwl.tcrm.coreParty.component to the list of imported packages. As you recompile – it should resolve the error.

Now that all compilation problems have been resolved, we are ready to deploy our extension onto the server.

Deploying your new behavior extension to MDM

Once the implementation of the behavior extension has been developed, we are ready to deploy it onto the server. There are two steps involved into the deployment:

- Deploying code to the server
- Executing generated SQL to insert required metadata

Deploying code to the server

Our customized behavior extension can be deployed to the server as a Composite Bundle Archive (CBA) as follows:

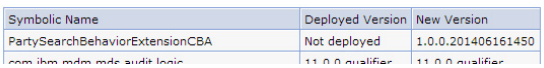
1. Make sure the customized code has been built and export the CBA containing the behavior extension by right-clicking the CBA project and selecting **Export > OSGi Composite Bundle (CBA)**.
2. In the opened view, select **PartySearchBehaviorExtension** as the bundle to include. Click **Browse** and navigate to a selected export location, then click **Save**. If you do not explicitly provide the filename, the wizard will generate the appropriate name automatically.
3. Click **Finish**. The CBA containing the behavior extension has now been exported to selected location.
4. At this point, we will assume that the MDM instance is up and running. Let's open the WebSphere Administrative Console. We are looking to import our new CBA into the internal bundle repository. Go to **Environment > OSGi bundle repositories > Internal bundle repository**. In the opened view, click **New**, choose **Local file system**, specify the location of the CBA exported above, and **Save your progress**.
5. Once the CBA has been imported, attach this new bundle to the MDM application. Go to **Applications > Applications Types > Business-level applications**. Choose MDM application from the opened view. In the next open view, open the MDM .eba file.
6. We are now looking at the properties of the MDM Enterprise Bundle Archive (EBA). To attach our CBA, go to **Additional Properties** section and select **Extensions for this composition unit**.
7. If this is the first extension you've deployed on your instance, the list of attached extensions will be empty. Click **Add** and check the CBA imported above, then click **Add**. Wait for the addition to complete, then **Save your changes**.
8. You may think that we are done, but not quite. We've only updated the definition of the EBA deployment by adding our extension. The MDM OSGi application itself has not been updated and even if you restart the server, your new behavior extension will not be picked up. So you must update the MDM application to the latest deployment by returning to the EBA properties view.

Figure 14. Update to the latest deployment button becomes clickable

Before we attached our extension, the button was grayed out, and the comment stated that the application is up to date. But since we've update our application with a new extension bundle, we need to update it to the latest deployment. Click **Update to latest deployment**.

9. In the next view, you can see that the PartySearchBehaviorExtensionCBA attached to the MDM EBA will now be deployed.

Figure 15. The behavior extension bundle being deployed



At this point, scroll down and click **Ok** to proceed. It may take several minutes depending on your system hardware.

10. At this point, WebSphere will take you through three views, offering multiple information summaries of the deployments and several customization options. There is no need to customize anything, so click **Next** three times, followed by**Finish**. At this point, the application will update. It may take some time; please allow 5–10 minutes to complete depending on underlying hardware. Once it is complete, save your changes.

At this point, the MDM application has been updated to the latest deployment which includes our extension.

Now we need to deploy our custom metadata to the database. This metadata will govern the behavior of our extension in ways discussed above.

Deploy metadata onto the MDM database

As mentioned, the Workbench generates database scripts that insert the required configuration into the metadata tables of the MDM repository. This metadata is generated based on the parameters we provided for our behavior extension as part of the **Creating extension project** section. In order to deploy this metadata to the database, run the database scripts listed under the resources/sql folder that are appropriate for your database type. Conversely, if you need to remove extension from the server, you would need to run the `rollback` scripts provided in the same folder.

Note: In the case where some portion of the script fails, please investigate the error because it may render the extension useless. Potential reasons for an error may include residual data from previous extension (rollback was not run when extension was removed), incorrect database schema, etc.

Once the scripts have been successfully run, you're your behavior extension has now been successfully deployed. Restart your WebSphere server so that your new metadata gets picked up when the application runs next.

Testing deployed code using remote debugging

Now that all of the aspects of the behavior extensions have been deployed, we are ready to test it out. Run a `searchPerson` transaction. It is required to have at least one person in the database so you can actually search and yield a successful search result to trigger your new extension. This test will show us that the extension is successfully deployed. Once the transaction returns as successful, go to the SystemOut.log of the WebSphere server located under the log folder of the WebSphere profile where MDM application is deployed. If the extension has deployed correctly, due to the following line in our custom code:

```
System.out.println("PartySearchBehaviorExtension has finished executing.");
```

You should be able to see this message in the logs:

```
[6/17/14 13:24:59:816 EDT] 000001b3 SystemOut O PartySearchBehaviorExtension has
finished executing.
```

Note: The log message is there for testing purposes only, and depending on the usage of the behavior extension can significantly impede performance. For that reason, please make sure to remove such debugging messages or put them into fine logging level before going into production. Such as:

```
logger.finest("PartySearchBehaviorExtension has finished executing.");
```

Configuring WebSphere Application Server debug mode

To observe the behavior of our extension more closely, put WebSphere server into the debug mode and connect MDM Workbench to the said server to debug our code step by step. To put your server in debug mode:

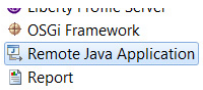
1. Go to WebSphere Application Server administrative console and navigate to **Servers > Server Types > WebSphere Application Server > <Name of your instance>**.
2. Once in the server configuration view, take a look at **Server Infrastructure** section and navigate to **Java and Process Management 7gt; Process definition**.
3. In the **Additional Properties** section, select **Java Virtual Machine**.
4. Once we are in the Java Virtual Machine view, navigate down to the **Debug Mode** checkbox, check it and provide the following settings in the **Debug arguments** textbox:
   ```
   -Dcom.ibm.ws.classloader.j9enabled=true -
   agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777
   ```
   Note that *7777* is the debug port to which the MDM Workbench will connect. Make sure this port does not conflict with any other assigned ports on the server, and set it accordingly.
5. Save the configuration and restart your server. It is now running in debug mode. If later you observe unexpected performance degradation and do not require debug mode any longer, make sure to take the server out of the debug mode using the same steps.

Configuring MDM Workbench to for remote debugging

Once the server is running in debug mode, we can go back to the MDM Workbench and configure it for debugging:

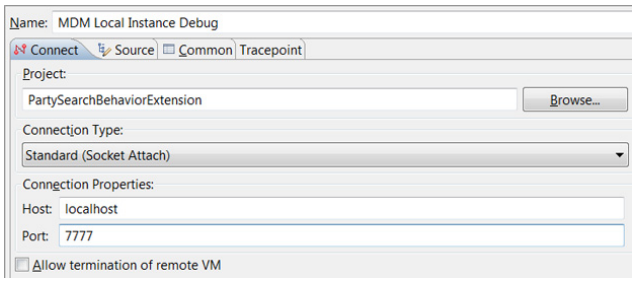
1. In MDM Workbench, go to **Run > Debug Configurations**.
2. Within the **Debug Configurations** window, double-click **Remote Java Application**. This will create a new remote Java application profile.

Figure 16. Remote Java application option in the debug-as listing



3. When configuring the remote Java application, let's name our configuration `MDM Local Instance Debug`. The project setting does not play a role; you may leave it empty or whatever the default populated value is. connection type should remain as standard. Lastly, connection properties should reflect the location of the MDM instance and debug port chosen above.

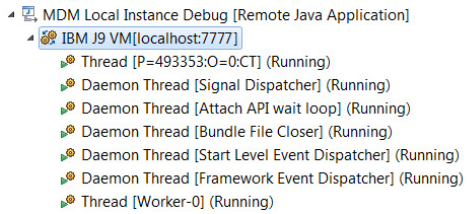
Figure 17. Remote debug configuration pane



We will not cover other tabs because the configuration we've done so far is sufficient.

4. Once configuration is complete, click **Apply > Debug** in order to attach to the MDM instance. The attach process may take a little time depending on the environment. Once it is complete, go to the debug perspective of your environment. You should observe the connected MDM instance if the attach was successful.

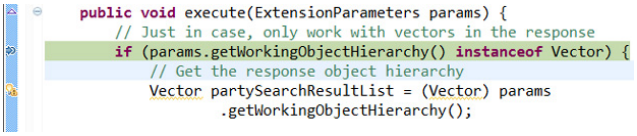
Figure 18. Successful debug connection showing the threads being debugged



You can see above that the instance is available along with all of the threads.

5. Set a breakpoint at the beginning of the behavior extension execute method and observe this breakpoint getting engaged once we run a searchPerson transaction.

Figure 19. Breakpoint being engaged in the behavior extension code



6. If you have multiple TCRMPersonSearchResultBObj coming back in the response, step through and observe the duplicates being removed.

Note that we can debug local and remote instances as described above, using Eclipse's Remote Java Application debug capabilities.

What we have accomplished

In this tutorial, we've gone through the steps of creating, configuring, deploying and testing a basic yet realistic behavior extension scenario for InfoSphere MDM.

We've covered two ways in which an extension template can be created: while the wizard option is straightforward and is preferable for a novice or a simple extension scenario, the model editor allows for more flexibility.

We've taken a look at the various configurations that apply to a behavior extension and outlined their effects on its execution. Additionally, we've covered the assets that get generated as a result of the configuration.

For the development step, we've created and analyzed the implementation of our behavior extension.

And finally, we've deployed, tested and debugged our behavior extension to make sure it performs as expected.

All of the above steps constitute a complete development process of an MDM Server behavior extension.

Related topic