1. Design Goals behind 6.1

The 6.1 version has been designed and built with the perspectives and requirements of our many stakeholders in mind. Some of this is based on direct feature requests and suggestions for improvement. The online discussion forums also provide feedback on the types of scenarios being solved, systems being connected and areas where users struggle. And, of course, was have the invaluable insights of our highly motivated TDI support staff, with a deep understanding of real-world integration, earned at the front lines, knee-deep in technical and design issues alongside our users.

For TDI developers, we continue to make TDI more flexible, more consistent, better documented and easier to use. As TDI projects grow in size and complexity, features for solution and component sharing have been added, as well as tools to facilitate deployment and solution handover. For TDI administrators, improvements have been made to significantly boost the reliability and manageability of TDI solutions and the Server itself.

As part of Tivoli Security, we take advantage of the wealth of security libraries and expertise in the portfolio, reducing the security exposure of TDI solutions – both newly crafted ones, as well as those built with previous versions.

And finally, thought and effort have gone into making TDI an easier (and more desirable) bundle. This means meeting strict guidelines and requirements for simple component-based installation, ease-of-use and flexible bundling control and integration.

Each of these goals is addressed by a set of new features and enhancements to existing ones. This introduction highlights how some of the additions and enhancements in TDI 6.1 Express work in concert to address one or more goals.

1.1 Overview

Although these following sections are not exhaustive, they will give you a quick overview to how changes in this version have been designed to support one or more of the above goals.

1.1.1 Ease of use/development

A big news item in TDI 6.1 Express is the AssemblyLine Debugger (page 81). This habit-forming feature lets you pop the hood on any AssemblyLine and then step your way through the processing flow, from component to component – even from Hook to Hook; all the time watching and changing data interactively. It even works with Server Mode Connectors. And, as you can imagine, this tool is a powerful, visual teaching aid for understanding how TDI works.

Improvements to the features for creating and editing Configs on remote TDI Servers make building and maintaining a distributed TDI implementation even easier.

While version 6.0 introduced the concept of the Branch and Loop, removing the need for "hiding" much of your AL logic away in Hooks, this release carries this further by adding Else-If and *Else* Branches, as well as the new *Switch* construct (page 19). Together with AssemblyLine Operations (page 50), the Switch makes building flexible, reusable ALs even easier.

TDI 6.1 Express has also *harmonized* the way that you work with Properties (External Properties, Global/Solution Properties, Java Properties, and so forth) by offering a unified Property Store framework (page 62). In addition, the new TDI Expressions feature lets you exploit the new Properties framework – as well as any other configuration

settings and data in your solution – when setting up your components, building Attribute Maps, Link Criteria and Conditions that are dynamically controlled based on AL state and external commands (page 69)[1].

Used often for synchronizing data, TDI 6.1 Express boasts several improvements to its Change Detection componentry (page 78) that make these features work faster and result in more robust solutions.

There are several new components, like the SMTP-based SendMail FC (page 75) for sending emails and the Java FC (page 73) that lets you browse Java class libraries and then make function calls – all without writing a line of script! Other existing components have been enhanced: for example, the JDBC Connector (page 77) now offers Delta mode for simplifying database synchronization.

The enhanced AssemblyLine Connector (page 52) leverages AssemblyLine Operations feature (page 50) for building 'Adapters'. The concept of Adapters is that you construct an AssemblyLine with Operations defined for each branch of logic in the AL – like 'EnableAccount', or 'ReturnGroupMembers' – and then you Publish the AssemblyLine (page 55). This pretty much squeezes the entire AL into a single Connector with a Mode setting for each Operation. Easy to share, easy to reuse.

There are also some usability improvements, like the Tooltips for AssemblyLine components that now show more information, including enabled Hooks; as well the new Copy-to-Library feature Attribute Maps, and even copying and pasting of individual Attributes between maps (page 57).

And to make code sharing simpler, TDI offers a Resource organizer (page 54). Scripts, AttMaps, Connectors, ALs and more can be dragged in and out of your Resources and into your Configs, Your Resources are mapped to a directory structure on disk (personal or shared over the network) which in turn can be organized as you and your team wants.

1.1.2 Ease of Deployment

This release boasts a new version of the TDI Administration and Monitoring Console, AMC v.3 (page 79), which not only unifies a multi-Server/solution installation into a single status overview (with custom monitoring and point-and-click drill-down), but also features the new Action Manager (page 80): a failure/response engine that lets you build high availability into any TDI deployment.

The "jars" folder has been reorganized (page 15), giving you a "Patches" sub-directory that the loader uses to override any class libraries found elsewhere. So now you can apply fixes to libraries and components without tampering with the originally installed software.

There is a Command Line Interface (CLI) for TDI (page 61) that lets you load, run and control your solutions directly from the command prompt or scripts/batch-files. And you can now use custom log Appenders (page 38) to tailor log-handling according to customer requirements,

Not to mention the new Property Store framework (page 62) that unifies handling of all properties (Ext Props, solution/global, Java, and more) and TDI Expressions (page 69). Expressions take the ability to tie and External Property to a component parameter even further, allowing you to define settings that are based on any Property Store, Attribute value or even other parameters. So if you instruct your Connector or FC to initialize whenever a parameter changes (page 44), and this parameter is based on a property value, then you can cause the component to re-initialize by changing the property via AMC or the CLI!

[1] Be also sure to check out the new Command Line Interface (CLI) described on page 61. This tool even lets you retrieve and set properties (in any Property Store) from the command line.

1.1.3 Ease of Bundle

One major undertaking in this release has been the removal of code and libraries that where not in compliance with IBM regulations and legal practice (for example, replacing the Rhino JavaScript engine with the IBM JSEngine – described on page 15). This in itself makes it easier for product teams to consider TDI for bundling.

Also, TDI has been certified as an *Express* product in this latest version, a quality stamp with regards to install-ability and ease-of-use; but without sacrificing any of the flexibility or scalability of the toolkit or the solutions you can build.

The API has been enriched to include access to new features, and improvements have been made to existing libraries.

Finally, the new Solution Install provides control over which components to install, making it easier for a bundler to include only those parts of the system as desired.

1.1.4 Security

There is a slew of new and improved features that help you secure your solutions more easily than ever, including the new TDI Server Hooks (page 17) that let you customize TDI's security model. One feature that makes securing your Config easier is the automatic secure storage of password parameters (page 23), as well as Attribute protection (page 24) to keep sensitive data out of logs and trace files. Furthermore, individual properties – both user-defined, and configuration settings (Global-/Solution-Properties) can now be encrypted (page 67).

Both the API (page 31) and the AMC v3 console (page 79) support authentication via LDAP, and the TCP-based components now offer SSL support as well as to return relevant TCP header information.

Also added/improved is documentation around configuring and administrating the security features of TDI and AMC v3, including examples to make these features easier to understand and work with.

1.1.5 High Availability

Even though AMC v3, Action Manager the CLI and TDI Expressions have been mentioned before, it's worth repeating here in the context of High Availability. TDI now also supports custom exit/return codes (page 22), and there is even a feature for signaling JVM failures (page 21).

Connector Reconnect has been turned into broader Connection Failure feature (page 45), and there is finer control over Connector and FC initialization (page 44). There is also support for custom TDI 6.1 Express events notifications (page 30) so you can build inter-AL/Server communications using your own vocabulary. Furthermore, the new System Queue (page 57) lets you easily send data between AssemblyLines pumping away in different parts of your infrastructure.

In addition to the AssemblyLine Pooling provided whenever you use a Server Mode Connector, TDI 6.1 Express now lets you define Global Connector Pools (page 43) that are shared between ALs. And using the enhanced AL Connector (page) this lets you also set up global AL Pools.

2. What's New

There's a lot to absorb here, so to help you get your head around all the new and improved features an index has been added to the end of this document that should give you context/topic-specific views of the contents here.

Changes and new features can be found throughout TDI: both in the CE and Server, as well as in AMC (now v3) and new stand-alone tools.

2.1 Compatibility

The following information is described in greater detail elsewhere in this document. A summary is provided here for your convenience.

2.1.1 Replace of Rhino JavaScript with IBMJS

The only requirement that may cause some compatibility issues is the replacement of the Rhino JavaScript engine with the IBM jsengine. Great care has been taken to minimize possible compatibility issues, and the IBM jsengine is being updated as issues are found. See page 15 for more information on this change.

2.1.2 TMSXML Format for all Messages

PD Serviceability Enhancements have been introduced, includingTMSXML formatting of messages for all TDI components (Connectors, EventHandlers, Parsers, etc). In TDI 6.1 Express, the TMSXML format messages are automatically converted to properties file during the build process. Regardless, TDI 6.1 Express components may be dropped into TDI 6.0 build.

2.1.3 Cloudscape/Derby Upgrade

The Cloudscape database bundled with TDI, and used by default to provide System Store functionality, has been upgraded to the latest version, Derby version 10.1. As a result, existing Cloudscape database must be converted to the newest db level. This is handled by the installer for the standard "Cloudscape" directory (default System Store setup), which results in a new "TDISysStore" directory/database after completed conversion.

The conversion utility – which is found under the "tools\CSMigration" folder of the installation directory – can also be run manually for any other databases that you need to convert:

```
migrateCS <newCSdirectoryDB> <newCSdirectoryDB>
```

Note that the new CS directory/db must be different from the old one.

2.2 Tombstones

6.0 and previous versions of TDI did not keep track of configurations or AssemblyLines that had terminated. Therefore, administrators had no way of knowing when their AssemblyLines last ran – without going into the log of each one. Bundlers or integrated applications that initiated ALs via API calls could not query their status after they'ld terminated.

The solution is the new **Tombstone Manager** that creates a *tombstone* for each AssemblyLine[2] as it terminates. This marker stores a timestamp, as well as the AL exit status and other information as show in the table below. This enables the following functionality.

- It enables the new AMC v3 to provide desired status screen of an entire TDI server configuration, including *last-run* info.
- The Action Manager (part of AMC v3) uses tombstones for AL scheduling and health checks.
- Bundlers and other clients of the TDI Server API (including ALs that launch other ALs) can query status on AssemblyLines that they run asynchronously
- The new Command Line Interface (CLI - described on page 61) tool can retrieve status on running and stopped ALs

Tombstone managed can be configured for a Config via the Configs -> Tombstones folder in the Config Browser, or specific AssemblyLines via the Config tab for each AL.

2.2.1 Tombstone Data in the System Store

The Tombstone Manager uses the System Store for persisting its data. This is done in the form of a table

Each tombstone contains the following information:

Data item	*Type*	*Description*
Component Type ID	Number	0 – Config Instance 1 – AssemblyLine 2 - EventHandler
Event Type ID	Number	0 – Stop Event
Start Time	dateTime	When ConfigInstance, AssemblyLine and EventHandler are started
Tombstone Create Time	dateTime	When the tombstone record is created
Component Name	String	AssemblyLine name, EventHandler name or Config Instance ID
Configuration	String	Config ID for AssemblyLines and EventHandlers
Exit Code	Number	0 – Normal termination 1 – Error
Error Description	String	When the component terminated with an error, this field contains text description of the error.
GUID	String	Global Unique Identifier – a unique string for each tombstone record created

[2] Although EventHandlers are deprecated, they are supported by the Tombstone Manager in order to enrich existing solutions.

Statistics	com.ibm.di.entry.Entry	A TDI Entry object whose Attributes keep various statistics data like number of Entries retrieved, modified, deleted, etc. Only relevant for AssemblyLines. If new data has to be added in the future new attributes might be added to this Entry object.
UserMessage	String	A user defined message associated with this tombstone. The user is provided with an interface to specify that through scripting on AssemblyLine runtime.

The **Statistics** Attribute returns an Entry object that contains its own set of Attributes. These contain the same information as is displayed in the log when an AL completes. These Attributes are:

- "add" – total number of "add" operations performed
- "mod" – total number of "modify" operations performed
- "del" – total number of "delete" operations performed
- "get" – total number of "getNext" (Iterations) performed
- "request" – total number of requests accepted when there is a Server mode Connector in the AssemblyLine.
- "callReply" – total number of "callReply" operations performed
- "err" – total number of errors encountered
- "skip" – total number of 'skip' operations performed
- "lookup" – total number of "lookup" operations performed
- "ignore" – total number of "ignore" operations performed
- "reconnect" – total number of "reconnect" operations performed
- "exception" – the exception text if the component terminated with an exception
- "getTries" – total number of "getTries" operations performed
- "getclientTries" – total number of "getclientTries" operations performed
- "nochange" – total number of "nochange" operations performed
- "branchtrue" – total number of "branchtrue" operations performed
- "branchfalse" – total number of "branchfalse" operations performed
- "loopstart" – total number of "loopstart" operations performed
- "loopcycles" – total number of "loopcycles" operations performed
- "reconnectTime" – time in ms after last reconnect

2.2.2 Config -> Tombstones

TDI now offers a Tombstone Manager which keeps track of when AssemblyLines *last stopped*. Previous versions allowed you to query the status on running ALs, but no information on when they last ran. In order let you configure Action Manager to trigger on rules like "AL has not run for 24 hours", the Tombstone Manager is a necessity.

However, you still have to turn this feature on. This is done under the Config node in the Config Browser where there is now a new Tombstones selection. Choosing this item brings up the Tombstones dialog which has three checkboxes for turning on Tombstone creation for Configs, ALs and EHs.

This means whenever a *stop event* occurs and that setting is enabled (e.g. if Config Tombstones is enabled and a Config Instance is stopped; or similarly for ALs and EHs) then a tombstone is written that can later be accessed via API calls, or used by the AMC v3 Action Manager for building a trigger.

2.3 New Hooks

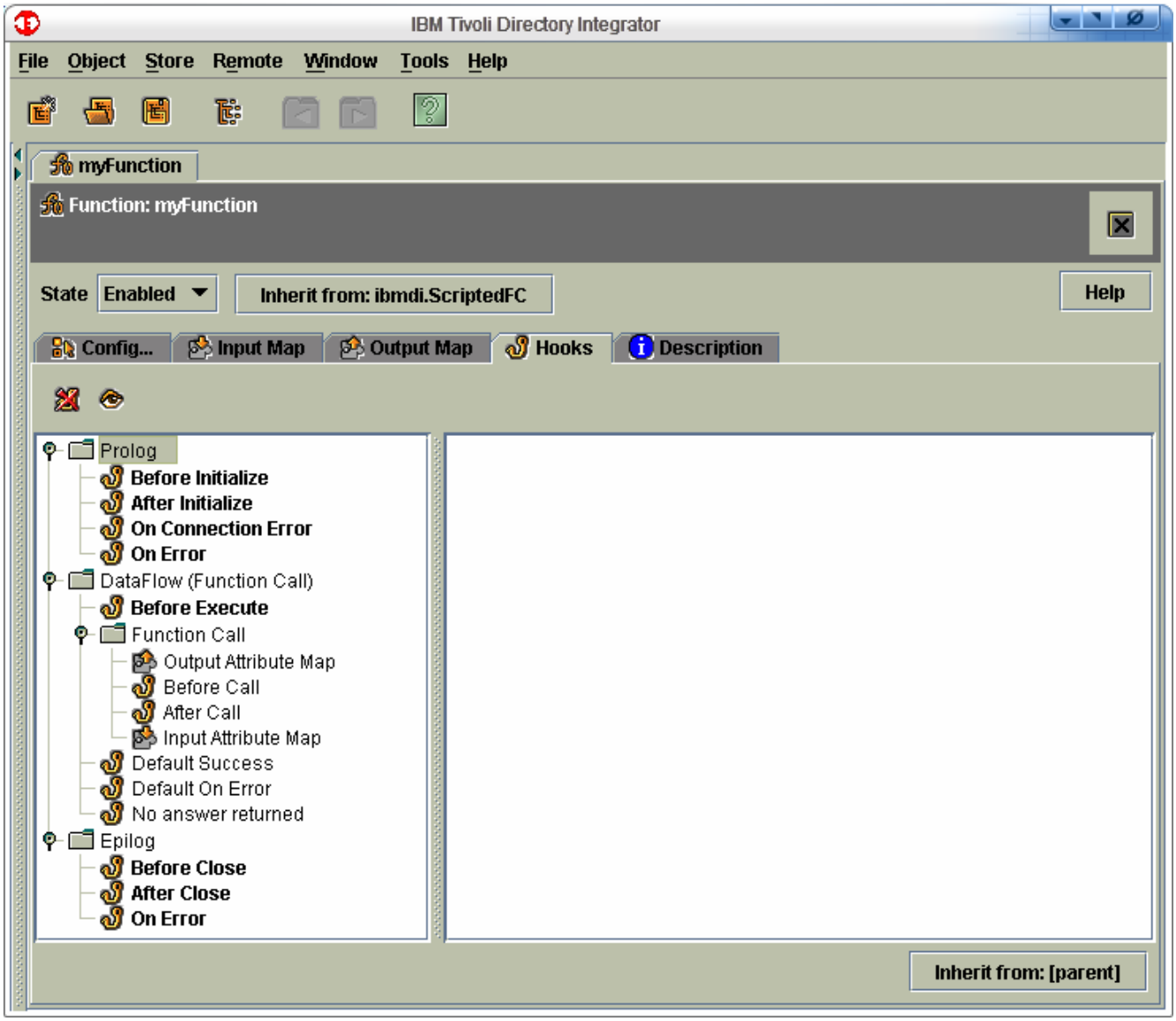
2.3.1 Add additional hooks to FCs

The following additional hooks were added to Function Components:

- Hook.before_initialize=Prolog - Before Initialize
- Hook.after_initialize= Prolog - After Initialize
- Hook.after_initialize= Prolog - On Connection Error (called if init. exception is connection/io type)
- Hook.initialize_fail= Prolog - On Error

- Hook.before_execute=DataFlow - Before Execute

- Hook.before_close= Epilog - Before Close
- Hook.after_close= Epilog - After Close
- Hook.close_fail=Epilog - On Error

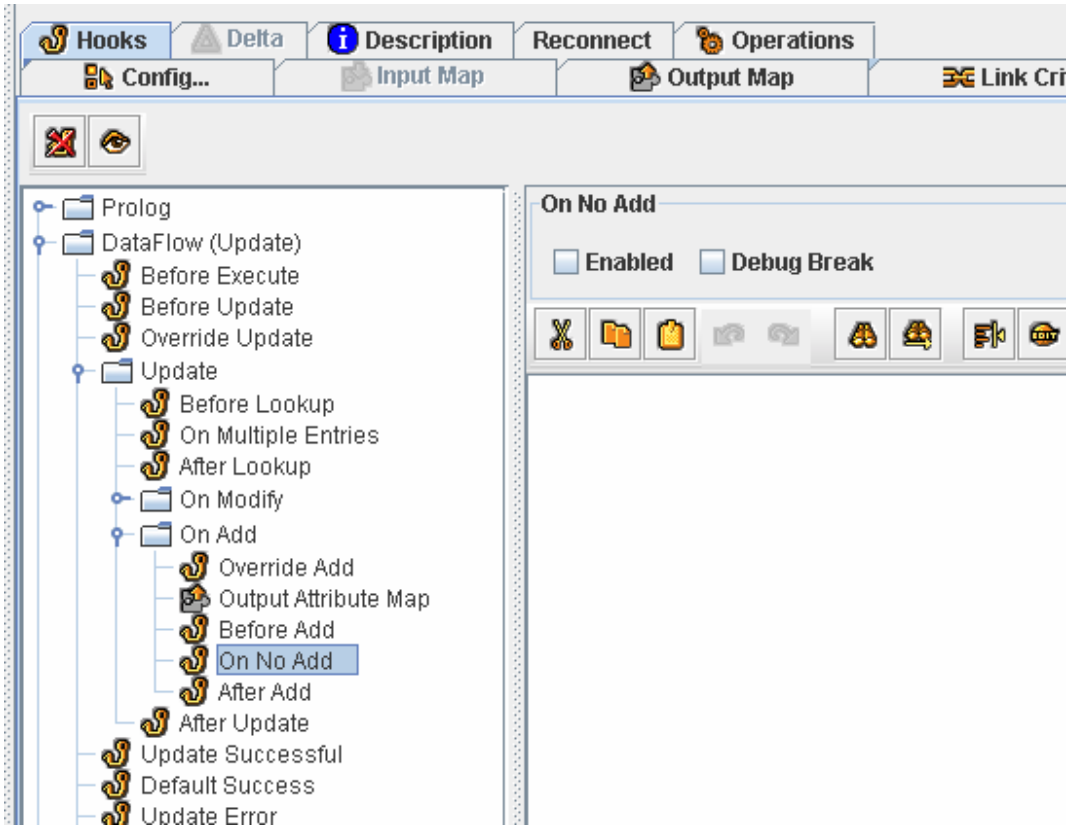


2.3.2 Operation Abandon Hooks

A new Hook was added for *add* operations (AddOnly and Update Modes), "On No Add", which is called when the Entry to add (i.e. the *conn* object) is empty. After this Hook is called, nothing is added, and the connector will log an *ignore*. For *modify* operations, no new Hook is required, since we use the "On No Changes" Hook, which can also be active due to Compute Changes behaviour in Update Mode.

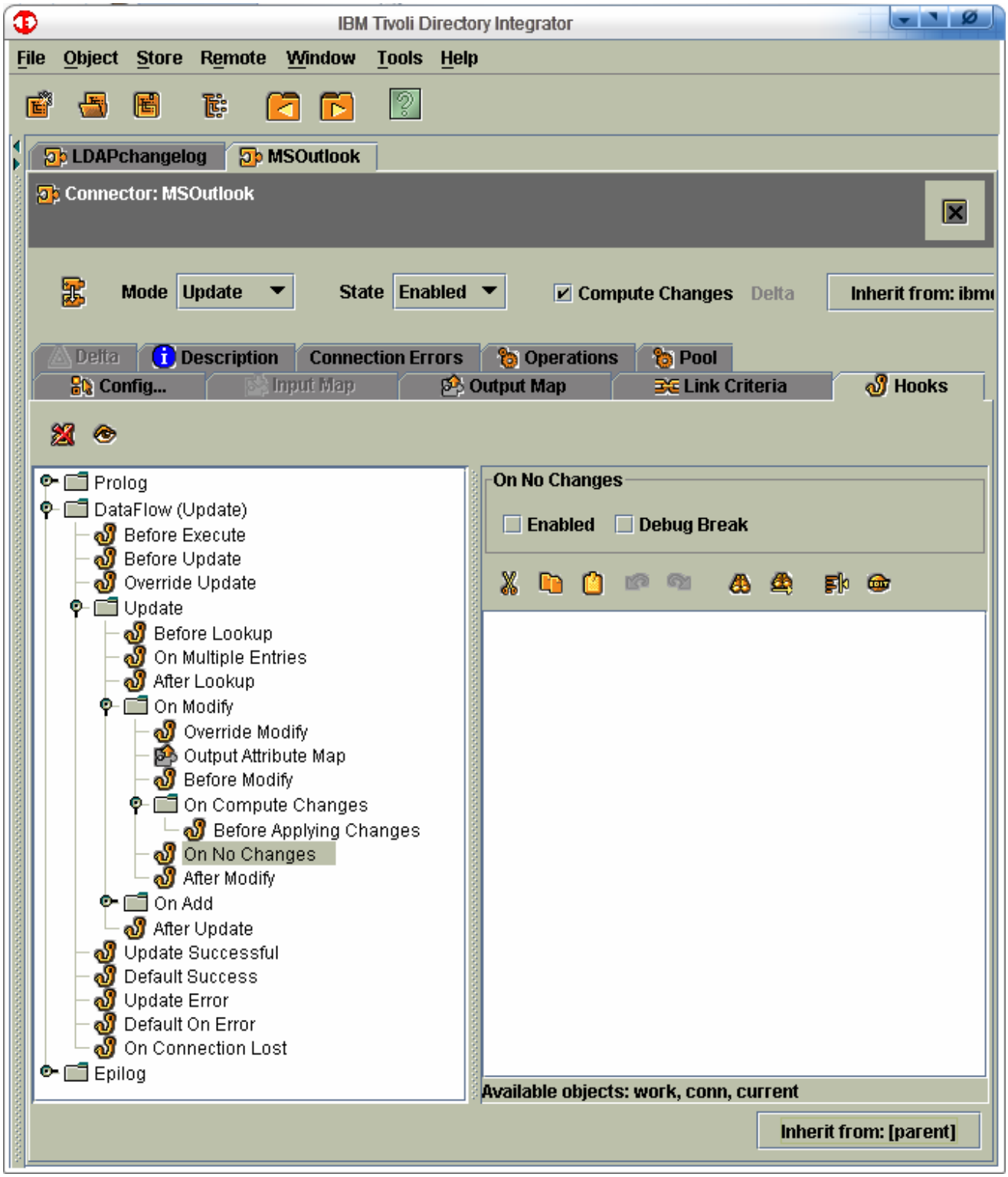
2.3.2.1 Changes for add operations (AddOnly and Update modes):

TDI no longer throws an ignoreEntryException when the attribute map is explicitly empty, as it used to do. Instead, flow goes to the Before Add Hook where Attributes can be added to conn via script. If conn is still empty after the Before Add Hook, then the new On No Add Hook (internal name: abandon_add) is called. At the same time, TDI logs a warning about an explicitly empty map (again, only if conn remains empty).



2.3.2.2 Changes for modify operations (Update mode):

TDI no longer throws an ignoreEntryException when the attribute map is explicitly empty, as it used to do. Instead, flow goes to the Before Applying Changes Hook where Attributes can be added to conn via script. Note that this Hook will only be called if Compute Changes is turned off; or if there are changes that need to be written. If conn is still empty after the Before Applying Changes Hook, then the On No Changes Hook is called. At the same time, TDI logs a warning about an explicitly empty map (again, only if conn remains empty).


IBM Tivoli Directory Integrator
File Object Store Remote Window Tools Help
LDAPchangelog
MSOutlook
Connector: MSOutlook
Mode Update
State Enabled
Compute Changes
Delta
Inherit from: ibm
Delta
Description
Connection Errors
Operations
Pool
Config...
Input Map
Output Map
Link Criteria
Hooks
Prolog
DataFlow (Update)
Before Execute
Before Update
Override Update
Update
Before Lookup
On Multiple Entries
After Lookup
On Modify
Override Modify
Output Attribute Map
Before Modify
On Compute Changes
Before Applying Changes
On No Changes
After Modify
On Add
After Update
Update Successful
Default Success
Update Error
Default On Error
On Connection Lost
Epilog
On No Changes
Enabled
Debug Break
Available objects: work, conn, current
Inherit from: [parent]

2.4 JavaScript (hello, JSengine; goodbye Rhino and BSF)

TDI 6.1 Express no longer uses the Rhino JavaScript engine, replacing this with IBM's own JSengine.

The use of the IBM JSengine is optimized in that ScriptEngine saves the "compiled" expressions derived from evaluating a JavaScript code segment. Hence, ScriptEngine provides similar functionality to Rhino for enhancing execution speed. The TDI ScriptEngine (which wraps the JSengine inside TDI) works in the same way as it did with Rhino, so your ALs won't notice any difference. And you be sure and tell us if they do!

Advantages of the IBM JSengine include:

- Better platform for debugging features. Many of the new Debugger/Stepper features are possible because of the jsengine. Check out section *2.49 AssemblyLine Debugger/Stepper* on page *81* for more details on this powerful tool.
- The eval() method now also allows registration of functions, not just variables as it did in Rhino.
- Improved error messages (which warrants a section on its own; the next one).

2.4.1 Improved error messages

The IBM JSengine JavaScript engine (jsengine) provides better clear and precise error messages (where possible). To facilitate finding syntax errors, the error message and the exception from the IBMJS engine are displayed. It also shows the line and column numbers in the script.

Example of jsengine information displayed:

If your script had the following code in its third line:

```
this is an error;
```

Then the following error information is displayed.

```
com.ibm.jscript.parser.ParseException: Syntax error at line 3, column 6. Invalid 'is'.
```

The first error encountered in the above line was "is", since "this" is a reserved word in JavaScript.

2.4.2 No support for other script languages than JavaScript

It is no longer possible to choose the scripting language for a component or AssemblyLine. Scripting is always done in JavaScript.

2.5 Library Loader Enhancements

In addition to providing a manual method for the customer to be able to specify a single additional directory to pick up libraries from that are loaded by TDI, specific files and directories of jar files can be configured.

In addition, the organization of the jars directory has been changed to provide better organization of class files.

NOTE: If the value of the com.ibm.di.loader.userjars variable is set in global.properties or solution.properties, idi.inf files in JAR files there are used. If the value is set later on, idi.inf files in JAR files there will not be used; idi.inf files in JAR files added using the addUserjars() method will not be used; JAR files containing such information should be placed in the jars folder, or should be added using the om.ibm.di.loader.userjars variable in global.properties/solution.properties.

2.5.1.1 Custom Specification of JAR files

“”

The previous version provided a property, “com.ibm.di.loader.userjars”, for specifying a single directory containing jar files. This property is extended in 6.1 to allow solution builders to specify more than one directory or jar file, separated by the java property "path.separator". The “path.separator” is ":" on UNIX/Linux platforms and ";" on Windows platforms. Directories are searched recursively by the TDILoader for jar files containing classes and resources. The TDI loader behaves otherwise as it always has, and only files with a ".zip" or ".jar" extension are searched.

A new method to UserFunctions, loadJarFile(String) has also been added. The method allows a user to dynamically add jar files while running TDI. The String parameter can be either a single jar file or a directory containing jar files (or new directories, which are searched recursively). For example,

```
system.addLoaderJarFile("c:\\myjardirectoy");
```

NOTE: JAR files added using the com.ibm.di.loader.userjars or the loadJarFile() method, cannot contain idi.inf files (the information in the idi.inf files will not be used, and is only read during initial loading at system startup). JAR files containing such information must be placed in the jars folder. This is because of the way TDI constructs the system namespace today.

2.5.1.2 Restructuring of the TDI “jars” sub-directory

The TDI ”jars” directory has been restructured. The JAR files in this folder have been moved into subdirectories to organize them better and the TDILoader has been updated to understand the directory changes.

The following specifies the new directory structure under the ”jars” directory and what JAR files go under each category:

- ”connectors” - Contains all TDI Connector JAR files.
- ”eventhandlers” - Contains all TDI EventHandler JAR Files.
- ”functions” - Contains all TDI Function Component JAR Files.
- ”parser” - Contains all TDI Parser JAR files.
- ”plugin” - JAR files needed for the TDI server to communicate with the plugins.
- ”common” - The core server JAR files.
- ”patches” - Used to hold patch JAR files for the support stream.
- “3rdparty\IBM” - Contains all of the JAR files our product needs from other IBM products.
- “3rdparty\others” – Contains all of the JAR files our product needs from non-IBM products.
- ”ce” – Contains the JAR files that are needed by the CE and not the server.

The TDILoader has been updated to first search the directories in the following order in every directory:

1. "patches"
2. "common"
3. "connectors"
4. "eventhandlers"
5. "functions"
6. "parsers"
7. "plugins"

When looking for a class, TDI uses the first instance found. Since the order is now specified (it was not specified earlier), it now possible to drop fixed components and other jars into the "patches" sub-directory, overriding installed classes. This makes it easy to test patches without corrupting an installation (or having to make backup copies of .jar files first). Furthermore, this makes it easy to see if an installation has any patches installed, since they are all in the same place.

2.6 TDI Server Hooks

The TDI server and configuration instances provide a method for TDI components to invoke custom Server-level Hooks. A Server Hook is a function name that is defined in a script file. Function implementations are provided by simply dropping script files in the "*serverhooks"* directory of the solution directory[3]. Note that all files found in this sub-directory are executed when the Server starts up, allowing function definitions to be registered[4].

Upon startup, TDI loads and executes all scripts in the "*serverhooks"* directory before any configuration instances are started. Those functions that define standard TDI Server Hook functions are prefixed with "***TDI_"***., and these are executed at various points during operation, as described below.

2.6.1 Scripting Server Hook functions

All TDI Server Hook functions have the following JavaScript signature:

```
/**
 * @parameter "main" - The configuration instance invoking the function
 * @parameter "source" - The component invoking the function
 * @parameter "user" - Arbitrary parameter information from the source
 */
function TDI_functionName(main, source, user) {
}
```

The *"main"* and *"source"* parameters always provide access to the Config Instance and calling component, respectively. The *"user"* parameter is used for different purposes in the various Hook functions, as shown in the table over standard Server Hook functions below.

[3] Calls to these hooks are synchronized to avoid potential multi threading issues.

[4] Note that the TDI Server has its own script engine instance, so variables and functions defined here will not be available directly to script code in AssemblyLines.

Function Name	Called By (source)	User Parameter and Expected return value
TDI_ALStarted	CI	Called when an AssemblyLine is started. user = The AssemblyLine that started
TDI_ALStopped	CI	Called when an AssemblyLine stopped. user = The AssemblyLine that stopped
TDI_ConfigStarted	Server	Called when a config instance started. user = The configuration instance
TDI_ConfigStopped	Server	Called after a config instanced stopped. user = The configuration instance
TDI_Shutdown	Server/CI	Called immediately before the TDI server is terminating the JavaVM (e.g. System.exit()). user = Exit status (integer)

Note that in all the above methods the return value is ignored.

Access to TDI Server Hook functions is provided through the `main.invokeServerHook() method. This function` is synchronized to prevent more than one thread executing a hook at a time. All calls are invoked synchronously so the caller will wait for the function to return. As a result, care should be taken not to spend too much time in a server hook.

As noted above, scripts are defined and made available by creating files in the "*serverhooks"* subdirectory of the solution directory. Scripts that contain sensitive information should be encrypted with the Server-API before adding it to the directory. The *serverapi/cryptoutils* tool is availabe for encrypting script files. Note that TDI will automatically try to decrypt an encrypted file.

Furthermore, the files in the *serverhooks* directory are loaded and executed after first *sorting* the file names using case-sensitive sort with the standard collating sequence for the platform. All files in the top-level directory are loaded before files in any subdirectories are processed.

Some examples of using Server Hooks are:

- A custom object that you always want loaded in TDI for use from your own scripting could be instantiated from a JavaScript snippet hooked into the server hook "on TDI startup". This gives you more control than simply referring to the class under the Java Libraries folder in the Config Browser.
- Starting one or more custom ALs that create an audit log for these events, and/or propagate these events to other systems via some transport (SNMP, HTTP, JMS, etc.).
- Implement some corporate security policy that's invoked every time a config is loaded or AL started

2.6.2 Developer Background info

As described in the previous section, the com.ibm.di.server.RS class (script variable *"main"*) has a new method for invoking Server Hooks:

```
/**
 * Invokes a server hook.
 *
 * @param name The name of the hook (also the filename)
 * @param caller The object invoking the hook
 * @param userInfo Arbitrary information to the hook from the caller
 */
```

```
public Object invokeServerHook(
      String name,
      Object caller,
      Object userInfo) throws Exception;
```

Note that this call can return a Java Object (any type), so even though TDI ignores this during Server Hook execution, you can make use of returned values in your own scripted calls.

2.7 Loop/Branch/Switch

In addition to addition of ELSE-IF and ELSE logic for Branches, Loops have been updated with a new *continue* method, as well as new keywords for the system.exitBranch() call. Finally, there is a new Switch component providing functionality for implementing AssemblyLine Operations (described on page 50).

2.7.1 new exitBranch keywords

The system.exitBranch() now supports a number of new keywords:

Calling `system.exitBranch("Branch")` and or `system.exitBranch("Loop")` causes flow continue *after* the current, innermost Branch or Loop (depending on which keyword is used)..

The "**Flow***"* keyword causes flow to exit the Flow section of the AssemblyLine, continuing either to the Response behavior in the case of a Server Mode Connector; or to an active Iterator to read in the next Entry; or to AL shutdown (Epilogs, …)..

The "**Cycle***"* keyword passes control to the end of the current AL cycle, and does not invoke Response behavior in Server Mode Connectors.

All other values used in the exitBranch() call causes a break out of the branch/loop having the specified name. So, for example, the call `exitBranch("IF_LookupOk")` sends the flow after the *containing* Branch or Loop called "IF_LookupOk". Note that unlike `system.skipTo()`, which will pass control to any named AL component, `exitBranch()` will cause processing to continue *after* the specified Loop/Branch.

2.7.2 Programmatic *continue*

The following new methods are available in the *system* object (UserFunctions):

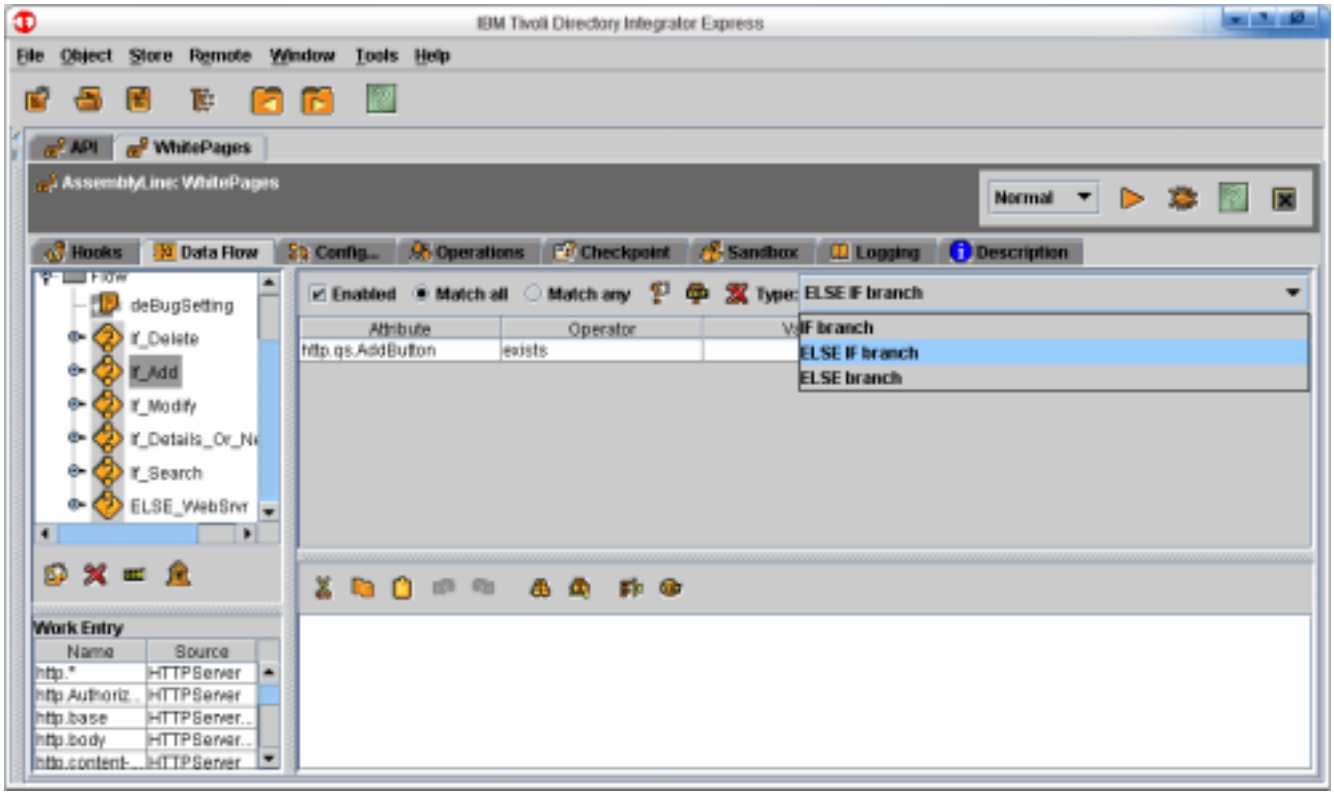
```
void continueLoop() throws com.ibm.di.exceptions.ContinueLoopException
```

```
void continueLoop (String name) throws com.ibm.di.exceptions.ContinueLoopException
```

Both of these methods throw a `com.ibm.di.exceptions.ContinueLoopException. This exception causes the LOOP to continue. In case a loop name is provided, the program flow is transferred to the LOOP component with that name.`

2.7.3 IF, ELSE-IF and ELSE

Branches now offer a drop-down for selecting the type of Branch. These options are IF (which is default for a Branch, and all you had in the previous version), ELSE-IF and ELSE.



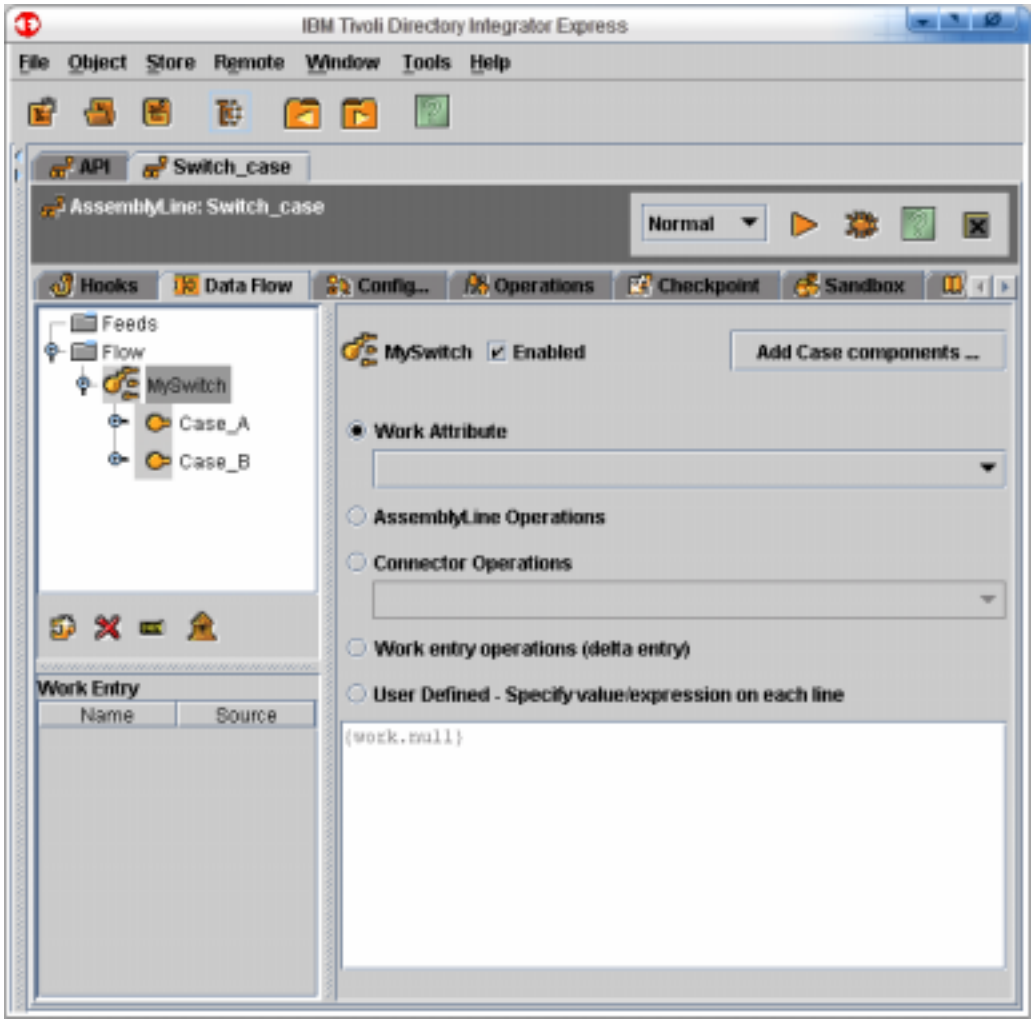
The AL component icons for the Branch have been changed to reflect the type. Note that if an invalid Branch type is selected (e.g. an ELSE with no preceding IF or ELSE-IF), then the component name will appear in red onscreen to indicate the invalid placement.

2.7.4 Switch/Case Components

As an alternative to AL logic as a series of IF-ELSE Branches, and in order to provide a more straightforward tool for implementing AL Operations, a new Switch component is available.

A Switch can be added to an AL any place a Branch or Loop can appear. Case components can only appear immediately under a Switch (and the CE enforces this).

More on AL Operations can be found under section *2.20 AssemblyLine Operations* on page 50.



A Switch can be based on one of the following mechanisms:

- On the value of a work Entry Attribute,
- Based on the AL Operation specified when this AssemblyLine was invoked,
- Based on Operations defined for a Connector (some Connectors can return a fixed set of *verbs* (i.e. operations) in an Attribute, for example the LDAP Server Connector or a Connector using the DSML Parser),
- The delta operation code set in the work Entry (e.g. *add, modify, delete, and so forth)*,
- A user-defined Expression (described in section *2.31 Expressions* on page 69).

Each method will return some value which is then used to match against Case components set up under the Switch. A Case is always based on an Expression. Also, note that only Cases that are matched are executed, so you do not have to exit the Switch manually (as is the case for Switch-case in many 3GL programming languages).

2.8 Improve Termination and Cleanup for Critical Errors

There are a number of TDI methods which allow catching and handling of TDI internal errors as well as errors occurred in TDI Connectors, Parsers, Function Components, EventHandlers. These methods comprise:

- error hooks, in which Javascript code can be written to handle an error - this method is accessible to TDI users
- Java try-catch-finally blocks which make sure that a minor failure does not break the TDI server as well as that all errors are handled appropriately – such blocks are already put in place in the core TDI server classes

The new JVM Shutdown Hook feature improves the reliability of the TDI server. Java shutdown hooks allow a piece of code to perform some processing after Control-C is pressed, or when the Java Virtual Machine (JVM) is shutting down for some other reason, even System.exit.

Users can specify an external program to be started when the JVM is shutting down. This external program is started from within the JVM shutdown hook.

This external program is configured via an optional property in the global/solution.properties file:

jvm.shutdown.hook=<external application executable>

Shell scripts and batch-files can also be specified as the value of this property.

When the JVM shutdown hook is called, nothing can be done to prevent the JVM termination. However, with the execution of an external program it is possible to perform customizable operations: e.g. sending a message that the TDI server has been terminated, carrying out cleanup operations, or even restarting a new TDI server if so desired.

2.9 Custom exit/return codes in TDI

A new method has been added to the com.ibm.di.server.RSInterface interface: *shutdownServer(int aExitCode)*. This new method is available in JavaScript through "**main**" variable, which references RSInterface and offers functions like runAL(). The specified aExitCode is then passed to the process that started TDI, allowing the caller to react to different exit codes, for example, in the script or batch-file used to launch TDI.

2.9.1 Access to this feature via TDI API calls

The Server API *Session* interfaces is extended to provide the *shutDownServer(int aExitCode)* method. The invocation of this method will result in termination of the TDI Server with the supplied exit code.

A new method *shutDownServer(Integer aExitCode)* has been added to *DIServer* MBean so that it can be accessed from the JMX context as well.

The com.ibm.di.server.RSInterface has gained a new method to terminate the TDI Server with a specific exit code:

```
/**
 * Set the shutdown request flag and specify an exit code
 */
public void shutdownServer (int aExitCode);
```

New methods are also added to the following Server API Interfaces:

1) com.ibm.di.api.local.Session :

```
/**
 * Shuts down the TDI Server with the specified exit code.
 * @throws DIException if an error occurs while shutting down the server.
 */
public void shutDownServer (int aExitCode)
    throws DIException, RemoteException;
```

2) com.ibm.di.api.remote.Session :

```
/**
 * Shuts down the TDI Server with the specified exit code.
 * @throws DIException if an error occurs while shutting down the server.
 */
public void shutDownServer (int aExitCode)
    throws DIException, RemoteException;
```

3) com.ibm.di.api.jmx.mbeans.DIServerMBean :

```
/**
 * Shuts down the TDI Server with the specified exit code.
 * @throws DIException if an error occurs while shutting down the server.
 */
public void shutDownServer (Integer aExitCode)
    throws DIException;
```

2.10 Securing Configs, Passwords and Sensitive Data

TDI saves configuration information in an XML file (Config file) which contains clear text for all configuration values. This often includes sensitive information like passwords. With version 6.1, TDI not only supports encryption of the entire configuration file, but also protecting individual values or settings.

This release can also be set up to automatically handle any component parameters tagged as *passwords.* Values entered into these configuration fields are delegated to the specified Password Property Store[5]. The parameter itself is then set with an Expression[6] that references the newly created password property. So, as passwords are entered or changed in the password field, the are actually value is never visible or stored in the Config itself.

Note that changes will not be made to existing Configs. If you want previously defined Config password parameters stored in the Password Store as well, then you must define a Password Store and re-enter the passwords themselves.

2.10.1 Default and user-defined parameter protection

The password protection mechanism is directly related to the configuration panels offered to the user. The configuration panels, or forms, contain descriptions of each parameter and its syntax. One type of parameter syntax is the "*password*" type, which causes the CE to use the special password edit field for user input. Whenever the value for a password syntax component parameter is changed, the value entered is saved to the designated Password Store. If no such Property Store is configured, then password values are still saved in clear text in the configuration file.

The new *setProtectedParameter(name,value)* method in the *com.ibm.di.config.interfaces.BaseConfiguration* interface will query the associated *MetamergeConfig* object for the default password store. If one is configured, a unique property name is generated the first time a call to setProptectedParameter is called. This key is used as the

---

[5] This behavior is only available once you have designated a Password Store. For more information on the new Property Store feature, including Password Store configuration, see section *2.30 Property Store Framework* on page *62*.

[6] The powerful, new Expressions feature is detailed in section *2.31 Expressions* page *69*.

key in the password store. The same property name is written to the configuration file as a standard property reference. When the value is later retrieved, standard property resolution takes place to retrieve the actual value from the password store. Hence, there is not an accompanying *getProtectedParameter(name)* for retrieval of protected parameters. The type of protection of the value is not defined by this feature since the protection mechanism may vary from between the various Connectors that can be used for a Property Store.

2.10.2 New methods in the API

The following methods have been added to the `com.ibm.di.entry.Attribute` and `com.ibm.di.entry.AttributeInterface` classes:

```
public void setProtected(boolean protect)
```

If the parameter is **true**, **try** to protect the Attribute values by not dumping them in log files

```
public boolean getProtected()
```

Returns **true if** the values should not be dumped in log files

The following method was added to the com.ibm.di.entry.Entry class:

`public void setAttribute (Object name, Object value, boolean protect)`

name The attribute name
value The attribute value. If this parameter is null, then the attribute is removed.
protect If this parameter is true, do not dump the Attribute values in log files

The following method was added to com.ibm.di.server.TaskCallBlock:

`public void setConnectorParameter (String connectorName, Object parameterName, Object parameterValue, boolean protect)`

If the last parameter is true, do not write the value of the parameter in log files

Methods that have been modified to not dump protected Attribute values:

`Log.dumpEntry( Entry e )` - this will also affect e.g. the dump() and dumpEntry() methods in AssemblyLine (task) and RS (main).

`Attribute.toString()` - Which also affects Entry.toString()

`Attribute.toDeltaString()` - Which also affects Entry.toDeltaString()

`TaskCallBlock.setConnectorParameters (AssemblyLine task)`

Also modified

`Entry.mergeAttributeValue(Object p1, AttributeInterface p2)` - If either Atribure is protected, the merged Attribute is protected

`Entry.merge (Entry e, boolean mergevalues)` - Make sure merged Attributes are protected if either of the old Attributes are.

2.11 Sensitive Data (Attributes) in logs and traces

TDI solution builders need a way to protect sensitive data, such as passwords, from being printed in clear text when tracing on the solution is needed. Therefore in TDI 6.1 Express some of the methods dealing with the Attribute class have been enhanced to say whether an attribute is protected or not. If the attribute is marked as protected and tracing is on, stars are output instead of the actual value.

Note: When *connection* parameters are found in the TaskCallBlock (TCB), the values will never be logged directly by TDI. The fact that parameters were given are logged, but not the values themselves can make troubleshooting more difficult. If the solution needs to be debugged, those values can be dumped manually, for example using scripting from the AL Debugger/Stepper, or from within script code in the AL itself.

2.12 Autocommit for the Delta Engine

In previous TDI versions (pre-6.1), snapshots written to the Delta Store (a feature of the System Store) during Delta Engine processing were committed immediately. As a result, the Delta Engine would consider a changed entry as *handled* even though processing the AL Flow section failed[7].

This limitation has been addressed in 6.1 through a new parameter – "Commit" – that has been added to the Connector Delta tab. The setting of this parameter now controls when the Delta Engine commits snapshots taken of incoming data to the System Store.

The options available are:

After every database operation	The default (and backwards compatible) value where, snapshots are committed to the System Store immediately as they are computed during the iteration.
On end of cycle	Wait with the commit until the AL completes the current cycle. This is the **recommended** setting to use.
On Connector close	Then the commit is delayed until the AL is finished and Connectors closed. Although delaying commit until the end of AL processing can boost performance, it can also result in situations where some changes have been successfully propagated, and yet the Delta Engine snapshots do not reflect this.
No autocommit	Commit of snapshot must be handled manually through script with the commitDeltaState() method of the Connector in Iterator Mode: myIterator.commitDeltaStore();

As shown in the table above, the default value is " After every database operation".

If this parameter value is "After every database operation", then System Store (Derby) transaction is committed after each delta store operation. In this case the performance will be the worst, but the changes made to the database will never be lost. They might be applied twice however in case of an unexpected stop.

If this parameter value is "On end of AL cycle", then System Store operations are committed at the end of each AssemblyLine iteration. This will in most cases improve the performance and prevent double-processing of changes.

If this parameter value is "On Connector close", then System Store operations are committed only at the end of the AssemblyLine (state MS_CLOSECONN of the AssemblyLine). Again, it is important to keep in mind that there may be a problem if the AssemblyLine is unexpectedly stopped in the middle of cycling. If the AL fails, then no snapshots are updated in the Delta Store.

[7] This is similar to the previous limitation found in Change Detection Connectors where the *last-change* marker was written to the System Store before the delta Entry processing was completed.

If this parameter value is "No autocommit", then System Store operations will never be committed automatically. The user must provide a script executing the public method "commitDeltaState()" of the Connector object. If the user does not execute a custom script to commit the connection, then at the end of the AssemblyLine (state MS_CLOSECONN) connection the changes are rollbacked by invoking the method rollbackDeltaState().

2.12.1 Controlling the Delta Engine via API calls

You can add your own custom scripts to commit or rollback the delta store connection regardless of what "Commit" value is chosen. This can be done in the Iterator's hooks, for example in "After GetNext" or "End of Data" by adding script like:

<the name of the connector>.commitDeltaState();
<the name of the connector>.saveDeltaState(); (alias for commitDeltaState() method)
or
<the name of the connector>.rollbackDeltaState();

If the name of the connector is "filesystem1" the script to commit the delta store connection will look like:
filesystem1.commitDeltaState(); or filesystem1.saveDeltaState();

2.13 Server API Notifications Enhancements

The Server API has been enhanced in several areas, including authentication. locking of Config files, custom notifications, Server shutdown events and documentation. These improvements are described in the following sections.

2.13.1 Server API Script Object

In order to make script access to the API easier, a new "**session**" variable is now available that references a local session to the TDI Server.

2.13.2 Remote Config Editing

Loading a remote Config for editing is a new concept in TDI 6.1 Express and operates differently than opening a remote Config in TDI 6.0. Configurations loaded for editing are not started in the usual way on the TDI Server.

There are now two options for loading a configuration for edit:

- Either the configuration is only loaded for editing and cannot be started at all,
- Or the configuration is loaded for editing and a temporary Config Instance is started on the Server so that the configuration can be tested while being edited.

In both cases, the configuration loaded for editing is independent of the Configs loaded and running (normally) on the Server. As a result, you could have the same Config both running on the Server as well as a second instance opened for editing (with or without a temporary Config Instance). This means that the CE does not give you access to changing a running Config; this functionality is only available via AMC v3.

As a result, when you save the Config back to disk on the TDI Server, this operation does not affect any running copy of the same configuration. Instead, there are API calls (like those used by AMC) to perform a "reload" operation, bringing in the last version of the Config from the disk.

Finally, the Server API will not allow concurrent modification of the same configuration by multiple users. When a user gets a configuration for editing, the Server API locks that configuration and will not allow other users to get it for editing until it is released (or the lock times out).

2.13.2.1 TDI Config Folder

A new TDI Server property *api.config.folder* is now available in Solution/Global Properties for specifying a folder on the local disk. This folder (and its sub-folders) is where Configs that can be browsed and loaded via the API are to be stored.

For example:

```
api.config.folder=configs
```

This means that all configuration files placed in "<TDI Solution Directory>/configs" (and sub-folders) are eligible for browsing and loading through the Server API, both locally and remotely.

The Server API provides new calls for browsing the files and folders in the directory specified by the "api.config.folder" property.

2.13.2.2 Load for editing

In TDI 6.0 configurations can be edited only after the corresponding Config Instance has been started on the TDI Server. Then there are API calls for getting the Config object, setting the Config object back (probably after editing) and saving the configuration on the disk.

TDI 6.1 Express does not allow modification of the Config object of an active Config Instance. Server API users will still be able to get the Config object for an active Config Instance, but the following calls for setting the Config object and saving it on the disk will throw an exception when executed on a normal running Config Instance:

- `ConfigInstance.setConfiguration(MetamergeConfig configuration)`
- `ConfigInstance.saveConfiguration()`
- `ConfigInstance.saveConfiguration(boolean aEncrypt)`

When a configuration is loaded for editing with a temporary Config Instance it, you are then able to execute the `setConfiguration(...)` method in order to test the changes applied to the configuration. The `saveConfiguration(...)` methods will however still throw exceptions.

TDI 6.1 Express presents new Server API calls for loading configurations for editing and for saving the edited configurations on the disk.

2.13.2.3 Configuration Locking

The Server API internally tracks all configurations loaded for editing. When another Server API user requests a configuration already loaded for editing, the method call will fail with exception.

A new Server API call has been added for checking whether a configuration is currently loaded for editing (locked). The lock on a configuration is released when the user that loaded the configuration for editing saves it back or cancels the update.

Furthermore, the Server API provides an option to specify a *timeout value* for keeping a configuration locked. When that timeout is reached for a configuration the lock is released and the user that locked the configuration will not be able to save it before loading it again. A new property "api.config.lock.timeout" is available in Global/Solution Properties for specifying the timeout value in minutes. When the property is left empty or is set a value of 0, this means that there is no timeout. The default value for this property is 0.

The timeout logic is handled in a separate thread running in the TDI Server. This thread is activated only when "api.config.lock.timeout" is set to a value greater than 0 and will check for and release expired locks each 30 seconds.

Note that there is a special call for forced releasing of a lock on a loaded Config. Only Server API users with the admin role are able to execute it.

2.13.2.4 Load for editing with temporary Config Instance

This is a special version of the "load for edit" mechanism (as described in the "Load for editing" section) that causes a temporary Config Instance to be started as well. This allows for testing of the Config and its AssemblyLines while they are being changed, providing valuable functionality to tools like the TDI Config Editor.

This temporary Config Instance is automatically stopped when the configuration is released or when the lock on the configuration expires.

Note that temporary Config Instances are independent of the normal long running Config Instances on the Server. For example, consider a normal Config Instance for "rs.xml" running on the Server. At the same time, the "rs.xml" configuration can be loaded for editing with a temporary Config Instance started to allow for testing. As a result, both a temporary Config Instance for "rs.xml" will be running in addition to the normal long running "rs.xml" Config Instance already loaded in the TDI Server.

The same locking mechanism applies for configurations loaded for editing with a temporary Config Instance. This means that a configuration can be loaded for editing only once regardless of whether it has been loaded for editing with a temporary Config Instance or without.

2.13.2.5 New Server API event for configuration update

A new Server API event "di.ci.file.updated" is fired whenever a configuration that has been locked is saved on the TDI Server.

This feature allows Server API clients to get notified on changes in Configs they are using, for example to reload them in order to run the latest version.

2.13.2.6 New API calls

All configurations are identified through the relative file path of the configuration file according the TDI Server configuration Config folder. All paths specified as parameters are relative to Config folder itself (so "./" references the folder specified by the "api.config.folder" property).

The following new calls have been added to the local and remote Server API *Session* objects, as well as the JMX interfaces:

- **`public boolean releaseConfigurationLock(String aRelativePath) throws DIException;`**

  Administratively releases the lock of the specified configuration. This call can be only executed by API users with the Admin role.

- **`public boolean undoCheckOut(String aRelativePath) throws DIException;`**

  Releases the lock on the specified configuration, aborting all changes being done. This call can only be executed from a user that has previously checked out the configuration and if the configuration lock has not timed out.

- **`public ArrayList listConfigurations(String aRelativePath) throws DIException;`**

Returns a list of the file names of all configurations in the specified folder. The configurations file paths returned are relative to the TDI Server configuration codebase folder.

- **`public ArrayList listFolders(String aRelativePath) throws DIException;`**

Returns a list of the child folders of the specified folder.

- **`public ArrayList listAllConfigurations() throws DIException;`**

Returns a list of the file names of all configurations in the directory subtree of the TDI Server configuration codebase folder. The configurations file paths returned are relative to the TDI Server configuration codebase folder.

- **`public MetamergeConfig checkOutConfiguration(String aRelativePath) throws DIException;`**

Checks out the specified configuration. Returns the MetamergeConfig object representing the configuration and locks that configuration on the Server.

- **`public MetamergeConfig checkOutConfiguration(String aRelativePath, String aPassword) throws DIException;`**

Checks out the specified password protected configuration. Returns the MetamergeConfig object representing the configuration and locks that configuration on the Server.

- **`public void checkInConfiguration(MetamergeConfig aConfiguration, String aRelativePath) throws DIException;`**

Saves the specified configuration and releases the lock. If a temporary Config Instance has been started on check out, it will be stopped as well.

- **`public void checkInConfiguration(MetamergeConfig aConfiguration, String aRelativePath, boolean aEncrypt) throws DIException;`**

Encrypts and saves the specified configuration and releases the lock. If a temporary Config Instance has been started on check out, it will be stopped as well.

- **`public void checkInAndLeaveCheckedOut(MetamergeConfig aConfiguration, String aRelativePath) throws DIException;`**

Checks in the specified configuration and leaves it checked out. The timeout for the lock on the configuration is reset.

- **`public MetamergeConfig createNewConfiguration(String aRelativePath, boolean aOverwrite) throws DIException;`**

Creates a new empty configuration and immediately checks it out. If a configuration with the specified path already exists and the aOverwrite parameter is set to *false* the operation will fail and an Exception will be thrown.

- **`public ConfigInstance checkOutConfigurationAndLoad(String aRelativePath) throws DIException;`**

Checks out the specified configuration and starts a temporary Config Instance on the Server. This Config Instance will be stopped when the configuration is checked in or when the lock on the configuration expires. The method returns the ConfigInstance object. The MetamergeConfig object can be retrieved through the ConfigInstance object.

- **`public ConfigInstance checkOutConfigurationAndLoad(String aRelativePath, String aPassword) throws DIException;`**

Checks out the specified password protected configuration and starts a temporary Config Instance on the Server. This Config Instance will be stopped when the configuration is checked in or when the lock on the configuration expires. The method returns the ConfigInstance object. The MetamergeConfig object can be retrieved through the ConfigInstance object.

- **`public ConfigInstance createNewConfigurationAndLoad(String aRelativePath, boolean aOverwrite) throws DIException;`**

Creates a new empty configuration, immediately checks it out and loads a temporary Config Instance on the Server. If a configuration with the specified path already exists and the aOverwrite parameter is set to *false* the operation will fail and an Exception will be thrown. The temporary Config Instance will be stopped when the configuration is checked in or when the lock on the configuration expires. The method returns the ConfigInstance object. The MetamergeConfig object can be retrieved through the ConfigInstance object.

- **`public boolean isConfigurationCheckedOut(String aRelativePath) throws DIException;`**

Checks if the specified configuration is checked out on the Server.

2.13.3 Access to 6.1 Functionality

The API has been enhanced to give access to new 6.1 functionality like the System Queue.

2.13.4 Server shutdown event

A new Server API event notification has been added to signal Server shutdown events. This event is available to Server API clients and JMX clients, both in local and remote context, and is of type "di.server.stop" for both the Server API and JMX notification layers. Note that this event object also conveys the Server boot time.

This feature is not to be confused with the new feature described in section *2.8 Improve Termination and Cleanup for Critical Errors* on page 21, which is meant for dealing with JVM termination.

2.13.5 Custom Server API event notifications

Server API functionality has been added for sending custom, user-defined event notifications. The following new call is added to both the local and remote Server API *Session* objects, as well as to the *DIServer* MBean so that it can be accessed from the JMX context as well:

```
public void sendCustomNotification (String aType, String aId, Object aData)
```

The invocation of this method results in broadcasting a user defined event notification. The parameters that must be passed to this method have the same meaning as the respective parameters of standard Server API notifications.

- The *aType* parameter specifies the type of the event. The value given by the user will be prefixed with the "user." prefix. For example if the type passed by the user is "process.X.completed" the type of the event broadcasted will be "user.process.X.completed". A client application can register for all custom events specifying a type filter of "user.*".
- The *aId* parameter can be used to identify the object this event originated from. The standard Server API events use this value to specify a Config Instance, AssemblyLine or EventHandler.
- The *aData* parameter is where the user can pass on any additional data related to this event; if the event is expected to be sent and received in a remote context, this object has to be serializable.

More detailed information about the Server API notification mechanism can be found in the new *Server API Developers Guide*.

2.13.6 Authentication

In addition to previously supported authentication based on an SSL channel with client side authentication, now offers a new option for using custom authentication. Furthermore, SSL client authentication can be switched off.

This is done via a new property in Solution/Global Properites: **api.remote.ssl.client.auth.on**.

When this property is set to "true" the TDI Server will require client authentication on the SSL channel. When client authentication is turned on, the mechanism for SSL-based Server API authentication from TDI 6.0 can be used. When this property is set to "false", SSL-based Server API authentication cannot be used. When the property is not specified a value of "false" is assumed.

The following authentication options are available:

- **SSL-based authentication (the mechanism available in TDI 6.0)**

  Only works when *api.remote.ssl.client.auth.on=true* (you will also need *api.on=true*, *api.remote.on=true*, *api.remote.ssl.on=true*).

  SSL-based authentication is triggered when *createSession()* with no parameters is invoked. The user is authorized as per the rights assigned to the SSL certificate user ID in the Server API User Registry.

  Note: When SSL is used and the remote client application uses Server API listener objects, then the client application must have its own certificate that is trusted by the TDI Server (this is analogous to the setup for SSL client authentication). If there is no client certificate trusted by the TDI Server, the listener objects will not work and the remote client application will not be able to receive notifications from the TDI Server.

- **Custom authentication**

  Only works when *api.custom.authentication* is set to a JavaScript authentication file.

  Custom authentication is used when *createSession(username, password)* is invoked. This method works regardless of whether SSL is used and whether SSL client authentication is used. The user is authorized as per the rights assigned to the *username* user in the Server API User Registry.

- **Host-based authentication**

  Only works when *api.remote.ssl.on=false*. Then invocations of *createSession()* without parameters from all hosts specified on the *api.remote.nonssl.hosts* property are successfully authenticated and granted admin authority.

- **LDAP authentication**

  use of the LDAP Authentication mechanism is transparent to Server API users. The general Server API createSession(Username, Password) method will perform LDAP authentication when the TDI Server is configured to use LDAP Authentication. The Server API will use LDAP Authentication when the *api.custom.authentication* property in Global/Solution.properties is set to the "[ldap]" value:

  ```
  api.custom.authentication=[ldap]
  ```

2.13.6.1 Custom authentication

A Server Hook has been added to allow the use of custom JavaScript code to perform user-specific authentication. This is controlled by the TDI Server configuration property **api.custom.authentication** (found in Global/Solution Properties) For more details on the new Server Hooks, see section *2.6 TDI Server Hooks* on page 17.

This new *api.custom.authentication* property points to a JavaScript text file on the disk that contains custom authentication code. When the *api.custom.authentication* property is specified (and points to an existing script file) then the JavaScript code contained in the specified file is executed for each authentication request based on the

new Server API *createSession()* methods with username and password parameter. Note that these methods will not work unless this property is present and correct. Instead, the SSL-based authentication mechanism is used,

The authentication script has access to the predefined script object **userdata**. This object provides the following two public members (variables):

- **`userdata.username`** – contains the name of the user requesting authentication
- **`userdata.password`** – contains the password provided by the user

The script is free to perform whatever checks and authentication actions, however it must return whether the authentication is successful through the **ret** object:

- set **`ret.auth = true`** to specify that the authentication is successful
- set **`ret.auth = false`** to specify that the authentication is not successful; in this case the authentication script can provide additional information for why the authentication failed through the **ret.errordescr** variable (for example *`ret.errordescr = "Invalid user name"`*) and **ret.errorcode** (for example *ret.errorcode = 42*). The description and error code fields are conveyed by the AuthenticationException thrown by the ServerAPI on unsuccessful authentication.

The authentication script also has access to the **main** script object, providing RSInterface methods. This is useful for logging custom messages in the TDI Server log file (for example *main.logmsg("Authentication failed for user : " + userdata.username)*).

2.13.6.2 Example LDAP authentication

A ready to use JavaScript file that performs authentication against an LDAP Server is provided with the TDI Server, serving as an example of how to use the new mechanism for custom authentication.

The JavaScript file is named "ldap_auth.js" and is located in the "examples/auth_ldap" folder. To deploy this sample LDAP authentication mechanism, copy that file to your TDI solution folder (preferrably some sub-directory) and specify *api.custom.authentication=ldap_auth.js* in Global/Solution Properties. Note that you will also need to edit the script file in order to specify the LDAP parameters to use.

The JavaScript code in "ldap_auth.js" will try to bind to an LDAP Server with the specified username and password. If the bind operation is successful the script will indicate a successful authentication, otherwise the authentication is rejected.

2.13.6.3 LDAP Authentication

The LDAP Authentication mechanism for the TDI Server API is configured through the following Global/Solution Properties:

- **api.custom.authentication**

  This property points to a JavaScript text file on the disk that contains custom authentication code. If this property is not defined, then only the SSL-based authentication mechanism can be used and the new Server API *createSession* methods with username and password will not work.

  Setting this property to "**[ldap]**" enables the built-in LDAP Authentication mechanism, for example:

  ```
  api.custom.authentication=[ldap]
  ```

  All properties starting with "*api.custom.authentication.ldap.*" will only be taken into account when *api.custom.authentication* is set to *[ldap]*.

The following new configuration properties further specify how LDAP Authentication takes place:

- **api.custom.authentication.ldap.critical**

This parameter specifies the Server API behavior when the LDAP Authentication module cannot be initialized on startup.

If this parameter is set to "true" the Server API initialization will fail and the Server API will not be started.

If this parameter is missing or is set to "false" the Server API will log the LDAP Authentication initialization error but the Server API is started. An attempt to initialize the LDAP Authentication module will be made on each authentication request received by the Server API until the LDAP Authentication module is initialized.

- **api.custom.authentication.ldap.hostname**

  The LDAP Server hostname.

- **api.custom.authentication.ldap.port**

  The LDAP Server port number. For example, 389 for non-SSL or 636 for SSL.

- **api.custom.authentication.ldap.ssl**

  Specifies whether SSL is used to communicate with the LDAP Server. When set to "true" SSL will be used, otherwise SSL will not be used.

- **api.custom.authentication.ldap.searchbase**

  Specifies the LDAP directory location where user searches will be preformed. When this property is not specified user searches will not be performed.

- **api.custom.authentication.ldap.admindn**

  Specifies an LDAP Server administrator distinguished name that will be used for user searches. When this property is not specified anonymous bind will be used for user searches.

- **api.custom.authentication.ldap.adminpassword**

  Password for the LDAP Server administrator distinguished name.

- **api.custom.authentication.ldap.userattribute**

  Specifies the user id attribute to be used in searches. When this property is not specified user searches will not be performed. For example:

  ```
  api.custom.authentication.ldap.userattribute=cn
  ```

2.13.6.4 Authentication configuration examples

Following are some examples of authentication settings:

1. **Non-SSL configuration & custom authentication:**

   ```
   api.remote.ssl.on=false
   api.remote.nonssl.hosts=192.168.113.51, 192.168.113.52
   api.custom.authentication=ldap_auth.js
   ```

   SSL is not used.

   Authentication requests using *createSession()* with no parameters will only succeed if they are invoked from the localhost or from 192.168.113.51 or 192.168.113.52.

   Authentication requests using *createSession(username, password)* will only succeed if the ldap_auth.js successfully authenticates the user specified with the *username* and *password* parameters.

   Remote JMX clients will be authenticated only when the request comes from the localhost or from 192.168.113.51 or 192.168.113.52.

2. SSL (without client authentication) & custom authentication:

```
api.remote.ssl.on=true
api.remote.ssl.client.auth.on=false
api.custom.authentication=ldap_auth.js
```

SSL is used for remote Server API communication.

Authentication requests using *createSession()* with no parameters will fail because neither SSL client authentication is switched on, nor host-based authentication is not available.

Authentication requests using *createSession(username, password)* will only succeed if the ldap_auth.js successfully authenticates the user specified with the *username* and *password* parameters.

Host-based authentication is not available in this case regardless of the value of the *api.remote.nonssl.hosts* parameter, because *api.remote.ssl.on* is set to *true*.

The remote JMX layer will not be accessible because SSL is turned on but SSL client authentication is not used.

3. SSL with client authentication & custom authentication:

```
api.remote.ssl.on=true
api.remote.ssl.client.auth.on=true
api.custom.authentication=ldap_auth.js
```

SSL is used for remote Server API communication and the Server requires SSL client authentication.

Authentication requests using *createSession()* with no parameters will succeed when the SSL certificate of the client is present in the Server's trust store.

Authentication requests using *createSession(username, password)* will only succeed when the SSL client authentication is successful (the SSL certificate of the client is present in the Server's trust store) and the ldap_auth.js script successfully authenticates the user specified with the *username* and *password* parameters. In this case authorization is performed based on the *username* parameter from the *createSession(username, password)* call and not with the user identity from the SSL client certificate.

Host-based authentication is not available in this case regardless of the value of the *api.remote.nonssl.hosts* parameter, because *api.remote.ssl.*on is set to *true*.

Remote JMX clients are authenticated when the SSL certificate of the client is present in the Server's trust store.

4. SSL with client authentication & no custom authentication:

```
api.remote.ssl.on=true
api.remote.ssl.client.auth.on=true
api.custom.authentication=
```

SSL is used for remote Server API communication and the Server requires SSL client authentication.

Authentication requests using *createSession()* with no parameters will succeed when the SSL certificate of the client is present in the Server's trust store.

Authentication requests using *createSession(username, password)* will not succeed because custom authentication is not configured.

Host-based authentication is not available in this case regardless of the value of the *api.remote.nonssl.hosts* parameter, because *api.remote.ssl.*on is set to *true*.

Remote JMX clients are authenticated successfully only when the SSL certificate of the client is present in the Server's trust store.

2.13.6.5 Remote CE SSL Enhancement

In TDI 6.0 the Remote Config Editor panels contained a checkbox labeled "SSL". Users of the Remote Config Editor were supposed to specify whether SSL is enabled on the TDI Server. Actually the SSL option was not used for creating the connection, but was instead used by another piece of Remote CE functionality – running remote AssemblyLines and EventHandlers, as well as receiving their logs.

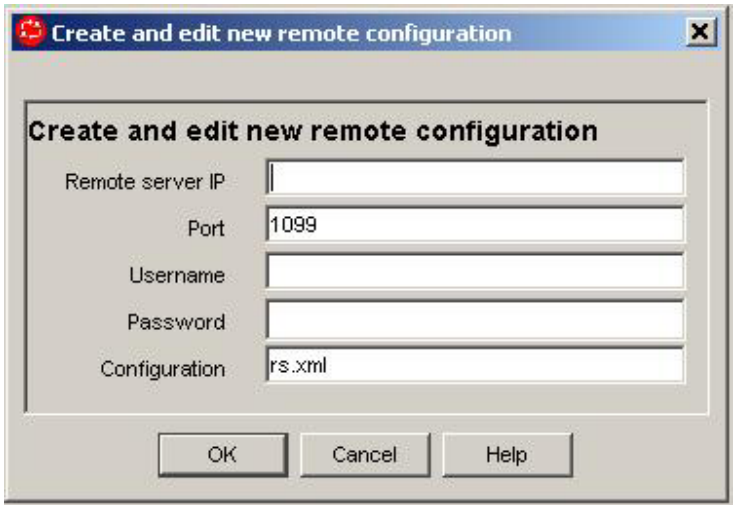
In TDI 6.1 Express the "SSL" option is added to the Config Editor panels. In addition, a new method is added to the Server API Session Interface:

```
public boolean isSSLon();
```

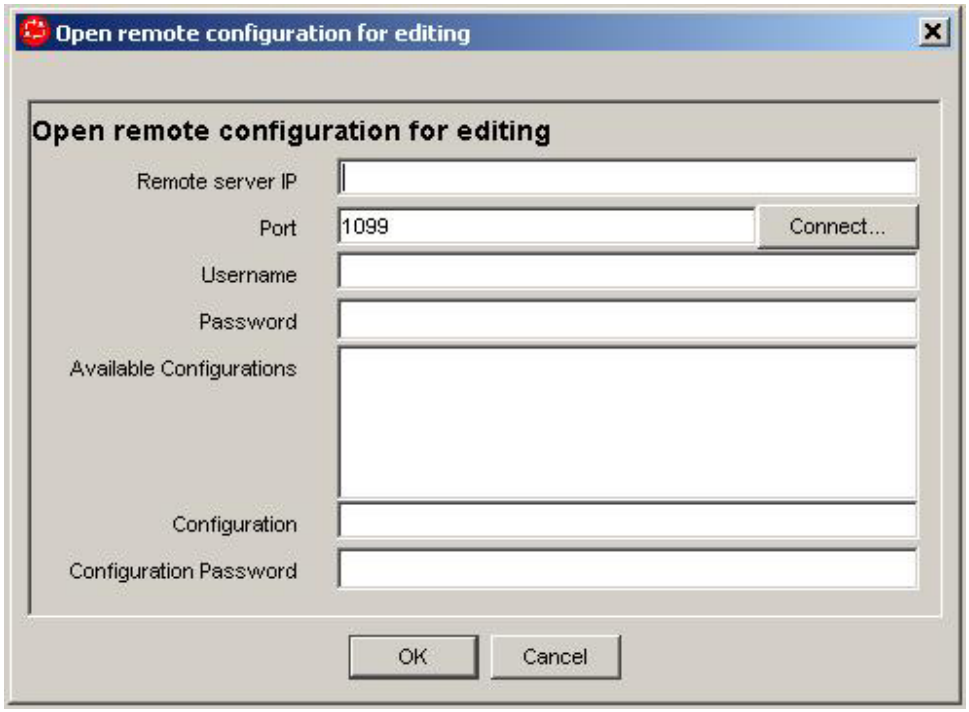
This method will return "true" if the current Session is over SSL.

The Remote Config Editor has been updated to use this new Server API method instead of forcing users to provide the "SSL" parameter thus reducing confusion and improving usability.

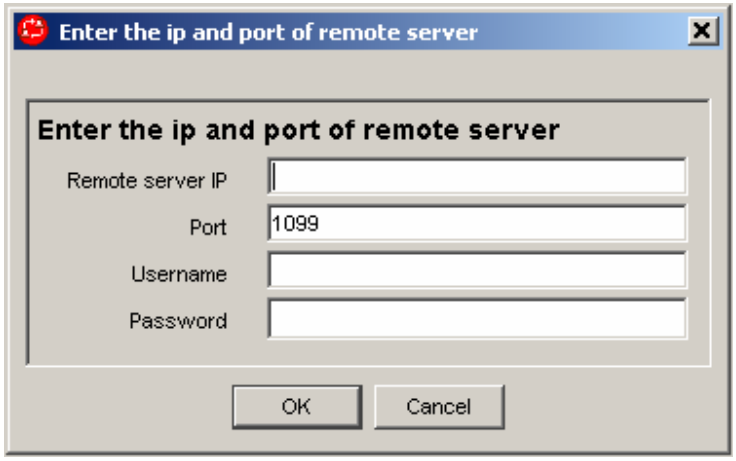
Furthermore, the New Remote Config dialog is updated to support the new authentication options, providing parameters for setting user name and password:



The dialog for opening a remote Config has also been updated:



And the Remote Server "About" screen:



A seen in the screenshots above, new "Username" and "Password" parameters have been added. Furthermore, the old "Password" field for the Open Remote Config dialog has been renamed to "Configuration Password".

2.13.6.6 Programmatic Interfaces (APIs)

2.13.6.6.1 New createSession() Server API method

A new call has been added to the local and remote Server API SessionFactory objects:

- **`public Session createSession (String aUserName, String aPassword) throws DIException;`**

This method creates a Server API Session if the provided username and password are successfully authenticated by the user specified custom authentication JavaScript code. See "**Error! Reference source not found.**" for more details.

The new method for creating local sessions is mirrored in the com.ibm.di.api.APIEngine class so that it is accessible from script context:

- **`public static com.ibm.di.api.local.Session getLocalSession (String aUserName, String aPassword) throws DIException;`**

2.13.6.7 Server API AuthenticationException

When custom authentication is used (see "**Error! Reference source not found.**") and the script specified by the user indicates that the authentication has failed, an exception of type **com.ibm.di.api.exceptions.AuthenticationException** is thrown by the *createSession(username, password)* method.

This exception object follows the public interface of the AuthenticationException class:

```
public class AuthenticationException extends DIException {

    /**
     * Returns error description as set by the authentication script in the
     * ret.errordescr field.
     */
    public String getErrorDescription() {
        …
    }

    /**
     * Returns error code as set by the authentication script in the
     * ret.errorcode field.
     */
    public Integer getErrorCode() {
        …
    }

    /**
     * Returns the string representation of the exception containing the error
     * message and error code values.
     */
    public String toString() {
        …
    }
```

2.13.6.8 NOTE: The Server API JMX layer does not support custom authentication

The remote JMX layer of the Server API does not support the custom authentication mechanism. It will ignore the *api.custom.authentication* property. Regardless of the value of this property and whether custom authentication is enabled or not for the Server API, the remote JMX layer will perform the following authentication:

- If SSL is turned on and SSL client authentication is turned on, the remote JMX layer will perform SSL-based authentication (as in TDI 6.0).
- If SSL is turned on and SSL client authentication is turned off, the remote JMX layer will not work.
- If SSL is turned off, the remote JMX client is successfully authenticated only if its host is specified on the *api.remote.nonssl.hosts* property. In this case it will be granted admin authority.

2.14 External properties file from the command line

External property files (e.g. any user-defined Property Store based on the ibmdi.Properties Connector) can now be specified from the command line when starting a TDI Server.

A new optional command line parameter [-f] can be used with the "ibmdisrv" server startup scripts.

The format of the command line argument is formed in key-value pairs in the following way:

–f <extProp1=file1,extProp2=file2, …>, where ***extProp?*** is name of the Property Store – which is based on the ibmdi-Properties Connector – and ***file?*** specifies the file to associate with Connector.

When this [-f] option is used to specify a properties file from the command line, the server changes the Property Store configuration in memory only, i.e. the server does not make this change permanent by changing the TDI Config file on disk – this change is valid for the current run of the TDI server.

If any property files are specified at the command line, they are valid only for the Config Instances specified with the [-c] command line option (which are loaded on TDI server startup). The property files specified at the command line do not have any impact on Config Instances which have not been explicitly named with the [-c] command line option (these can be Config Instances loaded by remote Server API client for example).

If a Property Store whose name has been specified with the [-f] command line switch cannot be found in a Config Instance, an error message is logged in the server log (ibmdi.log) in the Install-directory.

When a Property Store name is specified more than once with the -[f] command line switch then there are two effects: (1) a warning message is logged, and (2) the file specified last will take effect.

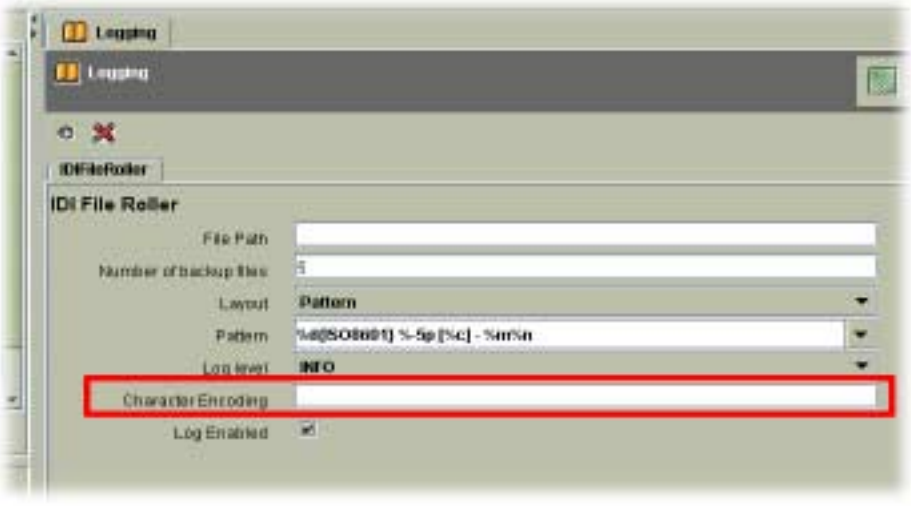
This feature is implemented in the com.ibm.di.server.RS Java class (referenced via the **main** variable when scripting). After the **reload()** method is called the MetamergeConfig object is loaded and for each Property Store specified on the command line the corresponding PropertyStoreConfig object is updated.

2.15 Logging and general PD enhancements

2.15.1 Character Encoding Parameter for All File Appenders

All panels showing loggers that write to encoding enabled streams have an extra parameter that lets the user define the character encoding for the stream.

All relevant logger panels have a new ENCODING_LABEL parameter.



2.15.2 Custom Appender Support

Log4j allows logging requests to print to multiple destinations. In log4j speak an output destination is called an *appender*. TDI Server supports several appenders (IDIFileRoller Appender, Console Appender, File Appender, Syslog Appender, NTEventLog Appender, DailyRollingFile Appender, SystemLog Appender). The logging environment for TDI 6.0 could not be extended with additional appenders. The TDI Server is enhanced in version 6.1 so that you can now add your own custom Appenders.

Custom Appenders are defined with a system property in global.properties/solution.properties file:

custom.appender.<CustomAppenderName>=<CustomAppenderClass> [CustomFormUserInterfaceClass]

where:

<CustomAppenderName> - the custom Appender name used in the Config Editor.

<CustomAppenderClass> - the custom Appender java class implementation. All custom Appenders must implement **com.ibm.di.log.CustomAppenderInterface** interface (see details in section "Programmatic Interfaces")

[CustomFormUserInterfaceClass] – an optional parameter that specifies custom GUI implementation for that Appender. Custom user interface class must implement **com.ibm.di.admin.ui.CustomAppenderUIInterface** see details in section "Programmatic Interfaces")

Here is an example for a custom Appender definition in global.properties:

custom.appender.customLog=com.ibm.di.log.CustomLogAppender com.ibm.di.admin.ui.CustomLogFormUI

2.15.2.1 Custom Appender Support GUI definition

There are three methods available for defining the user interface for custom Appenders:

1) **Using the default custom user interface form from "standard_forms.cfg" file.**

   If there is no custom GUI defined for this Appender (no optional parameter is set in the Appender definition in global.properties) and no appender specific user interface form is added to the appender jar file, then the Default form will be used. The Default form is shown in the "User Interface Panels" section of this document. The custom appender class is responsible for reading and processing the parameters in this form. The default form has the following definition in the "standard_forms.cfg" configuration file:

```
[form CustomAppender]
        title:Custom Logger
        parameterlist [
                Custom.AppenderParams
                $GLOBAL.com.ibm.di.log.level
                $GLOBAL.logenabled
        ]
        parameter {
                Custom.AppenderParams {
                        label:Appender Parameters
                        description:Parameters format is not defined and it is up to Appender
                implementation to parse this composite field.
```

```
                    syntax:textarea
                }
            }
        [end]
```

This is the default logger configuration screen. If there is no custom form class specified and no Appender specific form is added in appender jar file then the default form is used.

The panel's title is "Custom Logger".

2) Defining Appender specific form in external "idi.inf" file.

The following steps have to be followed for defining Appender specific configuration form:

1. Write a form definition in a file named "idi.inf"
2. Put "idi.inf" file in the jar file where the custom Appender classes are packaged.

You can examine forms of existing appenders as guidelines for writing your own. The "standard_forms.cfg" file contains definitions for TDI Appender forms and it is located in <TDI_install_dir>/jars/ce/miadmin.jar under "com\ibm\di\admin\templates".

The name of the form described in "idi.inf" should be formed with the Appender name concatenated with the "**Appender**" string.

For example if you have the following Appender definition in global.properties:

custom.appender.customLog=com.ibm.di.log.CustomLogAppender,

the name of the form should be **customLogAppender**

In this case you are free to define whatever parameters his Appender needs, and the custom Appender class should read and process these parameters.

3) Implementing custom java class for user interface.

In case the custom Appender configuration requires a more complicated configuration GUI screen then the name of the class implementing **com.ibm.di.admin.ui.CustomAppenderUIInterface** interface must be provided in the Appender definition in global.properties.

Note that new interfaces are introduced in TDI 6.1 Express that are used for implementing custom Appenders:

2.15.2.2 Developer Notes

All custom Appenders must implement CustomAppenderInterface.

com.ibm.di.log.CustomAppenderInterface:

```
package com.ibm.di.log;

import org.apache.log4j.Appender;

public interface CustomAppenderInterface extends Appender {

    /**
     * Initilizes the Custom Appender with its configuration parameters
     * @param aLogConfigItem the Custom Appender configuration object.
     */
    public void initialize(com.ibm.di.config.interfaces.LogConfigItem
aLogConfigItem);

}
```

Furthermore, all custom Appenders GUI forms must implement CustomAppenderUIInterface.

com.ibm.di.admin.ui.CustomAppenderUIInterface:

```
      public interface CustomAppenderUIInterface {

          /**
           * Initializes the custom Appender congfiguration panel.
           * aLogConfigItem the Custom Appender configuration object.
           */
          public void initialize(com.ibm.di.config.interfaces.LogConfigItem
      aLogConfigItem);

          /**
           * Returns the ready user interface panel.
           */
          public JPanel getPanel();
      }
```

The default custom Appender form uses the following configuration parameters:

"Custom.AppenderParams" parameter:

Custom Appender parameters.

Config Editor GUI control:

- label: "Appender Parameters"
- description: "Parameters format is not specified and it is up to Appender implementation to parse this composite field."
- type: textarea
- default value: empty

"com.ibm.di.log.level" parameter:

Specifies logging level.

Config Editor GUI control: globally defined for all appenders.

"logenabled" parameter:

Enables / Disables logging.

Config Editor GUI control: globally defined for all appenders.

2.15.3 Log4j logs folder

The default location of the logs generated by log4j has now been changed to "logs" folder. The change has been made in both the log4j.properties file and the ce-log4j.properties file.

Now both the ibmdi.log and the ibmditk.log are by default placed under the "logs" directory.

The `log4j.appender.Default.file` property in both log4j.pproperties file and ce-log4j.properties file has been modified to change the default folder to the logs folder and then place the corresponding files under the log folder.

2.15.4 Miscellaneous Serviceability Enhancements

In TDI 6.0, the *jlog,properties* file defined paths where the logs and trace files were placed. Furthermore, these log files had a default suffix of ".log", whereas the default message files should have had ".msg" as their file extension. In general, logs and trace files where difficult to locate. In order to improve the serviceability of TDI as well as consistence with other Tivoli Security products, the following enhancements have been made to 6.1:

- TDI 6.1 Express parsers now include trace information.
- Log, message and trace files are now stored in the following sub-folders of the common TDI installation directory:

  - Trace files are stored in "CTGDI/logs".

  - FFDC data in "CTGDI/FFDC/<date>".

  - Message logs in "CTGDI/logs" (note that their naming is standardized as well – see below)
- The name of the message log file will have prefix "msg" and suffix ".log".
- PD scripts – collect.bat (sh) and logcmd.bat (sh) – are provided in the "CTGDI/scripts" directory.
- The *jlog.properties* file is now placed under the Tivoli_Common_Dir/CTGDI/etc dir.

An example of the defaults set for the logs and traces are shown below.

```
jlog.snapmemory.className=com.tivoli.log.SnapMemoryHandler
jlog.snapmemory.description=Memory handler used to trace to memory
jlog.snapmemory.queueCapacity=10000
jlog.snapmemory.dumpEvents=true
jlog.snapmemory.snapFile=trace.log
jlog.snapmemory.baseDir=$Tivoli_common_dir$/CTGDI/FFDC/
jlog.snapmemory.userSnapFile=userTrace.log
jlog.snapmemory.userSnapDir=$Tivoli_common_dir$/CTGDI/FFDC/user/
jlog.snapmemory.triggerFilter=jlog.levelflt
jlog.snapmemory.msgIds=*E
```

```
jlog.snapmemory.msgIDRepeatTime=10000
jlog.snapmemory.maxFiles=10
jlog.snapmemory.maxFileSize=1000000
```

2.16 Global Connector Pooling

TDI version 6.0 introduced the concept of AssemblyLine pooling as a feature of the new Connector Server Mode. Now in 6.1 you can also define global pools of Connectors that can be shared between AssemblyLines in the same Config.

Library Connector provide a new tab called "Pool" where the following parameters can be set:

Enable Pooling	This checkbox toggles pooling for this Connector on/off.
Min Pool Size	Specifies the minimum number of Connectors instances this Pool will always contain.
Max Pool Size	Max size this pool can grow to.
Purge Interval	Specifies the time interval in seconds between when the Connector Pool will be regularly shrunk to its minimum size. If the number of Connectors currently in use is bigger than the minimum size of the Pool, the Pool will be shrunk to this number. If a value of **0** is specified, then the Pool will never shrink.
No. of Attempts to initialize	Specifies how many attempts to initialize the Connector will be made if errors occur during initialization. A default value of **1** is assumed if this parameter is missing or blank.
Sleep interval between initialization attempts	The pause in seconds between each two consecutive initialization attempts. A default value of **0** is assumed if this parameter is missing or blank.

When a Pool-enabled Connector is used in an AssemblyLine, its "Pool" tab is enabled in the context of the AL and offers two parameters:

Use Connector from Pool	Specifies if this instance of the Library Connector is to used a pooled connection, or simply create its own (standard AL Connector behavior).
Exhausted Pool Behavior	This setting controls what happens when the AL requests a Connector from the Pool which has reached its max size, but does not have a free Connector available. Default behavior is to "Wait" until an instance is released elsewhere and becomes available for this requestion. Option, this can be set to "Fail" which will cause an error/exception to be thrown.

Note that the Prolog Before and After Initialize Hooks for a Pooled Connector are not executed when the Pool Manager sets up the Connector. However, they are run during AssemblyLine and AL component initialization. Note however that even though these Hooks are called, the initialize() method of the Connector is not executed at this time. Since the Connector is already initialized by the Pool Manager, any parameter changes made in the Prolog Hook will not have any effect.

Similarly, the Epilog Before and After Close Hooks are not executed when the Pool Manager closes a Connector. They are run when the AL terminates, but as with the case of the Prolog Hooks, the actual terminate() method is not called at this time.

Reconnect Functionality is available for Pooled Connectors, and is controlled by the Connection Errors tab of the Connector.

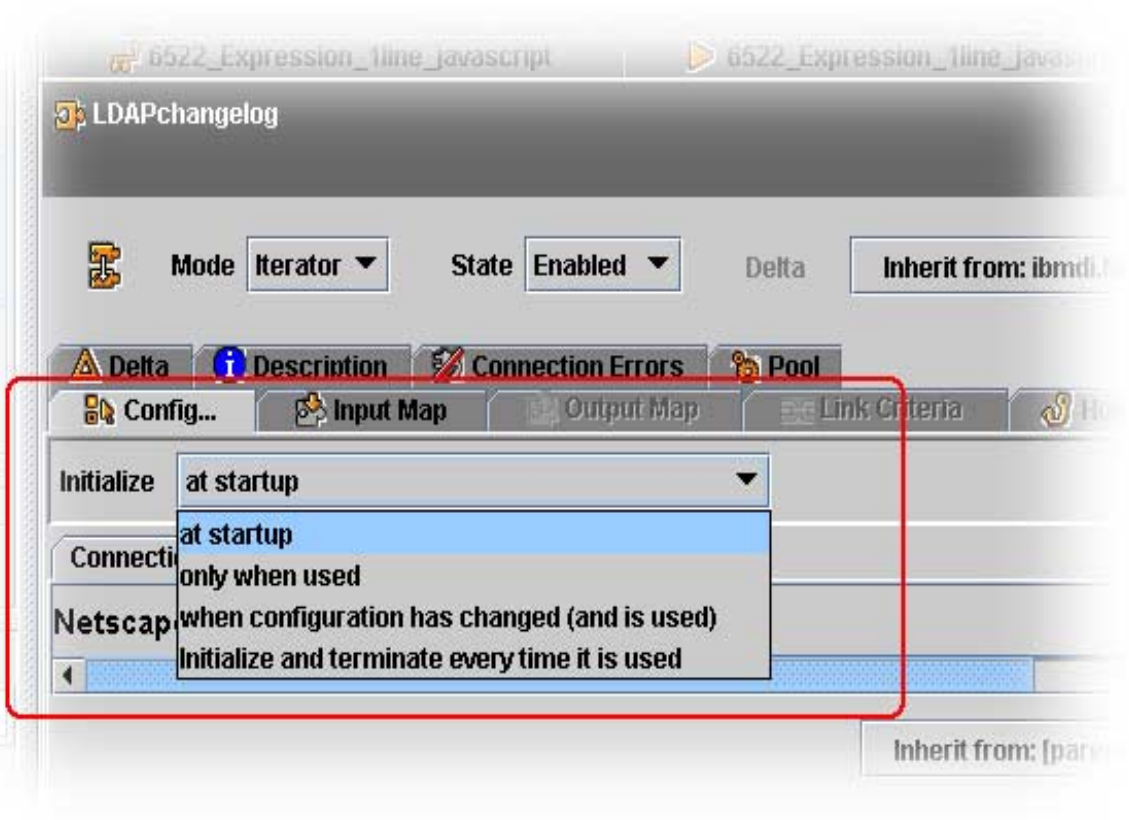
Note that you can pool the AssemblyLine Connector (see page 52 for more details on the AL Connector), which then allows you to create global pools of AssemblyLines[8].

Furthermore, by configuring the AL Connectors based on those in the Pool to "*terminate and Initialize every time"*, you are creating a dynamically shared pool. Every time the Connector is activated in an AL, its initialization process will result in a request to the Pool. After execution of this component is completed, it undergoes *termination*, which for a pooled Connector means being returned to the Pool again. See the next section for more on Initialization Control.

2.17 Connector and Function Initialization Control

It is now possible to control *when* a Connector or FC is initialized (as well as *how often*).

This is done from the "Initialize" drop-down at the top of a Connector or Function "Config" tab:



This selection list provide four options:

at startup	This is the default behavior and is consistent with pre-6.1 behavior. This component will initialize when the AssemblyLine starts up, or if part of a Global Pool, when the Pool Manager sets up the pool after the Config is loaded into a TDI Server.
only when used	Now the component will initialize the first time it is reached in the AL Flow. If this component is never executed during AL operation then it will never

[8] Not to be confused with the AL Pool Manager for Server Mode Connectors, which not only is limited to the scope of a single AL, but is actually a Pool of *Flow* sections to be used to handle incoming connection data.

	be initialized (for example, under a Branch that is never *true*, or a Loop that doesn't cycle even once). There is a possible resource and performance benefit from not initializing unnecessary components, but the trade-off will be pausing during AL operation while components and connections are set up.
when configuraiton has changed (and is used)	Before any Connector or Function operation is performed, check If the value of a parameter has changed since the last time this component was active during AL cycling. If so, then re-initialize it. This includes changes made programmatically from script: myLDAPconn.connector.setParam("ldapUsername", "neo"); or through the use of External Property assignments or TDI Expressions. The new Expressions feature makes it easy to tie parameter settings to Properties (which can be stored in databases or directories as well as files). This makes it easy for you to build Configs that re-configure themselves based on external events. For example, Connectors can be re-directed different servers by simply setting a property through the AMCv3 console; or as the result an Action Manager fail-over rule; or by logic from within the AssemblyLine or Config itself; or via a command issued to the Server using the TDI Command Line Interface (CLI).
Initialize and terminate every time it is used	Ostensibly the most costly of the Initialize settings, this choice means that the Connector or Function is closed and re-initialized every time it is active during AL operation. However, in combination with Pooled Connectors, it becomes a powerful tool in tuning TDI resource use and performance. A Pooled Connector set to initialize every time will cause ALs that use the component to dynamically share them. Instead of each AssemblyLine laying claim to a Pooled Connector for the duration of its operation, this shared resource is terminated after each operation. For a Pooled Connector, this means that it is sent back to the Pool Manager and available to the next AL that needs it. The same is true of the re-initialization of a Pool Connector, which becomes a request to the Pool Manager for the first available instance of this component. This combination of powerful features also lets you build Pools of AssemblyLines by pooling AL Connectors.

2.18 Enhance Connector Initialization Failure Handling

Although the Reconnect feature was introduced in version 6.0, it did not handle initial Connector initialization failures as desired. This mechanism has been enhanced in 6.1 to include a checkbox in the Connector Reconnect tab. The checkbox tells the TDI server to use the Reconnect parameters for initial connection failure in addition to the connection loss during AssemblyLine operation.

In addition, the actual exceptions that should be handled by Connection Loss behavior is now customizable.

By default, rules for defining which errors that trigger this behavior are coded into Connectors themselves. Additional rules can be added to compliment or override these built-in ones.

2.18.1 Low Level Details for Connection Failure during Connector Initialization

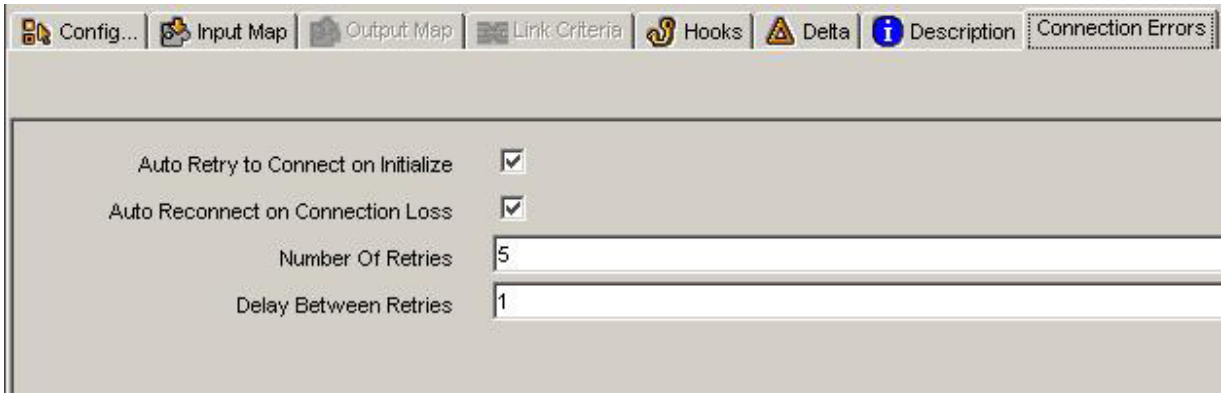
If a connection failure is detected on Connector initialization, the "Prolog" – "On Connection Error" hook introduced in TDI 6.1 Express is called.

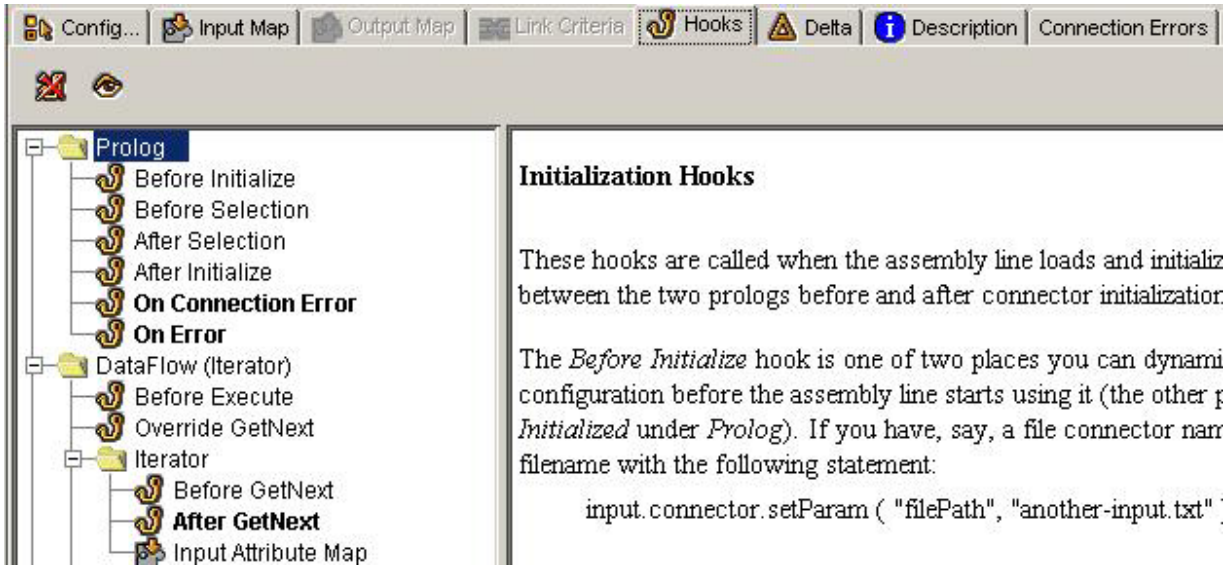
A Boolean parameter in the ReconnectConfig Interface as well as a corresponding User Interface control "Auto Retry to Connect on Initialize" checkbox are introduced in the Reconnect configuration tab.
If the "Auto Retry to Connect on Initialize" checkbox on the Connector tab is checked, TDI 6.1 Express will try to (re)connect x times according to what is set in the Reconnect tab. If the Connector has still not managed to connect, the flow will go to the "On Error" hook.
If a Connector successfully reconnects after the initial connection failure then the execution continues with the "After Initialize" hook except for Iterators where it continues with the "Before Selection" hook.

2.18.2 Changes to the CE (Config Editor)

The following changes have been made to the Connector tab previously labeled "Reconnect":

- The "Reconnect tab" is renamed to "Connection Errors" to reflect that it handles both lost connections and errors during initialization.
- The "Auto Reconnect Enable" label is changed to "Auto Reconnect on Connection Loss".
- A new "Auto Retry to Connect on Initialize" checkbox is added just above the "Auto Reconnect on Connection Loss", to specify that reconnect behavior should also occur for initialization errors.





The "Auto Reconnect on Connection Loss" and the "Auto Retry to Connect on Initialize" checkboxes can be set independently of each other.

2.18.3 Global Connector Pooling and Reconnect

When a Connector is pooled using the Global Connector Pool its initialization logic is invoked by the pool. Also, if the initialization fails the pool is capable of retrying the Connector initialization.

The Global Connector Pool logic for initialization of a pooled Connector as well as the error handling logic is implemented independently of the AssemblyLine logic for initialization and its error handling of Connectors which are **not** pooled.

That is why the initialization of Connectors which are pooled and Connectors which are not pooled is performed by different pieces of code and are thus different.

Please note that both Connectors which are pooled and Connectors which are not pooled are executed in an AssemblyLine context. This means that any errors which occur during the AssemblyLine processing are handled identically for Connectors which are pooled and for Connectors which are not pooled, i.e. the AssemblyLine treats both types of Connectors identically during AssemblyLine execution past the initialization of Connectors.

Differences between initialization of pooled and non-pooled Connectors

Both the AssemblyLine logic for Connector initialization and the Global Connector Pool logic for Connector initialization support retrying the initialization (up to a configurable number of times) in case the initialization fails. The difference is that in the AssemblyLine Context there are error hooks which are invoked after all initialization retries have failed; in the Global Connector pool, however, there are no hooks and thus it is impossible to take action in case all initialization retry attempts have failed.

2.18.4 Built-in rules configuration

The built-in rules reside in the .inf file of the Connector's jar file.

The rules for the particular Connector appear in the "connectors" section as a sibling of the "connectorConfig" sub-section like this:

```
[connectors <connector_class_identificator>]
      connectorConfig {
            <contents_here>
      }
      reconnectRules [
            {
                  <rule 1 here>
            }
            {
                  <rule 2 here>
            }
            …
            {
```

```
                <rule N here >
            }
      ]
      …
 [end]
```

Each rule has the following format:

```
{
      exceptionClass:<fully qualified name of the Java class of the exception>
      exceptionMessageRegExp:<regular expression in Java syntax>
      action:<'error', 'reconnect' or 'ignore'>
}
```

Parameters "`exceptionClass`" and "`exceptionMessageRegExp`" are optional – if not specified, the rule will match all exception classes and all exception messages respectively.

Regular Expression Syntax

For a detailed description of the regular expression syntax used in "`exceptionMessageRegExp`", please see the the JavaDoc of the `java.util.regex.Pattern` class at following URL
http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

Example

```
[connectors ibmdi.ReconnectTest]
      connectorConfig {
            connectorType:com.ibm.di.connector.ReconnectTestConnector
      }
      reconnectRules [
            {
                  action:reconnect
            }
            {
                  exceptionClass:java.io.IOException
                  exceptionMessageRegExp:.*file not found.*
                  action:error
            }
      ]
      description:Reconnect Test
[end]
```

2.18.5 User-defined rules configuration

The list of user-defined rules is configured in a text file named "reconnect.rules" in the "etc" subfolder of the TDI install/solution folder.

Each rule is placed on a single line. The format of a rule is the following:

```
<connector_class>:<connector_name>:<exception_class>:<action>:<regular_expression>
```

where

- `<connector_class>` is the fully qualified name of the Java class of the Connector
- `<connector_name>` is the name of the Connector as inserted in the AssemblyLine
- `<exception_class>` is the fully qualified name of the Java class of the exception
- `<action>` can be on of 'error', 'reconnect' or 'ignore'
- `<regular_expression>` is a Java regular expression as described in the JavaDoc of the `java.util.regex.Pattern` class at http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

Notes

- Each part except the action can be empty. If a part is empty that means "match-all".
- Each part is mandatory – even if it is empty the surrounding colons must be present. (Consequently on each line there must be at least 4 colons – each colon separating two adjacent parts of the rule. At least 4, because the regular expression may contain colons too. These colons will not interfere with the rule parsing because the regular expression comes last in a rule.)
- No redundant white space is allowed.
- The regular expression starts just after the fourth colon and spans till the end of the line.

Example

```
com.ibm.di.connector.ReconnectTestConnector:myconnname:java.io.IOException:error:.*\Wfatal\W.*
::java.io.IOException:ignore:
```

2.19 Disabling AL components via the Task Call Block (TCB)

In TDI 6.0 each of the components (Connectors, Function Components, etc.) hosted by an AssemblyLine are created and initialized on AssemblyLine Initialization.

In TDI 6.1 Express it is possible to programmatically specify that certain AssemblyLine components must not be created or initialized on AssemblyLine initialization. This is done by disabling those components.

AssemblyLine components are enabled by default.

In order to enable/disable a component in the AssemblyLine one has to call the `com.ibm.di.server.TaskCallBlock.setComponentEnabled(String name, boolean enabled)` method on the TaskCallBlock (TCB) object of the AssemblyLine. The `name` argument of the method specifies the name of the component to be enabled/disabled. The `enabled` argument of the method specifies whether the component is to be enabled or disabled.

The actual enabling/disabling of the AssemblyLine components happens in the `com.ibm.di.server.TaskCallBlock.applyALSettings(AssemblyLineConfig alc)` method. This method is invoked on AssemblyLine initialization. As the initialization of the AssemblyLine progresses the components which have been marked as disabled do not get created/initialized.

If a LOOP component is disabled then all components contained in that LOOP will also be disabled.

Even if a component id disabled from the Config Editor GUI, it can be enabled using this feature and vice versa.

```
com.ibm.di.server.TaskCallBlock.setComponentEnabled(String name, boolean enabled)
```

2.20 AssemblyLine Operations

Traditionally, AssemblyLines have performed a single function[9]. Although they could contain branches for dealing with specific situations or data values, an AL was traditionally designed to move data in one direction from a set of data sources to another set of targets. As a result, when you used the 6.0 web services components to expose an AL as a service, multiple ALs necessary to provide additional services.

TDI version 6.1 introduces the concept of AssemblyLine Operations, allowing you to implement any number of distinct functions to be performed by an AL. Each Operation has an associated set of Input and Output Maps for defining both parameter values passed in when an Operation is called, as well as Attributes returned after the called AL Operation is finished. This extends and replaces the single set of Call and Return Attribute Maps found for AssemblyLines in previous versions.

Once you have defined Operations for an AssemblyLine, the new Switch-Case (section *2.7.4 Switch/Case Components* on page 20) contructs let you easily implement the logic in the AL to deal with them. Furthermore, both the AL Function (FC) and the AL Connector have been enhanced to support AL Operation calls.

AssemblyLines with Operations can be *published* as "Adapters", using the AL Publishing feature described in section 2.22 on page 55. These Adapters show up as Connectors and can easily be added to other AssemblyLines or Config Connector Libraries.

If you drop the improved Web Service Receiver Server Connector into an AssemblyLine, it can generate the WSDL for the AL based on its Operations and the asssociated Attributes.

And, of course, AL Operations are accessible through API calls. As a result, the new CLI (commandline interface, described on page 61) for TDI and AMC v3 (page 79) both offer features for calling specific AL operations, and for passing Attribute values between the calling AL and the called AL.

---

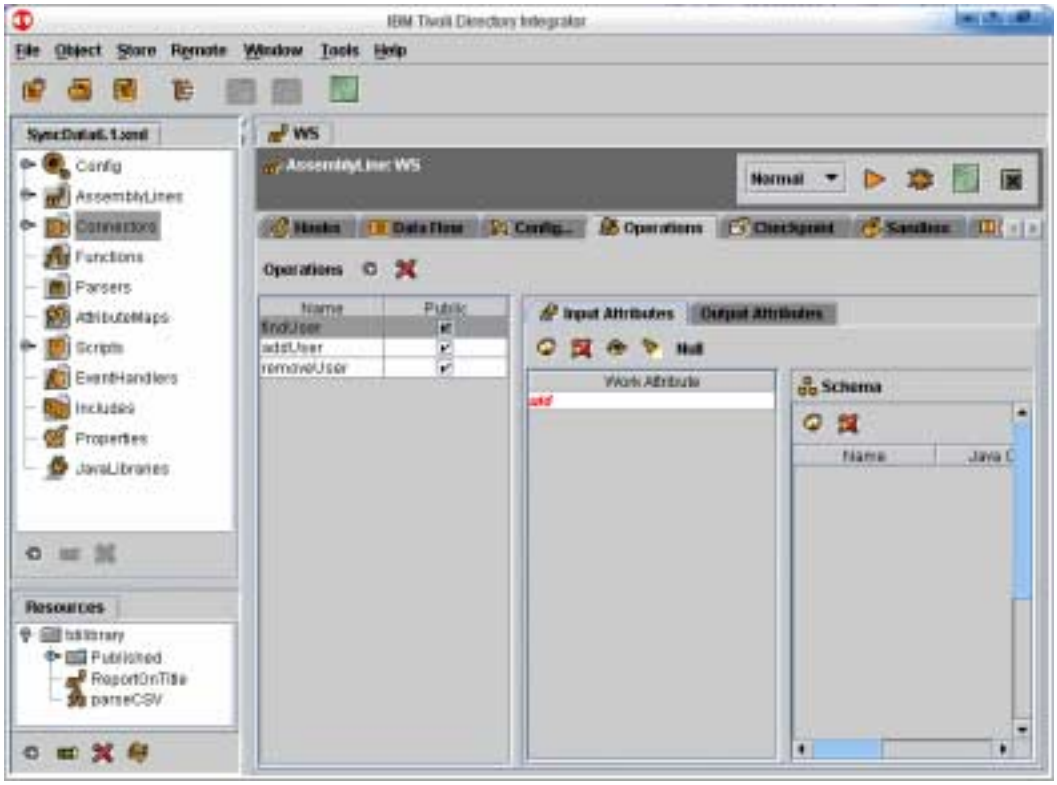
[9] Of course, you could always engineer your AL with branching logic, changing its behavior by passing in an Attribute value with the operation you want to perform, or setting a property or global variable. One example of an AL that deals with multiple "commands" (read: operations) is the sample v.6.0 WhitePages Config published here:

http://www.tdi-users.org/twiki/bin/view/Integrator/WhitePages

The WhitePages AL supports a number of browser-based operations which are handled by the Branches "IF_Delete", "IF_Add", "IF_Modify", and so forth. The new AL Operations feature makes this type of functionality easier to build, use and maintain.

2.20.1 Defining AL Operations

Operations are managed in the "Operations" tab of an AssemblyLine.



The *Insert* and *Delete* buttons at the top of the Operations list are used to add and remove Operations for an AssemblyLine. Select an Operation in the list in order to edit the Input and Output Maps associated with it.

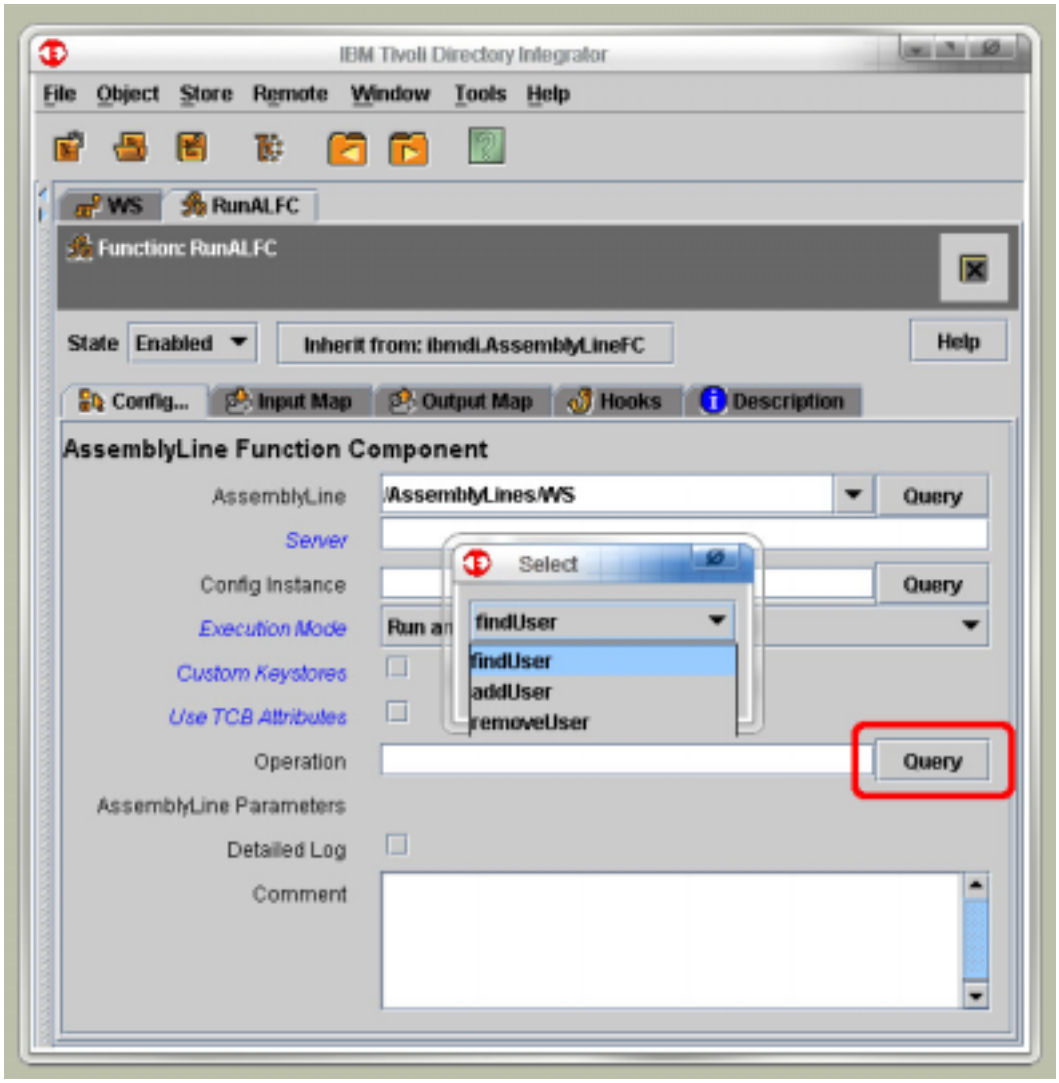
If no operations are defined, then TDI considers the AL to have a single "default" operation, allowing existing Configs to run without modification.

2.20.2 Calling AssemblyLine Operations

In addition to API calls, there are a number of Components for running and controlling AssemblyLines. These have all been enhanced to support AL Operations.

2.20.2.1 The AL FC

This component now offers a Query button for retrieving Operations from the configured AL.



Once you have selected an Operation, the relevant *call* and *return* Attributes appear in the Schemas for the Input and Output Map of the FC[10].

2.20.2.2 The AL Connector

The *AssemblyLine-caller* Connector, called the AL Connector allows you to exploit an AssemblyLine as a single component in another AL. Furthermore, it enables the development of custom Connector *Modes* that represent the various operations implemented by the called AL.

---

[10] A bit confusing at first, the Output Map of a component (AL FC, AL Connector) is what is passed *out* to the AL when it is called. They correspond to the Input Map of that AL Operation (everything is relative ;). Conversely, when the called AL completes (either running, or a single cycle in manual mode) Attributes are returned via the called AL's Output Map which is passed to the calling component's Input Map.

This component works in a similar fashion to the AL FC, except that it provides direct access to AL Operations via the AL Connector's Mode setting. So in addition to implementing the standard Modes like Iterator, Lookup and AddOnly, your AL Connector can make use of any mode you can dream up, like EnableAccount Mode, ConvertToMetric Mode or ReadMergedFiles Mode.

Note that in order to provide the standard Connector Modes, you must implement operations for each of the standard Connector Interface functions: like selectEntries and getNextEntry for Iterator Mode, and findEntry and deleteEntry for Delete Mode. The list of standard Modes that can be implemented this way is:

Mode	AL Operations required
AddOnly	putEntry
Call/Reply	callReply
Delete	findEntry *and* deleteEntry
Iterator	selectEntries *and* getNextEntry
Lookup	findEntry
Update	findEntry *and* putEntry *and* modEntry

Note that since AL Connectors can be stored in the Connector Library folder of the Config Browser, and these can be designated as a *pooled* Connector, it is not possible to configure global AssemblyLine pools.

2.20.3 Using Operations from JavaScript

Specifying the operation to call, as well as the Attribute values to pass into the called AL are both done in the Task Call Block (TCB).

```
// Set up a new TCB and choose the “findUser” Operation.
//
var myTCB = system.newTCB();
myTCB.setOperation(“findUser”);

// Now I have to pass in the ”uid” Input Attribute
// defined in the example screen shown earlier in this section.
// The value I want to use is in a work Entry Attribute called “userID”.
//
myTCB.setOperationInitParam(“uid”, work.getString(“userID”));

// Now I can call the AL Operation.
//
var al = main.startAL( “WS”, myTCB );
al.join(); // Wait for it to complete
var resEntry = al.getResult(); // Return the resulting Attribute
```

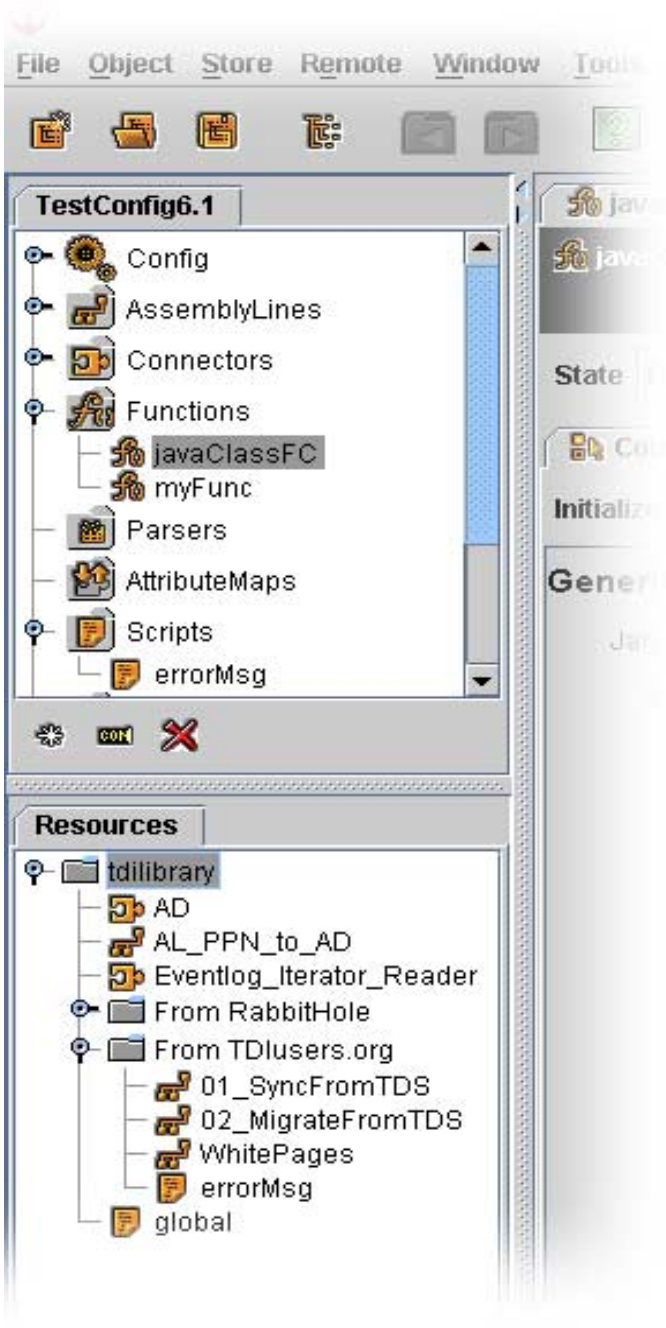
A more efficient way of calling AL Operations is by using the AssemblyLine's Manual Mode feature. If you start the AL in Manual Mode, then it fires up all components and returns without actually doing any processing:

```
myTCB.setRunMode(“manual”);
var al = main.startAL( “WS”, myTCB );
```

Now you can make calls to different AL Operations each time you cycle the AssemblyLine manually with the executeCycle() method.

2.21 Resource library/project dev model

TDI now provides a simpler method for sharing AssemblyLines and components by introducing the TDI Resource library. Onscreen in the Config Editor, the new "Resources" navigator appears just below the Config Brower:



The Resources shown here reflects a directory structure located at the path specified in the Solution or Global Property by the "com.ibm.di.admin.library.dir" property. By default, this is set to "<User Home Dir>/tidlibrary", but can be reassigned to point to any other folder, including shared space on a network disk.

ALs and components can be dragged to-and-from between the Config Browser and Resources. However this works the same as dragging between multiple Configs open in the Config Editor: you only get the items you drag-

and-drop[11]. So if you drag an AssemblyLine to Resources (or to another open Config), it won't bring along any Library components that are being inherited from.

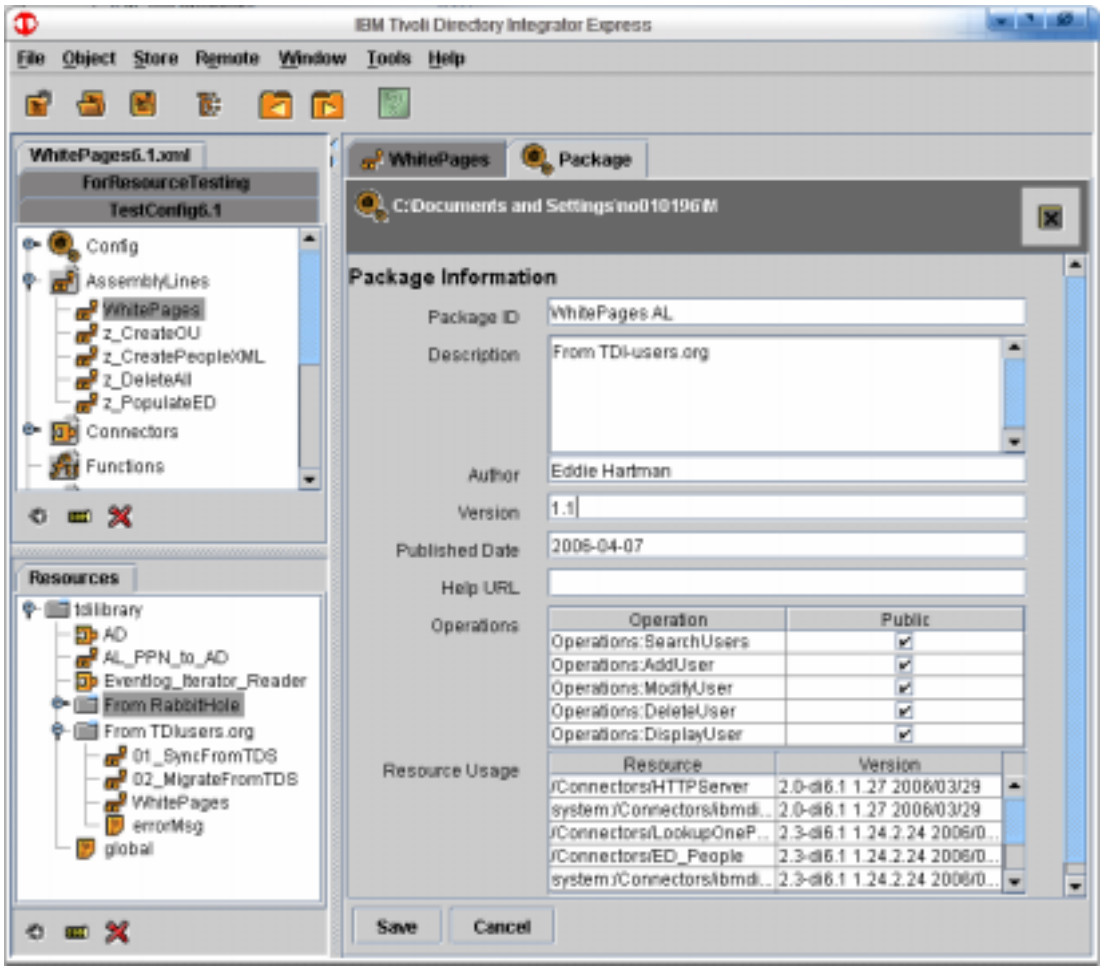
The answer to this dilemma is the new AssemblyLine Package Publishing feature

2.22 Publishing AssemblyLines (Adapters)

The concept of "Adapters" is to allow you to publish an AL as though it were a Connector. Although this type of functionality has been around for a while, AL Publishing makes this powerful feature easy-to use.

2.22.1 Publishing a *Package*

When you publish an AssemblyLine, TDI resolves all dependencies to inherited Library components, as well as processing the operations defined for this AL. Publishing is done by right-clicking on an AL in the Config Browser and then select "Publish...", which brings you to the Package Information screen:



[11] Note that instead of dragging ALs between Configs, if you right-click and Copy it instead, then inherited components are also copied. When you then paste into another Config, you get all the library components that this AL is dependent on.

Here you can give this package an ID and other related info. The Package Publisher analyses your AL and shows you the AL Operations published and components inherited from your Library. When you press the Save button, a copy of the AssemblyLine is created and *flattened* – removing any inherited dependencies and unused Config settings – and then written to the Adapter directory. This is by default a folder called "Adapters" under your Solution Directory.

TDI reads this folder during CE or Server startup, making any packages published here available as Connectors. Packages can simply be copied to the "Adapters" folder in another Solution Directory. If you put a package in "jars" sub-folder of the TDI Installation Directory, then it is available regardless of which Solution Directory is used[12].

2.23 EventHandler transition complete

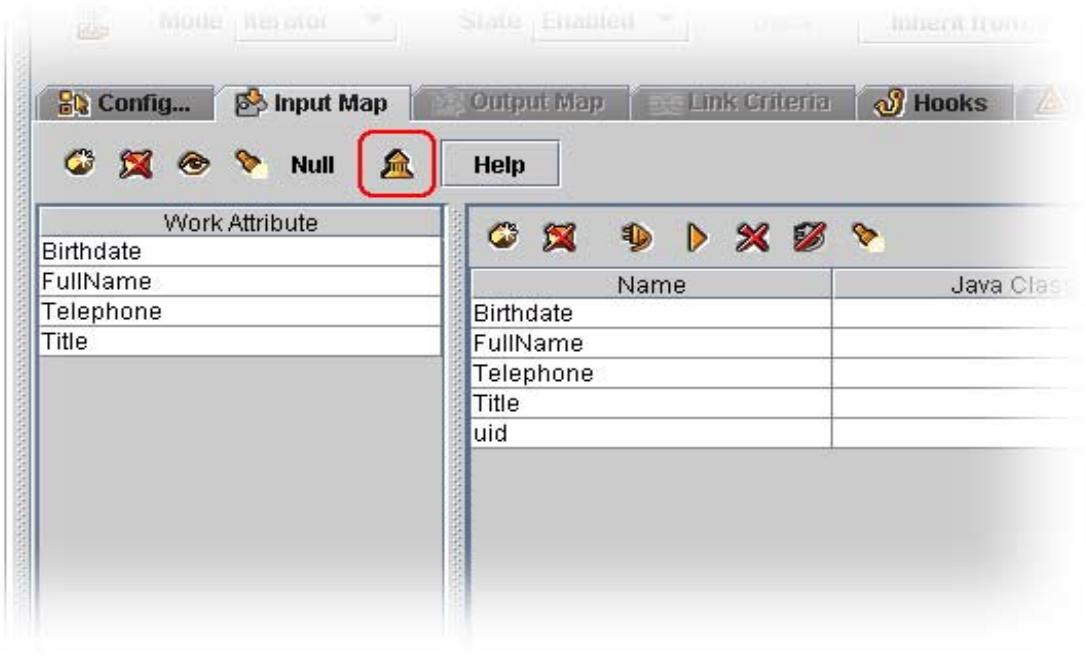
EventHandlers are deprecated in 6.1. This means that if you create a new Config, no folder called "EventHandlers" will be visible in the Config Browser window of the Config Editor.

However, TDI still supports pre-6.1 Configs using EventHandlers. If you open such a Config, then the Config Browser will contain the "EventHandler" library folder, and the TDI Server still supports their operation.

[12] It is recommended that you create a sub-folder under "jars" called "Adapters" in order to keep your packages separated from the other library files here.

2.24 Library Feature & Copy/Paste for Attribute Maps

AttMap components, while already supporting drag-and-drop to Input or Output Maps on Connectors and Functions, now provide the reciprocal feature: there is a new "Copy to Library" button for all Input and Output Maps that allows you to copy these as AttMaps to the Config Library.



Note that a simply mapped Attribute can be used for either Input or Output Maps. However, JavaScript or Expression based Attribute mappings will probably contain explicit references to Entry objects like work or conn. As such, you must make sure that they fit the context that you want to use them in.

2.24.1 Copy/Paste of Attributes

You can now also right-click any Attribute in a map and copy it, allowing you to paste the entire Attribute Map definition for this item to any other Map or Schema (like for an AL Operation).

2.25 Copy/Paste for Config objects

Although Copy/Paste of Config objects (ALs, Connectors, FCs, etc.) was introduced in version 6.0, this feature has been enhanced and corrected in the latest release.

You can now easily Copy ALs and components and then paste them into another Config. You can also exchange them via IM chats, emails and text files, since the copy-buffer is filled with the TDI Config XML definition of the selected item. This makes passing stuff around simple and easy, and is a great tool for support and online assistance (e.g. ICT/NotesBuddy, forums, …).

Note: Make sure you get the whole <MetamergeConfig> node in your copy command, including the start and end tags.

2.26 SystemQueue

In addition to the System Store, which was available in previous versions, TDI 6.1 Express now offers a System Queue. This is a built-in queue infrastructure to facilitate communications between ALs, in much the same way that the MemQ feature (and components) do. The big difference is that the System Queue enables data transfer across multiple TDI Servers.

By default, the System Queue uses the bundled MQe (WebSphere MQ Everyplace), but can be configured to leverage other JMS-compliant queuing systems.

2.26.1 Enabling the System Queue

Enabling the System Queue is done in the Global/System Properties by setting the property called **systemqueue.on** to "true". This causes the System Queue service to be started when the TDI Server starts.

2.26.2 SystemQueue Connector

TDI offers a SystemQueue Connector that can be used to Iterate off a queue, as well as add Entry data to one.

2.27 Complex XML Handlers: the SDO-based FCs

TDI 6.0 featured the Castor JavaToXML and Castor XMLToJava Function Components, which serialized Java bean objects to XML and parsed XML into Java bean objects respectively. These two components (in fact, the entire Castor library and all related componentry) have been removed in TDI 6.1 Express and replaced with greater functionality provided by the Eclipse Modelling Framework[13] (EMF) library.

As a result, two new FCs have been added: the SDOtoXML and XMLtoSDO Function components, both of which leverage Service Data Objects (SDOs) to let you work with arbitrarily complex XML documents.

This section contains an overview of this functionality. For details, refer to the Tivoli Directory Integrator 6.1 Express Reference Guide.

2.27.1 XMLtoSDO FC

This FC parses an XML document and creates SDO objects connected in a tree-like structure resembling the XML structure.. Once in this format, the XML data and hierarchy can easily be accessed via scripted function calls, for example, in scripted or Expression-based Attribute Maps, or in Script components (SCs) or Hooks.

For each XML element and XML Attribute a Data Object (SDO) is created. Then TDI Entry Attributes are created for some of these. The name of the Entry Attribute consists of the names of the full chain of ancestors of the element this TDI Attribute represents. Individual XML elements are separated in the Attribute name by the "@" character. When an XML *attribute* is represented, the name of the XML attribute is appended to the name of the XML element using the "#" symbol as a separator.

All Attribute names start with the "DocRoot" text which represents the XML root. The type of the value of the Entry Attribute is:

- either the standard Java object when the XML element/attribute value is a primitive type (java.lang.String, java.lang.Integer, java.lang.Boolean, etc.)

[13] The EMF web page "http://www.eclipse.org/emf" provides full information about the Framework and the related libraries. The SDO object is of type *org.eclipse.emf.ecore.sdo.EDataObject.*

- or a Service Data Object when the XML element is a complex XML structure. This will be an object of type *org.eclipse.emf.ecore.sdo.EDataObject*.

XML elements that have a common parent element are called siblings. Sibling elements with the same name are grouped in a multi-valued TDI Attribute.

Important: Entry Attributes are not created for XML elements with an ancestor element that has a sibling with the same name. Those can only be accessed through the multi-valued Attribute representing the siblings.

Let's take the following sample XML document:

```
<?xml version=”1.0”?>
<database name=”Persons”>
      <description>This is a sample database</description>
      <person>
            <name>Ivan</name>
            <age>21</age>
      <person>
      <person>
            <name>George</name>
            <age>32</age>
      <person>
</database>
```

After this XML document is processed by the EMF XMLToSDO Function Component, an Entry with the following Attributes will be created:

- **DocRoot** – a Service Data Object representing the XML root
- **DocRoot@database** - a Service Data Object representing the “database” XML element
- **DocRoot@database#name** – a java.lang.String object representing the “name” XML attribute of the “database” XML element.
- **DocRoot@database@description** - a java.lang.String representing the “description” XML element (which is a child of the “database” XML element).
- **DocRoot@database@person** – multi-value attribute whose values are Service Data Objects representing the individual “person” XML elements.

“*DocRoot@database@person@name*” would not be a valid TDI Attribute in this case because more than one person XML element exists in the XML document at the same level (i.e. identically named siblings).

The Function component will provide an option to use namespace prefixing in which case all XML element names in the Entry Attribute name will be prefixed with the corresponding namespace, for example “*DocRoot@namespace1:database@namespace2:person*”.

See section “**Error! Reference source not found.**” for more details on how XML elements and attributes are mapped to TDI Entry Attributes.

2.27.2 SDOtoXML FC

This component uses an XML schema (XSD) to serialize an SDO to an XML document.

The Function Component receives an Entry whose Attributes represent an XML document. The types of the Entry Attribute values are either Java classes representing primitive types or Service Data Objects (SDOs) representing complex XML elements.

The Entry Attribute names describe the XML hierarchy in exactly the same manner as the EMF XMLToSDO Function Component constructs Attribute names: All Attribute names start with "DocRoot" which represents the XML root. Subsequent elements down the XML hierarchy are separated with the "@" character. If the TDI Entry Attribute represents an XML attribute the "#' character is used to separate the name of the XML attribute from the name of the XML element containing this attribute.

It is possible that the TDI Entry passed contains only Entry Attributes corresponding to the actual data. For example, the Entry may contain an Attribute "DocRoot@database@person" without containing an Attribute "DocRoot@database" – the EMF SDOToXML Function Component will automatically create the "database" XML element in the XML document it builds. The EMF SDOToXML Function Component uses the XML Schema to track and create all XML elements that are ancestors of the specified XML element or attribute.

It might happen that the Entry contains Attributes specifying XML elements that are contained in other XML elements specified by Entry Attributes, for example the Entry contains both "DocRoot@database@person" and "DocRoot@database" Attributes. In this case the Attributes are processed starting from the one that is closest to the root, continuing with the one closest to it and so on – the last one will be the most specific XML element that is contained in all the other. This order of processing provides the option to change specific details in a bigger XML context.

Let's say, for example, that you want to change just the "DocRoot@database@person" element but you want to leave the other parts of the "DocRoot@database" element untouched. You would read the XML document with the EMF XMLToSDO Function Component and map in both the "DocRoot@database" and "DocRoot@database@person" Attributes.

In your EMF SDOtoXML FC, you map out the "DocRoot@database" Attribute as-is, but change the value(s) of "DocRoot@database@person" as desired – e.g. in the Input Map of the XMLtoSDO FC, or the Output Map of the SDOtoXML FC, or elsewhere in your AL. The EMF SDOToXML Function Component will first process the "DocRoot@database" applying the unchanged content to the resulting XML and it will then override the "person" child of the "database" element with whatever is provided in the "DocRoot@database@person" Entry Attribute.

In case a multi-valued Attribute is provided together with an Attribute specifying a child or other successor of that element, the function Component will signal an error (throw exception) because it cannot be determined to which of the sibling XML elements, this successor applies. For example, if "DocRoot@database@person" is provided and contains two values (thus specifying two XML "person" elements at the same level) and also "DocRoot@database@person@name" is provided, the Function Component would not know to which "person" element of the two existing this "name" element applies to.

2.27.3 Namespaces

The names of the elements in the Entry Attribute name can be XML namespace prefixed. The names of the elements are prefixed with the namespace URI or with the prefixes defined in the "namespaceMap" parameter.

For example, in order to construct the following XML document:

```
<?xml version="1.0"?>
<database xmlns="www.ibm.com" xmlns:tmp="www.tmp.com" name="employees">
      <person>
            <name>Ivan</name>
            <tmp:age>21</tmp:age>
      <person>
</database>
```

the following TDI Entry can be passed to the EMF SDOToXML Function Component:

- DocRoot@ibm:database#ibm:name
- DocRoot@ibm:database@ibm:person@ibm:name
- DocRoot@ibm:database@ibm:person@www.tmp.com:age

The namespace prefixes used assume that the "namespaceMap" parameter contains the "ibm" prefix set to "www.ibm.com" and no namespace prefix is defined for "www.tmp.com" (that is why it is used directly in the Attribute name).

2.27.4 Discover Schema is supported

Since the Attribute mapping for both the EMF XMLToSDO Function Component and EMF SDOToXML Function Component is based on the XML Schema, both components provide a convenient Discover Schema functionality at design time. Users don't need a real XML instance document, they only need the XML Schema definition.

2.27.5 Error Flows

The following errors can occur during the execution flow.

- In both function components the XSD File parameter is required. If the parameter is not specified an exception is thrown on initialization.
- If the specified XSD File cannot be found an exception is thrown on initialization.
- If the specified XSD File is not a valid XML Schema file an exception is thrown.
- If the "namespaceMap" parameter has an invalid syntax an exception is thrown on initialization.
- If an invalid XML document is passed to the EMF XMLToSDO Function Component an exception is thrown.
- If an XML document not conforming to the XML Schema is passed to the EMF XMLToSDO Function Component an exception is thrown.
- If an Entry Attribute passed to the EMF SDOToXML Function Component has a name referring to an invalid XML element or attribute an exception is thrown.

2.28 Command Line Interface (CLI)

TDI 6.1 Express offers a stand-alone command line tool for accessing and controlling a local or remote TDI Server. This tool allows you to connect to a running Server and then start and stop Configs and ALs, reload Configs, query status, read and set properties, send events and shut down the Server.

This program is located in the <TDI Installation Directory>/bin folder and is called **tdisrvctl**. It offers a complete usage display if you fire it up without parameters (or with invalid ones).

Some typical examples of usage are:

tdisrvctl –op status

which results in a listing of loaded Configs, as well as the status of all ALs and EHs.

tdisrvctl –op report –c "c:/tdiwork/MyConfig.xml"

which produces a textual report of the Config specified.

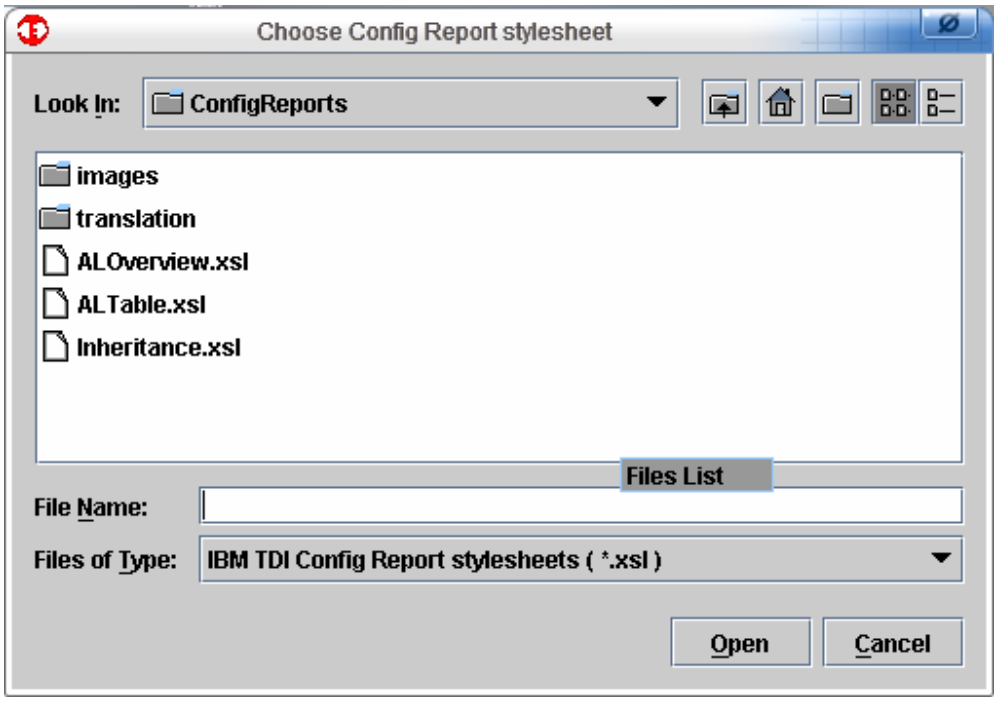
Note that in the above two examples, the default host and port are used (localhost:1099). Also, for the "op report" example above, note that you must enter the full path to the Config, *except* if this Config is located in the TDI Configs folder (specified by the **api.config.folder** Global/Solution property). You can request the list of Configs stored in the Configs folder with the following command:

tdisrvctl –op report –l

This will return a list of Config files found for that Server.

2.29 Config Reports

By right-clicking on an AssemblyLine or component in the Config Browser (or on the "AssemblyLines" folder itself) you get a option called "Config Report". Selecting this brings up a File Browser dialog where you can choose which Config Report to run.



Each report is an XSL transformation stylesheet that the CE can apply to the Config XML. The ALOverview Report produces an overview of a single AssemblyLine (or multiple if selected for the "AssemblyLine" folder), and is handy as a hard-copy of your AL[14]. The Inheritance Report details how Library componets are being utilized both by other components in the Library as well as in AssemblyLines. The last report, ALTable is a simple example stylesheet.

In each case, once the report to run is selected, TDI generates an XML view of the Config in memory and then transforms it with the chosen report stylesheet. The resulting HTML is then displayed in a browser window.

Note that since you have the full source to these report stylesheets, you can use them to create your own custom reports..

2.30 Property Store Framework

TDI solutions are packaged into one or more configuration files (Configs in XML format). These Configs hold the settings for end point connections, data flow and a host of other features. Although a Config can hold everything necessary for a complete solution, users often leverage external data sources – called External Properties files – to make their solutions configurable *outside* the Config Editor.

[14] Note that although inheritance is indicated in the report, inherited settings are not shown.

In addition to External Properties, TDI has a number of system and Java properties that are set in various configuration files, like global.properties and solution.properties, as well as those configured for the JVM itself.

Some of these properties are implicitly used by TDI components, like external properties tied to parameters, and system store settings found in global.properties; whereas others are used explicitly by the solution developer through scripting:

```
if (system.getExternalProperty("sortReturnedData") == "true")...
```

You can define multiple External Properties, and have TDI search one or all of these. However, External Properties were managed and handled separately from other properties, like Java properties, was done using other methods. Furthermore, only External Properties could be tied to component parameters.

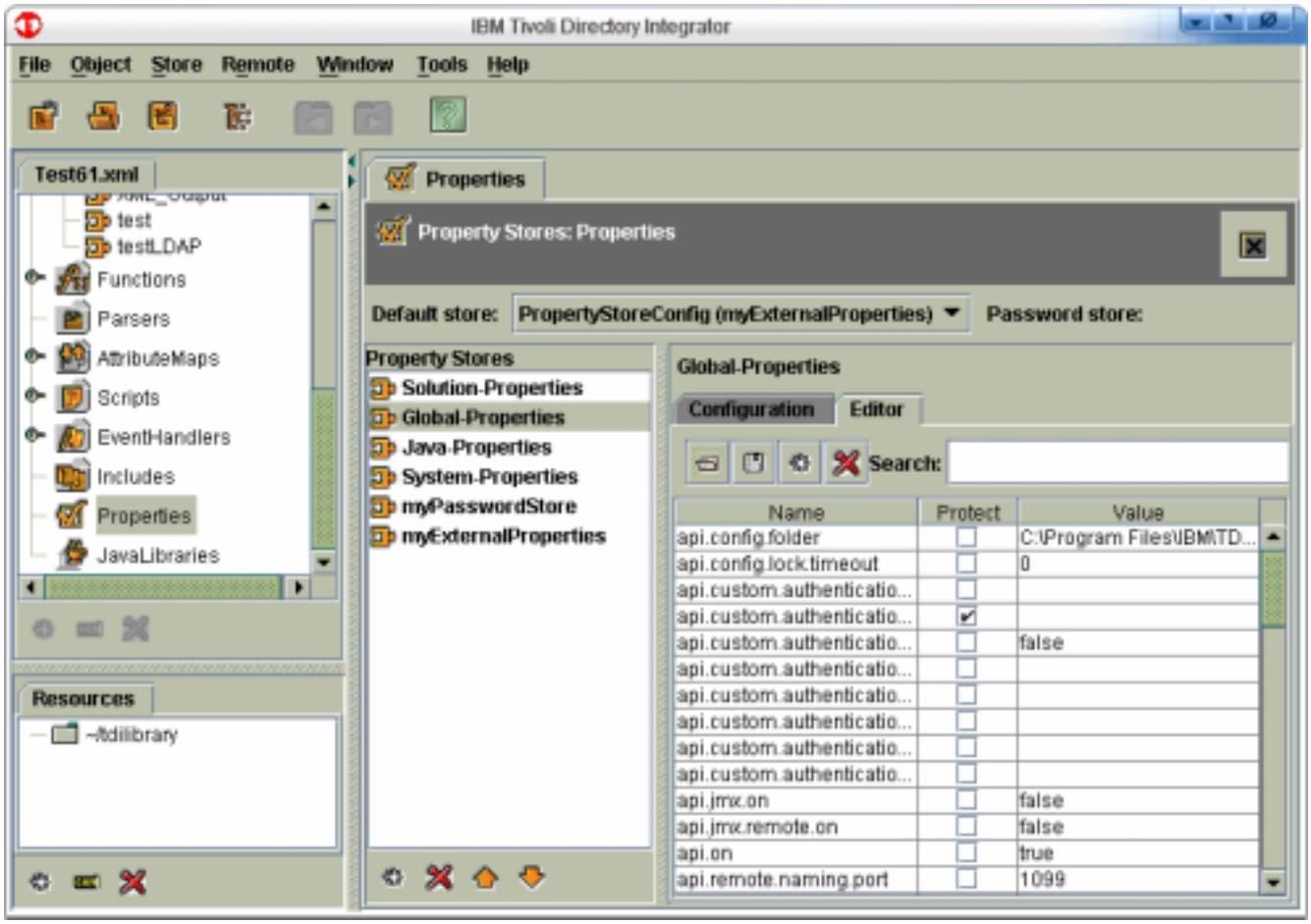
TDI 6.1 Express introduces the ***TDI Properties Framework***, providing a common interface for managing and using all TDI-related properties. It builds on Connector technology, allowing you to read and write properties to a broad range of systems and data stores (not just files as in previous versions).

2.30.1 Overview of Properties

As before, Properties can be used to control the configuration of components. Together with the new TDI Expressions feature[15], you can now compute these values using Attribute values, Java Properties and even JavaScript code.

TDI 6.1 Express also extends the use of Properties and Expressions to Attribute Maps, Link Criteria and Conditions. And, of course, it is fully accessible from script and via the API.

The Properties selection in the Config Browser opens up the Properties window.



Solution-, Global-, Java- and System- properties are available by default. **Solution-** and **Global-Properties** are stored in the solution.properties and global.properties files, respectively. **System-Properties** are also available by default, and are kept in the System Store, giving you access to user-defined properties without having to set up an External Properties file[16]. Finally, Java-Properties, which are only kept in memory (not stored anywhere), provide access to configuration settings of the JVM itself.

---

[15] Be sure to read section *2.31 Expressions* on page 69 for more information on this powerful new tool.

[16] If you run the System Store in networked mode, then this allows you to easily share System-Properties between solutions.

There is a row of buttons at the bottom of the Property Stores list for creating new Stores based on Connector-technology; for deleting existing ones; and for moving Stores up and down in the list. This is significant since TDI always uses the first instance of any key-pair read, and starts searching from the top of the Property Store list[17].

At the top of the Properties window are two drop-downs: **Default store** and **Password store**[18], both of which can be set to reference one of the defined Property Stores.

Default store The Default store is the one that is searched first, as well as where new property values created via API calls go. The designated Default Store takes precedence over the sorting-order of the Property Store list[19].

On reading a property, the first Property Store that has a presence for the property is used. Conversely, when writing a property, the first Property Store that has a presence for the property will be used to write it back. This ensures that the location of the property remains consistent when reading and writing properties[20]. Of course, this behavior can be overridden by explicitly naming the store to be used in the API call.

Password store Here is where TDI will automatically store password parameters for Connectors, removing this sensitive information from the Config file and into an encrypted store. See section *2.10 Securing Configs, Passwords and Sensitive Data* on page 23 for more details.

2.30.2 Editing Properties

To bring up the details of a Property Store, simply select it in the list.

All Properties have **Configuration** and **Editor** tabs. When you add new properties then these are always *Connector-based* types and part of their configuration is *where and how* to store these key-value pairs. As a result, user-defined Property Stores have an extra **Connector** tab in their Details view:

---

[17] This means that you can decide which of the two "historically overlapping" Property Stores (global or solution) has precedence. Previous versions always read global.properties first and then solution,properties, and TDI used the *last* key-pair read. As a result, settings in solution.properties overrode those in global.properties. Now you can change this, or even create a Property Store that sorts first, allowing you to override any setting in another Store without having to changing it directly.

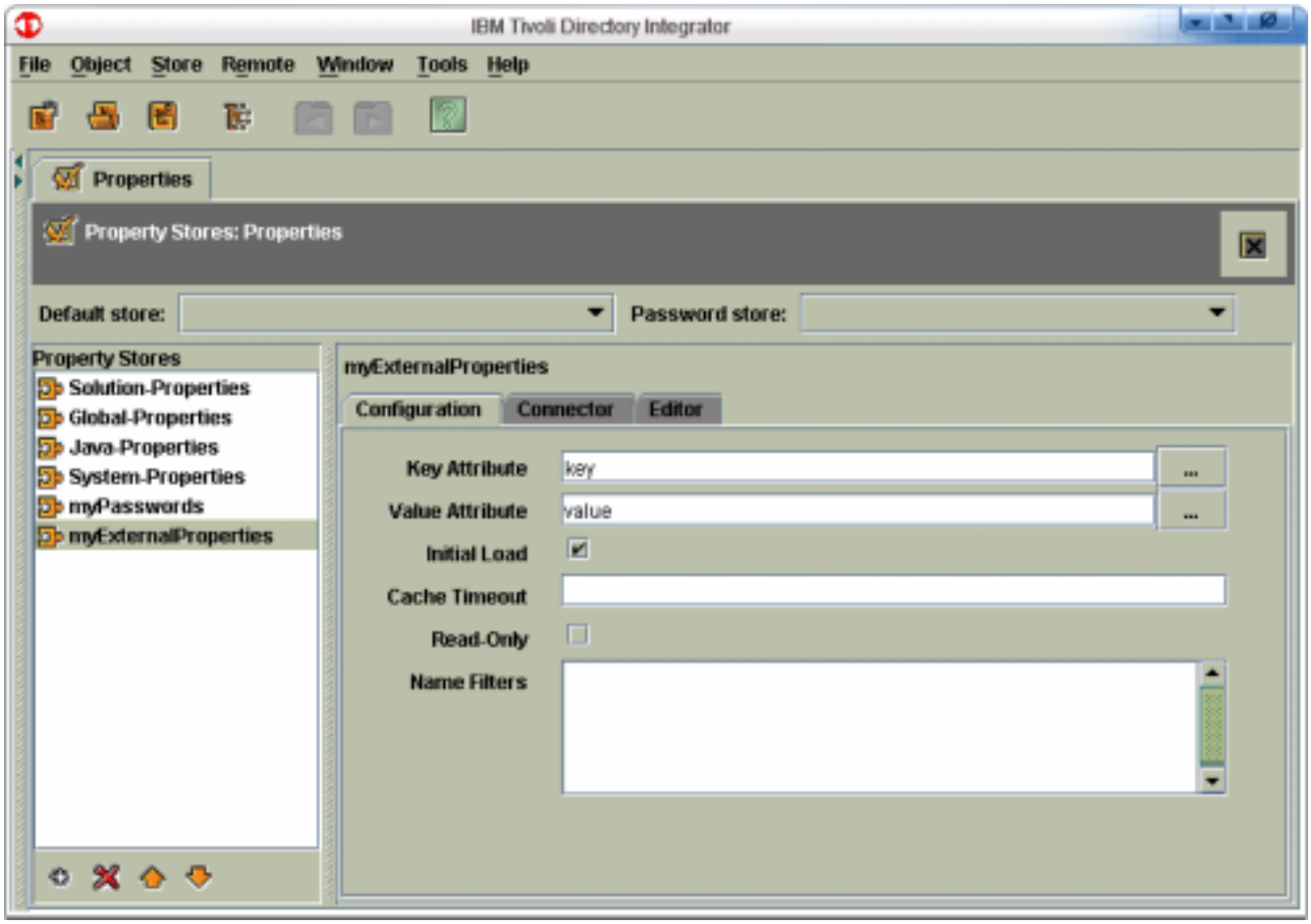
[18] Note that depending on the size of the TDI Config Editor window, some drop-downs are not displayed correctly (as is the case with "Password store" in the above screenshot if the window is sized any smaller).

[19] When TDI 6.1 opens older Configs, existing External Properties defined there are defined in the Property Store list, and the current default store ("_Default") will added at the top.

[20] All property names (keys) are case-sensitive, unless the store used for persistence behaves differently.

2.30.2.1 Configuration tab

This tab defines the layout (Schema) and behavior of this Property Store. These settings include the *schema* of the Property Store (for example, the database column names if your Property Store is JDBC-based), and a Read-Only flag to control access to these properties in this Store.



The first two parameters, **Key Attribute** and **Value Attribute** define the schema of the Property Store.

The **Initial Load** switch is relevant only when local caching is enabled (see the "Cache Timeout" parameter below) and the connector supports Iterator mode. If true, then the Connector iterates once through the entire store to build a complete local in-memory cache. This is instead of doing searches when individual properties are accessed.

Cache Timeout is set to the number of seconds that a property can be cached before another call to the connector is made to refresh the contents. Legal values are:

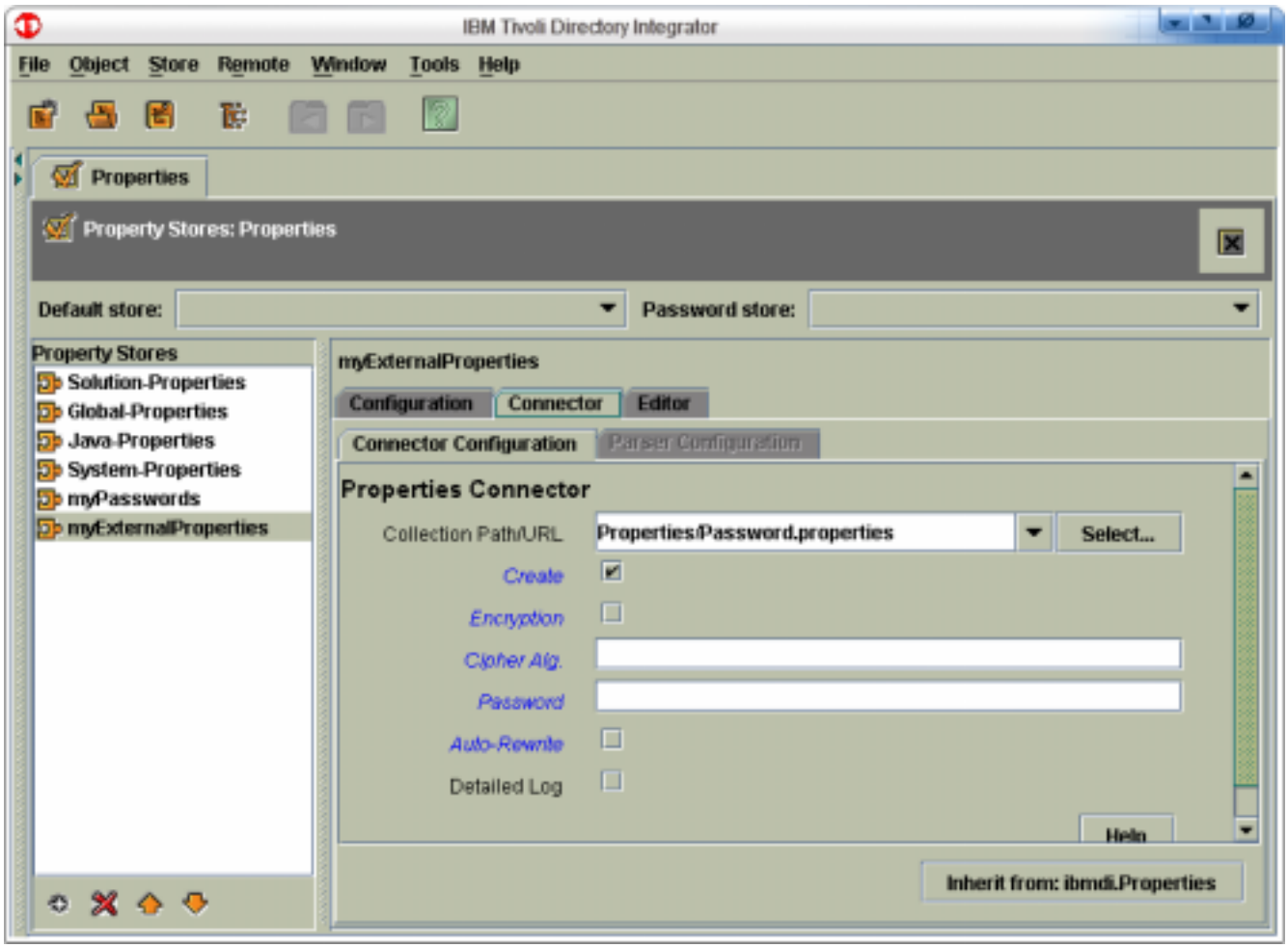
- -1 = No timeout (always cache)
- <no value set>= Cached disabled
- 1->MAX_INT = time out in seconds

The **Read-Only** flag controls whether this Store can be written to or not via API calls (note that your Password Store should ***not*** be Read-Only).

Finally, the **Name Filters** are used to control which properties this store is to provide access to. The store could contain additional properties, but only those that conform to the Name Filters are visible via the Property Manager calls.

2.30.2.2 Connector tab

This tab is only available for user-defined Property Stores (i.e. like External Properties in previous versions).



First, note the "Inherit from:" button at the bottom-right part of the Connector tab. Here you can control which type of Connector to use for this Property Store. The default setting is the ibmdi.Properties Connector, which reads and writes the standard External Properties format used in previous versions.

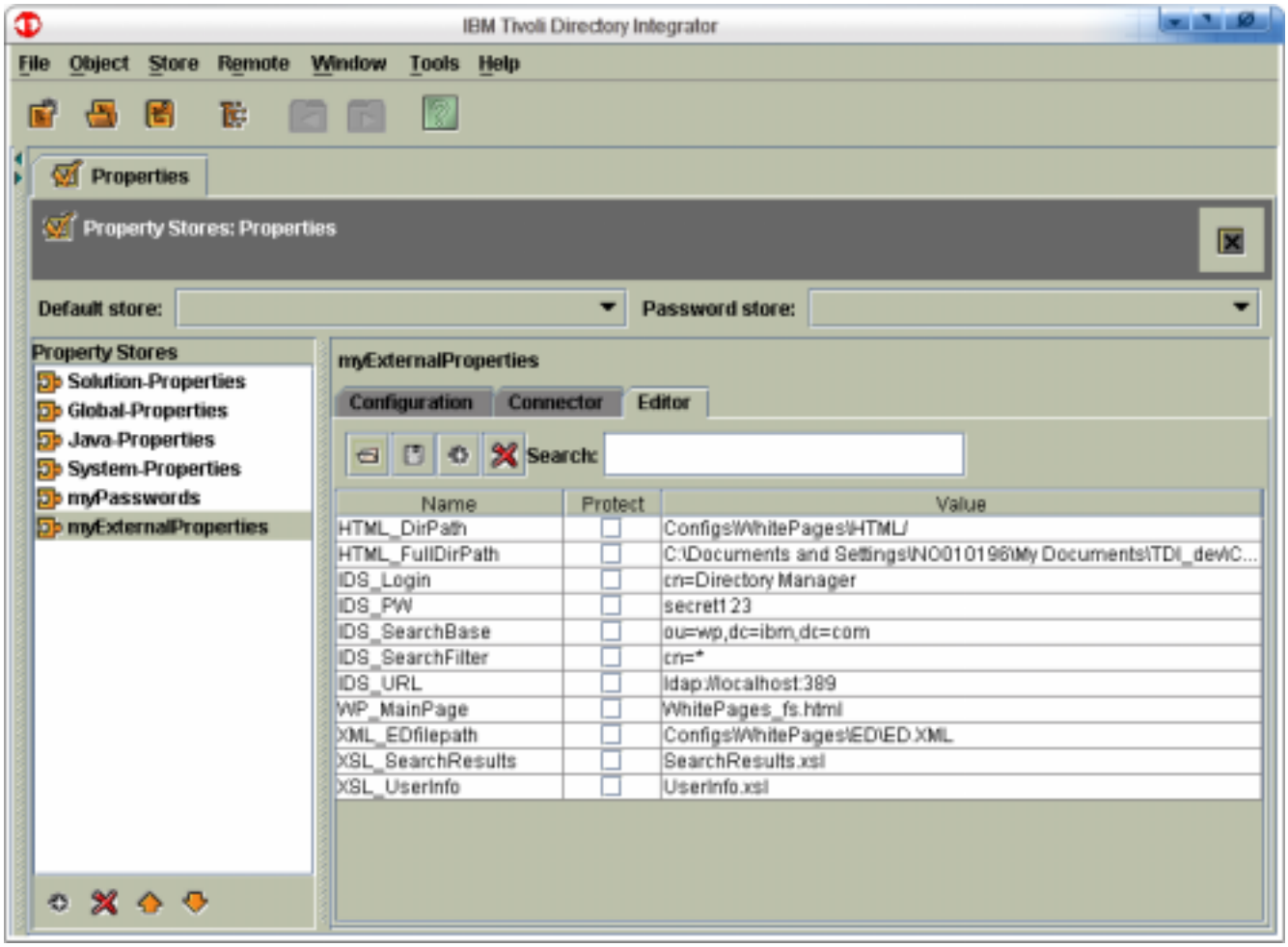
The other parameters shown in this tab are dependant on the type of Connector used. For the ibmdi.Properties Connector, you have the same options previously available for External Properties (Encryption, Cypher Algorithm, etc.), as well as a couple of new ones: "Create", which will cause this file to be created automatically when properties are accessed, and "Auto-rewrite" which is used in conjunction with properties tagged for encryption (the "protect" checkbox) in the **Editor** tab. Setting Auto-rewrite to true will cause TDI to re-write the file with protected values encrypted (prefixed in the file with the standard "{protect}-" phrase). Note that the ibmdi.Properties Connector is also available as a standard Connector for use in AssemblyLines, and that it writes a header like the following:

```
##{PropertiesConnector} savedBy=NO010196, saveDate=Tue Feb 14 14:55:38 CET 2006
```

As you can see, even user-defined Properties support the comments (lines that start with "#").

2.30.2.3 Editor tab – encrypting of individual properties

Here is where you actually make changes to a Property Store.



Above the property list is a toolbar with features for loading properties into the editor, commiting (saving) properties out to their associated data store (except for Java Properties), and to add or delete properties.

To change a property name or value, simply double-click on this value in the grid. The "Protect" checkbox tells TDI to encrypt this property (and the store is re-written if the Auto-Rewrite flag is turned on – see previous section).

Finally, the search bar gives you a free-text search through the list of properties. Each type you press ENTER after filling in a search string will cause the cursor to move to the next matching property. Note that this search is NOT case sensitive.

NOTE: Remember that you must press the Commit button to have TDI persist any changes you make here.

2.30.2.4 Encryption of Individual Properties

In addition to encrypting an entire Property Store, you can now also encrypt individual properties. This done either by preceding a text-file based property with the "{protect}-" keyword, or by selecting the "protect" checkbox.

2.30.3 Accessing Properties from JavaScript

In addition to the existing UserFunctions (the "system" object) functions like getExternalProperty() and getJavaProperty, there are new methods for working with the Properties framework that manages all types of Properties: `getTDIProperty()` and `setTDIProperty()`.

`Both functions come in two flavors, one which names on the property in question and another which allows you to specify a specific Property Store as well:`

```
var filePath = system.getTDIProperty("SAP.exportXML.filePath");
```

or

```
var deleteAccts = system.getTDIProperty("myExtProps", "disableAccountFlag");
```

`The first call above causes the Properties framework`

2.31 Expressions

When configuring an AssemblyLine, a component or similar settings, you are provided with input fields where parameters are configured. Here you can choose to enter/select a value directly, or by clicking on the label of the parameter, open up a Parameter Details dialog where you can choose a Property that will provide this parameter value (e.g. External Properties, as well as all the other Property Stores described in section *2.30 Property Store Framework* on page 62).

In addition to this simple substitution, TDI now offers an Expressions feature that allow you to compute parameters and other settings at run-time, making your solutions dynamically configurable.

This feature expands on the External Properties handling found in previous versions. In addition to support for simple External Properties references (fully backwards compatible), Expressions provide more power in manipulating AssemblyLine and component configuration settings during AL/component initialization and execution.

Expressions can also be used for Attribute maps, as well as for Conditions and Link Criteria, alleviating much of the scripting previously required to build dynamically configured solutions.

TDI provides an Expressions editor to facility building these expressions.

2.31.1 Overview of Expressions in TDI

The Expressions feature is built on top of the services provided by the standard Java **`java.text.MessageFormat`** class. The MessageFormat class provides powerful substitution and formatting capabilities. Here is a link to an online page outlining this class and its features:

http://java.sun.com/j2se/1.4.2/docs/api/java/text/MessageFormat.html

In addition to features described in the above class, TDI provides a number of run-time objects that can be used in expressions – although the availability of some objects will depend on run-time state (e.g. whether conn/current defined, or the error Entry): The Expressions syntax provides a *short-hand* notation for accessing the information in these objects, like Attributes in a named Entry object, or a specific parameter of a component.

TDIReference	Value	Availability
work.*attrname[.index]*	The *work* entry in the current AssemblyLine. The optional *index* refers to the *n*'th value of the attribute. Otherwise the first value is used. So this Advanced Attribute Map:	AssemblyLine

	ret.value = work.getString("givenName") + " " + work.getString("sn"); can be expressed simply as: {work.givenName work.sn}	
conn.*attrname[.index]*	The *conn* entry in the current AssemblyLine The optional *index* refers to the *n*'th value of the attribute. Otherwise the first value is used.	AssemblyLine during attribute mapping
current.*attrname[.index]*	The *current* entry in the current AssemblyLine The optional *index* refers to the *n*'th value of the attribute. Otherwise the first value is used.	AssemblyLine during attribute mapping for Modify
config.*param*	The configuration object of the "scoped" component/AL: Furthermore, if "config" is used in the parameter of a Connector, Parser or Function, then it refers to the config object of that component's *Interface* (e.g. JDBC Connector, or XML Parser) *param* is the name of the parameter itself, as if you were to make a call to getParam() or setParam(). For example, for the JDBC Connector you could make the following reference: {config.jdbcSource]	AssemblyLine EventHandler Connector Parser Function Component
alcomponent.*name.param*	The component Interface parameter value of a named AssemblyLine component. *name* is the name of the AssemblyLine component *param* is the parameter name of the *name* object So, the following Expression: {alcomponent.DB2conn.jdbcSource} is equivalent to the following scripted call: DB2conn.connector.getParam("jdbcSource");	AssemblyLine
property[:storename].name property[:storename/bidi].name	A TDI-Properties reference. The optional *storename* targets a specific Property Store. If no *storename* is specified, then the default store is used. *name* is the property name *bidi* will, when present, cause setting the parameter value to forward the call to the referenced Property Store. When *bidi* is present no other substitution patterns or text is allowed.	Always.
javascript<<*EOF*	*Embedded* script code used to generate a value for	Always.

script code … // Must contain "return" *EOF*	the Expression. This script must *return* a value. The "*EOF"* text shown is an arbitrary string[21] that terminates the javascript snippet. The javascript is collected up until a single line with the *EOF* string is encountered (or no EOF is flag is set – see the note below). Note that embedded JavaScript is evaluated using the AssemblyLine's script engine instance, so you have access to all variables otherwise present for scripting. *Note: also that there is a short-hand form of adding JavaScript that works for input fields that do not support multiple lines (like Link Criteria or the names of Attributes in maps) and can therefore not have the necessary EOF line:* {javascript return work.getString("givenName") + " " + work.getString("surName")}	

As mentioned in the table above, embedded JavaScript in Expressions have access to the AssemblyLine's script engine. As a result, even script variables defined elsewhere in the AssemblyLine can be accessed. Note that if you reference a variable or object that is not one of those specifically listed in the tables shown in this section, the Expression evaluator will check with the AL's script engine to see if it is defined there.

With that in mind, here are the explicitly available variables for Expressions used in component parameters, Link Criteria and Branches/Loops/Switch/Case components:

2.31.2 Expressions in component parameters

When used for a component parameter, the following objects are of special interest:

Object	Value
config	The component's Interface configuration object.
mc	The MetamergeConfig object of the config instance (config.getMetamergeConfig())
work	The work Entry of the AssemblyLine
task	The AssemblyLine object

As an example, let's say you have a JDBC Connector with the Table Name parameter set to "Accounts". You could then click on the **SQL Select** parameter label and then enter this into the Expression field at the bottom of the dialog:

```
select * from {config.jdbcTable}
```

This will take the Table Name parameter and create the following SQL Select statement:

```
select * from Accounts
```

Or you could get more advanced, and try something like this for the **SQL Select** parameter:

[21] You can use anything you like, but we should stick to the "EOF" convention to make our solutions easier to read and share.

```
SELECT {javascript<<EOF
      var str = new Array();
      str[0] = "A";
      str[1] = "B";
      return str.join(",");
EOF
} FROM {property:mystore.tablename} WHERE A = '{work.uniqueID}'
```

The embedded JavaScript will return the value "A,B" which is then used to complete the rest of the Expression. If you have a Property Store called "mystore" with a "tablename" property set to "Accounts", and there a "uniqueID" Attribute in the work Entry with the value "42", the final result will be:

SELECT A,B FROM Accounts WHERE A = '42'

Note that for a parameter to be based on an Expression, there must be a special preamble stored in the Config and which is not displayed onscreen in the CE: "@SUBSTITUTE". Simply entering curly braces will not cause Expression evaluation to be done for the parameter value. Instead you have three choices when tying Expressions to parameters:

1. Press Ctrl+E to open the Expressions Editor dialog while in the parameter input field,
2. Click on the Parameter label and enter the Expression directly into the field labeled "Expression" at the bottom of the Parameter Information Dialog.
3. Type the special preamble, @SUBSTITUTE, manually into the parameter input field, followed by the Expression. For example:

```
@SUBSTITUTEhttp://{property.myProperties:HTTP.Host}/
```

2.31.3 Expressions in LinkCriteria

Expressions in Link Criteria provide a similar list of pre-defined objects. Again, note that you also have access to any other objects or variables currently defined in the AssemblyLine's script engine.

Object	Value
config	The component's Interface configuration object.
mc	The MetamergeConfig object of the config instance (config.getMetamergeConfig())
work	The work Entry of the AssemblyLine
task	The AssemblyLine
alcomponent	The component itself, or a named component.

So, for example, let's say that you want to set up the Link Criteria for a Connector so that the Attribute to use in the match is determined at run-time. In addition to standard data Attributes in the work Entry, there is also a "matchAtt" Attribute with the string value "uid". In this case, the following Expressions used in Link Criteria:

{work.matchAtt} EQUALS {work.uid}

Is equivalent to this:

uid EQUALS $uid

2.31.4 Expressions in Branches, Loops and Switch/Case

The list of Expression objects here is similar to that for Link Criteria:

Object	Value
config	The component's Interface configuration object.
mc	The MetamergeConfig object of the config instance (config.getMetamergeConfig())
work	The work Entry of the AssemblyLine
task	The AssemblyLine
alcomponent	The Branch/Loop component

As in the example shown for Link Criteria above, you have can use Expressions for both the Attribute Name and the Operand of a Condition. You can also use Expressions to configure Switch and Case components.

2.31.5 Scripting with Expressions

You can also use Expression directly from JavaScript code. Here is an example that builds an expression using the new ParameterSubstitution class:

```
var ps = new com.ibm.di.util.ParameterSubstitution("{work.FullName} -> {work.uid}");

map = new java.util.HashMap();

map.put("mc", main.getMetamergeConfig());
map.put("work", work);

task.logmsg(ps.substitute(map));
```

Resulting in the following log messages when run for several iterations in my test AssemblyLine:

```
14:35:29  Patricia L Adowski -> adowski
14:35:29  Gerard C Agocha -> agocha
14:35:29  Jose M Agudelo -> jagudelo
14:35:29  Alfredo Aguirre -> aguirr
14:35:29  Shahid Ahmad -> sahmad
14:35:29  Kevin L Ahuna -> ahunakl
```

2.32 Java FC: Simplified browse/call Java objects and methods

This Function Component let's you open a .jar file, browse and select a method, then populates the schemas for Input and Output Maps with the required parameters.

To set this FC up, first select a .jar file by entering its path or using the Browse button to browse to it. Once the .jar is selected, you then press the Select button next to the second parameter (Java Class) to choose which class to use. Finally, once the class is selected, the drop-down in the third parameter line is populated with the methods offered by this class. Simply select the function that you want to invoke.

Now if you bring up the Output Map you will see that the schema is filled out with the input parameters required when calling the selected method. Furthermore, the Input Map schema contains the return value from the function.

At this point, you can easily drop your FC into an AssemblyLine in order to call this Java function as part of the flow.

2.33 General Enhancements to TCP-based components

2.33.1 SSL support enhancements

TCP-based components now all support SSL (where applicable).

2.33.2 TCP attributes available in all TCP based connectors

TCP-based components, like the HTTP Server Connector, now have a switch in their Config screens for returning TCP headers as Attribute values. When this flag is unchecked, TCP headers are stored as *properties* in the returned Entry object[22].

2.33.3 TCP Connection Backlog parameter

Server Mode Connectors based on TCP protocols have a new parameter called **Connection Backlog** that controls the queue length for incoming connections.

2.34 Secure Remote Command Line FC

The new Secure Remote Command Line FC (or RCL FC for short) enables command line system calls to be executed on remote machines using any of the following protocols: RSH, REXEC, SSH or Windows. The RCL FC uses the RXA toolkit to connect to remote machines, execute the commands and return the results. The returned output can then be parsed, to be consumed as one value at a time and detect any problems with the executed command.

2.35 TDI event components

(read/write, local/remote)

[22] Properties are accessed using the .getProperty() and .setProperty() methods of the Entry object.

2.36 DSML v2 enhancements

The DSML v2 library used is now that from ITDS, making TDI's use of this XML format compliant with the DSML v2 standard[23]. This means that in addition to support the DSML operations Search, Modify, Add, Delete, ModifyDN and Compare, both Auth and Extended operations are now also handled. Furthermore, the DSMLv2 Parser (described in the next setction) now supports the optional "requestID" DSMLv2 attribute and the optional "control" DSMLv2 elements.

2.36.1 DSMLv2 parser (delta tagging, auth/extended operations,)

In addition to support for standard DSML operations (described above), invalid XML characters (as per the XML specification) are now removed from the "dsml.error" Entry Attribute before serializing this attribute into DSML.

This Parser is initialized to parse or create a DSMLv2 batch request or response. When used with components like the DSML v2 SOAP Connectors, Individual calls to read or write Entries will result in parsing or creation of individual DSML requests or responses (as parts of the batch request or response).

Delta tagging of Entries return during DSML v2 Parsing has also been added, making the

SOAP support is also added to the DSMLv2 Parser, providing now the option of using DSML SOAP binding, in which case the Parser is able to process and create SOAP messages.

2.36.2 DSMLv2 SOAP Server and Client Connectors

The DSML v2 Connector has been replaced by two new components: a DSML v2 SOAP Server Connector, as well as a corresponding Client Connector.

The DSML v2 SOAP Connector give you access to a variety of DSML servers over HTTP. The DSML v2 SOAP Server Connector listens for DSMLv2 requests over HTTP. Once it receives the request, it parses the request and sends the parsed request to the AssemblyLine workflow for processing. The result is then sent back to the client over HTTP. Both Connector support SOAP DSML binding.

2.36.3 ITIM DSML EH (new parser library)

This component uses the new libraries, but remains backwards compatible with the 6.0 version of this component. Note that although EventHandlers have been deprecated, this component is still used by IBM Tivoli Identity Manager (TIM) to drive TDI ALs as end-point agents.

2.37 SMTP "send email" FC

This new component uses the javax.mail package to offer a simple method for connecting to SMTP servers and sending e-mails. It can send e-mails to multiple recipients and includes an option to attach multiple files with different MIME types as well.

The contents of the mail (body) must be passed to the FC via the Output Map. Setting other e-mail properties – like the *from*, *to* and *subject* settings, as well as attachments and the details of the SMTP server to use – can be done by setting parameters in the Config tab of the FC, as well as via Attributes in the Output Map of the FC. The output schema for this component is:

[23] Note that the new DSML v2 components are not compatible with the DSML dialect used to communicate with ITIM. The DSML v2 standard is detailed here: http://www.oasis-open.org/committees/dsml/docs/DSMLv2.doc.

OutputSchema

- **attachments**

A multivalued attribute. Each value specifies an attachment file to be added to the e-mail. Each value is either the absolute file path or a file path relative to the working (solution) directory. If the attribute is present it overrides the value of the *attachments* FC parameter.

In order to attach each file with different MIME type, users can provide the MIME type of attachment after the name of the file separated by '>'. For example:

```
SomeDocument.pdf>application/pdf
```

- **body**

String object that contains the body text of the mail. This Attribute is required in the Output Map, and an exception is thrown if it is not present.

- **from**

The attribute specifies the content of the *from* field in the mail. If the attribute is present it overrides the value of the *from* FC parameter.

- **recipients**

The attribute should be a comma separated list of the recipients of the mail. If the attribute is present it overrides the value of the *recipients* FC parameter.

- **smtpServerHost**

The attribute specifies the address of the SMTP server used to send the mails. If the attribute is present it overrides the value of the *smtpServerHost* FC parameter.

- **smtpServerPort**

The attribute specifies the port of the SMTP server used to send the mails. If the attribute is present it overrides the value of the *smtpServerPort* FC parameter.

- **subject**

The attribute specifies the subject of the mail. If the attribute is present it overrides the value of the *subject* FC parameter.

The FC also returns a single Attribute:

InputSchema

- **status**

This attribute is a **java.lang.String** object containing value "OK" if the mail has been sent successfully.

2.38 Common Base Event (CBE) Generator FC

The CBEGeneratorFC is used for creating CBE event objects. The FC allows you to select between either generating a CBE Java object, which can then be passed to a Comment Event Infrastructure (CEI) Server via the TEC web services, or as an XML document that adheres to the the Hyades CBE Logging format and that can then be used to write a CBE-style logfile.

Retrieving the CBE event object as an XML document is done via a scripted called to the getCBELogXml() function, which is provided by the CBEGeneratorFC. For example, this could can be used in a script that is called *after* the CBE event object has been generated:

```
var cbe = work.getObject("event");
xmlString = com.ibm.di.fc.cbe.CBEGeneratorFC.getCBELogXml(cbe,false);
```

Since this is a *static* method, then you can make the above call directly to the class. You could also use the FC in your AL to make the call. Let's say, your CBEGeneratorFC is called "MakeCBE", then the code above could look like this:

```
var cbe = work.getObject("event");
xmlString = MakeCBE.function.getCBELogXml(cbe,false);
```

The resulting XML string can be stored back in the work Entry using the work.setAttribute() method.

2.39 Web Services enhancements

The web services components leverage AL Operations to define services and their parameters. Based on the Operations set, as well as the Input and Output Attributes defined for each of these, the

2.40 JDBC Connector enhancements

The JDBC Connector has be improved in a number of different ways. In addition to offering a configuration parameter for setting the SELECT statement (used for Iterator and Lookup mode operation), you can also define the INSERT, UPDATE and DELETE statements to be used.

Furthermore, the JDBC Connector now supports Delta mode (previously only available for the LDAP Connector), allowing you to simplify synchronization ALs.

Also, there is a new "Extra Provider Parameters" to allow you to pass system-specific commands for setting up the connection to the database. The Extra provider parameters in the config tab is a text area where we can enter the name:value pairs. Each name:value pairs MUST be on a separate line in the text area. The name and value MUST also be separated by a colon, e.g:

readOnly:true

securityMechanism:KERBEROS_SECURITY

Finally, the JDBC Connector normally tries to use prepared statements in an effort to optimize database performance. However, if this fails, TDI will now fall-back to sending a normal statement. You can instruct the Connector to *never* use prepared statements by unselecting the new "Use Prepared Statement" checkbox.

2.41 JMS Connector supports other (& custom) JMS providers

The JMS Connector now works with other message queues, in addition to the already supported IBM MQ. Although support for IBM MQ is included, you can now plug in other message queues by supplying your own JMS initiator class.

2.42 HTTP Server Connector enhancements

There is a new parameter called **Connection Backlog** that controls the queue length for incoming connections, as well as a **TCP Data as Properties** flag.

Another checkbox has been added to control **Chuncked Transfer Encoding**, which is handy when working with large HTTP messages (e.g. a big **http.body**).

In addition, there are now settings to provide basic HTTP authentication (login dialog), including binding with an LDAP Server Connector

Furthermore, there is a new parameter called **Connection Backlog** that controls the queue length for incoming connections.

2.43 AssemblyLine Connector and FC

Both the AL Connector and the AL FC have been enhanced to support AL Operations. For more information on the both these components can be found on page 51.

2.44 Harmonized Change Detection Handling

The Change Detection Connectors[24] have been reworked to make them behave in the same way, as well as provide the same parameter labels for common settings. These Connectors are:

- IBM Directory Server Changelog (TDS)
- AD Changelog v2 (Active Directory)
- Domino Change Detection
- Netscape Changelog (openLDAP, SunOne, iPlanet, etc.)
- RDBMSChangelog (DB2, Oracle, SQL Server, and so forth)
- zOS Changelog

This harmonization[25] is described in more detail below.

2.44.1 Delta Operation Code tagging

All Change Detection Connector set the delta operation code of the Work Entry. In addition, the TDS and Netscape Changelog Connectors also tag Attributes and their values.

2.44.2 Parameter Harmonization

Each Connector now provides a similar set of parameters to control where change processing is to begin. In addition, they all leverage the System Store to keep track of the Iterator's *cursor* in the set of changes, allowing you to easily build synchronization AssemblyLines that resume processing correctly on start-up. Furthermore, their polling intervals and timeout settings are controlled by similar parameters.

The Config tab of each Change Detection Connector boasts a **Start at…** parameter that can be set to "0" (which means *start at the beginning*), to "EOD" (End of Data, which will cause the Connector to return only new changes). Some Connectors permit you to set this value to a specific changelog number, or timestamp.

[24] The LDIF Parser is also used to read in entries with delta operation codes set (incremental LDIF), but is not part of the harmonization described here.

[25] Part of the harmonization process has been the conversion of all Changelog EventHandlers to Connectors that operate in Iterator mode and perform the same function.

The above **Start at…** value is typically used to begin initial processing of changes. However, in order to have the Connector keep track of the last change handled, and to pick up after this one on its next run, you have two additional settings: **Iterator State Key** and **State Key Persistence**.

The **Iterator State Key** must be set to a text value (id) that is unique for this Config, as well as all others that share the same System Store. This value is used as a key for storing the state of iteration for this Change Detection Connector. And it is retrieved from the System Store in order to prepare the Connector for resuming synchronization.

The second parameter state-related parameter, **State Key Persistence**, controls *when* the current Iterator State is saved to the System Store. The recommended setting is **End of cycle**.

2.44.3 Exchange Changelog Connector

The Exchange Changelog Connector connects to Microsoft Exchange Server 5.5. Microsoft Exchange is usually used with Windows NT4, while users of Windows 2000 Server and above would normally use Active Directory. Furthermore, Microsoft Exchange Server 2000 and later releases use Active Directory. As a result, the Exchange Changelog Connector is only applicable in Windows NT4 environment, and will therefore be deprecated since Microsoft dropped extended support for Exchange 5.5 at the end of 2005.

2.44.4 zOS Changelog Connector

This is a replacement for the zOS Changelog EventHandler.

2.44.5 JNDI-based Connectors

All the JNDI-based Connectors, including the Changelog Connectors for IBM Directory Server, Netscape/iPlanet and ActiveDirectory now offer settings for *extra provider parameters*. These are used to control connection settings (like timeout values, or controlling connection pools)

2.44.6 AD Changelog components

Both the Active Directory Changelog Connector (v.1) and Active Directory Changelog EventHandler are deprecated in TDI 6.1 Express, replaced by the Active Directory Changelog (v.2) Connector.

2.45 FTP Connector

This component now supports IPV6.

2.46 New AMC (Admin and Monitor Console)

AMC v3 is an entirely new implementation that is built to run on WAS Express (bundled), WAS, and Tomcat, and lets you view multiple solutions and server in a single status screen. You can also configure custom monitoring of anything you can access and check via a TDI AssemblyLine by simply designating such flow as being a "Health AL".

But just as exciting, is the replacement for the old AMC v2 FOS feature (Fail Over Service) which is called *Action Manager*, and which gives you the tools to build a failure-response framework around your TDI solutions.

- Entirely new implementation
- Single view monitors multiple solutions
- Custom "health check" displays results on monitor screen
- Action Manager triggers displayed as icons
- Authentication service provides differentiated access
- Drill down to details with alerts and messages from each solution

2.46.1 AMC Action Manager

Action Manager (AM) provides all the functionality offered by AMC v2 FOS and much more, giving you the flexibility to define what "failure" means (AL stops, or hasn't been run in xxx hours, or free disk < nnnn, or network link down, or...) as well as how to respond. AM ties into AMC v3 allowing you to view AM logs and modify AM behavior.

- **Replaces "FOS" in TDI version 6.0**
- **Drives content into AMC monitor**
- **Consists of *Rules* that**
  - **monitors TDI solutions**
  - **have a single Trigger and any number of Actions**

Trigger

- **Time**
- **Failed API**
- **AL stopped**
- **TDI Event**
- **Query AL**
- **On property value**

Actions

- **Run AL, stop AL (which one?)**
- **Disable and enable Rule**
- **Trigger Rule**
- **Send TDI event**
- **set/copy/inc/dec properties**
- **Log to AMC logger**

2.47 Inheritance of SC's and Scripted components

For example, let's say that you've scripted a Parser and put it in your Library. Then you set up a FileSystem Connector and attach the Parser to it. Now when you click on the Config -> Parser tab of the Connector, you see the *inherited* JavaScript from your Library Parser.

Above the text editor panel is the label: *Script*. When this text is displayed in *italics* then it means that the script code in the editor below is inherited. If the label is written in normal style it signals that the code is a local copy.

As with all Connector settings (e.g. Input/Output Maps, Link Criteria, Config…), Inheritance is controlled by the **Inherit From** button at the bottom right-hand corner of the tab. Note that even though inheritance may be specified, you may still be working on a local copy of this code. To reinstate broken inheritance, simply press the Script label above the text editor. This brings up the Parameter Information dialog, where you can press the **Use original value** button to restore the link.

If you try to change an inherited script, you are asked if you wish to break inheritance. If you answer no, nothing happens (the script is not changed). If you accept the break, then you will be working on a local copy of this script. Your Library Parser will remain unchanged. In this scenario, the label *Script* would cease to be in *italics* once inheritance was broken.

2.48 Java 1.5

TDI version 6.1 now includes IBM Java version 1.5.

2.49 AssemblyLine Debugger/Stepper

The AL Debugger (the little bug icon next to the Run button) has been revamped into a full-blown AssemblyLine Stepper. Cool new features include:

- Static, temporary and conditional breakpoints
- Watch lists for displaying data (Attributes, variables, JavaScript snippets, Expressions[26])
- The ability to interactively execute script code during AL operation.
- Support for Server Mode Connectors.
- Step Next Component, Step Next Possible Breakpoint (Step Into) and Continue options

But don't let yourself get spooked by these technical details, or the term "Debugger"; it all simply means that building, maintaining and even understanding your own AssemblyLines just got a lot easier. This is definitely a feature that you will want to try out asap. Just grab an existing AssemblyLine (completed or in-progress) and press the "Run Debugger" button (or right-click and select "Debug", or just press Alt+D) to start taking advantage of this powerful tool.

2.49.1 AL stepper main window

When you start your AL in Debugger, TDI performs the following actions: First, the CE starts up a separate instance of the TDI Run-time Server (complete with its own JVM). It then connects to this Server via the API and pipes over

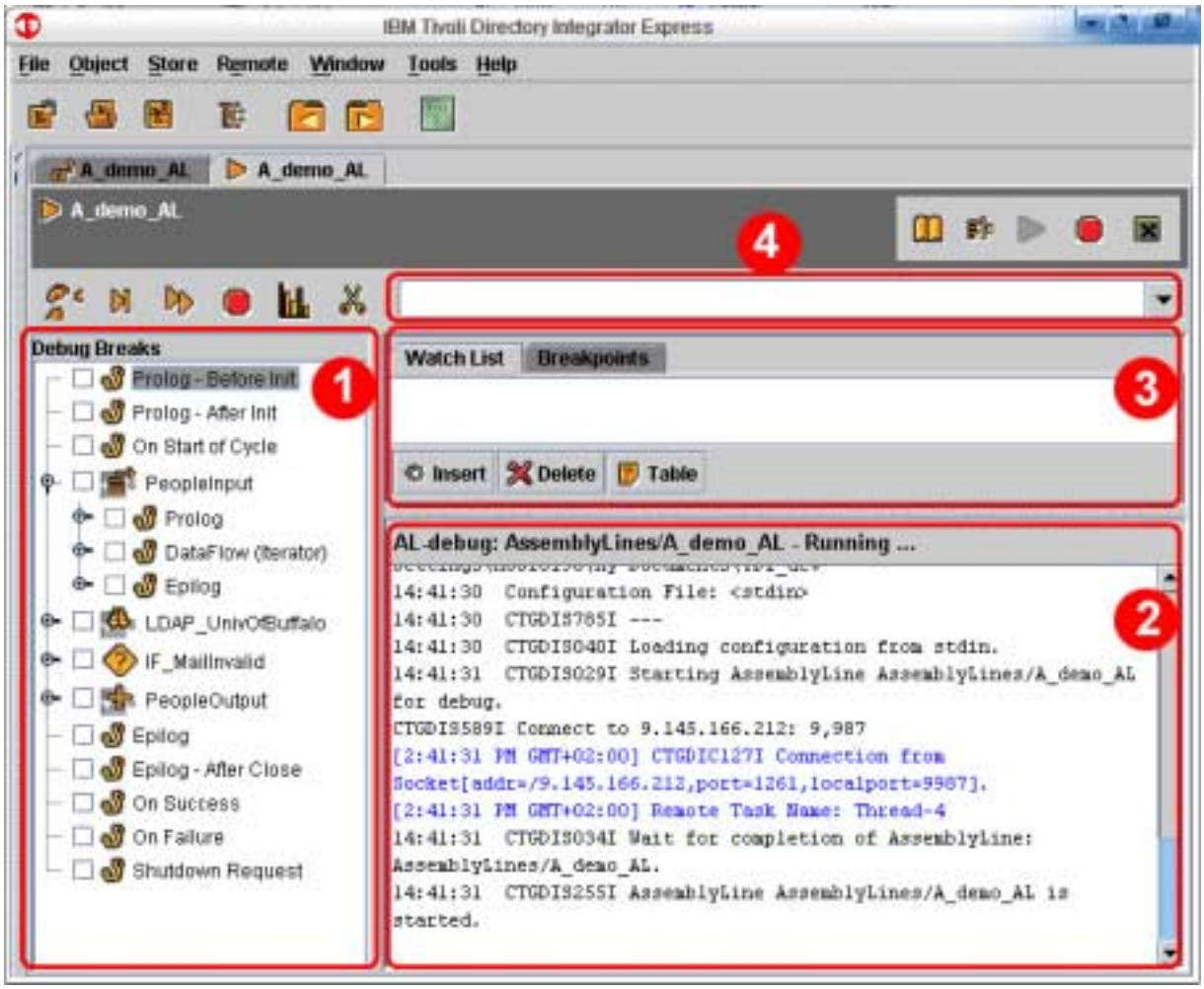
[26] Be sure and read about the powerful new Expressions feature described in section *2.31 Expressions* on page 69.

the Config[27]. Furthermore, it instructs the Server to execute this single AL and to shutdown upon completion; and to send all log output to the console so that it can be captured and displayed on screen.

As you probably noticed, this is exactly what happens when you press the "Run" button. The big difference for Debugging/Stepping is that the Server is told to run the AL in *interactive* mode, instead of the normal *run-to-completion* mode.

Interactive mode lets you step through the execution of your AL, viewing and modifying any data (even script variables) handled by your AssemblyLine. It allows you to set *Breakpoints* which cause AL processing to stop at the places where you have set them.

In addition, the CE presents you with a screen that is divided onto four major areas:



1	The AL Hook List. This tree-view includes the Hooks for the AL as well as any components. The checkbox next to each Hook allows you to set or clear a Breakpoint at the start of the Hook, meaning that execution will stop here. Note that you can also right-click on any Hook and select "Run and break here" in order to run the AL to this point.

[27] Note that this is the Config in memory that contains the AL to run. You still need to press the Save button (or Ctrl+S) to ensure that your Config has been persisted to disk. Note also that if you are currently editing Property values (e.g. Ext Props) then you will need to save these, or close the Property window, in order for the Run-time Server to have access to any modifications.

2	This area is the Log Output window, and shows the log output (just like for the standard Run window), but also includes *debug message* in blue text. Debug messages include status info sent by the Server, as well as the output from debug commands like task.debugBreak() or task.debugMsg(), both of which have a single parameter: the object or message to output[28]..
3	The Watch List provides a view of Attributes and script snippets that you want evaluated and displayed for each step of your AL. There is a "Table" button to get this list in tabular form where you also see the previous value of each item in addition to the current one. There is "Breakpoints" tab in this area as well, giving you an overview of all Breakpoints set, and allowing you to enter a JavaScript snippet next to any Breakpoint that must evaluate to *true* before execute stops at this stop. This is what is called a *conditional Breakpoint*, since it is only active based on the return value of the condition set.
4	This is the Evaluate input field. Whatever you enter here is immediately evaluated as JavaScript and the results shown in the Output window. So if you type "work" here followed by the ENTER key, the contents of work are displayed in the Log Output window as debug output. If you instead enter "task.dumpEntry(work)" then since this method returns null you will see this as debug output and the Entry dump will appear just after it as regular log output. But the real power here is the ability to call Java and JavaScript methods to drive components and manipulate data interactively: work.setAttribute("Hello", "World");

At the top of the AL Hook List are three Step mode buttons with the following meanings, from left-to-right:

Step Into	This button moves execution from one Hook to the next Hook (or scripted Breakpoint), and lets you see exactly how the logic of your AL is flowing.
Step Next	Moves processing to the top of the next component in the AL.
Continue	Causes AL execution to resume until the next active Breakpoint is hit. Note that if no Breakpoints are set (or their conditions do not evaluate to *true*), then the AL will continue until completion.

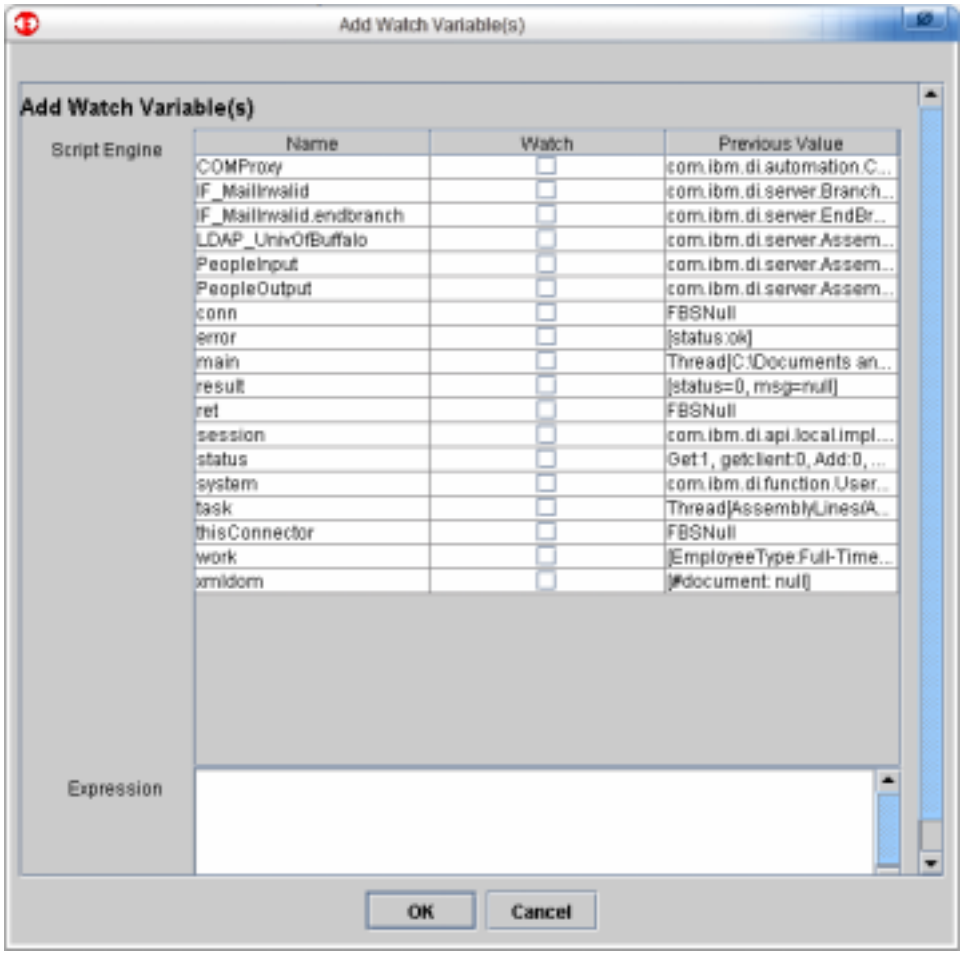
After these Stepping modes are buttons for stopping the AL, displaying statistics and clearing the Log output buffer.

Note that none of these buttons are active until your AssemblyLine pauses for interactive input. This happens when it first starts up, when you use either the Step Input or Step Next buttons, or if the AL processing hits an active Breakpoint.

[28] Any object passed to these methods must be serializable to string, like an Entry or Attribute is.

2.49.2 Inserting an Expression or object into watch list

Pressing the Insert button under the Watch List brings up the Add Watch Variable(s) dialog.



Here you can select objects to Watch by check the box for that item (the "Watch" column) or by writing a TDI Expression into the Expression editor window at the bottom of the dialog.

As mentioned above, you can view your Watch variable either as a list, or in tabular format by using the "Table" button at the right of this button row. In Table view, TDI displays both the current data content of any item, as well as its previous value.

NOTE: Debugger settings like Watch items and Breakpoints are lost when you close the Debugger window. There it is advised to simply Stop the AL, make changes to the source AssemblyLine and then re-start the Debugger. That way, preferences can be reused in successive debugging sessions.

2.50 Password change plug-ins

MQe can now secure communications through certificates. In addition, there is now a password change plug-in for Unix PAM.

2.51 "Express" readiness

TDI has now been Express certified, raising the bar for ease of use and installation, and making the toolkit more broadly available to our SMB customers and partners.

2.52 System Store

The DDL (SQL statements for creating tables and indexes) necessary to configure TDI to leverage Microsoft SQL Server and Oracle as its System Store are now provided.

2.53 Documentation

As part of the ongoing process of improving TDI documentation, a number of guides and video tutorials have been created (which can be found here: http://www.tdi-users.org/twiki/bin/view/Integrator/LearningTDI).

Furthermore, the manuals have been updated as well as the JavaDocs, and a number of new documents have been prepared:

- Developing Components
- Change Propagation in TDI (data sync.)
- Error Handling
- Using Loops and Branches
- Server API Developer Guide

2.54 "Response" section removed from AL flow

The *Response* section is removed from the AssemblyLine component list, leaving only Feeds and Flow. Instead, the **Output Map** and **Response Hooks** are now part of the Server Mode Connector itself onscreen. The execution of *response* behavior is still done after Flow section processing is finished.

Note that `system.skipEntry()` will continue to skip *response* behavior, and as such, no reply will be made by a Server Mode Connector. If you would prefer to simply skip the remaining Flow section components and yet send the response, use the `system.exitBranch("Flow"); call instead.`

2.55 TDI can be started as more than one Windows service

The TDI Windows service wrapper has been enhanced so that you can start TDI as multiple service instances.

2.56 Iterators can be used in flow

You can now put a Connector in Iterator Mode into the *Flow* Section. As such, the Iterator will work in the same way as it would in the *Feeds*: it is initialized (including building its result set with the selectEntries call) during AL startup

and will retrieve one Entry (getNextEntry) on each cycle of the AL. However, an Iterator in the *Flow* section will not drive the AL itself, as it would do in *Feeds*.

2.57 Improved Tooltips for AssemblyLine Components

When you let your mouse hover over a component in an AssemblyLine, you now get informational tooltips up that tell you interesting things like which Hooks are enabled for a Connector or FC.

2.58 Invoke custom methods/objects through the Server API

Users sometimes need to implement their own functionality and be able to access it from the Server API - both local and remote. This is currently supported by the Server API, but it needs to be simplified so that you can drop a JAR file of your own in the TDI classpath and then access it from the Remote Server API without having to deal with RMI.

Two methods are added to the following classes and interfaces:

- com.ibm.di.api.remote.Session
- com.ibm.di.api.remote.impl.SessionImpl
- com.ibm.di.api.local. Session
- com.ibm.di.api.local.impl.SessionImpl

The two methods are:

public Object invokeCustom(String aCustomClassName, String aMethodName, Object[] aParams) throws DIException;

public Object invokeCustom(String aCustomClassName, String aMethodName, Object[] aParamsValue, String[] aParamsClass) throws DIException;

Note: The methods in com.ibm.di.api.remote.Session and com.ibm.di.api.remote.impl.SessionImpl are also declared to throw RemoteException due to the remote session.

Both methods invoke a custom method described by its class name, method name and method parameters.

These methods can invoke only static methods of the custom class. This is not a limitation, because the static method of the custom class can instantiate an object of the custom class and then call instance methods of of the custom class.

The main difference between the two methods is that the invokeCustom(String, String, Object[], String[]) method requires the type of the parameters to be explicitly set (in the paramsClass String array) when invoking the method. This helps when the user wants to invoke a custom method from a custom class, but also wants to invoke this method with a null parameter value. Since the parameter's value is null its type can not be determined and so the desired method to be called cannot be determined. If the user wants to invoke a custom method with a null value he/she must use the invokeCustom(String, String, Object[], String[]) method, where the desired method is determined by the elements of the String array which represents the types and the exact order of the method parameters. If the user uses invokeCustom(String, String, Object[]) and in the object array put a value which is null than an Exception is thrown.

2.58.1 Primitive types handling

These methods do not support the invocation of a method with primitive type of parameter(s). All primitive types in Java have a wrapper class which could be used instead of the primitive type.

2.58.2 Custom methods with no parameters

If the user needs to invoke a method which has no parameters he/she must set the paramsValue object array to null (and the paramsClass String array if the other method is used).

2.58.3 Errors

Several exceptions may occur when using these methods (please see 3.5 Error Flows section). Both local and remote sessions support these two methods, but the Server API JMX does not.

2.58.4 Security

API access to custom methods/objects requires Admin-level authorization for the connection to the TDI Server.

2.59 Support Unicode through ICU4J

The ICU4J library is used to provide IBM standard support for Unicode.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Z