Boomi Cloud™ API Management - Local Edition

SDK Guide

Version 5.6.2 | November 2024

Contents

Contents **2**

Overview **5**

The Local Edition SDK 5

SDK Components 5

System Requirements 6

Quick Start **8**

Step 1: Creating an Adapter Project 8

Pre or Post Processor 9

Custom Authenticator 9

Step 2: Packaging the Adapter 10

Step 3: Uploading the Build Adapters to Local Edition-Installer Build Job 11

Step 4: Building the Changing Local Edition-TM Docker image with Customer-Provided Adapters or Processors 12

Step 5: Configuring Endpoints for Processors 14

Tethered 15

Upgrading the Local Edition SDK 5.2 or 5.3 or 5.4 to Local Edition SDK 5.5 ..... **19**

Downgrading Local Edition SDK to an Earlier Version **21**

Configuring Network Proxy for Boomi Cloud™ API Management - Local Edition SDK **22**

Adapter SDK Package **23**

Boomi Cloud™ API Management - Local Edition Domain SDK 23

Boomi Cloud™ API Management - Local Edition Infrastructure SDK 23

Developing Processors and Authenticators **25**
SDK Domain Model 25
Extended Attributes 29
Pre and Post Processor Extension Points 30
Listener Pattern 30
Event Types and Event 30
Event Listener API 31
Importing Existing Adapters 31
Developing and Packaging Multiple Adapters 32
Using the Adapter SDK in an IDE 33
Creating an Adapter using Eclipse 33
Importing the Local Edition SDK into Eclipse IDE 34
Creating an Adapter using IntelliJ IDEA 37
Creating an Adapter using Apache NetBeans 40
Adding Third-Party Libraries in an Adapter 43
Adding third-party library for Eclipse environment 43
Referring to Third-party Libraries with Dependency 45

Debugging the Adapter **47**
Adding Logger Utility Class 47
Changing Log Levels 47
Checking Adapter Logs 50
Debugging SDK Processor Remotely 50

Implementing and Registering Processors and Adapters **57**
Implementing a Processor or Adapter 57
Creating a Pre-processing Adapter 57
Reading Body Content of Request 57
Modifying the Request Body 58
Terminating Further Processing for Unavailable Header 59
Creating a Post-processing Adapter 60
Add Custom Header to the Response to Client 60

Modifying Body Content of Response to Client 61
Creating a Custom Authenticator 62
Stopping a Processing Request on Authentication Failure 62
Continue Processing Request for Successful Authentication 63
Implementing the Event Listener 64
Implementing Lifecycle Callback Handling 65
Caching Content 67
Terminating a Call During Processing of an Event 68
Accessing and Using Extended Attributes 71
How to Send Response Body to Another Location Other Than Caller 73
How to Externalize Properties and Files from SDK-Built Adapters 74
Accessing Plan, Package and Application ID in Custom Processor 74

Chaining of Processors 76
Chaining of Processors Using Mashery_Proxy_Processor_Chain 78
Chaining of Processor Using Chain Adapter 80

FAQs 84

Boomi References 85

Overview

The Boomi Cloud™ API Management - Local Edition (CAM) Software Development Kit (SDK) is an extension application programming interface (API) using which the traffic manager capabilities can be extended to handle requests.

The API uses a listener pattern to perform callbacks during different stages of the request flow. Each step in the flow performs a specific task for fulfilling the request. With the Boomi Cloud API Management - Local Edition (Local Edition) SDK users can inject hooks in the following steps:

1. Pre process - Prior to invoking the API Server.
2. Post process - After receiving response from API server.

The Local Edition SDK

The Local Edition SDK is implemented as a Gradle project. The SDK project structure is same as that of Gradle multi-project configurations.

Note: You do not need to install Gradle separately. The Local Edition SDK takes care of it at the time of installation.

SDK Components

The contents of Local Edition SDK are:

- Scripts to create boiler plates for adapters
- Script to create distribution packages
- Examples
- Local Edition SDK libraries
- Gradle Build framework

- Javadocs for the libraries

```
|MasheryLocalSDK/
    |--createadapterproject.gradle
    |--pre-post-processor.template
    |--libs/
      |--org.slf4j.api_{version}.jar
      |--com.mashery.util_{version}.jar
      |--com.mashery.trafficmanager.sdk_{version}.jar
      |--com.mashery.http_{version}.jar
    |--docs/
      |--javadoc/
    |--create-adapter.sh
    |--create-adapter.bat
    |--build.subproject.gradle.template
    |--build-adapter.sh
    |--build-adapter.bat
    |--LICENSE
    |--authenticator-adapter.template
    |--gradlew.bat
    |--build.gradle
    |--common.gradle
    |--settings.gradle
    |--gradle.properties
    |--gradlew.sh
    |--gradlew.bat
    |--upgrade-sdk.sh
    |--upgrade-sdk.bat
    |--rollback-sdk.sh
    |--rollback-sdk.bat
    |--gradle/
      |--wrapper/
        |--gradle-wrapper.properties
        |--gradle-wrapper.jar
    |--examples/
```

System Requirements

The following table lists the system requirements.

Java SE 11 or higher version

Text editor or any Gradle enabled integrated development environment. For more information see, Using the Adapter SDK in an IDE.
Local Edition SDK

Quick Start

This section provides an overview of the basic workflow of Local Edition SDK.

Estimated time to get started using the steps in this section: 30 minutes.

Before you begin

Before getting started with developing the Local Edition environment, carry out the following steps:

1. Download the Local Edition artifact.
2. Locate the sdk.zip file (TIB_mash-local_***.tar.gz).
3. Extract contents of sdk.zip to a known location.

   > **Note:** Windows user can skip this step.

4. In the terminal or command prompt, navigate to the folder of the extracted contents.

   ```
   /home/user/MasheryLocalSDK$ chmod +x gradlew create-adapter.sh build-adapter.sh
   ```

Procedure

1. Creating an Adapter Project
2. Packaging the Adapter Project
3. Uploading the Build Adapters to -Installer Build Job
4. Building the Changing -TM Docker image with Customer-Provided Adapters or Processors
5. Configuring Endpoints for Processors

Step 1: Creating an Adapter Project

Describes how to create an adapter project.

Pre or Post Processor

Procedure

1. To create an adapter project, use the create-adapter script in the terminal or command prompt.
    - Command for Linux or MacOS X:

      ```
      /home/user/MasheryLocalSDK$ ./create-adapter.sh --project-name MyCustomAdpater --adapter-type TrafficEventListener --adapter-name MyCustomProcessor --package-name com.companyname.apim.adapter
      ```

    - Command for MS Windows:

      ```
      C:\Users\Administrator\MasheryLocalSDK> create-adapter.bat --project-name MyCustomAdapter --adapter-type TrafficEventListener --adpater-name MyCustomProcessor --package-name com.companyname.apim.adapter
      ```

2. Change the project name, adapter name, and package name.
   Running the script for the first time downloads the necessary gradle binaries.
3. Open the JAVA source MyCustomProcessor.java in a text editor and write the processing logic for pre and post processing.

Custom Authenticator

Procedure

1. To create an adapter project, use the create-adapter script in the terminal or command prompt.
    - Command for Linux or MacOS X:

      ```
      /home/user/MasheryLocalSDK$ ./create-adapter.sh --project-name MyCustomAdpater --adapter-type Authenticator --adapter-name MyCustomAuthenticator --package-name com.companyname.apim.adpater
      ```

    - Command for MS Windows:

```
C:\Users\Administrator\MasheryLocalSDK> create-adapter.bat --project-name MyCustomerAdapter
--adapter-type Authenticator --adapter-name MyCustomAuthenticator --package-name
com.companyname.apim.adapter
```

2. Change the project name, adapter name, and package name.
   Running the script for the first time downloads the necessary gradle binaries.
3. Open the JAVA source MyCustomProcessor.java in a text editor and implement authenticator logic.

> **Note:** For more information to edit in an IDE, see Using the Adapter SDK in and IDE.

Step 2: Packaging the Adapter

The build adapter script is used for compiling and packaging the adapter.

To package the adapter:

Procedure

1. Run the build script.
   - Command for Linux/MacOS :

     ```
     /home/user/MasheryLocalSDK$ ./build-adapter.sh
     ```

   - Command for Windows :

     ```
     C:\Users\Administrator\MasheryLocalSDK> build-adapter.bat
     ```

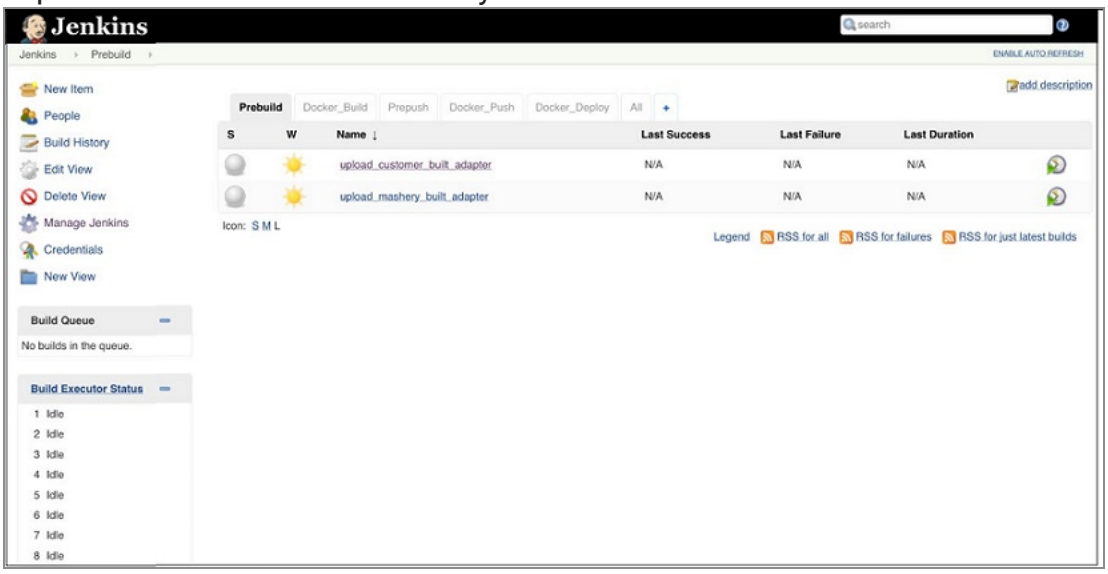
   The build script reports the compilation errors. In case of any reported errors rerun the build script.
2. The script generates an archive file under **MasheryLocalSDK > dist folder** with the name *tml-mashery-customer-extension.zip*. Upload this adapter to the Local Edition installer.

Step 3: Uploading the Build Adapters to Local Edition-Installer Build Job

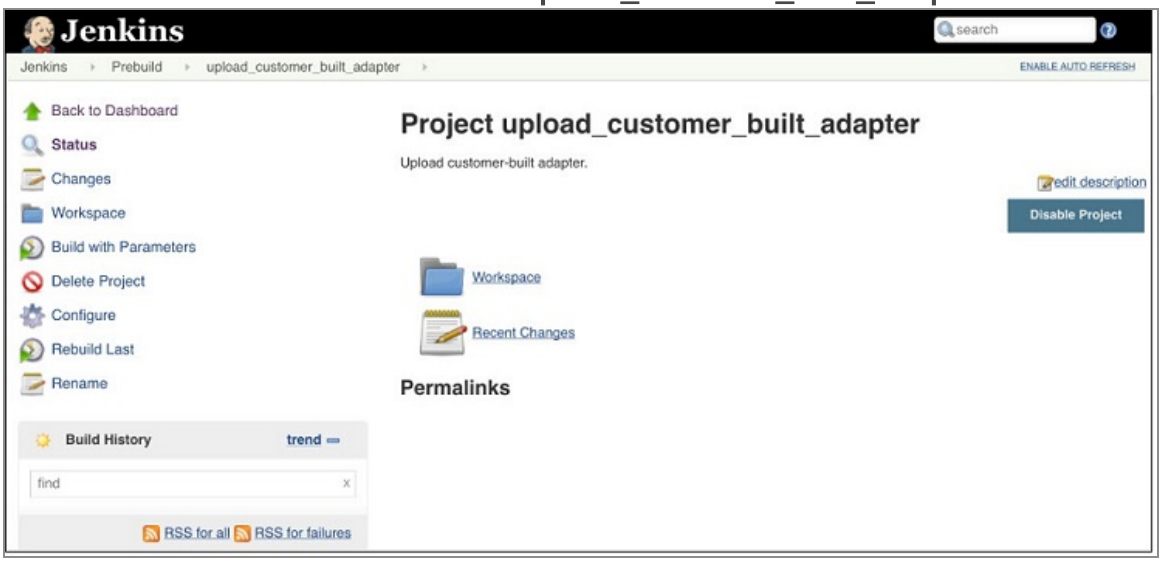
To upload a build adapter:

Procedure

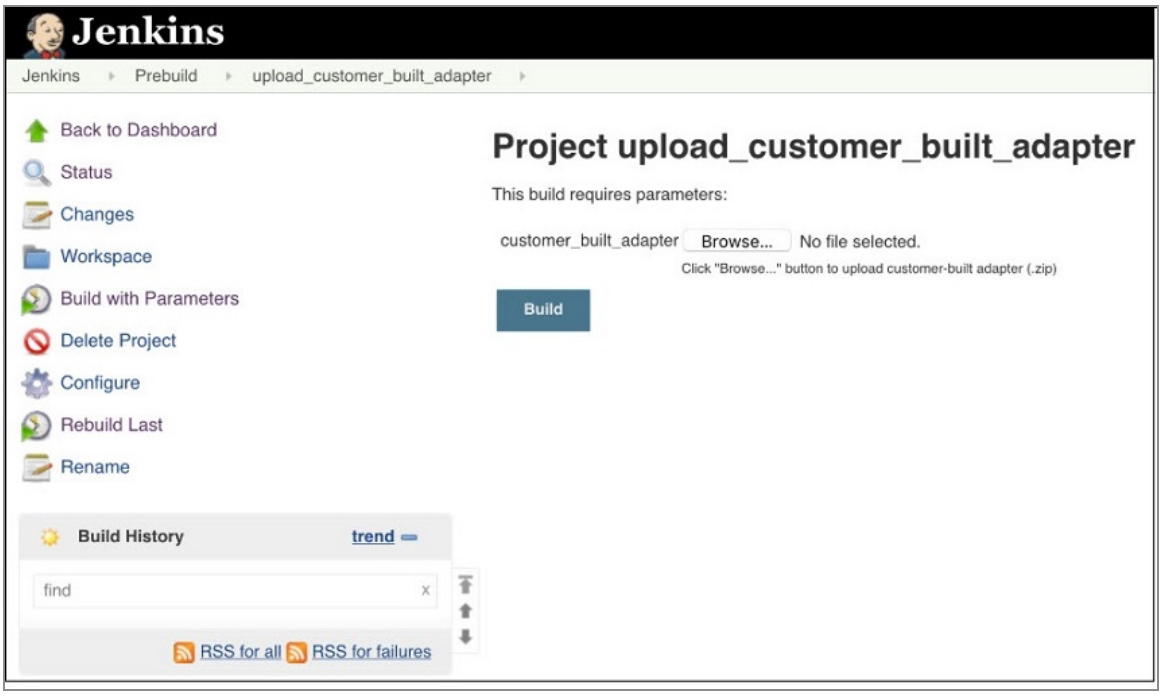
1. Open the installer's Jenkins in any browser.



2. Click the **Prebuild** tab and then click **upload_customer_built_adapter**.



3. On the left side of the window, click **Build With Parameters**, then click **Browse...**. Select the build adapter file.



4. Click **Build** to upload the adapter to Jenkins server.

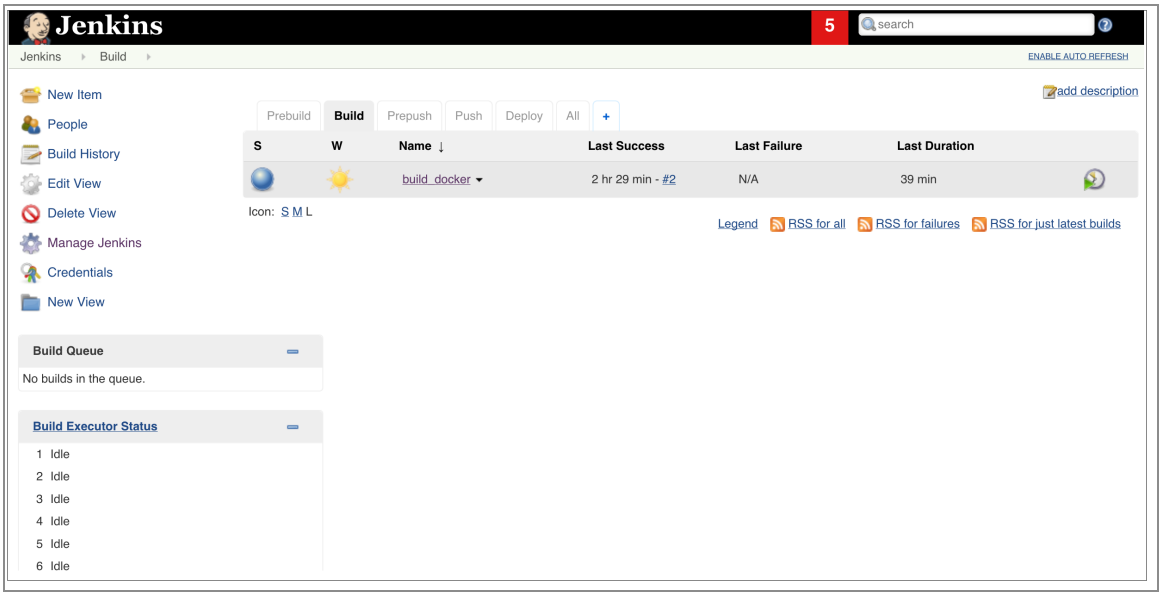
> **Note:**
>
> - A customer-built adapter is built by a customer using Local Edition SDK and it is a zip file.
> - The zip file is uploaded to /var/jenkins_home/userContent/proxy-extension in the tml-installer container.

Step 4: Building the Changing Local Edition-TM Docker image with Customer-Provided Adapters or Processors

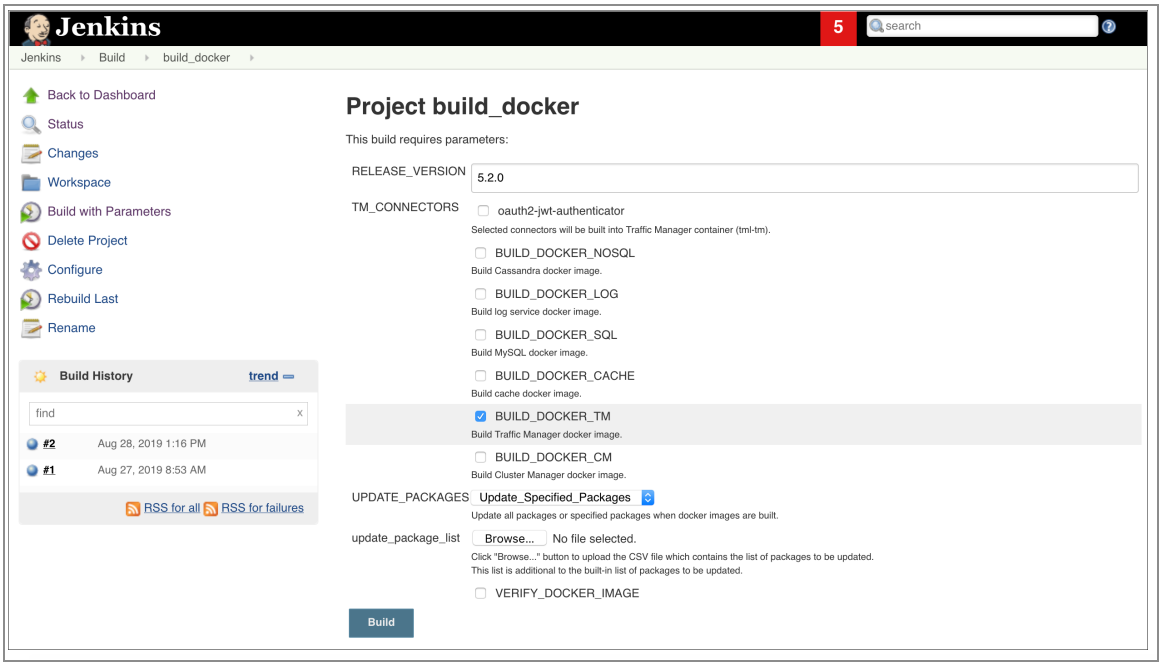
This section describes how to build the Local Edition-TM Docker image with adapters built by a customer.

1. After uploading the Adapter/Processor to Local Edition Local-Installer job, build the new Local Edition-TM Docker image.

2. Open the installer's Jenkins in any browser and click the **Build** tab.



3. Click on **build_docker** job and Click on **build with parameters**.



4. Once build is completed, the Docker images are available. For example,

```
tml-cm          v5.2.0.1    d1552b884be7    31 hours ago    926MB
tml-tm          v5.2.0.1    42b0077f22f8    31 hours ago    762MB
tml-cache       v5.2.0.1    182ebcbdd53d    31 hours ago    681MB
tml-sql         v5.2.0.1    1577c52c75df    31 hours ago    1.64GB
tml-log         v5.2.0.1    cc60d3310aaa    31 hours ago    752MB
tml-nosql       v5.2.0.1    7d8f2e75b89b    31 hours ago    650MB
```

Step 5: Configuring Endpoints for Processors

The endpoints of processors can be either Tethered or Untethered.

Untethered

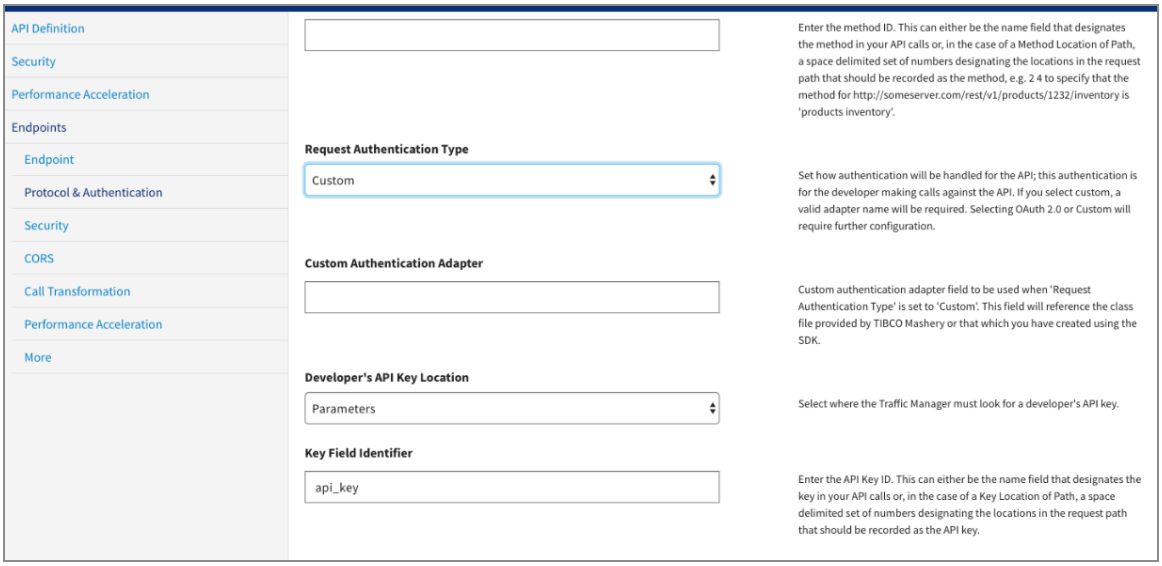
The Untethered configuration is a two-stage process:

1. Navigate to Config Manager GUI from your setup.
2. Navigate to API's endpoint.

Procedure

To register a custom authenticator:

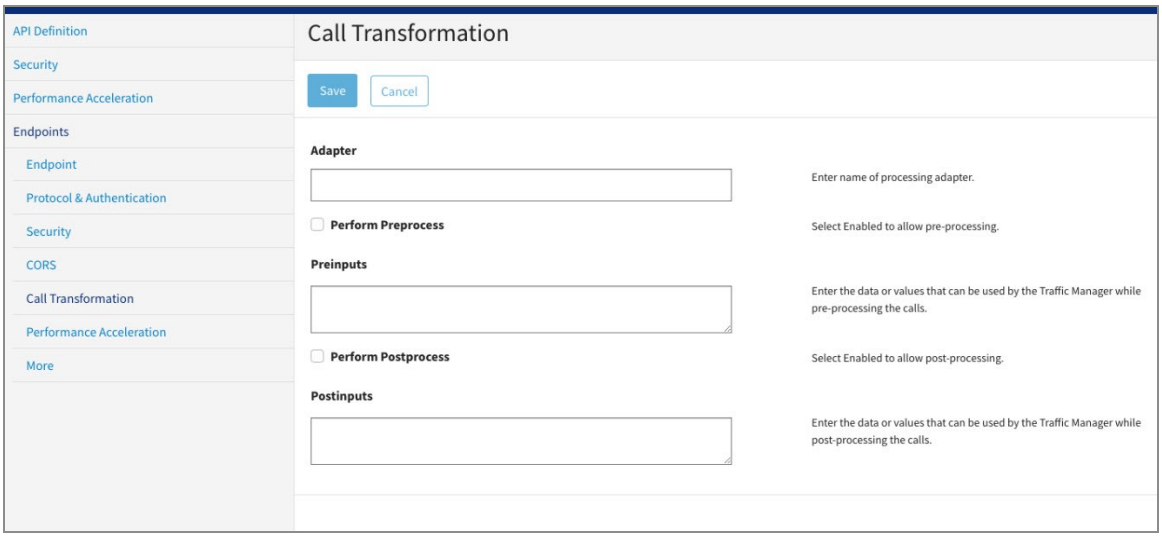
1. In the Configuration Manager, click **Endpoints** tab and then click **Protocol & Authentication**.
2. Select **Request Authentication Type** as **Custom**



3. In the **Custom Authentication Adapter** field, provide the processor bean's name from adapter com.companyname.mashery.adapter.MyCustomAuthenticator.

To register a processor:

4. In the Configuration Manager, select **Endpoints** tab and then click **Call Transformation**.
5. In the **Adapter** field, provide the processor bean's name from adapter com.companyname.apim.adapter.



6. Select the appropriate check box for performing the transformation Preprocess or Postprocess.

Tethered

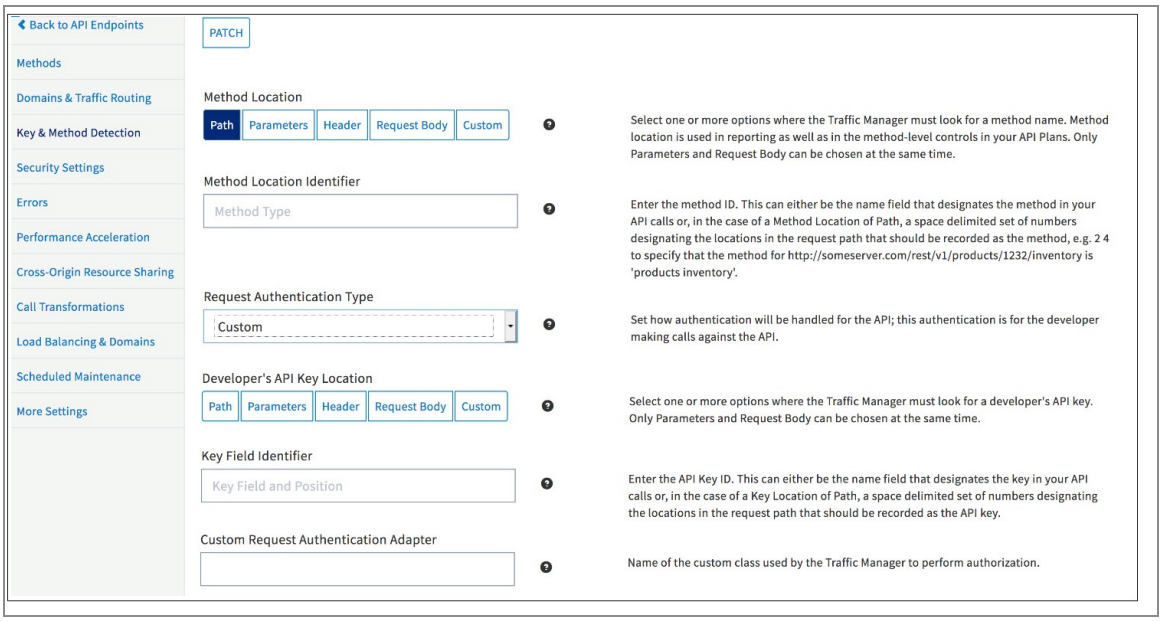
The tethered configuration is a two-stage process:

1. Navigate to API Design > Select an API.
2. Navigate to API's endpoint.

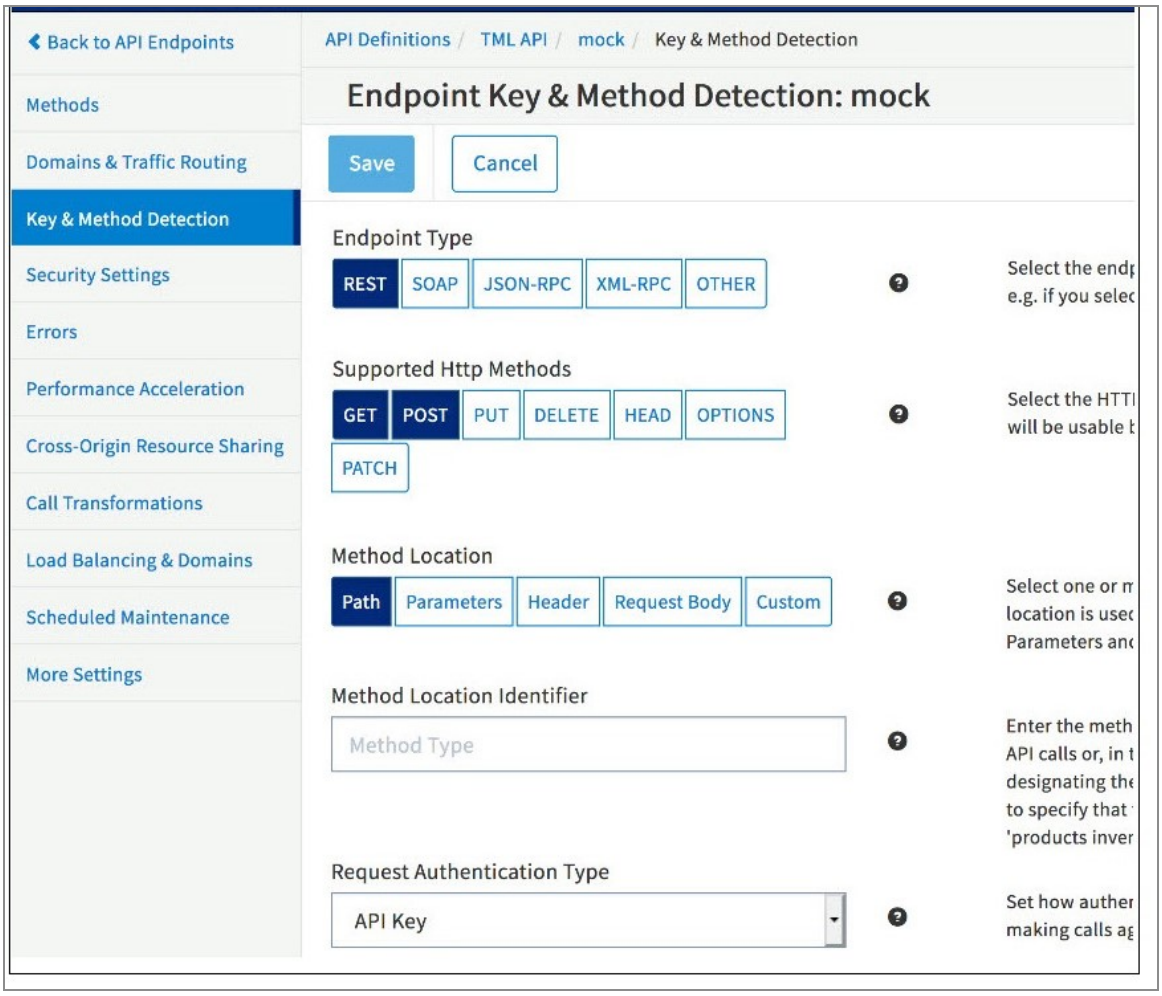
Registering a Custom Authenticator

Procedure

1. In the Configuration Manager, click **Key and Method detection** tab.
2. Select **Request Authentication Type** as **Custom**.

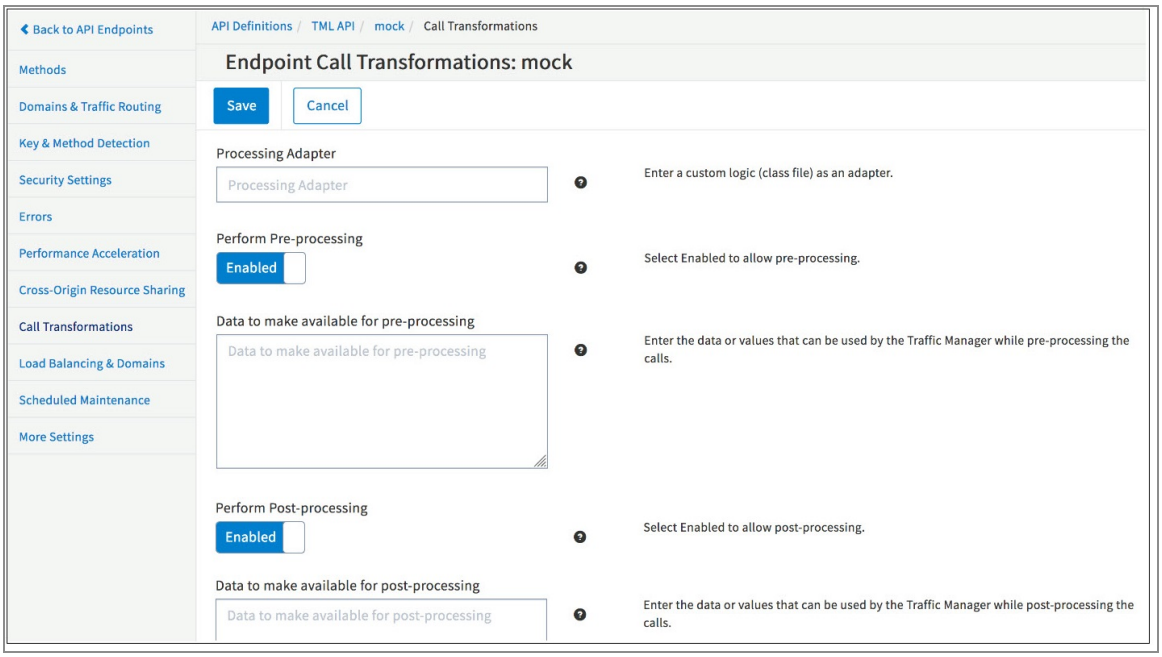


3. In the Customer Request Authentication Adapter field provide the processor bean's name from adapter com.companyname.apim.adapter.MyCustomAuthenticator.



Registering a Processor

4. In the Configuration Manager, select **Call Transformation** tab.
5. In the **Adapter** field provide the processor bean's name from adapter **com.companyname.apim.adapter.MyCustomProcessor** .



6. Select the appropriate check box for performing the transformation Preprocess or Postprocess.

Upgrading the Local Edition SDK 5.2 or 5.3 or 5.4 to Local Edition SDK 5.5

These instructions apply if you are upgrading from Local Edition 5.2 or 5.3 or 5.4 to Local Edition 5.5. Pre-built adapters can also be upgraded from earlier versions of Local Edition SDK.

Procedure

1. Extract the TIB_mash-local_5.5.0.GA*.tar.gz to your desired location and locate the sdk.zip file.
2. Extract contents of sdk.zip to a known location.

   Note: Windows users can skip this step.

3. Navigate to the location where you have extracted the sdk.zip file.

   In the command prompt, input the following:

   a. cd <extract location>/MasheryLocalSDK

   b. chmod +x upgrade-sdk.sh

4. In the command prompt, input the path of the extracted contents of Local Edition 5.2 or Local Edition 5.3 or Local Edition 5.4.
   - For windows:

     ```
     C:\Users\Administrator\MasheryLocalSDK> upgrade-sdk.bat -d "<path of TML 5.2 MasheryLocalSDK folder>"
     ```

     Or

     ```
     C:\Users\Administrator\MasheryLocalSDK> upgrade-sdk.bat -d "<path of TML 5.3 MasheryLocalSDK folder>"
     ```

     Or

```
C:\Users\Administrator\MasheryLocalSDK> upgrade-sdk.bat -d "<path of TML 5.4 MasheryLocalSDK folder>"
```

- For Linux/MacOS:

```
/home/user/MasheryLocalSDK$ ./upgrade-sdk.sh -d "<path of TML 5.2 MasheryLocalSDK folder>"
```

Or

```
/home/user/MasheryLocalSDK$ ./upgrade-sdk.sh -d "<path of TML 5.3 MasheryLocalSDK folder>"
```

Or

```
/home/user/MasheryLocalSDK$ ./upgrade-sdk.sh -d "<path of TML 5.4 MasheryLocalSDK folder>"
```

What to do next

The libraries, documents and example folder will get updated. You will be required to rebuild the processors and other extensions.

To rebuild the adapters:

```
build-adapter.sh
```

```
build-adapter.bat
```

Downgrading Local Edition SDK to an Earlier Version

The operation to rollback an upgrade to an earlier version of Local Edition SDK are similar to the upgrade path.

Input the following as applicable.

- For windows:

```
C:\Users\Administrator\MasheryLocalSDK>./rollback-sdk.bat--destination-sdk-path<OLDER_SDK_PATH>--target-adapters<ADAPTER_NAME_1#ADAPTER_NAME_2>
```

- For Linux/MacOS:

```
/home/user/MasheryLocalSDK$./rollback-sdk.sh--destination-sdk-path<OLDER_SDK_PATH>--target-adapters<ADAPTER_NAME_1#ADAPTER_NAME_2>
```

Note: Target adapter is optional.

Configuring Network Proxy for Boomi Cloud™ API Management - Local Edition SDK

Incase your internet access is governed by a network proxy, complete the following steps to configure the SDK using proxy.

Procedure

1. Edit the Gradle settings file at <extract-location>/MasheryLocalSDK/gradle.properties.
2. Configure HTTPS URI.

   ```
   systemProp.https.proxyHost=www.somehost.org
   systemProp.https.proxyPort=proxy_port
   ```

3. Configuration proxy for HTTP URI.

   ```
   systemProp.http.proxyHost=www.somehost.org
   systemProp.http.proxyPort=proxy_port
   ```

   For more information on advance configuration like credentials and no proxy settings, see Accessing the Web Through a Proxy.

Adapter SDK Package

The Adapter SDK defines the Traffic Manager domain model, tools and APIs and provides extension points to inject custom code in the processing of a call made to the Traffic Manager.

> **Note:** The DIY SDK adapters need to be coded and compiled using JDK 1.6 or any lower version.

The Adapter SDK package contains the following:

- Boomi Cloud™ API Management - Local Edition Domain SDK
- Boomi Cloud™ API Management - Local Edition Infrastructure SDK

Boomi Cloud™ API Management - Local Edition Domain SDK

Boomi Cloud™ API Management - Local Edition SDK packaged in com.mashery.trafficmanager.sdk identifies the traffic manager SDK and provides access to the Local Edition domain model which includes key objects such as Members, Applications, Developer Classes, Keys, Packages.

Boomi Cloud™ API Management - Local Edition Infrastructure SDK

Boomi Cloud™ API Management - Local Edition Infrastructure SDK provides the ability to handle infrastructure features and contains the following:

- **Boomi Cloud™ API Management - Local Edition HTTP Provider**

The HTTP provider packaged as com.mashery.http provides HTTP Request/Response processing capability and tools to manipulate the HTTP Request, Response, their content and headers.

- **Boomi Cloud™ API Management - Local Edition Utility**

  The utility packaged as com.mashery.util provides utility code which handles frequently occurring logic such as string manipulations, caching, specialized collection handling, and logging.

Developing Processors and Authenticators

This section provides the details of the SDK domain model, the objects that can be used in custom processors and authenticators. It also guides about advanced use cases in developing and packaging extensions.

The following are covered:

- SDK Domain Model
- Importing Existing Adapters
- Developing Multiple Adapters
- Using the Adapter SDK in an IDE
- Adding Third-Party Libraries in an Adapter

SDK Domain Model

The Traffic Manager domain model defines the elements of the Traffic Manager runtime.

The following table highlights some of the key elements:

Element	Description	Usage
User	A user or member subscribing to APIs and accesses the APIs.	com.mashery.trafficmanager.model.User
API	An API represents the service definition. A service definition has endpoints defined for it.	com.mashery.trafficmanager.model.API

Element	Description	Usage
Endpoint	An Endpoint is a central resource of an API managed within Local Edition. It is a collection of configuration options that defines the inbound and outbound URI's, rules, transformations, cache control, security, etc. of a unique pathway of your API. An Endpoint is specialized as either an API Endpoint or a Plan Endpoint. This specialization provides context to whether or not the Endpoint is being used as part of a Plan or not.	• Generic endpoint entity representation: com.mashery.trafficmanager.model.Endpoint • API endpoint entity representation: com.mashery.trafficmanager.model.APIEndpoint • Plan endpoint entity representation: com.mashery.trafficmanager.model.PlanEndpoint
Method	A method is a function that can be called on an endpoint and represents the method currently being accessed or requested from the API request. A method could have rate and throttle limits specified on it to dictate the volume of calls made using a specific key to that method.	• Generic method entity representation: com.mashery.trafficmanager.model.Method • API method entity representation: com.mashery.trafficmanager.model.APIMethod • Plan method entity representation: com.mashery.trafficmanager.model.PlanMethod

Element	Description	Usage
	A Method is specialized as either an API Method or Plan Method. The specialization provides context to whether or not the Method belong to a Plan.	
Package	A Package is a mechanism to bundle or group API capability allowing the API Manager to then offer these capabilities to customers/users based on various access levels and price points. A Package represents a group of Plans.	com.mashery.trafficmanager.model.Package
Plan	A Plan is a collection of API endpoints, methods and response filters to group functionality so that API Product Managers can manage access control and provide access to appropriate Plans to different users.	com.mashery.trafficmanager.model.Plan
API Call	The API Call object is the complete transaction of the	com.mashery.trafficmanager.model.core.APICall

Element	Description	Usage
	incoming request received by the Traffic Manager and the outgoing response as processed by the Traffic Manager. It provides an entry point into all other entities used in the execution of the request.	
Key	A key is an opaque string allowing a developer to access the API functionality. A key has rate and throttle controls defined on it and dictates the volume of calls that can be made to the API by the caller. A Key can be specialized as an API key or Package Key. This specialization provides context to whether the key provides access to an API or a specific Plan in a Package.	• Generic key entity representation: com.mashery.trafficmanager.model.Key • API key entity representation: com.mashery.trafficmanager.model.APIKey • Package key entity representation: com.mashery.trafficmanager.model.PackageKey
Application	An application is a developer artifact that is registered by the developer when he subscribes to an API	com.mashery.trafficmanager.model.Application

Element	Description	Usage
	or a Package.	
Rate Constraint	A Rate Constraint specifies how the amount of traffic is managed by limiting the number of calls per a time period (hours, days, months) that may be received.	com.mashery.trafficmanager.model.RateConstraint
Throttle Constraint	A Throttle Constraint specifies how the velocity of traffic is managed by limiting the number of calls per second that may be received.	com.mashery.trafficmanager.model.ThrottleConstraint
Customer Site	A customer specific area configured through the developer portal.	com.mashery.trafficmanager.model.CustomerSite

Extended Attributes

The traffic manager model allows defining name-value pairs on different levels of the model. The levels are identified here:

- Application
- Customer Site
- Key (both API Key and Package Key)
- Package
- Plan
- User

For more information, see Accessing and Using Extended Attributes.

Pre and Post Processor Extension Points

This version of the SDK allows extensions for Processors only. This means that only pre and post processing of requests prior to invocation of the target host are allowed.

Listener Pattern

The extension API leverages a listener pattern to deliver callbacks to extension points to allow injecting custom logic.

A call made to the traffic manager is an invocation to a series of tasks. Each step in the workflow accomplishes a specific task to fulfill the call. The current API release only allows customization of the tasks prior to invoking the API server (pre-process) and post receipt of the response from the API server (post-process). The callback API handling these extensions is called a Processor.

The pre-process step allows a processor to receive a fully-formed HTTP request targeted to the API server. The processor is allowed to alter the headers or the body of the request prior to the request being made to the server. Upon completion of the request and receiving the response the Traffic Manager allows the processor to alter the response content and headers prior to the response flowing back through a series of exit tasks out to the client.

Event Types and Event

The transition of the call from one task to the next is triggered through events and an event is delivered to any subscriber interested in receiving the event. The SDK supports two event-types which are delivered synchronously:

- **Pre-Process Event type**: This event is used to trigger any pre-process task.
- **Post-Process Event type**: This event is used to trigger any post-process task.
- **Authentication Event type**: This event is used to trigger any custom authentication.

The subscribers in this case will be Processors registered in a specific manner with the Traffic Manager API.

Event Listener API

The Traffic Manager SDK provides the following interface and is implemented by custom processors to receive Processor Events.

```
package com.mashery.trafficmanager.event.listener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
/*** Event listener interface which is implemented by listeners which wish to handle Traffic events.
Traffic events will be delivered via this callback synchronously to handlers implementing the
interface.
The implementers of this interface subscribe to events via annotations. E.g. Processor events need
to handle events by using annotations in the com.mashery.proxy.sdk.event.processor.annotation */
public interface TrafficEventListener {
  /*** The event is delivered to this API @param event*/
  void handleEvent(TrafficEvent event);
}
```

Importing Existing Adapters

Adapters developed using versions prior to Local Edition 5.x SDK can be imported in the new version of the SDK.
To import existing adapters:

Procedure

1. Create a new adapter project using the create-adapter.sh/bat script.

   > **Note:** The project name and other package name must be same as the existing adapter project that you want to import.

2. Copy the existing source package into the newly created sub-project.
3. Copy the third-party libraries into the lib folder of the sub-project.

   > **Note:** For an existing adapter being developed as a maven-based project, provide the gradle form of dependencies in the sub-project's build.gradle file. For more information, see the Referring to Third-Party Libraries topic.

Developing and Packaging Multiple Adapters

A project in SDK terminology is a bundle that will be packaged and deployed to the traffic manager, and will show up as an OSGi bundle in the traffic manager.

A bundle can either package a single adapter or multiple adapters per user choice. Adapters' boiler plate code can be generated using the adapter creation script.

Developing Multiple Adapters

- One project per adapter

```
./create-adapter.sh -p <New adapter project name> -c <New adapter full package name> -a <New Adapter class name>
```

Example

```
./create-adapter.sh -p DemoProject1 -c com.tibco.apim.examples1 -a DemoAdapter1
```

```
./create-adapter.sh -p DemoProject2 -c com.tibco.apim.examples2 -a DemoAdapter2
```

Running the above two commands will create two new projects under the extract location <extract location>/MasheryLocalSDK/ with one adapter class each.

- Multiple adapters per project
  1. Create a new project with a new adapter class.

     ```
     ./create-adapter.sh -p <New adapter project name> -c <New adapter full package name> -a <New Adapter class name>
     ```

  2. Create an adapter class in same project.

     ```
     ./create-adapter.sh -p <Existing adapter project name> -c <Existing/New adapter full package name> -a <New Adapter class name>
     ```

  Example

  ```
  ./create-adapter.sh -p DemoProject1 -c com.tibco.apim.examples1 -a DemoAdapter1
  ```

```
./create-adapter.sh -p DemoProject1 -c com.tibco.apim.examples1 -a DemoAdapter2
```

Running these two commands creates a Demo project with two adapter classes in package. com.tibco.apim.examples1.

- Bundling Multiple adapters

  Use the build-adapters.sh script to compile; build jars and bundle them as a deployable artifact. The script ensures that the adapters are packaged properly.

Using the Adapter SDK in an IDE

You can create adapters using the following IDE:

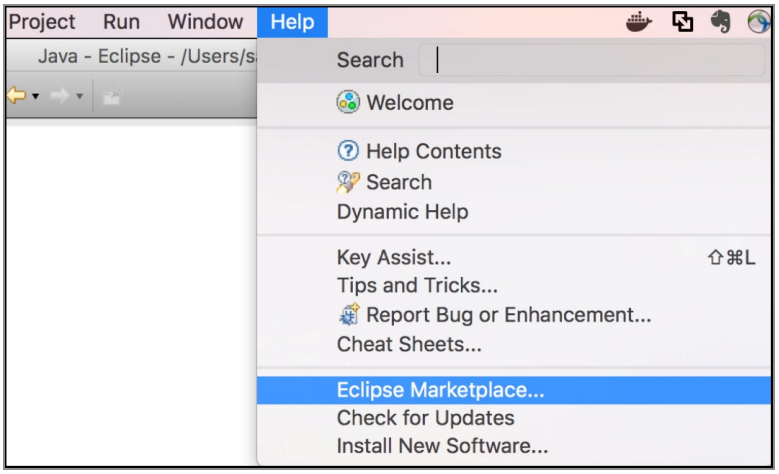
- Eclipse
- IntelliJ Idea
- Apache Netbeans

Creating an Adapter using Eclipse

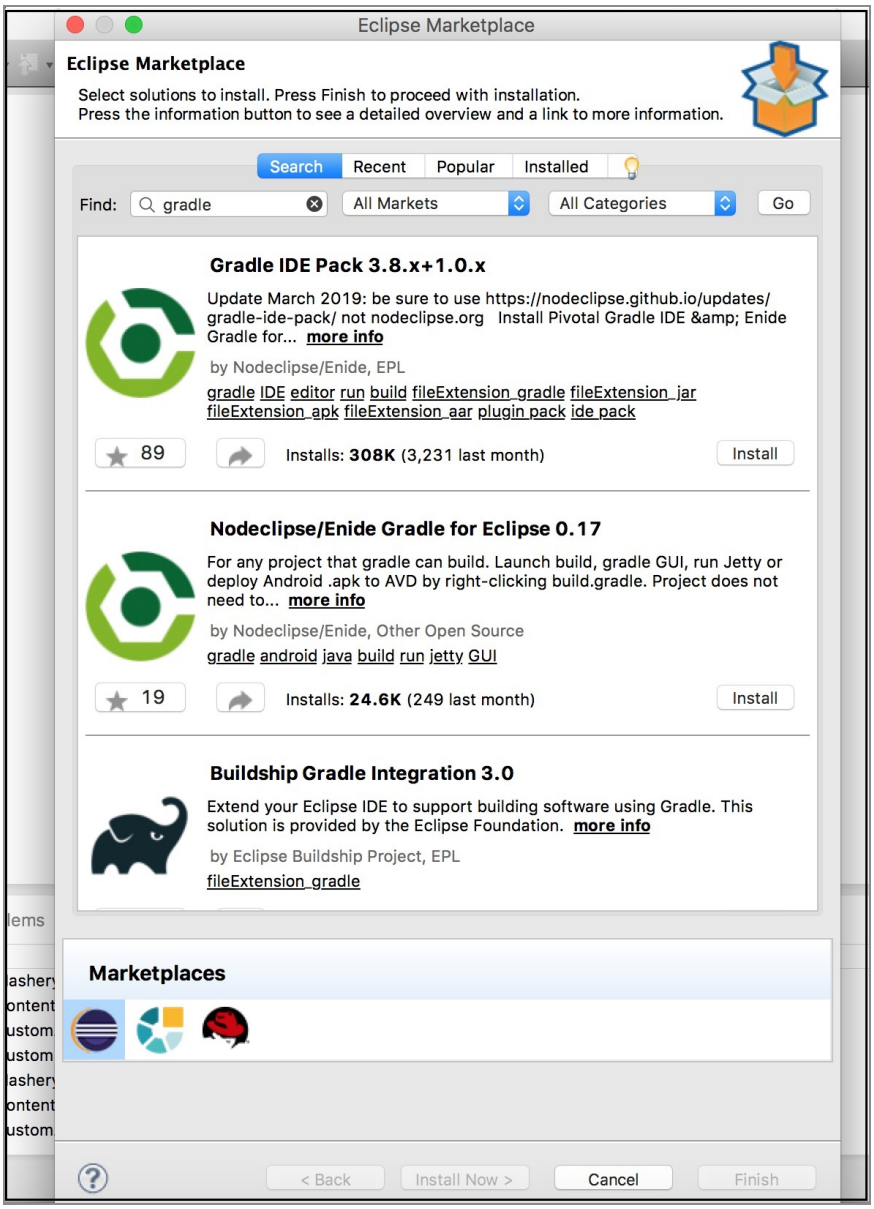
Preparing Eclipse:

Procedure

1. On the Eclipse user interface, point the **Help** tab and click **Eclipse Marketplace**.



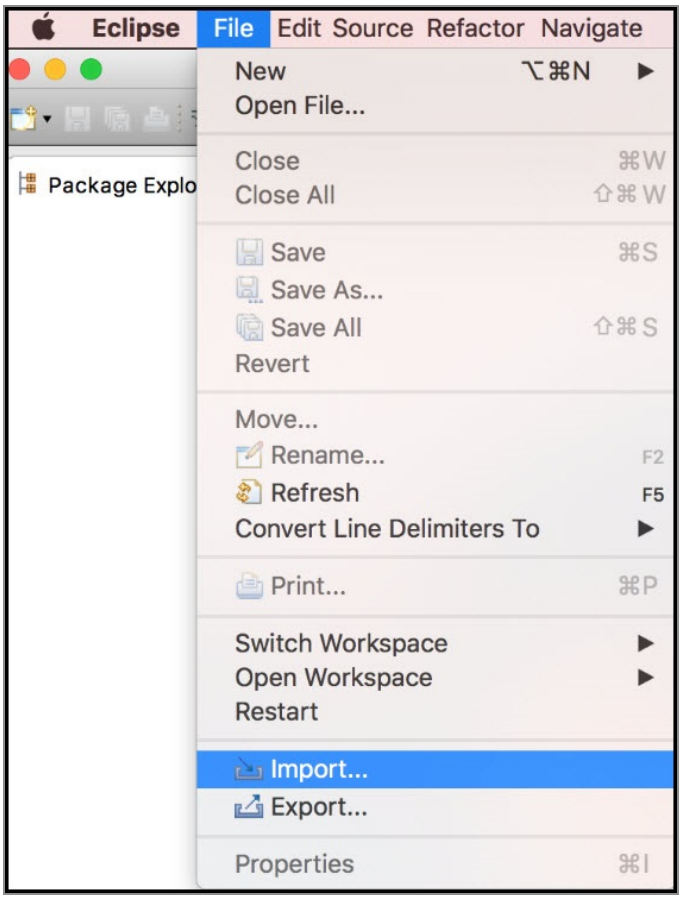
2. Search Gradle and install.



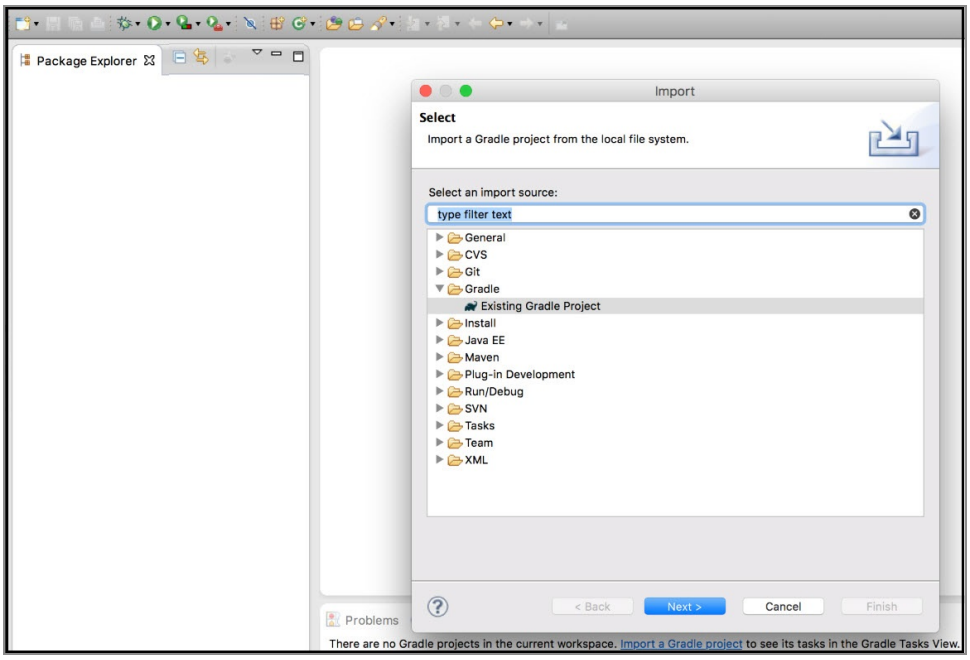
Importing the Local Edition SDK into Eclipse IDE

Procedure

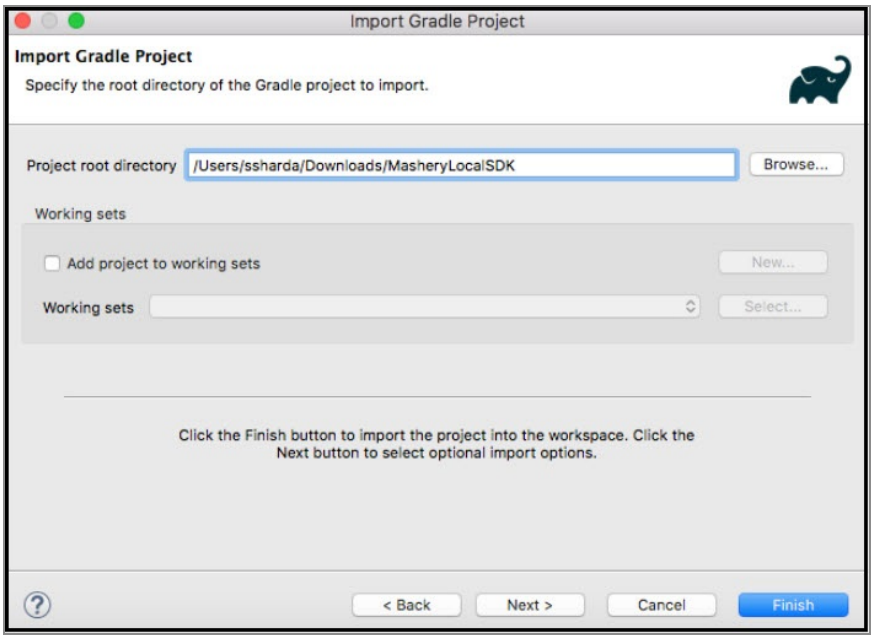
1. On the **File** tab, click **Import**.



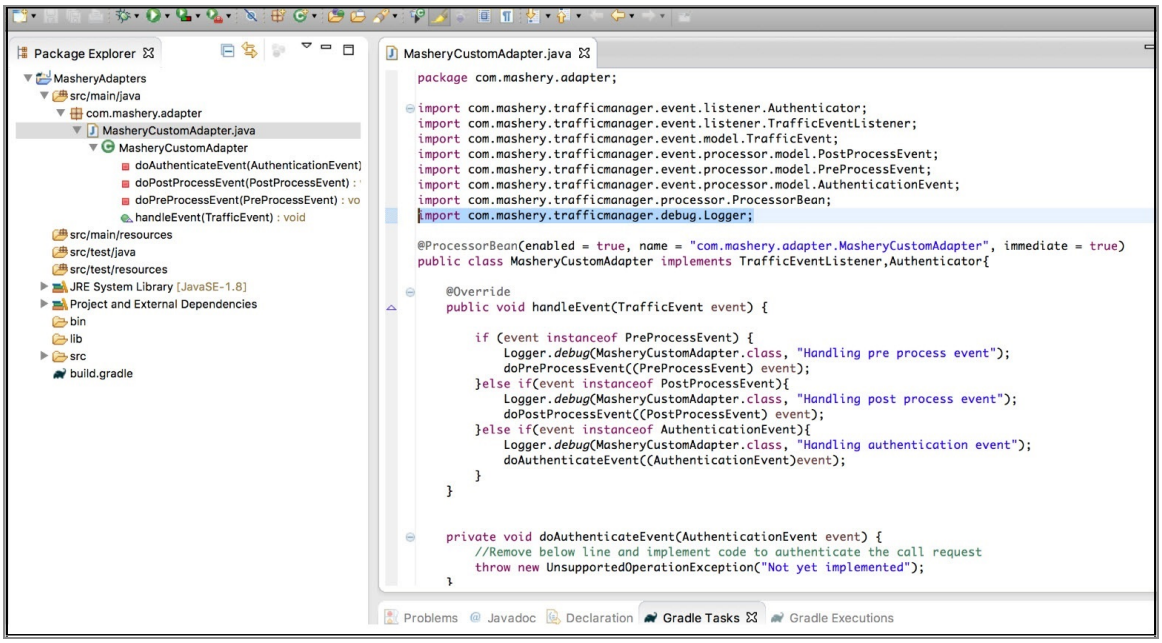
2. In the **Import** dialog box, select the Gradle project.



3. The **Import Gradle Project** wizard opens, follow through till the end. Close the wizard by clicking **Finish**.



4. The explorer shows the uploaded project with its contents.



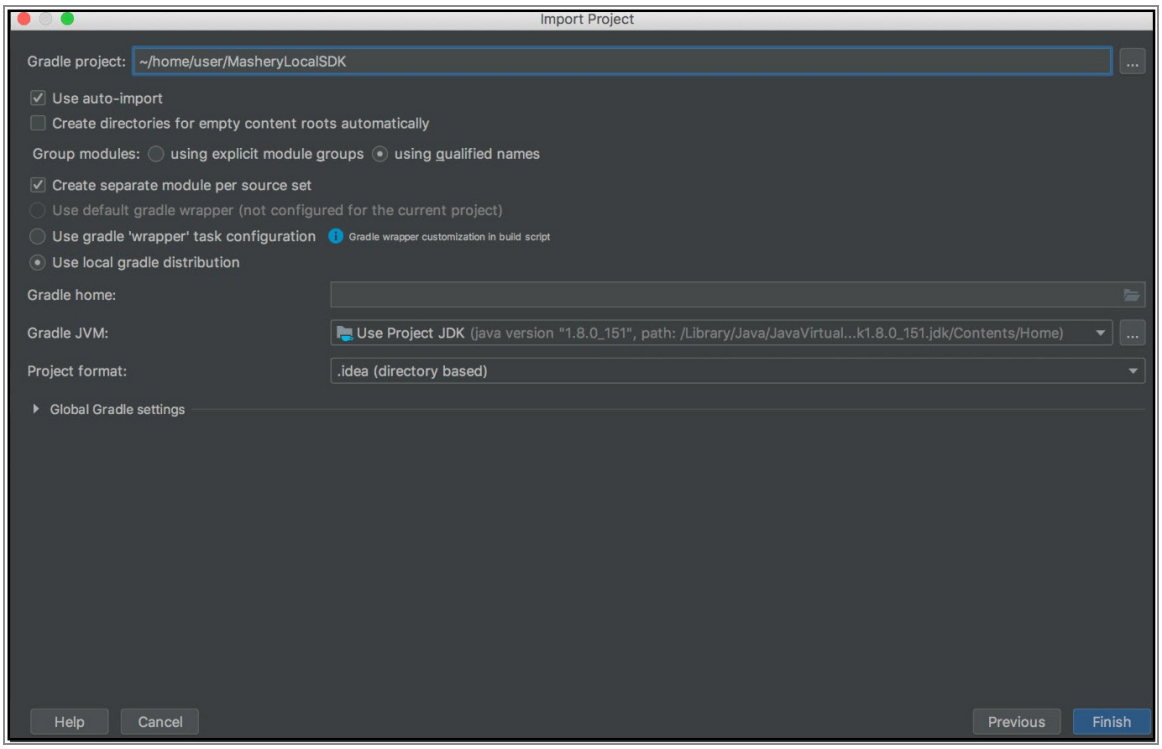
Creating an Adapter using IntelliJ IDEA

Preparing IntelliJ IDEA:

Procedure

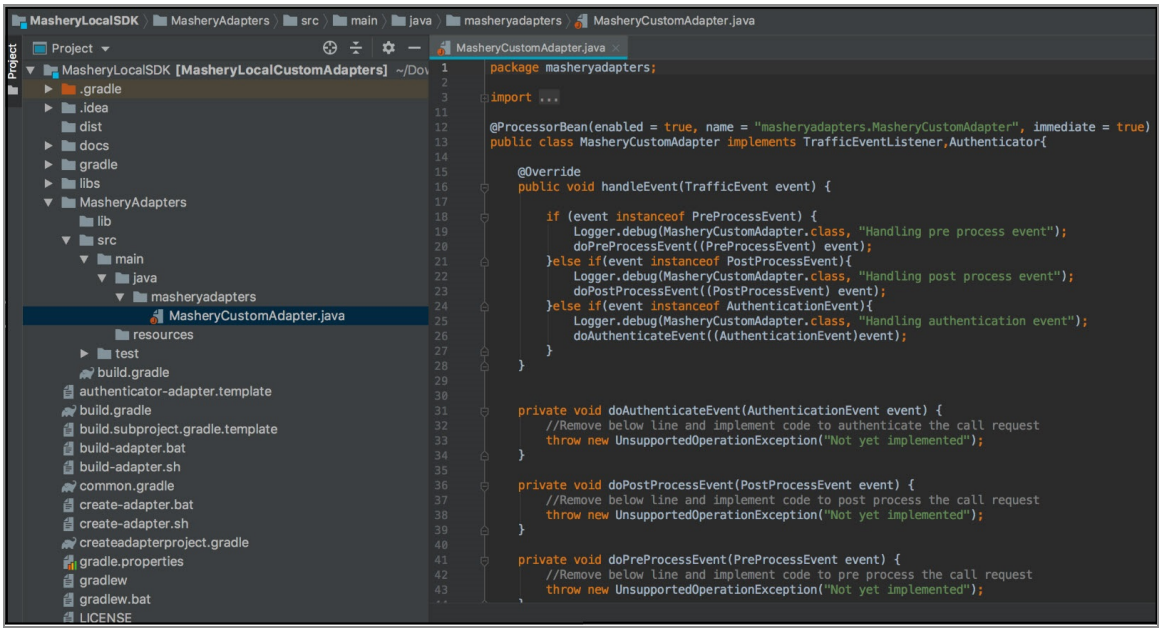
1. Open IntelliJ IDEA and click **Import Project**.

2. In the **Import Project** dialog box, navigate to the Local Edition SDK folder as input for **Gradle project**.



Select the Gradle options as shown in the image.

3. Click **Finish** to upload the project. The content of the project can be seen in the project view.



Copy required third party libraries in the lib folder of the sub project.

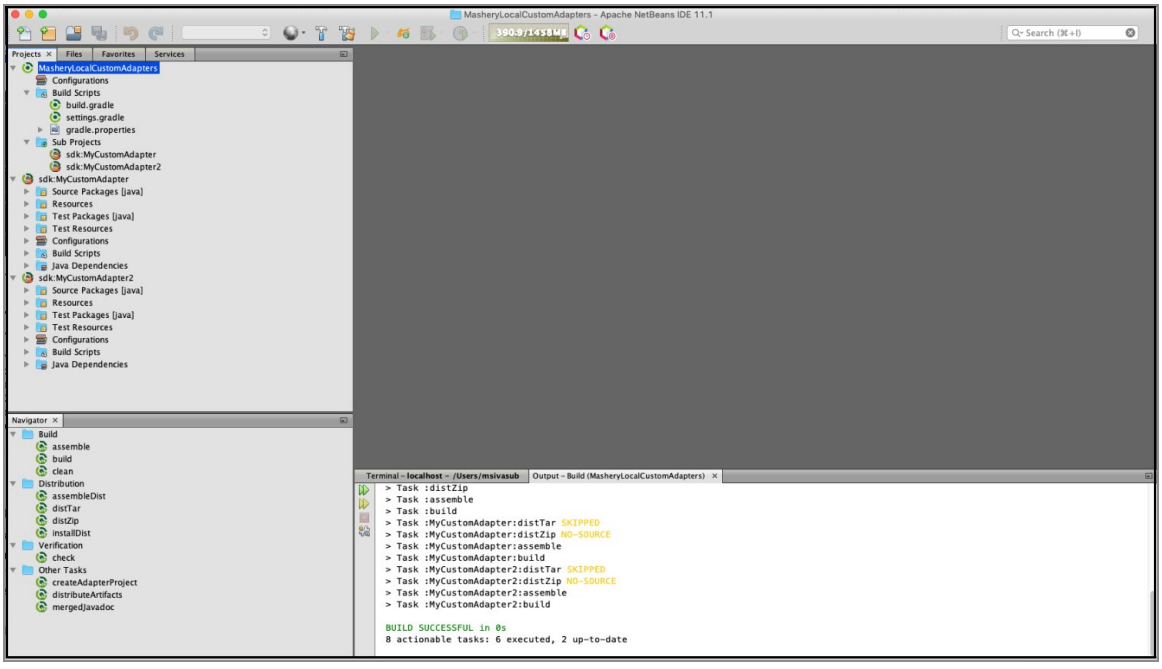
Creating an Adapter using Apache NetBeans

To create an adapter using Apache NetBeans:

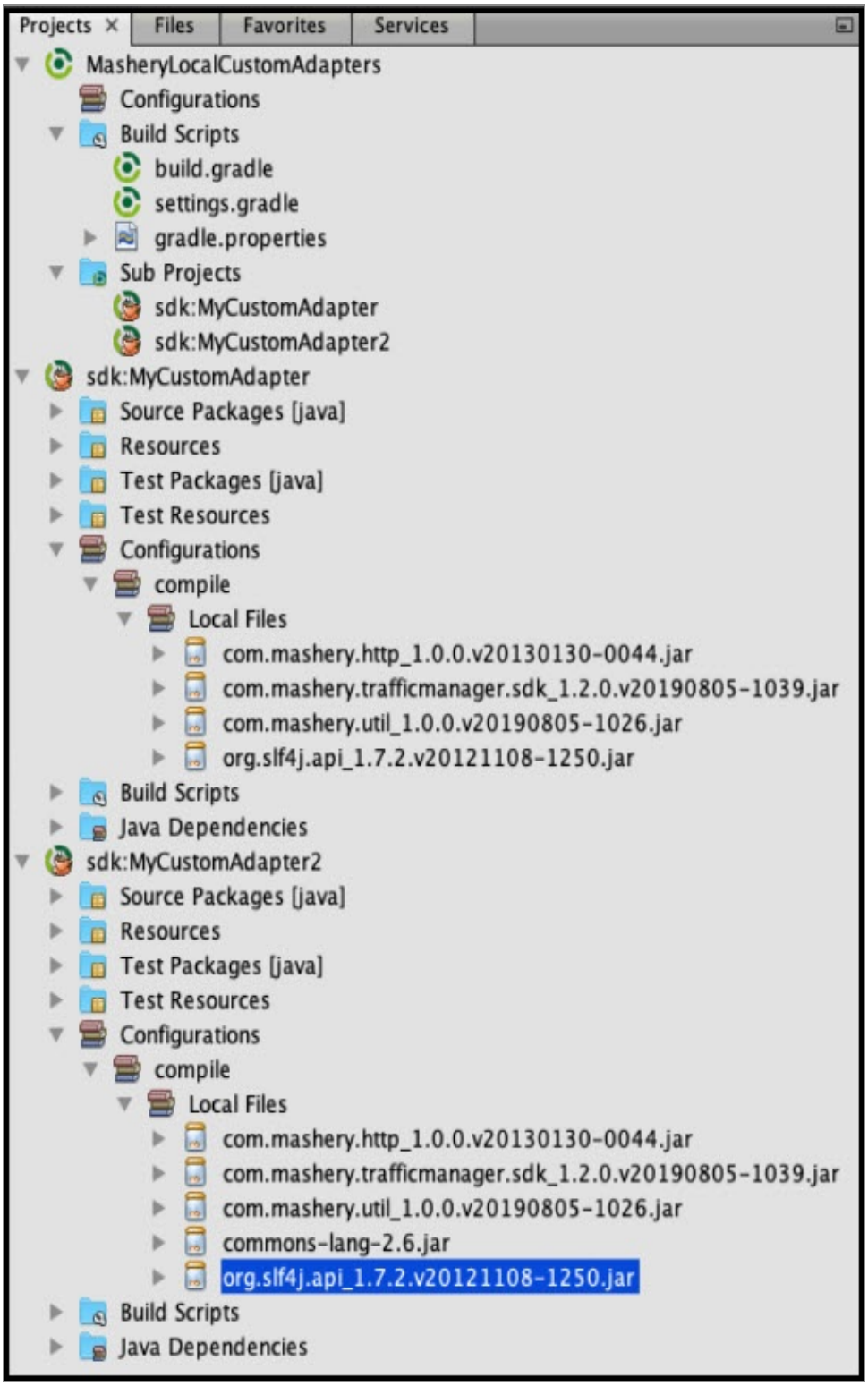
Procedure

1. Open the **Apache NetBeans** software.
2. On the **File** menu, click **Open Project**.
   - For Windows, use: Ctrl+Shift+O
   - For Mac, use: CMD+Shift+O
3. Navigate to the Local Edition SDK folder. It will be annotated with  . Click **Open**.

   The primer build will take few minutes to compile.
4. On the Project node, right click and select **Clean and Build**.
5. On the Project node, right click and select **Open Required projects** and click **Open all projects**.
6. The newly created project is now visible.



The libraries for each of the sub projects can be viewed by expanding the configuration node.

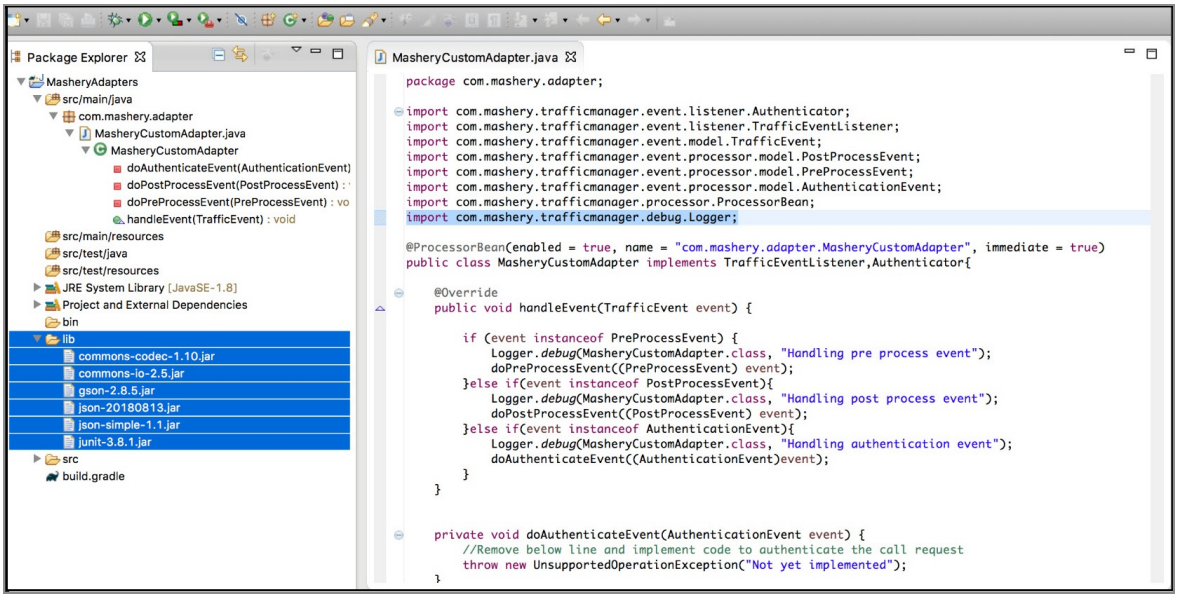

Projects
Files
Favorites
Services
MasheryLocalCustomAdapters
Configurations
Build Scripts
build.gradle
settings.gradle
gradle.properties
Sub Projects
sdk:MyCustomAdapter
sdk:MyCustomAdapter2
sdk:MyCustomAdapter
Source Packages [java]
Resources
Test Packages [java]
Test Resources
Configurations
compile
Local Files
com.mashery.http_1.0.0.v20130130-0044.jar
com.mashery.trafficmanager.sdk_1.2.0.v20190805-1039.jar
com.mashery.util_1.0.0.v20190805-1026.jar
org.slf4j.api_1.7.2.v20121108-1250.jar
Build Scripts
Java Dependencies
sdk:MyCustomAdapter2
Source Packages [java]
Resources
Test Packages [java]
Test Resources
Configurations
compile
Local Files
com.mashery.http_1.0.0.v20130130-0044.jar
com.mashery.trafficmanager.sdk_1.2.0.v20190805-1039.jar
com.mashery.util_1.0.0.v20190805-1026.jar
commons-lang-2.6.jar
org.slf4j.api_1.7.2.v20121108-1250.jar
Build Scripts
Java Dependencies

Adding Third-Party Libraries in an Adapter

You can use third-party libraries in an adapter.
To enable third-party libraries in classpath and run time:

Procedure

1. Copy the third party jars into the adapter sub project/lib folder.

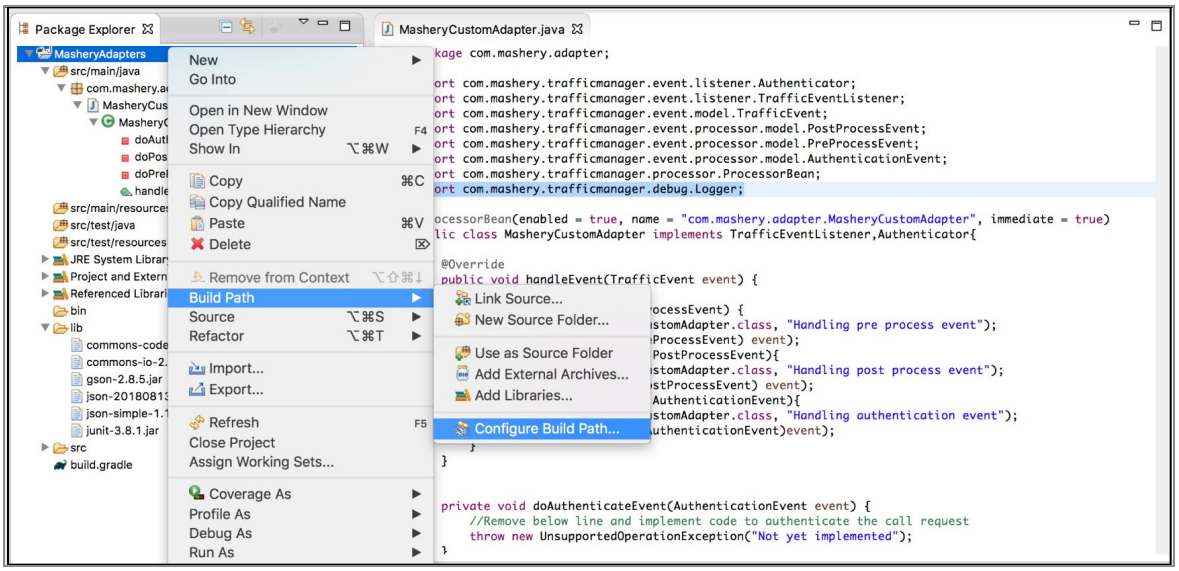


2. In IntelliJ IDEA and Apache NetBeans, rebuild the main project from within the IDE for the newly-added jars to be included.
3. For Eclipse: Configure build path of the project to add all the third-party jars.

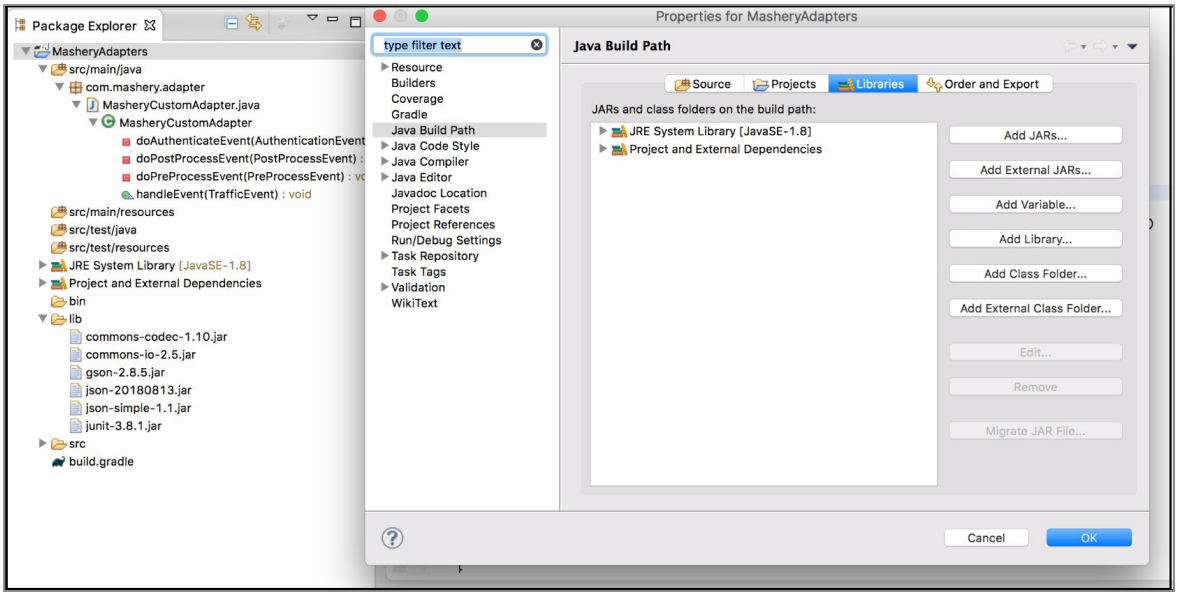
Adding third-party library for Eclipse environment

Procedure

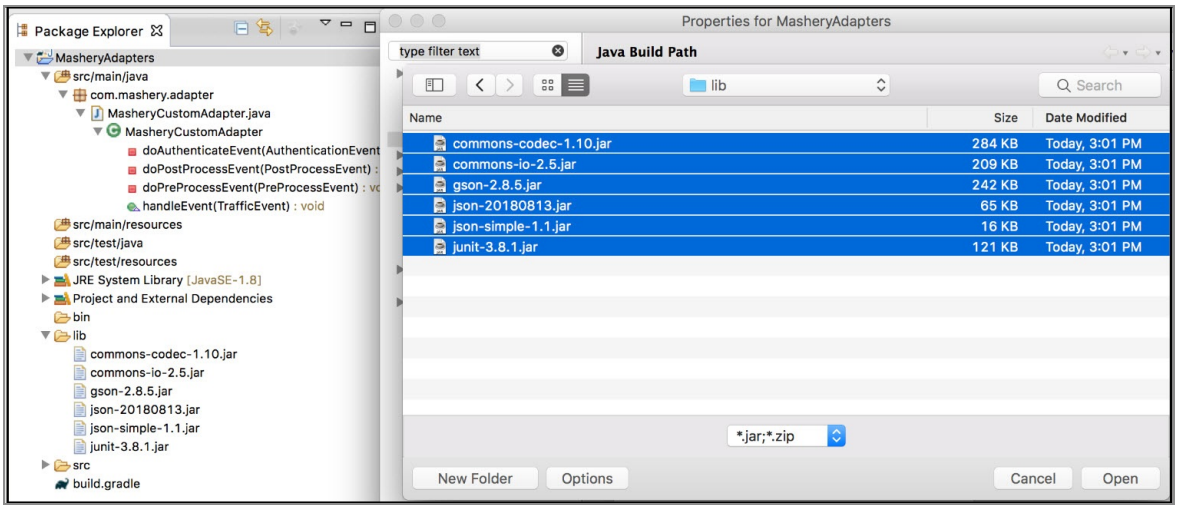
1. In the project structure, right click the **MasheryAdapter** folder, point **Build Path** and click **Configure Build Path**.



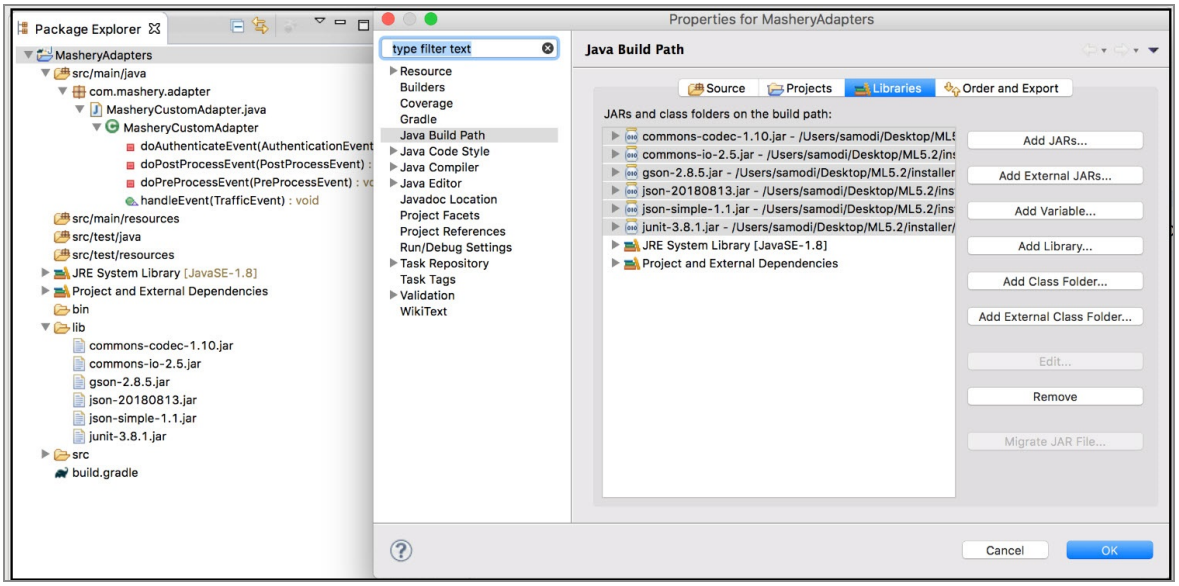
2. Navigate to the library location and click **Add External JARs**



3. Select the files you wish to add and click **Open**.



4. The files are added to the library. Click **OK**.



Referring to Third-party Libraries with Dependency

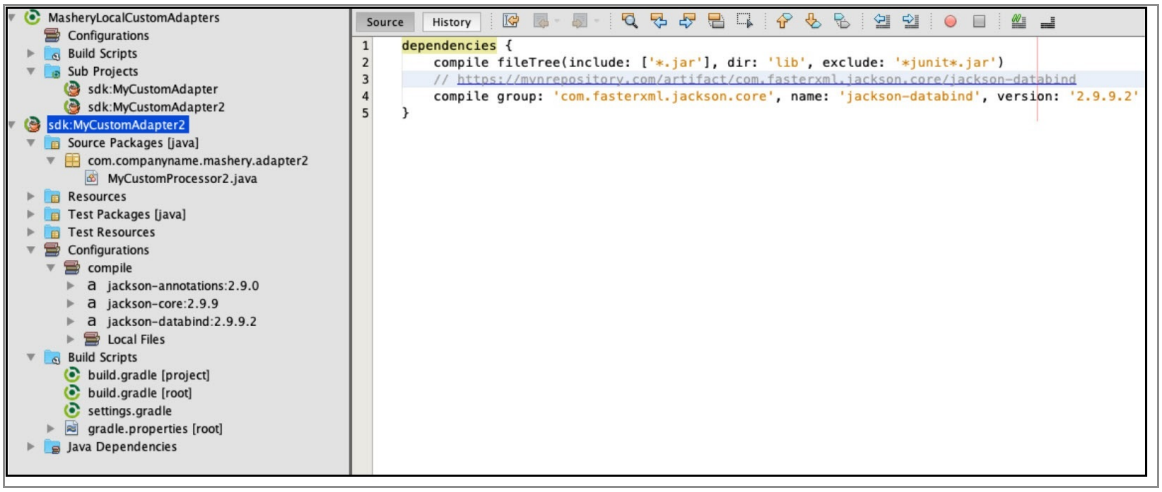
You can use third party libraries for an adapters. Few third-party libraries have transitive dependencies. You have to include these dependencies in the lib folder.

Gradle supports reference to third-party libraries and then resolve the transitive dependencies.
To refer third -party dependency for an adapter sub project:

Procedure

1. Navigate to the JCenter repository (https://bintray.com/bintray/jcenter) or to the Maven central repository (https://mvnrepository.com/repos/central).
2. Select the required library and click **Gradle**.
3. Copy the dependency text.

   The dependency text will be in the form of compile 'org.apache.commons:commons-lang3:3.9
4. Open the build.gradle file of the sub-project in a text editor.
5. In the dependenciessection, add the dependency text of step 3.



> **Note:** Adding the direct dependency on jackson-bind, includes the transitive dependency for jackson-core and jackson-annotations.

Debugging the Adapter

Stages of debugging an adapter are as follows:

1. Adding logger utility class.
2. Changing LogLevels.
3. Checking adapter logs.

Adding Logger Utility Class

To debug the adapters, use **com.mashery.trafficmanager.debug.Logger**, a new utility class. This logs "debug"," info","warn", and "error log" statements.

Example statement

```
com.mashery.trafficmanager.debug.Logger.info(MyAdapter.class,"Info statement for request param {} and value {},paramName, paramValue)
```

Procedure

1. In the adapter class, add:

```
com.mashery.trafficmanager.debug.Logger.info(MyAdapter.class,"Info statement for request param {} and value {},paramName, paramValue)
```

> **Note:** Use parameterized messages patterns instead of concatenating variables to the message.

Changing Log Levels

The default log level for messages from the adapter is 'INFO'. To change the LogLevel to DEBUG, use the Cluster Manager commands. To change LogLevel for adapters, log

into the tml-cm container using docker exec or kubectl exec as applicable to the deployment method.

To list components:

Run the command:

```
clustermanager ls components
```

Example component ID listing

```
clustermanager ls components

Component ID                         Component Type      Component Name      Component Status
Component Host      Component Agent Port   Component Service Port(s)
------------------------------------ ------------------- ------------------- ------------------- ------------------- ----------
------------ --------------------------
bb32be63-2a72-455e-9ffd-3e65174d119a  logservice          log                 ACTIVE              10.0.0.9
     9080                24224
d9927ae6-8135-4360-8031-4493e2cfd9e6  sql                 sql                 ACTIVE              10.0.0.11
 9080                3306
d2b75938-bcd5-40e9-b260-41abb0418ba7  nosql               nosql               ACTIVE              10.0.0.5
      9080                9042
fca40c79-36eb-4599-a0b4-bbe417d7bbbd  cache               cache               ACTIVE              10.0.0.13
       9080                11212,11211,11213,11214,11215,11216
1bdbc338-946d-42df-a535-5f72b0144372  trafficmanager      tm                  ACTIVE
10.0.0.16           9080                8080
388ca01b-1442-4a5e-856c-775325af8bbd  trafficmanager      tm                  ACTIVE
10.0.0.15           9080                8080
```

The component types for the Traffic Manager is trafficmanager.

To see current LogLevel for custom adapters

Run the command:

```
clustermanager ls loggers --componentType trafficmanager --componentId <component id>
```

Output of List Loggers

```
clusterId [1cac153d-48db-4d5c-b3fc-9d25673ee536] and zoneId [a127e639-737c-4220-b42f-
da9f14e939dc]
Using cluster name [Tibco APIM-LE  Reference Cluster]
```

```
Using Zone name [local]
Using Component ID [1bdbc338-946d-42df-a535-5f72b0144372] of type [trafficmanager]
LoggerName                                                    LogLevel   Process
------------------------------------------------------------ ---------- --------------------
...A list of other loggers...                                  INFO
MasheryCustomAdapter                                             INFO
```

To change log level for custom adapters

Run the command:

```
clustermanager set loglevel --componentType trafficmanager --logLevel DEBUG --loggerName
MasheryCustomAdapter
```

Acceptable values for log level are:

- OFF
- INFO
- DEBUG
- WARN
- ERROR

Note: It will take some time for the Log Level change to be reflected in the traffic manager.

To verify that the LogLevel has changed

Command to list loggers on traffic manager:

```
clustermanager ls loggers --componentType trafficmanager --componentId <component id>
```

```
clusterId [1cac153d-48db-4d5c-b3fc-9d25673ee536] and zoneId [a127e639-737c-4220-b42f-
da9f14e939dc]
Using cluster name [Tibco APIM-LE  Reference Cluster]
Using Zone name [local]
Using Component ID [1bdbc338-946d-42df-a535-5f72b0144372] of type [trafficmanager]
```

```
LoggerName                                    LogLevel   Process
------------------------------------------------------------ ---------- --------------------
...A list of other loggers...                   INFO
MasheryCustomAdapter                              DEBUG
```

Checking Adapter Logs

The logs are all stored on the Log service (tml-log) pod or container at /mnt/data/tml-tm/<hostname/ipaddress>/tmdata/proxy_error/proxy_error.log and /mnt/data/tml-tm/<hostname/ipaddress>/tmdata/proxy_debug/proxy_debug.log.

Open the log file and search for messages containing text: *MasheryCustomAdapter*.

Debugging SDK Processor Remotely

Note: You can debug remotely only in the quick start mode.

Preparing Traffic manager container

1. Set up the cluster and connect to the tml-tm container.

   ```
   docker ps | grep -i tml-tm
   ```

   ```
   docker exec -it {tml-tm-container_id} bash
   ```

2. Enable **Javaproxy with debugger agent** by executing the following command:

   ```
   Edit the file /opt/javaproxy/proxy/proxy.ini and add below line at the end
   -agentlib:jdwp=transport=dt_socket,address=8001,server=y,suspend=n
   ```

   Save the changes.

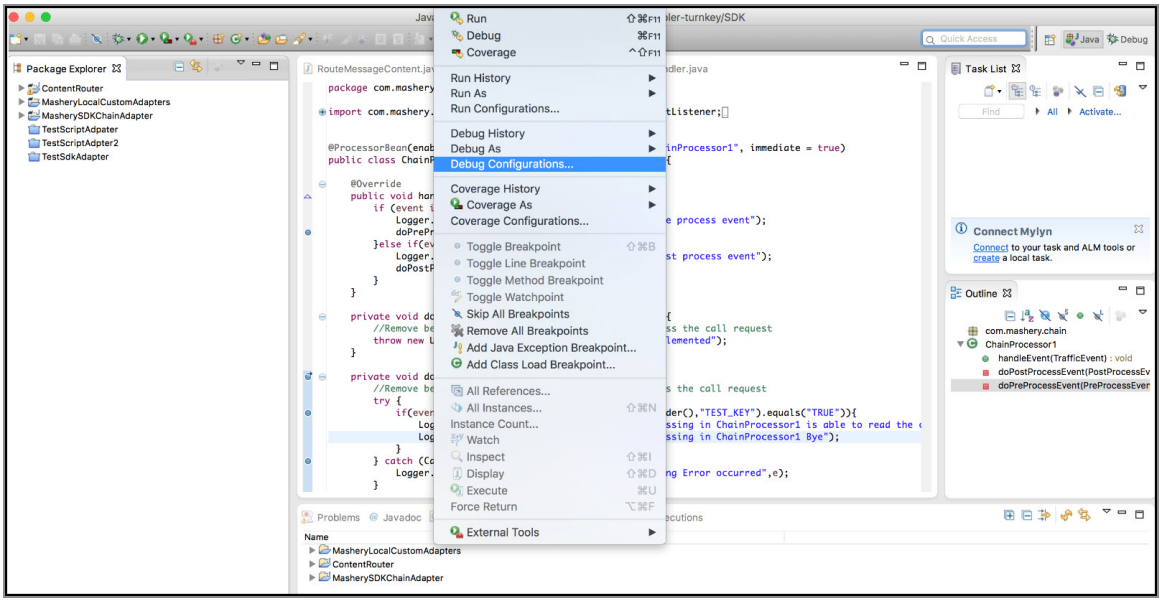
3. Restart the proxy as shown below.

   ```
   /etc/init.d/javaproxy restart
   ```

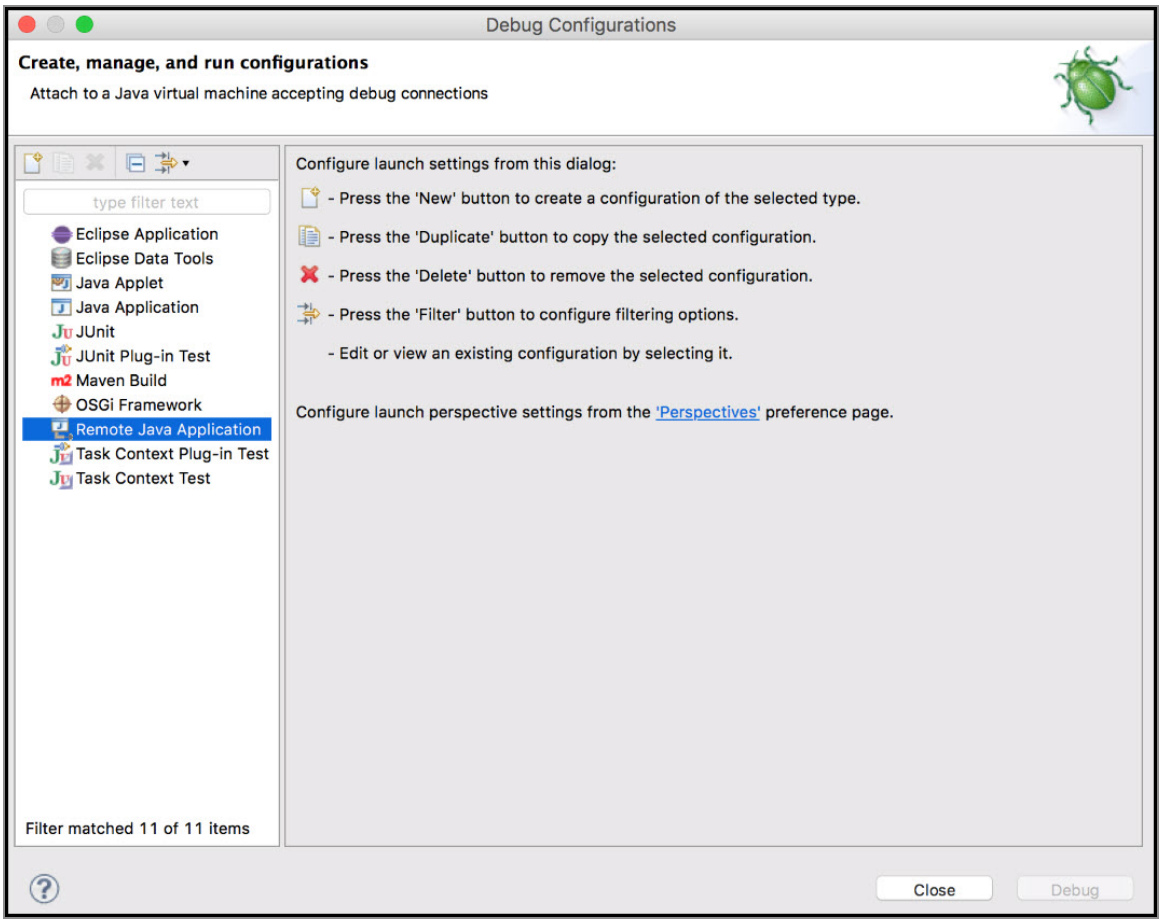
Preparing the IDE

Eclipse

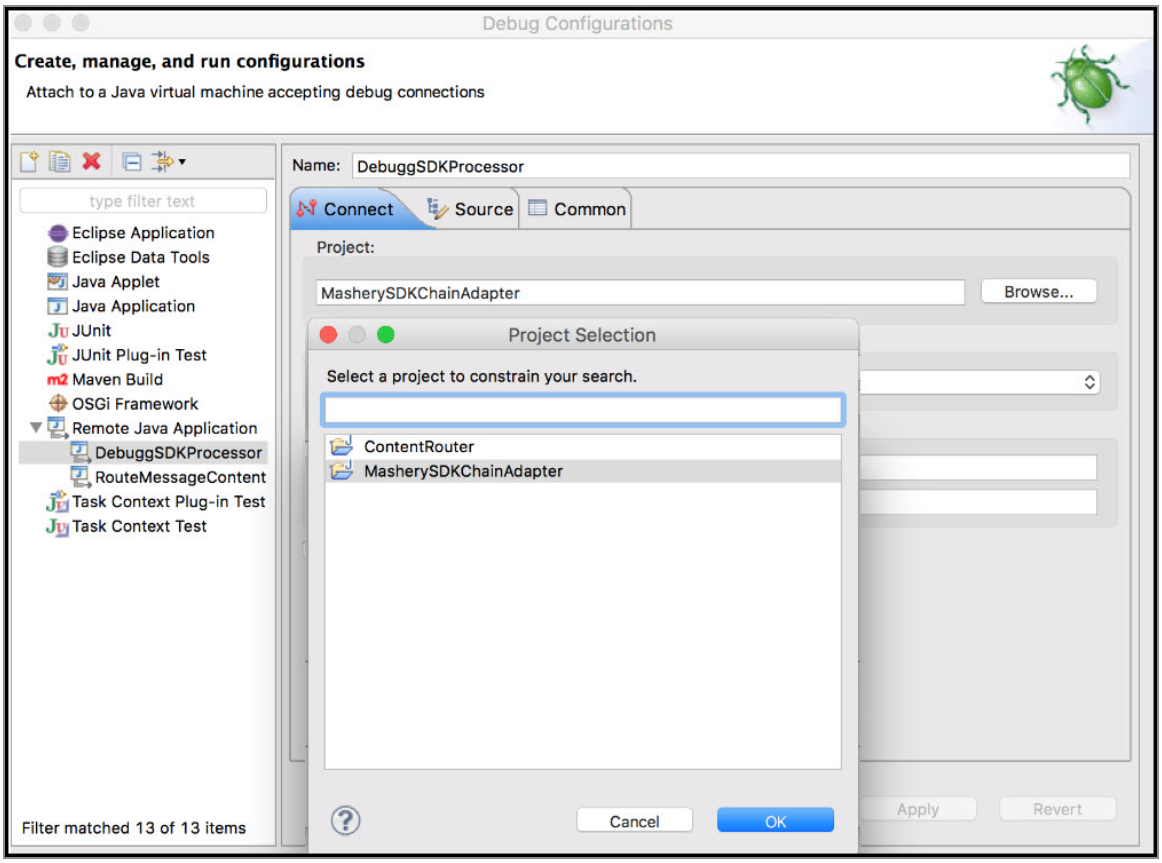
1. Open IDE and configure the debugger to connect to a remote port.
2. Click **Debug Configurations**.



3. The **Debug Configurations** window opens, here click **Remote Java Application**.



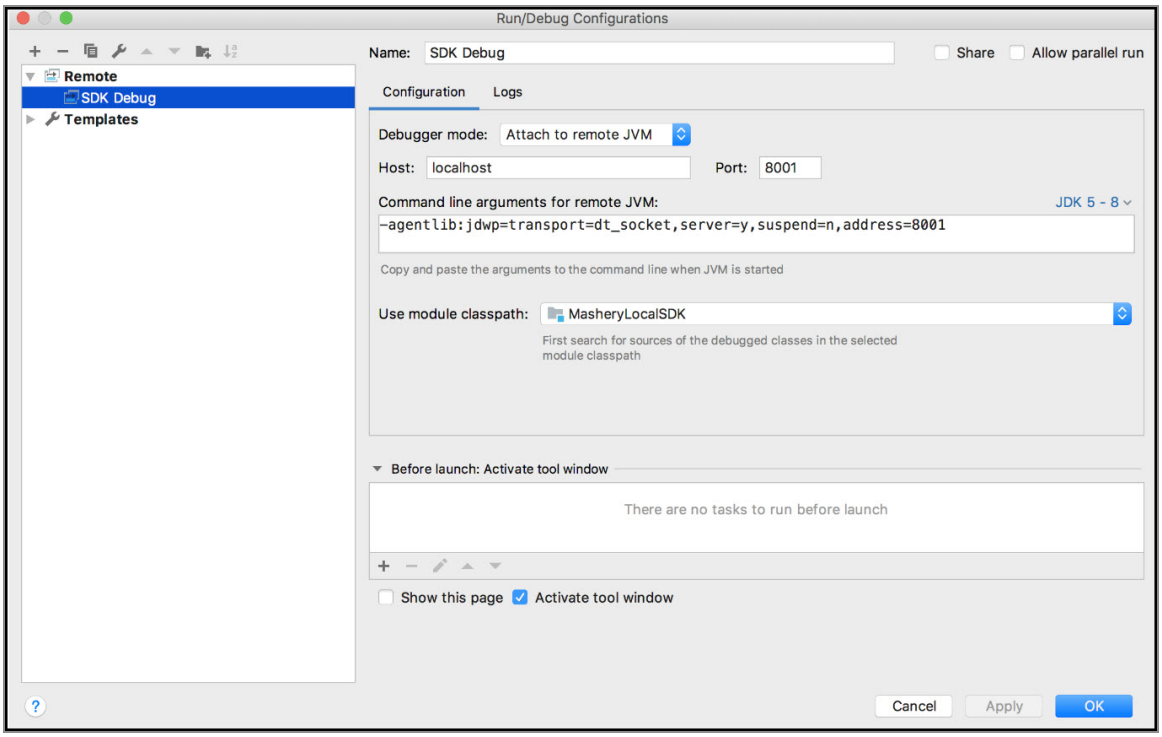
4. Click **New** button  , to create a new configuration. The right side of the screen displays options to create new configuration.
5. In the **Connect** tab, select the **Project** by clicking **Browse**. In the **Project selection** window, select the SDK Processor's project.



6. In the **Connection Properties** field, for the **Host** field, input the node's IP and for **Port**, input 8001. Click **Apply** and then click **Debug**.

IntelliJ Idea

1. From the main menu, click **Run** option and select **Edit Configuration**.
2. Click  button and select **Remote** from the configuration list.
3. In the tree structure, click **SDK Debug**. In the **Debugger mode** field, select **Attach to remote JVM**. For the **Host** field, input the node's IP and for **Port**, input 8001.

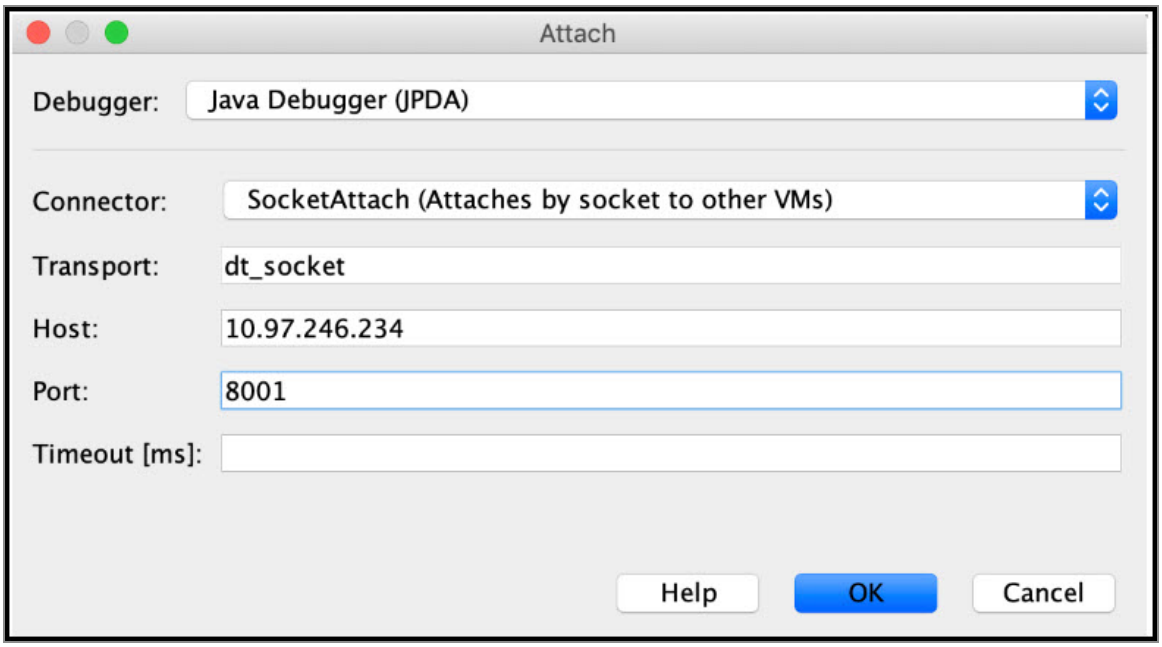


On successful completion, you can see the toolbar in the IntelliJ Idea window. Click the green debug icon,when you wish to debug.

Apache NetBeans

1. In the **NetBeans** window, click **Debug** menu, and then click **Attach Debugger**.

2. The **Attach** dialog box opens, here provide the input for **Debugger**, **connector** and **Transport**.



3. For the **Host** field, input the node's IP and for **Port**, input 8001. Click **OK**.

The debug view opens. Set the required breakpoints in the processor code and call an end point.

Fix, Build and Deploy

Fix

Fix the issues found on debugging by fixing the code.

Build

1. Run the built script.

```
/home/user/MasheryLocalSDK$ ./build-adapter.sh
```

> **Note:** The build script reports the compilation errors. In case of any reported errors rerun the build script.

2. The script generates an archive file as tml-mashery-customer-extension.zip under MasheryLocalSDK. Upload this file to the Local Edition installer.

Deploy

1. Find the container ID of the tml-tm container using the following command:

```
docker ps | grep -i "tml-installer"
```

2. Copy the upload-deploy-sdk-processor.sh script from the installer to the local path.

```
docker{tml-installer-contianer-id}:/var/jenkins_home/docker-deploy/docker-swarm/system/upload-deploy-sdk-processor.sh {path_to_save}
```

3. Use the upload-deploy-sdk-processor.sh script to upload and deploy the already built SDK processors in the tml-tm container.

```
./upload-deploy-sdk-processor.sh --help
Usage: ./upload_deploy_sdk_processor.sh [--adapter-zip <ADAPTER_FULL_PATH>] [--tml-tm-container <TM_CONTAINER>]
   ( --adapter-zip | -a ) -  path/filename of the adapter zip to be uploaded to container
   ( --tml-tm-container| -c ) - TML-TM Container ID where adapter to be uploaded
Example: ./upload-deploy-sdk-processor.sh --adapter-zip

/home/user/MasheryLocalSDK/dist/tml-mashery-customer-extension.zip --tml-tm-container 9c9e09ea1f95
```

4. Log in the tml-tm container,

```
docker exec -it {tml-tm-container-id} bash
```

and restart thejavaproxy service using the following command:

```
/etc/init.d/javaproxy restart
```

Implementing and Registering Processors and Adapters

Implementing a Processor or Adapter

Creating a Pre-processing Adapter

Pre-processing an adapter comprises the following three stages:

1. Reading body content of request.
2. Modifying the request body.
3. Terminating further processing for unavailable header.

For details on how to access and use extended attributes, such as sending the response body to another location other than the caller, see Accessing and Using Extended Attributes.

Reading Body Content of Request

Procedure

1. In the request, get the ContentSource.
2. From the ContentSource, get the inputStream.
3. Read content from the InputStream.

```
ContentSource body = event.getCallContext().getRequest().getBody();
final InputStream inputStream = body.getInputStream() ;
//use input stream to read content
//Do something with the content
```

> **Note:**
> Refer working code in examples/ReadRequestBody.java.

Modifying the Request Body

To modify the request body in the pre-processing stage employ the content source and content producer to read from and write to the HTTP request.

Procedure

1. Get Content source from request.
2. Get input stream from Content source.
3. Read content from the input stream.
4. Modify the content.
5. Create a content producer.
6. Set the request body to the created content producer.

```
ContentSource body = event.getCallContext().getRequest().getBody();
final InputStream inputStream = body.getInputStream() ;
//use input stream to read content
//modify content
httpReq.setBody(new ContentProducer() { //set new content body
   .........
   public void writeTo(OutputStream out) throws IOException {
      out.write("modified content")
      out.flush();
      out.close();
   }
});
```

> **Note:**
> Refer to the working code in examples/ModifyRequestBody.java.

Terminating Further Processing for Unavailable Header

Checking for Header

1. Get the headers from either HttpClientRequest or HTTPServerRequest of an event.
2. Check the header.
3. If the header is not present or value is not proper then you can terminate the request and Local Edition would not send the request to target server.
4. You would be able to set the status code and status message in case of termination of call in pre/post processing.

```
private void doPreProcessEvent(PreProcessEvent event) throws IOException {
   MutableHTTPHeaders headers = event.getClientRequest().getHeaders();
   String custHeader = headers.get("X-Custom-Header");
   if(null == custHeader || !custHeader.equals("Allowed")){
      event.getCallContext().getResponse().getHTTPResponse().setStatusCode
(HttpURLConnection.HTTP_BAD_REQUEST);
      event.getCallContext().getResponse().getHTTPResponse().setStatusMessage
("Custom Header is not set in the client request");
      event.getCallContext().getResponse().setComplete();
   }
}
```

Checking Parameter

1. Get CallContext from event.
2. From CallContext get ApplicationRequest.
3. You can check for any parameter passed in the QueryData of ApplicationRequest.
4. If the parameter is missing then user/developer can terminate the request and Local Edition would not sent the request to target server. You can terminate the process and set the status code and status message.

```
private void doPreProcessEvent(PreProcessEvent event) {
   //Remove below line and implement code to pre process the call request
```

```
    String custParam = event.getCallContext().getRequest().getQueryData().get
("customParam");
    if (custParam == null) {
        event.getCallContext().getResponse().getHTTPResponse().setStatusCode
(HttpURLConnection.HTTP_BAD_REQUEST);
        event.getCallContext().getResponse().getHTTPResponse().setStatusMessage
("Custom Header is not set in the client request");
        event.getCallContext().getResponse().setComplete();
    }
}
```

Creating a Post-processing Adapter

Post-processing for an adapter comprises of:

1. Add custom header to the response to client.
2. Modifying body content of response to client.

Add Custom Header to the Response to Client

Procedure

1. Get TrafficManagerResponse from the call context of an event.
2. Get HTTPServerResponse from TrafficManagerResponse.
3. Get Headers from HTTPServerResponse and add new header to MutableHTTPHeaders list.

```
private static final String CUSTOM_HEADER="X-CUSTOM-HEADER";
private static final String CUSTOM_HEADER_VALUE="POST-PROCESSED";

@Override
public void handleEvent(TrafficEvent event) {
    if(event instanceof PostProcessEvent){
        Logger.debug(AddHeaderPostProcessor.class, "Handling post process event");
        doPostProcessEvent((PostProcessEvent) event);
    }
}
```

```
private void doPostProcessEvent(PostProcessEvent event) {
   MutableHTTPHeaders headers = event.getCallContext().getResponse
().getHTTPResponse().getHeaders();
   if(event.getCallContext().getResponse().getHTTPResponse().getStatusCode() == 200){
      headers.add(CUSTOM_HEADER,CUSTOM_HEADER_VALUE);
   }

}
```

Note:
Refer to the working code in examples/ AddHeaderPostProcessor.java.

Modifying Body Content of Response to Client

To modify the request body in the pre-processing stage, employ the content source and content producer to read from and write to the HTTP request.

Procedure

1. Get HTTPClientResponse from the call context of an event.
2. Get ContentSource from the HTTPClientResponse.
3. Get InputStream from ContentSource and convert it into String.
4. Add or Modify the String content and set the body of HTTPServerResponse with new content. (HTTPServerResponse can be obtained from TrafficManagerResponse of call context of an event.)

```
private static final String CUSTOM_HEADER="X-CUSTOM-HEADER";
private static final String CUSTOM_HEADER_VALUE="POST-PROCESSED";

@Override
public void handleEvent(TrafficEvent event) {
   if(event instanceof PostProcessEvent){
      Logger.debug(AddHeaderPostProcessor.class, "Handling post process event");
      doPostProcessEvent((PostProcessEvent) event);
   }
}

private void doPostProcessEvent(PostProcessEvent event) {
```

```
   ExtendedAttributes attrs = (event).getKey().getExtendedAttributes();
    String strAllowed = attrs.getValue("EAV_CallAllowed");
    if(strAllowed != null && !Boolean.parseBoolean(strAllowed)){
       event.getCallContext().getResponse().getHTTPResponse().setStatusCode(401);
       event.getCallContext().getResponse().getHTTPResponse().setBody(new
StringContentProducer("{\"error\":\"Call not allowed for this key\"}"));
       event.getCallContext().getResponse().setComplete();
    }

}
```

Note:
Refer to the working code in examples/ AddBodyContentPostProcessor.java.

Creating a Custom Authenticator

A customer authenticator comprises of:

1. Stopping processing request on authentication failure.
2. Continuing processing request for successful authentication.

Stopping a Processing Request on Authentication Failure

Procedure

1. Get the headers from the HTTPServerRequest.
2. Check for authentication header.
3. Validate the value of authentication header. On validation failure, set the TrafficManagerResponse to complete.
4. Local Edition would terminate the request and return ERR_403_NOT_AUTHORIZED error.

Note: You cannot change the status code or status message from the adapter.

Unsuccessful Authentication

```
private void doAuthenticateEvent(AuthenticationEvent event)
        throws ProcessorException {
      //For example request doesn't contain the authorization header then user can terminate
the call by marking response as complete
      // in order to thrown 403 ERR_403_NOT_AUTHORIZED for the incoming request.
        HTTPHeaders headers = event.getServerRequest().getHeaders();
        if (headers != null) {
         String authorization = headers.get(HEADER_AUTHORIZATION);
         if ((null == authorization || authorization == "")
               || !authorization.startsWith(AUTH_BASIC)) {
           Logger.warn(MyCustomAuthenticator.class,"Error validating the authentication
header {}",HEADER_AUTHORIZATION);
            event.getCallContext().getResponse().setComplete();
         }
  }
```

Note: If the authentication fails to prevent further processing, set the following:

```
event.getCallContext().getResponse().setComplete();
```

Refer to the working code in examples/MyCustomAuthenticatorFailed.java.

Continue Processing Request for Successful Authentication

Procedure

1. Get the headers from the HTTPServerRequest.
2. Check for authentication header.
3. Validate the value of authentication header. On validation success, return from the adapter and continue processing.

Successful Authentication

```
if (userId.equals("userName") && password.equals("userPassword")) {
   Logger.info(MyCustomAuthenticatorFailed.class,"Basic Authentication is successful");
}
```

Implementing the Event Listener

To implement the event listener:

Procedure

1. Employ the Traffic Event Listener interface (introduced in Event Listener API) as shown in the following example:

```
package com.company.extension;
public class CustomProcessor implements TrafficEventListener{
    public void handleEvent(TrafficEvent event){
      //write your custom code here
     }
}
```

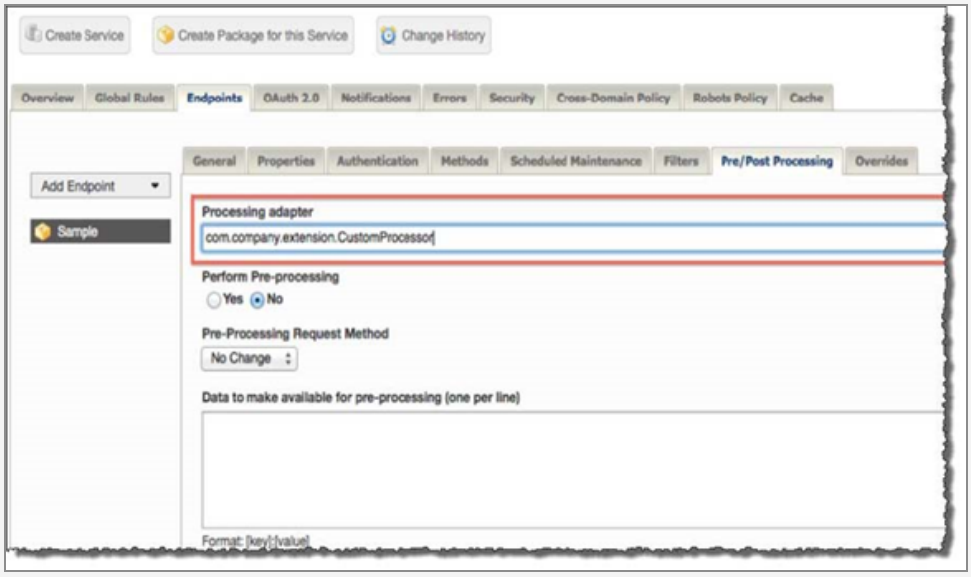
2. Annotate your code to ensure that the processor is identified correctly for callbacks on events related to the specific endpoints it is written to handle:

```
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true)
public class CustomProcessor implements TrafficEventListener{
    public void handleEvent(TrafficEvent event){
      //write your custom code here
   }
}
```

The annotation identifies the following properties:

- **enabled**: Identifies if the processor is to be enabled.
- **name**: Identifies the unique name of the processor as configured in API Settings (see marked area in 'red' in the following screenshot).
- **immediate**: Identifies if the processor is enabled immediately.

Note: The name used in the annotation for the Processor MUST be the same as configured on the portal for the Endpoint>Pre/Post Processing, as shown in the following screenshot:



Implementing Lifecycle Callback Handling

If you wish to have some initialization work done once and only once for each of the processors, then implement the following interface:

package com.mashery.trafficmanager.event.listener:

```
/*** The lifecycle callback which gets called when the processor gets loaded when installed and
released*/
public interface ListenerLifeCycle {
  /*** The method is called once in the life-cycle of the processor before the     processor is deemed
ready to handle requests. If the processor throws an exception, the activation is assumed to be a
failure and the processor will not receive any requests @throws ListenerLifeCycleException*/
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException;

 /*** The method is called once in the life-cycle of the processor before the processor is removed
due. The processor will not receive any requests upon     inactivation.*/
  public void onUnLoad(LifeCycleContext ctx);
}
```

The **onLoad** call is made once prior to the processor handling any requests and **onUnLoad** call is made before the processor is decommissioned and no more requests are routed to it.

The lifecycle listener can be implemented on the Processor class or on a separate class. The annotation needs to add a reference to the lifecycle-class if the interface is implemented (see highlighted property in **bold**).

```
package com.company.extension;
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true, lifeCycleClass="com.company.extension.CustomProcessor")
public class CustomProcessor implements TrafficEventListener, ListenerLifeCycle{
  public void handleEvent(TrafficEvent event){
    //write your custom code here
  }
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
  }
  public void onUnLoad(LifeCycleContext ctx){
  }
}
```

Note: The lifeCycleClass property should point to the class implementing the Listener LifeCycle interface. This also allows having a separate lifecycle listener interface as follows (note the different lifeCycleClass name).

The following example shows a different class implementing the LifeCycle callback:

```
package com.company.extension;
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true, lifeCycleClass="com.company.extension.CustomProcessorLifeCycle")
public class CustomProcessor implements TrafficEventListener {
  public void handleEvent(TrafficEvent event){
    //write your custom code here
  }
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
  }
  public void onUnLoad(LifeCycleContext ctx){
  }
}
public class CustomProcessorLifeCycle implements ListenerLifeCycle{
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
  }
  public void onUnLoad(LifeCycleContext ctx){
```

```
  }
}
```

Caching Content

You can interact with memcache using the Local Edition SDK. The memcached is a part of the Local Edition setup. A key-value pair can be stored in the cache.

The cache interface provided in the callback to the TrafficEventListener is:

```
package com.mashery.trafficmanager.cache;
/*** Cache API which allows extensions to store and retrieve data from cache*/
public interface Cache {
  /**
  * Retrieves the value from the cache for the given key
  * @param key
  * @return
  * @throws CacheException
  */
  Object get(String key) throws CacheException;
  /**
  * Puts the value against the key in the cache for a given ttl
  * @param key
  * @param value
  * @param ttl
  * @throws CacheException
  */
  void put(String key, Object value, int ttl) throws CacheException;
}
```

A reference to the cache can be found on the ProcessorEvent which is reported on the callback. Here is an example of how to access cache on callback:

```
package com.company.extension;
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true
public class CustomProcessor implements TrafficEventListener, ListenerLifeCycle{
  public void handleEvent(TrafficEvent event){
      ProcessorEvent processorEvent = (ProcessorEvent) event;
      Cache cacheReference = processorEvent.getCache();
      //Add data to cache
      try{
```

```
        cacheReference.put("testkey", "testValue", 10)
      }catch(CacheException e){
       //load data or load default data
      }
    //write your custom processor code here
    }
}
```

A reference to cache is also available on the lifecycle callback:

```
package com.company.extension;
public class CustomProcessorLifeCycle implements ListenerLifeCycle{
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
    Cache cache = ctx.getCache();
    // perform cache operations
  }
  public void onUnLoad(LifeCycleContext ctx){
  }
}
```

For more information, see examples/CacheAccess.java.

Terminating a Call During Processing of an Event

This version of the SDK allows a user to terminate a call during pre or post processing, or in authentication event handling. For example, if the request does not have a required URL parameter, Local Edition can be configured to terminate the call in the pre-processing.

> **Note:** All the headers, status code and status messages set in the custom processing is returned to the client as part of the response in pre processing and post processing.

Note: All the headers, status code and status messages set in the custom authentication would not be returned as part of response in case of Authentication Event handling (Custom authenticator). If you want to fail authentication request from the custom authenticator, then you need to terminate the call in order to throw "ERR_403_NOT_AUTHORIZED" for a request.

For example, if you want to terminate the call in authenticator, if request doesn't contain the authorization header, then the call can be terminated by marking the response as complete as shown in the following example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.mashery.http.HTTPHeaders;
import com.mashery.trafficmanager.debug.DebugContext;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.AuthenticationEvent;
import com.mashery.trafficmanager.event.processor.model.PostProcessEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.processor.ProcessorBean;
import com.mashery.trafficmanager.processor.ProcessorException;

@ProcessorBean(enabled = true, name = "CustomAuthentication", immediate = true)

public class CustomAuthentication implements TrafficEventListener,Authenticator {

   @Override
   public void handleEvent(TrafficEvent event) {
      try {
         if (event instanceof AuthenticationEvent) {
            authenticate((AuthenticationEvent) event);
         }
      } catch (ProcessorException e) {
      }
   }

   private void authenticate(AuthenticationEvent event)
         throws ProcessorException {
      //For example request doesn't contain the authorization header then user can terminate the call
by marking response as complete
      // in order to thrown 403 ERR_403_NOT_AUTHORIZED for the incoming request.
      if (headers != null) {
```

```
        String authorization = headers.get(HEADER_AUTHORIZATION);
        if ((null == authorization || authorization == "")
              || !authorization.startsWith(AUTH_BASIC)) {
           debugContext.logEntry("Final Value", "DIY-CUSTOM-AUTH-HEADER-FAILIURE");
           event.getCallContext().getResponse().setComplete();
        }
    }
}
```

If you want to terminate the call in pre or post processing, refer to the following example:

```
package com.mashery.processor;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.PostProcessEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.model.core.ExtendedAttributes;
import com.mashery.trafficmanager.processor.ProcessorBean;
import com.mashery.trafficmanager.processor.ProcessorException;

@ProcessorBean(enabled = true, name = "PrePostProcessing", immediate = true)

public class PrePostProcessing implements TrafficEventListener{

   private final Logger log = LoggerFactory.getLogger(PrePostProcessing.class);

   @Override
   public void handleEvent(TrafficEvent event) {
      try {
         if (event instanceof PreProcessEvent) {
            preProcess((PreProcessEvent) event);
         } else if (event instanceof PostProcessEvent) {
            postProcess((PostProcessEvent) event);
         }
      } catch (ProcessorException e) {
         log.error("Exception occurred when handling processor event");
      }
   }

  //In the below example we checking the query parameter's value to decide whether to terminate
the call or not.
   private void preProcess(PreProcessEvent event) throws ProcessorException {
```

```
        String complete = event.getCallContext().getRequest().getQueryData().get("preComplete");
        if (complete != null) {
            event.getCallContext().getResponse().getHTTPResponse().setBody(new
StringContentProducer("{\"response\": \"Terminated the call in pre-processing\"}"));
            event.getCallContext().getResponse().setComplete();
        }
    }
   //In the below example we checking the query parameter's value to decide whether to terminate
the call or not.
    private void postProcess(PostProcessEvent event) throws ProcessorException {
       String complete = event.getCallContext().getRequest().getQueryData().get("postComplete");
        if (complete != null) {
            event.getCallContext().getResponse().getHTTPResponse().setBody(new
StringContentProducer("{\"response\": \"Terminated the call in post-processing\"}"));
            event.getCallContext().getResponse().setComplete();
        }
    }
}
```

Accessing and Using Extended Attributes

Extended attributes can be set on customer site, key (API or package key), application, user, plan and package, and values . This is defined as per the model class.

The SDK includes an example custom processor that can access package Extended Attribute Values (EAV) as follows:

Accessing extended attributes of Customer site

```
CustomerSite customerSite = event.getEndpoint().getAPI().getCustomerSite();
ExtendedAttributes customerSiteEAVs = customerSite.getExtendedAttributes();
```

Accessing extended attributes of key

```
Key key = event.getKey();
ExtendedAttributes keyEAVs = key.getExtendedAttributes();
```

Accessing application extended attributes

```
Key key = event.getKey();
Application app = key.getApplication();
ExtendedAttributes appEAVs = app.getExtendedAttributes();
```

Accessing User extended attributes

```
Key key = event.getKey();
User user = key.getOwner();
ExtendedAttributes userEAVs = user.getExtendedAttributes();
```

Accessing plan and package extended attributes

```
CKey key = event.getKey();
if(key instanceof PackageKey){
   Plan plan = ((PackageKey)key).getPlan();
   ExtendedAttributes planEAVs = plan.getExtendedAttributes();
   //use planEavs

   //Packge extended attributes
   com.mashery.trafficmanager.model.core.Package pkg = plan.getPackage();
   ExtendedAttributes pkgEAVs = pkg.getExtendedAttributes();
   //USe pkgEavs
```

Accessing values from Extended Attributes

```
//Get Extended attributes from one of the objects viz.
User user =  event.getKey().getOwner();
ExtendedAttributes userEAVs = user.getExtendedAttributes();
String key = "user_defined_key";
String value = userEAVs.getValue(key);
Logger.debug(this.getClass(), "Found EAV for Key:{}, value: {}", key,value);
```

How to Send Response Body to Another Location Other Than Caller

Once a response is streamed, it cannot be streamed again since the target would have flushed and closed the stream. Use the com.mashery.http.server.HTTPServerResponse#setBody method to set a specialized content producer that will stream to a caller and also a secondary destination.

Defining a Content Producer

```
public class BiContentProducer implements ContentProducer{
   @Override
   public void writeTo(OutputStream out) throws IOException {
      BufferedInputStream in = new BufferedInputStream(origin.getInputStream());
      byte[]buffer = new byte[4096];
      int c;
      while((c = in.read(buffer)) != -1){
         out.write(buffer, 0,c); //send response bytes to caller
         secondaryStream.write(buffer);  //send response bytes to secondary output.
                              //OPtionally write to secondary stream in a separate thread.
      }
      out.flush();
      secondaryStream.flush();
      secondaryStream.close();
   }
}
```

Setting a Content Producer to the Response

```
try{
   BiContentProducer contentProducer = new BiContentProducer(origin, keyId);
   postProcessEvent.getServerResponse().setBody(contentProducer);
}catch(IOException e){
   Logger.error(AuditProcessor.class, "Error in creating respnse providr", e);
}
```

For more information, refer to examples/PipeResponseAdapter.java and examples/BiContentProducer.java in the Examples folder.

How to Externalize Properties and Files from SDK-Built Adapters

You can use the SDK to create a Local Edition adapter that can read the file content of /opt/mashery/containeragent/resources/properties/tml_tm_properties_final.json present inside the TM container.

After building the Local Edition adapter, upload it into the Installer and create the Local Edition cluster. The entries from file /tml_tm_properties.json are available.

Accessing Plan, Package and Application ID in Custom Processor

You can access the plan, package and application ID of a processor.

Access to the plan, package and application ID is explained using a sample processor as follows:

```
package com.tibco.apim.examples;

import com.mashery.trafficmanager.debug.Logger;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.model.core.PackageKey;
import com.mashery.trafficmanager.processor.ProcessorBean;

@ProcessorBean(enabled = true, name = "com.tibco.apim.examples.PlanAndPackageIdAccess",
immediate = true)
public class PlanAndPackageIdAccess implements TrafficEventListener{

   @Override
   public void handleEvent(TrafficEvent event) {
      if (event instanceof PreProcessEvent) {
         Logger.debug(PlanAndPackageIdAccess.class, "Handling pre process event");
         doPreProcessEvent((PreProcessEvent) event);
      }
   }
```

```
    private void doPreProcessEvent(PreProcessEvent event) {
        //Get Key from the event and type cast it to Package key.
        //From Package key plan, package and application details can be extracted.
        PackageKey key = (PackageKey)event.getKey();
        Logger.warn(PlanAndPackageIdAccess.class,"Plan Id is :" + key.getPlan().getId());
        Logger.warn(PlanAndPackageIdAccess.class,"Package Id is :" + key.getPlan().getPackage
().getId());
        Logger.warn(PlanAndPackageIdAccess.class,"Application Id is :" + event.getKey
().getApplication().getApplicationId());

    }
}
```

For more information, refer to working code in examples/PlanAndPackageIdAccess.java file.

Chaining of Processors

Local Edition can support multiple pre-processing and post-processing processors per endpoint. This is done by linking multiple processors into one configuration. The entire process is called *Chaining*.

> **Note:** You can chain only pre-processing and post-processing events. The authentication event cannot be chained.

The following table describes chaining between different events.

	Pre Process Event	Post Process Event	Authentication Event
Pre Process Event			
Post Process Event			
Authentication Event			

Types of Chaining

- Chaining Processor Using Mashery_Proxy_Processor_Chain

  The processor implements pre-process or post-process traffic events. These processors can either be provided by Boomi or incorporated with the SDK.

  To chain the processors use Mashery_Proxy_Processor_Chain.

- Chaining of Processor Using Chain Adapter

  These processors can implement pre-process or post-process traffic events and can be chained without using Mashery_Proxy_Processor_Chain and are provided by Boomi.

  The following are processors acting as chain adapter provided by Boomi.

- OAuth2JWT Authentication Connector
- OAuth2 Shared Token Connector

The following table provides information on various chaining combinations.

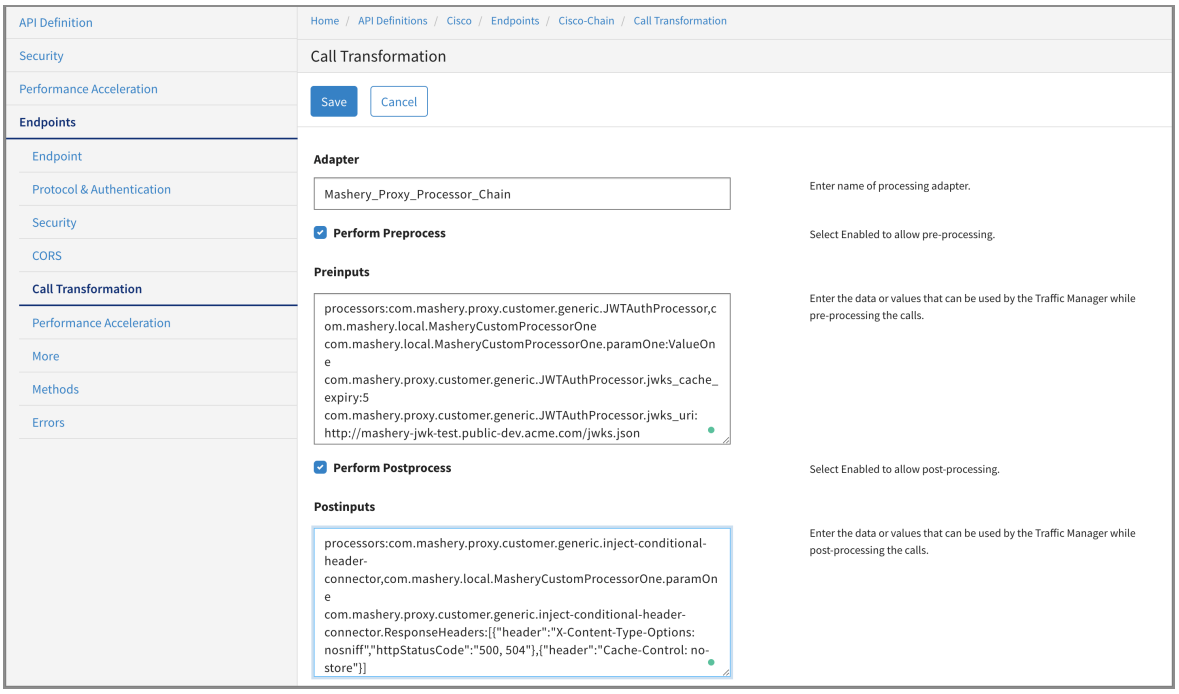
	SDK Based Adapters	Boomi Provided Adapters	Chain Adapter OAuth2JWTAuthenticationConnector	Chain Adapter OAuth2SharedTokenConnector
SDK Based Adapters	 Using Mashery_Proxy_Processor_Chain	 Using Mashery_Proxy_Processor_Chain	 Using OAuth2JWTAuthenticationConnector	 Using OAuth2JWTAuthenticationConnector
Boomi Provided Adapters	 Using Mashery_Proxy_Processor_Chain	 Using Mashery_Proxy_Processor_Chain	 Using OAuth2JWTAuthenticationConnector	 Using OAuth2JWTAuthenticationConnector
Chain Adapter OAuth2JWTAuthenticationConnector	 Using OAuth2JWTAuthenticationConnector	 Using OAuth2JWTAuthenticationConnector		
Chain Adapter OAuth2SharedTokenConnector	 Using OAuth2JWTAuthenticationConnector	 Using OAuth2JWTAuthenticationConnector		

Chaining of Processors Using Mashery_Proxy_Processor_Chain

Using Mashery_Proxy_Processor_Chain processors can be chained in an untethered environment and in a tethered environment.

Chaining Processors in an Untethered Environment

1. In the **Configuration Manager**, click the **Endpoints** tab and then click **Call Transformation**.



2. Input the **Adapter** name with built-in adapter as Mashery_Proxy_Processor_Chain.
3. As per requirement, select the checkbox for **Perform Preprocess/Perform Postprocess** or both.
4. In the **Preinputs** field, type the processors to be chained. The name of the processor is the processors bean name. The syntax is:

```
processors:PROCESSOR1,PROCESSOR2
```

   For example:

```
processors:com.mashery.local.MasheryCustomProcessorOne,com.mashery.local.MasheryCustomProcessorTwo
```

You can also provide configuration data as input. The syntax is:

```
PROCESSOR1.parameter:value
PROCESSOR1.parameter:value
PROCESSOR2.parameter:value
PROCESSOR2.parameter:value
```

Note:

- In the pre-input configuration, there must not be more than one key with name "processors".
- The processors to be chained must be input as comma separated list.

5. In the **Postinputs** field, input the name of the adapter that the traffic manager can use to post-process the calls, then click **Save**.
    - The input syntax for the **Postinputs** fields must be the same as that of the **Preinputs** field.
    - If you opt for only **Perform Postprocess**, the input for **Postinputs** field must be as follows:

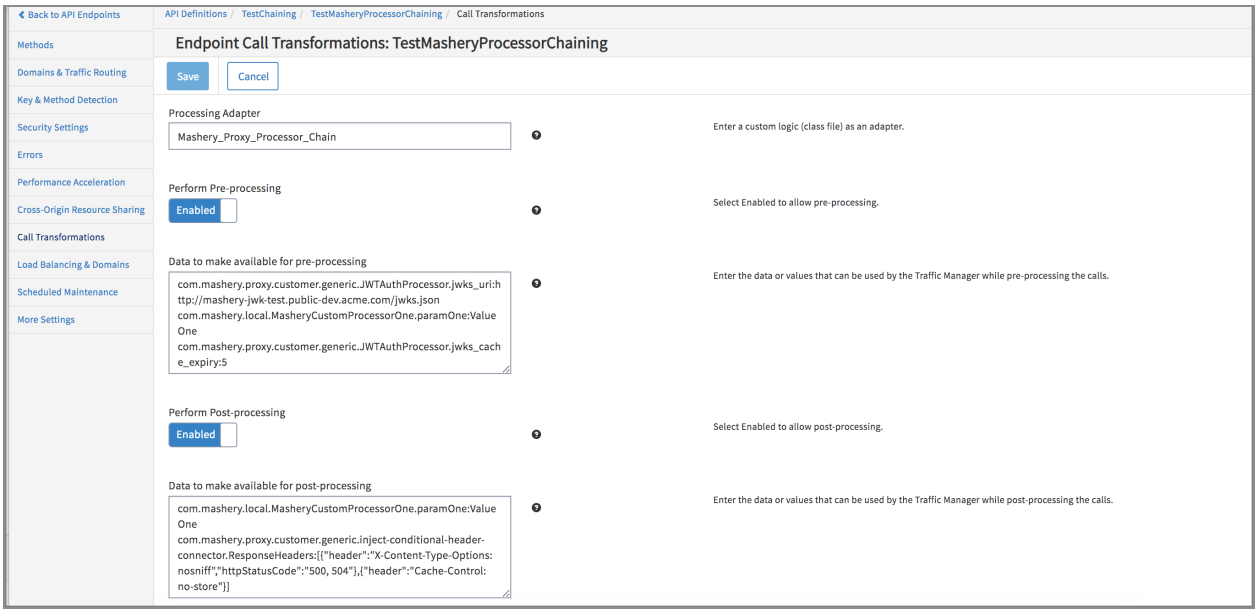
      ```
      processors:com.mashery.local.MasheryCustomProcessorOne,com.mashery.local.MasheryCustomProcessorTwo
      ```

      and

      ```
      PROCESSOR1.parameter:value
      PROCESSOR1.parameter:value
      PROCESSOR2.parameter:value
      PROCESSOR2.parameter:value
      ```

Chaining Processors in a Tethered Environment

The process for chaining processors in a tethered environment is the same as that of untethered. The interface of the control center appears as shown below:



What to do next

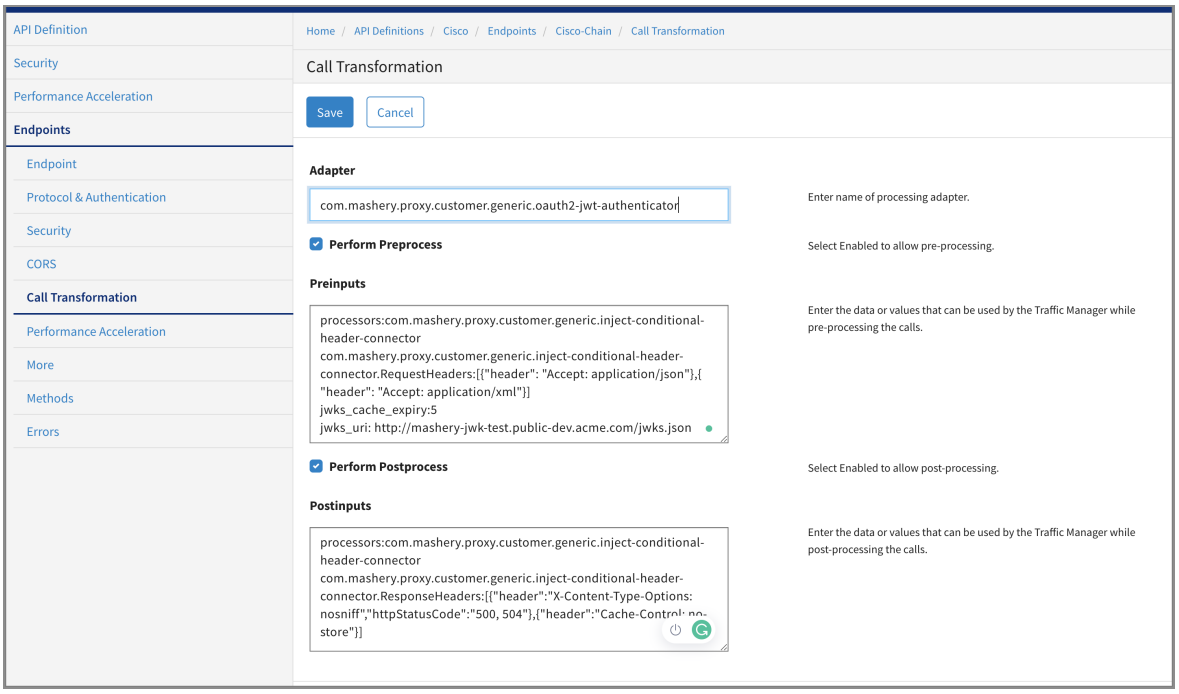
To test the changes, use the endpoint and make a traffic call.

Chaining of Processor Using Chain Adapter

Processors can be chained using a chain adapter in an untethered environment and in a tethered environment.

Chaining Processors in an Untethered Environment

1. In the **Configuration Manager**, click the **Endpoints** tab and then click **Call Transformation**.



2. In the **Adapter** field, input the name of the chain adapter. For example, com.mashery.proxy.customer.generic.oauth2-jwt-authenticator.
3. As per requirement, select the checkbox for **Perform Preprocess/Perform Postprocess** or both.
4. In the **Preinputs** field, type the processors to be chained. The name of the processor is the processors bean name. The syntax is:

```
processors:PROCESSOR1,PROCESSOR2
```

For example:

```
processors:com.mashery.local.MasheryCustomProcessorOne,com.mashery.local.MasheryCustomProcessorTwo
```

You can also provide configuration data as input. The syntax is:

```
PROCESSOR1.parameter:value
PROCESSOR1.parameter:value
PROCESSOR2.parameter:value
PROCESSOR2.parameter:value
```

Note:

- In the pre-input configuration, there must not be more than one key with name "processors".
- The processors to be chained must be input as comma separated list.

5. In the **Postinputs** field, input the name of the adapter that the traffic manager can use to post-process the calls, then click **Save**.
    - The input syntax for the **Postinputs** fields must be the same as that of the **Preinputs** field.
    - If you opt for only **Perform Postprocess**, the input for **Postinputs** field must be as follows:

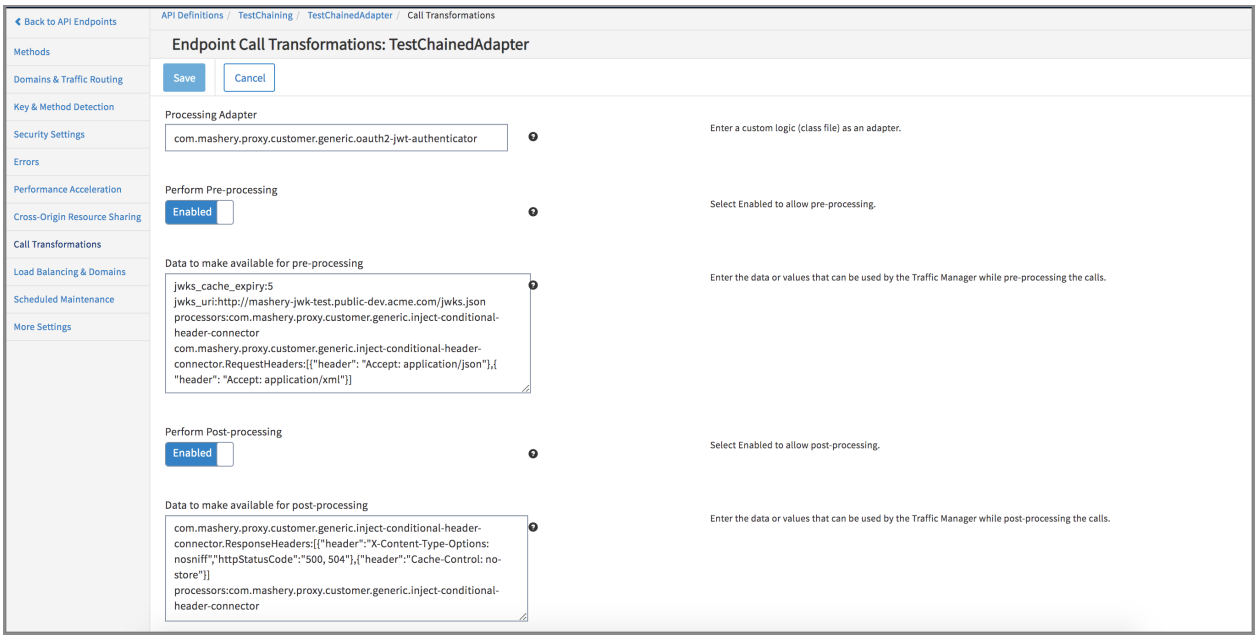
      ```
      processors:com.mashery.local.MasheryCustomProcessorOne,com.mashery.local.MasheryCustomProcessorTwo
      ```

      and

      ```
      PROCESSOR1.parameter:value
      PROCESSOR1.parameter:value
      PROCESSOR2.parameter:value
      PROCESSOR2.parameter:value
      ```

Chaining Processors in a Tethered Environment

The process for chaining processors in a tethered environment is the same as that of untethered. The interface of the control center appears as shown below:



What to do next

To test the changes, use the endpoint and make a traffic call.

FAQs

- **Do I need to install Gradle?**

  No. Gradle installation is not required. The SDK will take care of picking up the right versions of Gradle.

- **I already have Gradle. Will that affect working with the SDK?**

  No. The SDK will use the appropriate Gradle version.

- **Can I push the extracted SDK along with my adapter subprojects into Version control?**

  Yes. All required artifacts that need be version controlled are included and multi-authoring is governed by specific version control systems.

- **Can different sub projects contain different versions of same library?**

  Yes, but ensure the file names are different.

- **I have multiple Java SE versions, which one should I use?**

  It is recommended to use Java SE 8 or higher version. The SDK will take care of compiling to the required target version.

- **How do I change the java version to be used by the SDK?**

  Open gradle.properties file and add the following:

  ```
  org.gradle.java.home=/Path/To/Java SE HOME.
  ```

  The home directory will be different for different OSes.

- **How do I configure network proxy for Local Edition Local SDK?**

  Incase your internet access is governed by a network proxy, you can configure the SDK using proxy.

  For more information, see Configuring Network Proxy for Local Edition SDK.

Boomi References

Refer to these links to learn more about Boomi privacy policy, terms of service, and Boomi help documentation:

Privacy Policy
Terms of Service
Help Documentation