IBM Tivoli Directory Integrator 6.0
Server API Developers Guide

Version: 0.9, Date: 2006/03/01

1. Overview

The Server API of the IBM Tivoli Directory Integrator provides a set of programming calls that can be used to develop solutions and interact with the TDI Server locally and remotely. It also includes a management layer that exposes the Server API calls through the JMX interface.

The Server API includes calls that let you:

- Get information about the Directory Integrator Server
- Get information about components installed on the Server
- Read, Modify and Write configurations loaded by the Server
- Create and Load new configurations on the Server
- Start, Query and Stop AssemblyLines and EventHandlers
- Cycle manually through AssemblyLines
- Register for and Receive notifications for Server events
- Register for and Receive AssemblyLines and EventHandlers log messages

All calls can be invoked:

- *Locally, from the TDI Server JVM:*
  This type of access includes scripting in AssemblyLine or EventHandler hooks and also using the API from new components (Connectors, EventHandlers) implemented in Java and deployed on the Server
- *Remotely, from another JVM (on the local or a remote network machine), through RMI:*
  This type of access enables the implementation of solutions that remotely connect to TDI and manage processes within TDI or/and build business logic on top of TDI. It could be an application dedicated solely to TDI or an application that uses TDI to accomplish some of its goals.

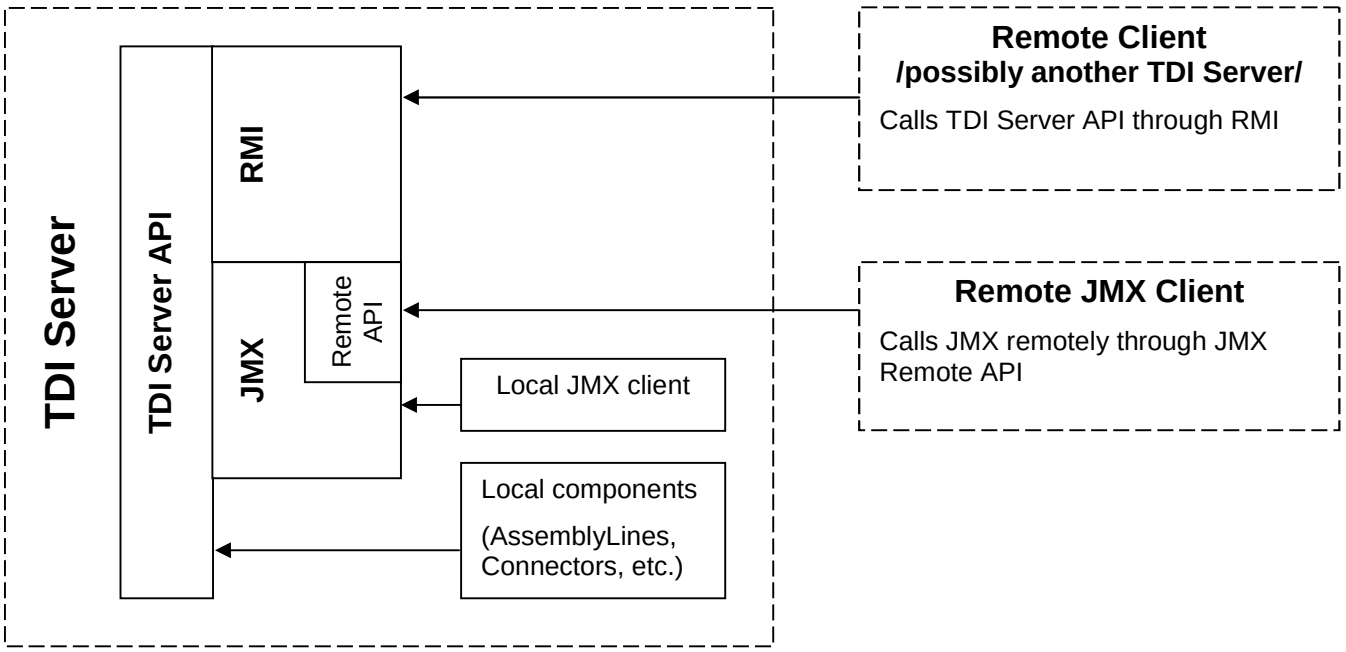
A management layer of the Server API exposes the Server API calls through JMX. This provides for Server manageability and enables plugging TDI into a managing infrastructure that speaks JMX. The JMX interface is accessible:

- Locally, as defined in the JMX 1.2 specification
- Remotely, through RMI as defined by the JMX Remote API 1.0 specification

The notifications emitted by the Server API internal engine are also available as JMX notifications.

Remote access to the Server API (including the JMX Remote API) is secured by using SSL with client and server authentication.

The different methods that can be used to access the TDI Server API are depicted on the diagram below:



Sample use case

A client (possibly an AssemblyLine on a TDI Server) needs to start an AssemblyLine on another TDI Server. The client could use the Server API and access it remotely through the RMI interface, using the Server API RMI client library.

In accordance with the security model described in the "**Error! Reference source not found.**" section, the client will first create an SSL session to the remote TDI Server using its own certificate. The Server will successfully authenticate the client if it has the client certificate in its trust store. Provided that the authentication is successful and the SSL channel is established, the client will be provided with an object that represents an entry point for calling Server API methods. Using that object the client will invoke the call for starting an AssemblyLine passing parameters that specify which AssemblyLine needs to be started. Before actually executing the method the Server API will check whether the client is authorized to execute that method – the identity of the client is determined through the client certificate used to establish the SSL channel. If the client is allowed to start this AssemblyLine the method will be executed and the AssemblyLine will be started, otherwise the method will not be executed and an error (exception) will be sent back to the client indicating that he is not authorized to perform this operation.

2. Local and Remote Server API interfaces

The Server API provides two sets of interfaces – for local and remote use. Both sets provide the same calls and functionality but they reside in different Java packages.

The package *com.ibm.di.api.local* contains the interfaces for local access and *com.ibm.di.api.remote* contains the interfaces for remote access to the Server through RMI.

Detailed specification of the local and remote interfaces and their methods can be found in the JavaDoc documentation shipped with the Directory Integrator (in "docs/api" folder under the root folder where the Directory Integrator is installed).

All interfaces in the remote package extend *java.rmi.Remote* and all their methods throw *java.rmi.RemoteException*. The interfaces for local access on the other hand do not extend *java.rmi.Remote* and their methods do not throw *java.rmi.RemoteException* which facilitates coding and is one of the reasons to have separate set of interfaces for local and remote access.

3. Server API structure

The structure of the local and remote interfaces is identical. The text below refers to the names of the Java interfaces only and is valid for the interfaces from both the local (*com.ibm.di.api.local*) and remote (*com.ibm.di.api.remote*) Server API Java packages.

The entry point to the Server API is the *SessionFactory* interface (*com.ibm.di.api.local.SessionFactory* for local use and *com.ibm.di.api.remote.SessionFactory* for remote use).

The *SessionFactory* interface provides a method called *createSession()*. It creates an API session for the user/entity that calls it and returns an object of type *Session*. It is this *Session* object that provides further access to the calls of the Server API.

Through the *Session* object one can get Server information or stop the Server itself, existing Config Instances can be obtained or new Config Instances can be loaded and created from scratch. Some of the calls of the *Session* object will return other Server API objects – for example *startConfigInstance(String aConfigUrl)* will return a *ConfigInstance* object. The *ConfigInstance* object gives access to the configuration data structure, to AssemblyLines and EventHandlers running in the Config Instance as well as calls for starting new AssemblyLines and EventHandlers. Some of its calls will also return Server API objects - *startAssemblyLine(String aAssemblyLineName)* for example returns an *AssemblyLine* object that you can use to query and perform different operations on the AssemblyLine.

To summarize, the *Session* object is the one that gives access to the hierarchy of Server API objects. All Server API calls are either invoked directly on the *Session* object or they are invoked on objects retrieved directly or indirectly through the *Session* object.

4. Security

Authentication is performed in the process of obtaining the *Session* object. Once obtained all methods called on the *Session* object or on other Server API objects retrieved directly or indirectly through this *Session* object are executed under the identity of the user that obtained the *Session* object.
Authorization is performed on each method call. Before executing the requested call, the Server will check whether the identity associated with the current session is authorized to execute that call.

Local sessions (from the TDI Server JVM) are always authenticated and are granted permissions to execute all Server API calls.

If the TDI Server is configured to accept a remote session without SSL, the session is authenticated and granted permissions to execute all Server API calls.
When the Server requires that SSL is used for remote sessions, the identity of the client is determined through the client certificate used to establish the SSL session. If the identity is

registered in the User Registry the authentication will be successful and the user will be able to execute the Server API calls he is granted access to.

5. Configuring the Server API

Configuring the Server API on the Server side includes specifying the relevant system properties in global.properties (solution.properties) and configuring the User Registry file.

5.1. Configuring the Server API properties

The Server API engine is configured through a set of properties in the global.properties file (or solution.properties if a solution folder is used). Refer to the TDI Administrator Guide, section "Remote Server\Configuring the Server API" for information on how to setup the Server API.

In Fixpack 1 of TDI 6.0 a new property has been added: **api.remote.nonssl.hosts**. It specifies a list of IP addresses to accept non SSL connections from (host names are not accepted). Use space, comma or semicolon as delimiter between IP addresses. This property is only taken into account when **api.remote.ssl.on** is set to *false*.

5.2. Setting up the User Registry

Refer to the TDI Administrator Guide, section "Remote Server\Authorization" for information and examples of how to setup the User Registry, assign user roles and encrypt/decrypt the User Registry file.

5.3. Remote client configuration

This section describes what setup is necessary on a remote client that will use the remote Server API.

Prerequisites:

- Java 1.4.2 or higher is required on the client side.

Configuring the client:

1. The following jar files must be included in the CLASSPATH of the remote side:
   - diserverapi.jar
   - diserverapirmi.jar
   - log4j-1.2.jar
   - miconfig.jar
   - miserver.jar
   - mmconfig.jar

   You can copy these jar files from the "<ITDI_root>/jars" folder.

2. When a non-IBM JVM or an IBM JVM prior to 1.4.2 is used on the client side, the file "xml.jar" from the "<ITDI_root>/_jvm/lib/" folder must be used on the client side. Copy "xml.jar" to the client machine and point the client's JVM system property **java.endorsed.dirs** to the folder where the "xml.jar" file is placed.

3. If custom non-TDI objects are used in the solution being implemented with the Server API (for example as Attribute values of an Entry that is transferred over the wire) the corresponding

Java classes have to be available on the client side as well. These classes must be serializable and they have to be included in the CLASSPATH of the client JVM.

5.3.1. SSL configuration of the remote client

There are two options for configuring SSL on the remote client. Which of them will be used depends on whether the Java System property **api.client.ssl.custom.properties.on** exists and its value.

Using Server API specific SSL properties
When the Java System property **api.client.ssl.custom.properties.on** is set to "true", then SSL is configured through the following TDI Server API-specific Java System properties:

- **api.client.keystore** – specifies the keystore file containing the client certificate
- **api.client.keystore.pass** – specifies the password of the keystore file specified by *api.client.keystore*
- **api.client.key.pass** – the password of the private key stored in keystore file specified by *api.client.keystore*; if this property is missing, the password specified by *api.client.keystore.pass*is is used instead.
- **api.truststore** – specifies the keystore file containing the TDI Server public certificate.
- **api.truststore.pass** – specifies the password for the keystore file specified by *api.truststore*.

Using the Server API specific SSL properties is convenient when your client application is using the standard Java SSL properties for configuration of another SSL channel used by the same application.

You can specify these properties as JVM arguments on the command line, for example:

```
java MyTDIServerAPIClientApp
      -Dapi.client.ssl.custom.properties.on=true
      -Dapi.truststore=C:\TDI\serverapi\testadmin.jks
      -Dapi.truststore.pass=administrator
      -Dapi.client.keystore=C:\TDI\serverapi\testadmin.jks
      -Dapi.client.keystore.pass=administrator
```

This example refers to the sample "testadmin.jks" keystore file shipped with TDI. Note that it contains both the client private key and also the public key of the TDI Server, so we use it both as a keystore and truststore.

Using the standard SSL Java System properties:
When the Java System property **api.client.ssl.custom.properties.on** is missing or when it is set to "false", then the standard JSSE system properties are used for configuring the SSL channel. Follow the standard JSSE procedure for configuring the keystore and truststore used by the client application.

You can specify these properties as JVM arguments on the command line, for example:

```
java MyTDIServerAPIClientApp
      -Djavax.net.ssl.keyStore=C:\TDI\serverapi\testadmin.jks
      -Djavax.net.ssl.keyStorePassword=administrator
      -Djavax.net.ssl.trustStore=C:\TDI\serverapi\testadmin.jks
      -Djavax.net.ssl.trustStorePassword=administrator
```

6. Using the Server API

6.1. Creating a local Session

If you are writing Java code that will be executed in the TDI Server JVM (for example a new Connector, or a Java class that you will access through scripting) and you want to execute Server API calls, you'll need a local Server API session.

You can obtain a local Server API session by calling:

```
import com.ibm.di.api.APIEngine;
import com.ibm.di.api.local.Session;

...

Session session = APIEngine.getLocalSession();
```

getLocalSession() is a static method of the *com.ibm.di.api.APIEngine* class. It creates and returns a new *com.ibm.di.api.local.Session* object. This session returned has admin rights and can execute all Server API calls.

6.1.1. Access to the Server API in a scripting context

You can get access to the Server API from a scripting context (for example from AssemblyLine hooks) by calling *Packages.com.ibm.di.api.APIEngine.getLocalSession()*.This will give you a local session object that you can use in your scripts.

Note: In TDI 6.1 a script object will be registered that will provide direct access to a local Server API session object.

6.2. Creating a remote Session

A client application that uses the remote Server API would first need to connect to the TDI Server and obtain a Server API Session.

Use the following Java code to lookup the RMI SessionFactory object and obtain a Server API Session.

```
import com.ibm.di.api.remote.Session;
import com.ibm.di.api.remote.SessionFactory;

...

SessionFactory sessionFactory = (SessionFactory)
  Naming.lookup("rmi://<TDI_Server_host>:<TDI_Server_RMI_port>/SessionFactory");

Session session = sessionFactory.createSession();
```

You need to replace <TDI_Server_host> and <TDI_Server_RMI_port> with the host and the RMI port of the TDI Server, for example
`Naming.lookup("rmi://127.0.0.1:1099/SessionFactory").`

The calls provided by the local and remote *Session* objects are identical. All examples below assume that you have already obtained a session and will operate in a remote context, i.e. the remote versions of the Server API interfaces will be used.

6.3. Working with Config Instances

The Config Instance is representing a configuration loaded on the TDI Server and the associated Server object. Each AssemblyLine or EventHandler is running in the context of a Config Instance. Through a Config Instance you can query the configuration of AssemblyLines, EventHandlers, Connectors, Parsers, start AssemblyLines and EventHandlers, get access to running AssemblyLines and EventHandlers and query their log files.

6.3.1. Getting access to running Config Instances

You can get access to all Config Instances running on the TDI Server by executing the following piece of code:

```
ConfigInstance[] configInstances = session.getConfigInstances();
for (int i=0; i<configInstances.length; i++) {
      // do something with configInstances[i]
}
```

The *getConfigInstances()* method will return an array with Config Instance Server API objects representing all Config Instances running on the Server.

6.3.2. Starting a Config Instance

In order to load a new configuration on the TDI Server you need to start a new Config Instance, pointing it to the XML configuration file:

```
ConfigInstance configInstance = session.startConfigInstance("testconfig.xml");
```

This will load the “testconfig.xml” configuration file (from the TDI solution folder) and start a new Config Instance object associated with that configuration. Once you get that Config Instance object you can use it to change the configuration itself, start AssemblyLines and EventHandlers or stop the Config Instance on the Server when you no longer need it.

6.3.3. Stopping a Config Instance

Assuming that you have a reference to the Config Instance Server API object, you can stop the Config Instance by calling:

```
configInstance.stop();
```

As you’ll need a reference to the Config Instance object, you have the following options:

- Keep that reference from where you started the Config Instance, i.e. *configInstance = session.startConfigInstance("testconfig.xml")*
- Retrieve the Config Instance object through its Config ID by calling *session.getConfigInstance (String aConfigId).* The Config ID is a unique identifier for each Config Instance running on the Server. It is created by the Server API when the

corresponding Server API Config Instance object is created. You can retrieve the Config ID through the Config Instance object by calling *configInstance.getConfigId()*.

- Iterate through all running Config Instances and find the one you need: *session.getConfigInstances()* will return an array of all running Config Instances.

6.4. Working with AssemblyLines

6.4.1. Getting access to the AssemblyLines available in a configuration

Assuming that you already have a reference to the Config Instance object, first you will have to obtain the MetamergeConfig object representing the configuration data structure for the whole Config Instance and then get the available AssemblyLines:

```
import com.ibm.di.config.interfaces.MetamergeConfig;
import com.ibm.di.config.interfaces.MetamergeFolder;
import com.ibm.di.config.interfaces.AssemblyLineConfig;

...

MetamergeConfig configuration = configInstance.getConfiguration();
MetamergeFolder configFolder =
    configuration.getDefaultFolder(MetamergeConfig.ASSEMBLYLINE_FOLDER);
String[] assemblyLineNames = configFolder.getNames();
if (assemblyLineNames != null) {
    for (int i=0; i<assemblyLineNames.length; i++) {
        System.out.println(assemblyLineNames[i]);

        // get the AssemblyLine configuration object
        AssemblyLineConfig alConfig =
            configuration.getAssemblyLine(assemblyLineNames[i]);
        // do something with alConfig ...
    }
}
```

This block of code prints to the standard output the names of all AssemblyLines in the configuration and demonstrates how to get hold of the AssemblyLine configuration objects. You can use the AssemblyLine configuration object to get more detailed information like what Connectors are configured in the AssemblyLine, their parameters, etc.

Note that the *MetamergeConfig*, *MetamergeFolder* and *AssemblyLineConfig* interfaces are not part of the Server API interfaces. They are part of the TDI configuration driver (see the import *clauses* in the example) and they are not remote objects. When *configInstance.getConfiguration()* is executed the *MetamergeConfig* object is serialized and transferred over the wire. Your code will then work with the local copy of that object.

6.4.2. Getting access to running AssemblyLines

You can get the active AssemblyLines either for a specific Config Instance or you can get all active AssemblyLines on the TDI Server for all running Config Instances.

Getting the active AssemblyLines for a specific Config Instance
You will need a reference to the Config Instance object. The following code will return all AssemblyLines currently running in the Config Instance:

```
AssemblyLine[] assemblyLines = configInstance.getAssemblyLines();
for (int i=0; i<assemblyLines.length; i++) {
    System.out.println(assemblyLines[i].getName());

    // do someting with assemblyLines[i]
}
```

Getting the active AssemblyLines for the whole TDI Server
If you want to get all AssemblyLines running on the Server, execute the following code:

```
AssemblyLine[] assemblyLines = session.getAssemblyLines();
for (int i=0; i<assemblyLines.length; i++) {
    System.out.println(assemblyLines[i].getName());

    // do someting with assemblyLines[i]

    // which Config Instance this AssemblyLine belongs to?
    ConfigInstance alConfigInstance = assemblyLines[i].getConfigInstance();
}
```

Note that this is executed at the session level and not for a particular Config Instance. If you need to know which Config Instance a running AssemblyLine belongs to, you can get a reference to the parent Config Instance object through the AssemblyLine object.

You can use the AssemblyLine Server API object to get various AssemblyLine properties, the AssemblyLine configuration object, AssemblyLine log, AssemblyLine result Entry as well as stop the AssemblyLine.

6.4.3. Starting an AssemblyLine

You can start an AssemblyLine through the Config Instance object to which the AssemblyLine belongs. You need to know the name of the AssemblyLine you want to start:

```
AssemblyLine assemblyLine = configInstance.startAssemblyLine("MyAssemblyLine");
```

You also receive a reference to the newly started AssemblyLine instance.

6.4.4. Starting an AssemblyLine in manual mode

The Server API provides a mechanism for manually running an AssemblyLine. In manual mode the AssemblyLine is not running in its own thread - instead, when you start it, it is only initialized. Iterations on the AssemblyLine are done in a synchronous manner when the *executeCycle()* method of the AssemblyLine object is called. This call blocks the current thread and when the AssemblyLine iteration is done it returns the result Entry object.

The following code will start the “TestAL” AssemblyLine in manual mode and execute 3 iterations on it. The result Entry from each iteration is printed to the standard output:

```
AssemblyLineHandler alHandler =
   configInstance.startAssemblyLineManual("TestAL", null);
Entry entry = null;
for (int i=0; i<3; i++) {
  entry = alh.executeCycle();
  System.out.println("TestAL entry: " + entry);
```

```
}
alHandler.close();
```

The *startAssemblyLineManual(String aAssemblyLineName, Entry aInputData)* method of the Config Instance object starts an AssemblyLine in manual mode and returns an object of type *com.ibm.di.api.remote.AssemblyLineHandler*. Through this object you can manually iterate through the AssemblyLine, you can pass an initial work Entry and various Task Call Block parameters, you can get a reference to the AssemblyLine Server API object and you can terminate the AssemblyLine when you are done with it.

You can imitate the AssemblyLine runtime behavior by calling *executeCycle()* until it returns NULL.

6.4.5. Starting an AssemblyLine with a listener

When you start an AssemblyLine through the Server API you can register a specific AssemblyLine listener that will receive notifications on each AssemblyLine iteration, delivering the result Entry, and also when the AssemblyLine terminates. Through this mechanism you can start an AssemblyLine from a remote application and easily receive all Entries produced by the AssemblyLine. The AssemblyLine listener will also deliver all messages logged during the execution of the AssemblyLine.

Your listener class must implement the *com.ibm.di.api.remote.AssemblyLineListener* interface (or *com.ibm.di.api.local.AssemblyLineListener* for local access). The methods you have to take care of are:

- *assemblyLineCycleDone(Entry aEntry)* – this method will be called at the end of each AssemblyLine iteration; the *aEntry* parameter represents the result Entry from the AssemblyLine iteration.
- *assemblyLineFinished()* – this method is called by the Server API when the AssemblyLine terminates.
- *messageLogged(String aMessage)* – this method is called by the Server API whenever a message is logged through the AssemblyLine logger. Thus you can get remote real time access to the log messages produced by the AssemblyLine.

A sample AssemblyLine listener class that only prints to the standard output all Entries received and all AssemblyLine log messages might look like:

```
import com.ibm.di.api.DIException;
import com.ibm.di.api.remote.AssemblyLineListener;
import com.ibm.di.entry.Entry;
import java.rmi.RemoteException;

public class MyRemoteALListener implements AssemblyLineListener {

  public void assemblyLineCycleDone(Entry aEntry)
    throws DIException, RemoteException
  {
    System.out.println("AssemblyLine iteration: " + aEntry.toString());
    System.out.println();
  }

  public void assemblyLineFinished()
    throws DIException, RemoteException
  {
```

```
    System.out.println("AssemblyLine terminated.");
    System.out.println();
  }

  public void messageLogged(String aMessage)
    throws DIException, RemoteException
  {
    System.out.println("AssemblyLine log message: " + aMessage);
    System.out.println();
  }
}
```

Once you have implemented your AssemblyLine listener class, you need to instantiate a listener object and pass it when starting the AssemblyLine:

```
MyRemoteALListener alListener = new MyRemoteALListener();
configInstance.startAssemblyLine("TestAL", null,
      AssemblyLineListenerBase.createInstance(alListener,true), true);
```

The *startAssemblyLine(String aAssemblyLineName, Entry aInputData, AssemblyLineListener aListener, boolean aGetLogs)* method specifies the name of the AssemblyLine, an initial work Entry, the listener object and whether you want to receive log messages – when *aGetLogs* is *false*, the *messageLogged(String aMessage)* listener method will not be called by the Server API.

Note that when you are registering a listener in a remote context, you have to wrap your specific listener in an AssemblyLine Base Listener class – this is necessary to provide a bridge between your custom listener Java class that is not available on the Server side and the Server API notification mechanism. A base listener class is created by calling the static *createInstance(AssemblyLineListener aListener, boolean aSSLon)* method of the *com.ibm.di.api.remote.impl.AssemblyLineListenerBase* class. You need to provide the object representing your listener class and specify whether SSL is used for communication with the Server or not (note that this is not an option for you to select whether to use SSL or not with this listener object; here you have to specify how the Server API is configured on the Server side – otherwise the communication for that listener will fail).

6.4.6. Stopping an AssemblyLine

You need a reference to the AssemblyLine object in order to stop it. You can keep the reference to the AssemblyLine object from when you started the AssemblyLine or you can iterate through all running AssemblyLines and find the one you need. Execute the following line of code to stop the AssemblyLine:

```
assemblyLine.stop();
```

6.5. Working with EventHandlers

Everything stated in section “Working with AssemblyLines” about AssemblyLines is valid for EventHandlers as well. You can work with EventHandlers in exactly the same manner using the corresponding EventHandler classes, interfaces and methods. Please consult the JavaDocs for the signatures of the EventHandler related classes, interfaces and methods.

Note: All EventHandlers will be deprecated in TDI 6.1. Connectors in Server or Iterator mode will be present and will provide the EventHandlers’ functionality.

6.6. Editing configurations

You can only edit a configuration loaded on the Server. The process of editing configurations consists of the following steps: (1) get the configuration object from the remote Server; (2) edit it locally; (3) set the edited configuration object back on the Server; (4) save the configuration to disk on the Server (do this if you want to keep your changes persistent in the configuration file):

```
// get the configuration object
MetamergeConfig configuration = configInstance.getConfiguration();

// modify locally the configuration object
// do something ...

// set the configuration object back on the Server
configInstance.setConfiguration(configuration);

// save the configuration on the disk on the Server
configInstance.saveConfiguration();
```

Note: In TDI 6.1 a new mechanism for editing configurations will be provided. It will not allow editing of the configuration of a Config Instance currently running on the Server. Instead the Server API will provide calls for loading and editing configurations independently of the running Config Instances. With TDI 6.1 you will still be able to use your code that does the actual modification of the configuration (the same MetamergeConfig interface is used) but you will have to obtain the MetamergeConfig object using different Server API calls.

6.7. Registering for Server API event notifications

The Server API provides an event notification mechanism for Server events like starting and stopping of Config Instances, AssemblyLines and EventHandlers. This allows a local or remote client application to register for event notifications and react to various events.

Applications that need to register and receive notifications should implement a listener class that implements the *DIEventListener* interface (*com.ibm.di.api.remote.DIEventListener* for remote applications and *com.ibm.di.api.local.DIEventListener* for local access). This class is responsible for processing the Server events. The *handleEvent(DIEvent aEvent)* method from the *DIEventListener* interface is where you need to put your code that processes Server events. Of course you may implement as many listener classes as you need, with different implementations of the *handleEvent(DIEvent aEvent)* method and register all of them as event listeners. A sample listener that just logs the event object might look like this:

```
import java.rmi.RemoteException;

import com.ibm.di.api.DIEvent;
import com.ibm.di.api.DIException;
import com.ibm.di.api.remote.DIEventListener;

public class MyListener implements DIEventListener
{
  public void handleEvent (DIEvent aEvent) throws DIException, RemoteException
  {
    System.out.println("TDI Server event: " + aEvent);
    System.out.println();
  }
}
```

Once you have implemented your listener you will need to register it with the Server API. If however you are implementing a remote application there is one extra step you need to perform before actually registering the listener object with the Server API – you need to instantiate and use a base listener object that will wrap the listener you implemented. The base listener class allows you to use your own listener classes without having the same Java classes available on the Server:

```
DIEventListener myListener = new MyListener();
DIEventListener myBaseListnener =
      DIEventListenerBase.createInstance(myListener, true);
```

The base listener object implements the same *DIEventListener* interface – its class however is already present on the Server and it can act as a bridge between your local client side listener class and the Server. A base listener object is created by calling the static method *createInstance(DIEventListener aListener, boolean aSSLon)* of the *com.ibm.di.api.remote.impl.DIEventListenerBase* class. The first parameter *aListener* represents the actual listener object and the second one specifies whether SSL is used or not by the Server API (note that this is not an option for you to select whether to use SSL or not with this listener object; here you have to specify how the Server API is configured on the Server side – otherwise the communication for that listener will fail).

When you have your listener object ready (or a base listener for remote access), you can register for event notifications through the *session* object:

```
session.addEventListener(myBaseListnener, "di.*", "*");
```

The *addEventListener(DIEventListener aListener, String aTypeFilter, String aIdFilter)* method of the *session* object will register your listener. The first parameter *aListener* is the listener object (or the base listener object for remote access), *aTypeFilter* and *aIdFilter* let you specify what types of events you want to receive:

- *aTypeFilter* specifies what type of event objects you want to receive. The currently supported events are:
  - **di.ci.start** – Config Instance started
  - **di.ci.stop** – Config Instance stopped
  - **di.al.start** – AssemblyLine started
  - **di.al.stop** – AssemblyLine stopped
  - **di.eh.start** – EventHandler started
  - **di.eh.stop** – EventHandler stopped

  You can either specify a specific event type like “di.al.start” or you can specify a filter using the “*” wildcard, for example “di.al.*” will register your listener for all Server events related to AssemblyLines, while a type filter of “*” or NULL will register your listener for all events.
- *aIdFilter* is only taken into account when *aTypeFilter* is not set to “*” or NULL. It lets you filter events depending on the object related to the event – for AssemblyLines this is the AssemblyLine name, for EventHandlers this is the EventHandler name and for Config Instances this is the Config Instance ID. For example, if you register your listener with *addEventListener(myListnener, "di.al.start", "MyAssemblyLine")* it will only be sent events when the “MyAssemblyLine” AssemblyLine is started and will not receive any other Server events.

If at some point you want to stop receiving event notifications from a listener already registered with the Server API, you need to unregister the listener. This is done through the same *session* object it was registered with by calling:

```
session.removeEventListener(myListener);
```

6.8. Getting access to log files

Section “Starting an AssemblyLine with a listener” described how we can use listeners to get AssemblyLine (or EventHandler) log messages in real time as they are produced.

The Server API provides another mechanism for direct access to log files produced by AssemblyLines or EventHandlers. This mechanism only provides access to the log files generated by the AssemblyLine or EventHandler SystemLog logger.
You don’t need a reference to an AssemblyLine or EventHandler Server API object to get to the log file. Also you can access old logs of AssemblyLines/EventHandlers that have terminated.

First you need to get hold of the *SystemLog* object:

```
SystemLog systemLog = session.getSystemLog();
```

You can then ask for all the log files generated by an AssemblyLine:

```
String[] alLogFileNames = systemLog.getALLogFileNames("C__Dev_ITDI_rs.xml",
    "TestAL");
if (alLogFileNames != null) {
  System.out.println("Availalbe AssemblyLine log files:");
  for (int i=0; i<alLogFileNames.length; i++) {
    System.out.println(alLogFileNames[i]);
  }
}
```

The *getALLogFileNames(String aConfigId, String aALName)* method is passed the Config ID (see “Stopping a Config Instance” for more details on the Config ID) and the name of the AssemblyLine. This will return an array with the names of all log files generated by runs of the specified AssemblyLine.

If you are interested in the last run of the AssemblyLine only, there is a Server API call that will give you the name of that log file only:

```
String lastALLogFileName = systemLog.getALLastLogFileName("C__Dev_ITDI_rs.xml",
    "TestAL");
System.out.println("AssemblyLine last log file name: " + lastALLogFileName);
```

When you have got the name of a log file you can retrieve the actual content of the log file:

```
String alLog = systemLog.getALLog("C__Dev_ITDI_rs.xml", "TestAL",
    lastALLogFileName);
System.out.println("TestAL AssemblyLine log: ");
System.out.println(alLog);
```

In cases where the log file can be huge you might want to retrieve only the last chunk of the log. The sample code below specifies that only the last 10 kilobytes from the log file should be retrieved:

```
String alLog = systemLog.getALLogLastChunk("C__Dev_ITDI_rs.xml", "TestAL",
     lastALLogFileName, 10);
System.out.println("Last 10K of the TestAL AssemblyLine log: ");
System.out.println(alLog);
```

The same methods are available for EventHandler log files. Consult the JavaDoc of the *com.ibm.di.api.remote.SystemLog* or *com.ibm.di.api.local.SystemLog* interfaces for the signatures and description of the EventHandler methods.

The Server API also provides methods for cleaning up (deleting) old log files.

You can delete all log files (for all configurations and all AssemblyLines and EventHandlers) older than a specific date. The sample code below will delete all log files older than a week:

```
Calendar calendar = Calendar.getInstance();
calendar.add(Calendar.DATE, -7);
systemLog.cleanAllOldLogs(calendar.getTime());
```

Another criterion you can use for log files clean up is the number of log files for each AssemblyLine or EventHandler. You can specify that you want to delete all log files except the 5 most recent logs for all AssemblyLines and EventHandlers:

```
systemLog.cleanAllOldLogs(5);
```

You can also delete the log files for AssemblyLines only or for EventHandlers only or for a specific AssemblyLine or EventHandler. The same two criteria are available: date and number of log files but in addition you can specify the name of an AssemblyLine or EventHandler or use calls that operate on all AssemblyLines or all EventHandlers only. Consult the JavaDoc of the *com.ibm.di.api.remote.SystemLog* or *com.ibm.di.api.local.SystemLog* interfaces for the signatures and the descriptions of all log clean up methods.

6.9. Server Info

Through the Server API you can get various types of information about the TDI Server itself like the Server version, IP address, operating system, boot time and information about what Connectors, Parsers, EventHandlers and Function Components are installed and available on the Server.

It is the *ServerInfo* object that provides access to this information. You can get the *ServerInfo* object through the *session* object:

```
ServerInfo serverInfo = session.getServerInfo();
```

You can then get and print out details of the Server environment:

```
System.out.println("Server IP address: " + serverInfo.getIPAddress());
System.out.println("Server host name: " + serverInfo.getHostName());
System.out.println("Server boot time: " + serverInfo.getServerBootTime());
System.out.println("Server version: " + serverInfo.getServerVersion());
```

```
System.out.println("Server operating system: " +
      serverInfo.getOperatingSystem());
```

You can also output a list of all Connectors installed and available on the Server:

```
String[] connectorNames = serverInfo.getInstalledConnectorsNames();
System.out.println("Connectors available on the Server: ");
for (int i=0; i<connectorNames.length; i++) {
  System.out.println(connectorNames[i]);
}
```

You can output more details for each installed Connector including its description and version:

```
String[] connectorNames = serverInfo.getInstalledConnectorsNames();
for (int i=0; i<connectorNames.length; i++) {
  System.out.println("Installed connector: ");
  System.out.println("    name: " + connectorNames[i]);
  System.out.println("    description: " +
      serverInfo.getConnectorDescription(connectorNames[i]));
  System.out.println("    version: " +
      serverInfo.getConnectorVersionInfo(connectorNames[i]));
  System.out.println();
}
```

6.10. Using the Security Registry

The Security Registry is a special Server API object that lets you query what rights a user is granted and whether he/she is authorized to execute a specific action. This is useful if an application is building an authentication and authorization logic of its own – for example the application is using internally a single admin user for communication with the TDI Server and it manages its own set of users and rights.

The Security Registry object is only available to users with the *admin* role. It is obtained through the *session* object:

```
SecurityRegistry securityRegistry = session.getSecurityRegistry();
```

You can then check various user rights. For example *securityRegistry.userIsAdmin("Stan")* will return *true* if Stan is granted the admin role; *securityRegistry.userCanExecuteAL ("User1", "rs.xml", "TestAL")* will return true only if Stan is allowed to execute AssemblyLine "TestAL" from configuration "rs.xml".
Check the JavaDoc of *com.ibm.di.api.remote.SecurityRegistry* for all available methods.

7. The JMX layer

The Server API provides a JMX layer. It exposes all Server API calls through a JMX interface locally and remotely (through the JMX Remote API 1.0).

Please refer to the TDI Administrator Guide, section “Remote Server\Configuring the Server API” for information on how to switch on and setup the JMX layer of the Server API for local and remote access.

7.1. Local access to the JMX layer

You can get a reference to the JMX MBeanServer object from the local Server JVM by calling

```
import com.ibm.di.api.jmx.JMXAgent;
import javax.management.MBeanServer;

...

MBeanServer jmxMBeanServer = JMXAgent.getMBeanServer();
```

The *getMBeanServer()* static method of the *com.ibm.di.api.jmx.JMXAgent* class will return an MBeanServer JMX object that represents an entry point to all MBeans provided by the JMX layer of the Server API. You can also register for JMX notifications with the MBeanServer object returned.
Note that the *getMBeanServer()* method will throw an Exception if it is called and the JMX layer of the Server API is not initialized.

7.2. Remote access to the JMX layer

The remote JMX access to the Server API is implemented as per the JMX Remote API 1.0 specification.

You have to use the following JMX Service URL for remote access:

service:jmx:rmi://<TDI_Server_host>/jndi/rmi://<TDI_Server_host>:<TDI_Server_RMI_port>/jmxconnector

You need to replace <TDI_Server_host> and <TDI_Server_RMI_port> with the host and the RMI port of the TDI Server, for example *service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxconnector*.

The sample code below demonstrates how a remote JMX connection can be established:

```
import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

...

JMXServiceURL jmxUrl = new
    JMXServiceURL("service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxcon
    nector");
JMXConnector jmxConnector = JMXConnectorFactory.connect(jmxUrl);
```

```
MBeanServerConnection jmxMBeanServer = jmxConnector.getMBeanServerConnection();
```

Similarly to the local JMX access the *MBeanServerConnection* object is the entry point to all MBeans and notifications provided by the JMX layer of the Server API. Once you get hold of that object you are in the JMX world.

For example, you can list all MBeans available on the JMX Server:

```
Iterator mBeans = jmxMBeanServer.queryNames(null, null).iterator();
while (mBeans.hasNext()) {
  System.out.println("MBean: " + mBeans.next());
}
```

7.3. MBeans and Server API objects

The JMX layer wraps the Server API objects in MBeans. The access to the MBeans is however straightforward - you can directly look up an MBean through the *MBeanServerConnection* object.

There is no session object in the MBean layer (the session and the security checks are managed through the RMI session). The methods for creating, starting and stopping Config Instances that exist in the Server API Session object can be found in the *DIServer* MBean in the JMX layer.

A list of the Server API MBeans available at some time on a TDI Server might look like this:

- ServerAPI:type=ServerInfo,id=192.168.113.222
- ServerAPI:type=ConfigInstance,id=C__Dev_ITDI_11_11_fp1_rs.xml
- ServerAPI:type=AssemblyLine,id=AssemblyLines/longal.618794016
- ServerAPI:type=DIServer,id=winserver
- ServerAPI:type=SystemLog,id=SystemLog
- ServerAPI:type=SecurityRegistry,id=SecurityRegistry
- ServerAPI:type=Notifier,id=Notifier

Each Config Instance, AssemblyLine or EventHandler is wrapped in an MBean. When the Config Instance, AssemblyLine or EventHandler is started the MBean is created automatically and it is automatically removed when the Config Instance, AssemblyLine or EventHandler terminates.

Refer to the JavaDoc of the Java package *com.ibm.di.api.jmx.mbeans* for all available MBeans, their methods and attributes.

7.4. JMX notifications

The JMX layer of the Server API provides local and remote notifications for all Server API events (see “Registering for Server API event notifications”). You have to register for JMX notifications with the *Notifier* MBean.

The JMX notification types are exactly the same as the Server API notifications:

- **di.ci.start** – Config Instance started
- **di.ci.stop** – Config Instance stopped
- **di.al.start** – AssemblyLine started
- **di.al.stop** – AssemblyLine stopped
- **di.eh.start** – EventHandler started

- **di.eh.stop** – EventHandler stopped

8. Miscellaneous

8.1. Concurrent use

The Server API does not isolate or coordinate simultaneous access by multiple users.

If for example several users are updating a Config Instance configuration (the *MetamergeConfig* object) the one that sets the configuration object back last, i.e. last calls *ConfigInstance.setConfiguration(…)*, will have his version of the configuration object saved, possibly overwriting the changes made by other users.

Note: In TDI 6.1 a new mechanism for editing configurations will be provided that will protect configurations against concurrent modifications. With TDI 6.1 you will still be able to use your code that does the actual modification of the configuration (the same MetamergeConfig interface is used) but you will have to obtain the MetamergeConfig object using different Server API calls.