Handling Exceptions/Errors with TDI

by Eddie Hartman

1 How Handle Errors in TDI Solutions

This document outlines the features and techniques used to deal with exceptions (and errors) in your TDI solutions. As you will see, although all errors are exceptions, not all exceptions can really be called errors. Confused? Then keep reading.

We'll begin with a look at how error messages are written and then move to a discussion of the error object itself, along with the mechanisms in TDI that are activated when one occurs. Examples of error logs are provided in order to help you navigate these. There is also a section on how to control logging in your solutions, including tips for preparing your AssemblyLines to deal with problem situations.

Note that the abbreviation "AL" is often used for "AssemblyLine" in the following text. The same applies to "CE", which is short for "Config Editor" – the TDI User Development Environment. This is common practice and will be the case in newsgroup postings, published solutions and other TDI literature.

In order to follow the content here you should already have a some experience with TDI; At the very least having completed the Getting Started tutorial [1].

2 Reading the error "dump"

The natural place to start this document is with a discussion of how to locate *where* the error is happening. To do this you must be able to interpret the details that TDI writes about the error to the log[1]. Although not an exhaustive list of possible errors, this chapter includes examples of some common types, as well as tips on how to deal with them.

When some problem causes your AssemblyLine to fail, TDI writes error information to the log. Here is an example of this:

```
14:29:36  [DB2e_GroupTable] Lookup
java.lang.Exception: No criteria can be built from input (no link
criteria specified)
   at com.ibm.di.server.SearchCriteria.buildCriteria(Unknown Source)
   at com.ibm.di.server.AssemblyLineComponent.lookup(Unknown Source)
   at com.ibm.di.server.AssemblyLine.msExecuteNextConnector(Unknown
Source)
   at com.ibm.di.server.AssemblyLine.executeMainStep(Unknown Source)
   at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
   at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
   at com.ibm.di.server.AssemblyLine.executeAL(Unknown Source)
   at com.ibm.di.server.AssemblyLine.run(Unknown Source)
14:29:36  Error in: NextConnectorOperation: java.lang.Exception: No
criteria can be built from input (no link criteria specified)
java.lang.Exception: No criteria can be built from input (no link
criteria specified)
   at com.ibm.di.server.SearchCriteria.buildCriteria(Unknown Source)
   at com.ibm.di.server.AssemblyLineComponent.lookup(Unknown Source)
   at com.ibm.di.server.AssemblyLine.msExecuteNextConnector(Unknown
Source)
   at com.ibm.di.server.AssemblyLine.executeMainStep(Unknown Source)
   at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
   at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
   at com.ibm.di.server.AssemblyLine.executeAL(Unknown Source)
   at com.ibm.di.server.AssemblyLine.run(Unknown Source)
```

The trick to reading this message is moving your attention to the top of the error output.

Although the *stack trace* that makes up most of the dump helps developers locate problems in TDI itself[2], details on where the problem originated are written first.

```
14:29:36  [DB2e_GroupTable] Lookup
java.lang.Exception: No criteria can be built from input (no link
criteria specified)
```

The first item shown is the name of the component where the error originated. This is written inside brackets (`[DB2e_GroupTable]` in the above example). After the component name comes the operation that failed (`Lookup`). The next line has both the internal *type* of the error, also called the *exception class* (`java.lang.Exception`), and the error message (`No criteria can be built from input (no link criteria specified)`).

Putting all this together: TDI is complaining that it cannot build Link Criteria for the Lookup operation in the DB2e_GroupTable Connector. This could be because no Link Criteria was

[1] When you run your AL from the Config Editor, the CE instructs the TDI Server to send log output to the *console* so that the CE can intercept these messages and display them onscreen for you. See section *5 Logging* on page 15 for details on how you can configure logging yourself.

[2] This is why you should always include the log output when reporting problems to support.

defined, or because you are referencing a Work Entry Attribute that was not found in the Work Entry when Link Criteria was built. Armed with this information, you can first check that you have set up the Link Criteria correctly. If this is the case then your next step would be to make sure that the Work Entry actually holds the Attributes you are referencing in the Value parameter of each Link Criteria. These will be prefixed with the special dollar symbol ($). One way to ensure their presence is to dump out the Work Entry to the log:

```
task.dumpEntry( work );
```

You should put this code in the Lookup On Error Hook of this Connector so that if the error happens, you will be able to see if the required Attributes are in place or not.

Our example AssemblyLine has more than one problem, and after correcting this first error we can see our next problem:

```
14:39:25  [DB2e_GroupTable] while mapping attribute "member"
undefined: undefined is not a function.
    ...
```

This time the error log looks a little different. You still get the component name and operation, which this time is during Advanced Mapping of the mail Attribute. But now the *error type* is written as "undefined". This is often the case for errors reported by the JavaScript engine itself. The message is clear enough though – `undefined is not a function` – and tells me that I've misspelled the name of a function call in my script code. After close examination I discover the typo: task.log**mgs**() instead of task.log**msg**().

Bugs in JavaScript code can be among the most difficult to track down. Some script errors cause your AL to crash during initialization (e.g. syntax errors), while others remain undetected until a block of code is executed for the first time. The above task.logmgs() example is an example of the latter, since the problem first arises when the JavaScript engine tries to call a method in the `task` object called "`logmgs()`".

One approach is to get someone else to look at your code. Unfortunately, this is not always an option. Alternatively, place several (correctly spelled) task.logmsg() calls throughout the problematic script. By seeing which messages get written before the crash, you are able to pinpoint the problem.

If the failing script is long and filled with if-tests, you can also try breaking it down to smaller snippets. This is generally good programming practice anyway, and allows you to perform simpler unit tests on each smaller block of code. Script logic that is frequently reused should be defined as functions in Scripts Components. Placing these in the Script Library will help minimize coding errors and make your solution easier to maintain and enhance.

Another method for isolating script errors is commenting out code until you find the snippet that is causing the problem.

Now let's look at a data source error:

```
14:48:25  [TDS_Update] AddOnly
javax.naming.directory.SchemaViolationException: [LDAP: error code 65 -
Object Class Violation]; remaining name 'uid=ehartman33,o=ibm,c=com'
```

The first line tells us that the AddOnly operation failed for the TDS_Update Connector. This time the class of the exception is system-specific and is coming from the underlying directory or driver/API class:

```
    javax.naming.directory.SchemaViolationException
```

The message tells us that the Entry we are trying to write is in violation of schema. You will need to check that your Output Map Attributes are correctly spelled (and are actually defined in the object class of the entry). If all else fails, try disabling Attributes until you find out which one(s) is failing.

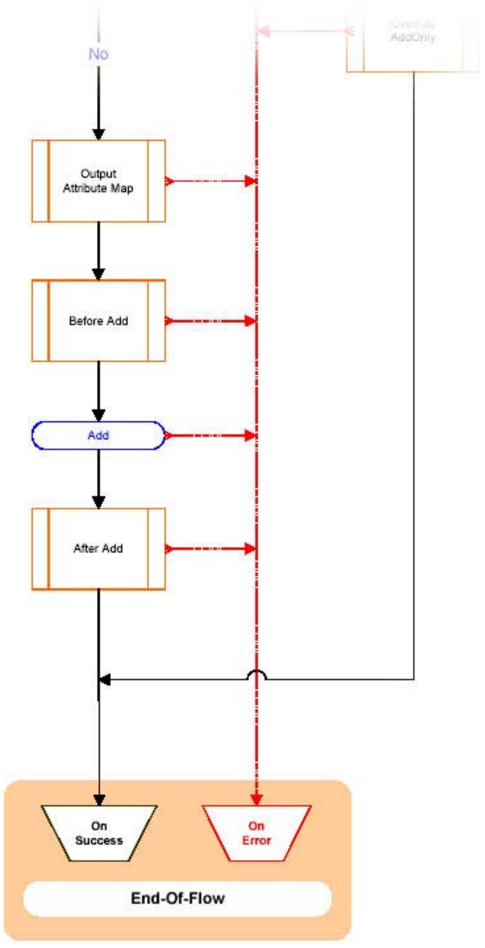
Now that we've looked at some example error logs and tips for correcting these, we will turn our attention to the error mechanism itself.

3 Errors = Exceptions

All errors in TDI are represented by Java objects called *exceptions*. Each type of error – both those internal to TDI itself, as well as errors coming from connected systems – has its own specific exception type.

When an error occurs, the corresponding exception is *thrown*, disrupting normal program execution[3]. The exception is *caught* by TDI and the Error flow is initiated, as described in the TDI *AssemblyLine and Connector mode flowcharts* (or Flow Diagrams for short) [2]. Although the next chapter covers Error Hooks in more detail, we will need to look at these briefly in order to understand exception handling in TDI.

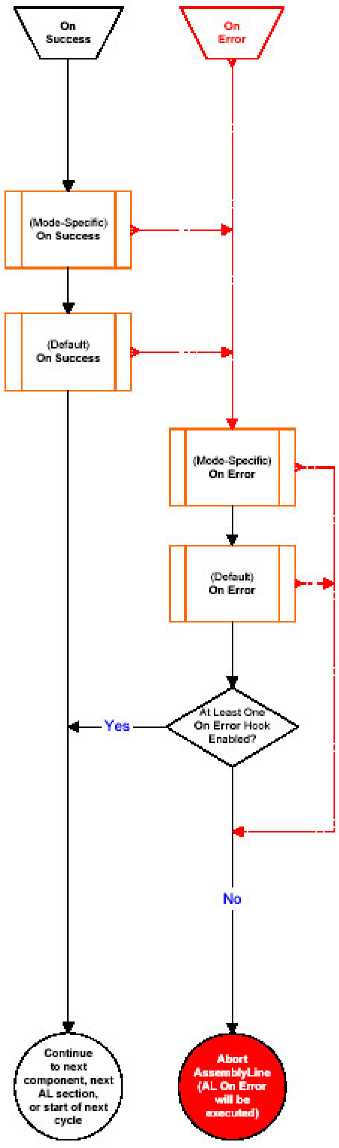
For example, let's take a look at part of the Flow Diagram for the AddOnly Connector mode:



As you can see, regardless of whether the error originates in a Hook, Attribute Map or internal Connector Interface operation (like *Add* in the above diagram), the Error Flow is initiated (drawn as a red dotted line in the diagram).

[3] Not all exceptions are errors. Some are used to signal changes in standard AssemblyLine processing. For example, methods like `system.skipEntry()` and `system.ignoreEntry()` throw special flow-control exceptions that do not initiate the Error Flow. Instead, these are caught by the AssemblyLine and handled as dictated by the exception type. Other exceptions are used to represent special situations like finding no match during a lookup operation, or finding multiple.

All Connector modes terminate in the same way, indicated by the orange box at the bottom of above diagram. Details of this behavior are described in the Flow Diagram page called *End-Of-Flow*.


On Success
On Error
(Mode-Specific) On Success
(Default) On Success
(Mode-Specific) On Error
(Default) On Error
At Least One On Error Hook Enabled?
Yes
No
Continue to next component, next AL section, or start of next cycle
Abort AssemblyLine (AL On Error will be executed)

In the case of successful operation, control is first passed to the Mode-specific On Success Hook (e.g. AddOnly Success for AddOnly mode) and then to the Default Success Hook. This latter Hook is shared by all modes.

Error flow has similar behavior: control goes first to the Mode-specific Error Hook and then to Default On Error. Just below the Default On Error Hook in the above diagram is a branch with the text: *At Least One Error Hook Enabled?* Notice how if the answer is *No* (i.e. no Error Hooks are enabled) then the AssemblyLine aborts with the error. On the other hand, if at least one of them *is* enabled then execution continues as if no error had occurred[4]. This is because TDI assumes that the problem has been dealt with by you.

However, this may not be desired behavior. For instance, if you are creating an ITIM Endpoint Adaptor using the DSML v2. EventHandler and have script code in Error Hooks

[4] Note that the error is still counted, and execution will halt if this count exceeds the Max. number of errors parameter setting in the Config tab of the AssemblyLine.

(e.g. writing to logs), then your AL will not report back any error to ITIM. This EventHandler expects the AssemblyLine to fail in the case of an error. So when the AL completes normally, the EventHandler reports back to ITIM that the operation was successful.

In order to "escalate" the error (instead of "swallowing" it) you have to *re-throw* the exception. To do this, you must first have access to the exception object itself. TDI provides you with this through the *error* object.

3.1 The error Object

The error object is an *Entry* – just like work and conn – and is available throughout the life of an AssemblyLine through the pre-registered script variable **`error`**. This *Java bucket* contains a number of Attributes that hold the exception itself, the error message associated with the exception, as well as details on where the error occurred. Here is a description of the various `error` Attributes.

Table 1 - Attributes of the `error` Object

status	The error status. The value of this Attribute is initially set to "ok". As soon as an error occurrs, it will be set to "fail". Furthermore, status is the only Attribute in the **error** object before the first exception is thrown. After an error is encountered, the other Attributes below are added.
connectorname	Name of the component where the error occurred. This is the name you have given this component in your AssemblyLine.
operation	The internal name of the operation that failed. For example, "get" for the getNext() operation of an Iterator. Note that this may not be the exact origin of the error, but rather the last operation performed. So, if you get an error in the Input Map of your Iterator, this Attribute will still have the value "get".
exception	This is the exception object itself.
message	Clear text message describing the error.
class	The Java class of the exception. This is typically a good place to start when trying to determine the type of error that has occurred.

So to re-throw an exception, you must use the value of the exception Attribute in the `error` Entry. Retrieving an Attribute value as an object is easily done using the Entry's `getObject()` method:

```
throw error.getObject( "exception" );
```

Note that throwing an exception from your Error Hook script breaks normal Error flow. If you throw an exception from the Mode-specific On Error Hook of a Connector then the flow will escalate to AssemblyLine error handling (e.g. the AL On Failure Hook) instead of continuing to the Connector's Default On Error Hook.

3.2 Exception Handling in Script

There are times when you want to deal with errors in your own script. For example, if you are calling Connector Interface methods manually, or other Java functions that can result in exceptions.

This is done using JavaScript's own exception handling feature: `try-catch`. The `try-catch` statement allows you to specify a snippet of code that you want to *try*. If an exception is thrown during script execution, you have also specified additional code to *catch* and handle the error. For example:

```
try {
   DB2e_Update.connector.putEntry( work );
} catch ( myException ) {
   task.logmsg( "** Error during DB2e add operation" );
   task.logmsg( "** " + myException );
}
```

The `try`-block in the above snippet is calling the `putEntry()` method of the DB2e_Update Connector's Interface. If this results in an error then the `catch` block is executed. The exception object itself is referenced using the variable name in parenthesis after the `catch` keyword (`myException` in the above example). This is not the actual Java exception (`java.lang.Exception`) but rather a JavaScript exception – which is a simple JavaScript type, like String, Number, Date or Boolean.

Note that `try-catch` will effectively "swallow" any exception in the specified `try`-block, including syntax errors. So you will want to make sure that your code is correct before wrapping it in this statement.

4 Error Hooks

The term “Error Hook” is used to denote a Hook that is called as the direct result of an exception being thrown. While Function Components offer a single Default On Error Hook, Connector actually have four distinct types, including Default On Error.

Mandatory	These are not strictly *Error* Hooks, but rather the result of special exceptions thrown during Connector operation. Mandatory Hooks are special in that if the execution flow in the Connector ever reaches one and this Hook is not at least enabled, an error occurs. There are only three mandatory Hooks: On No Match and On Multiple Entries Hooks after *lookup* operations and No Answer Returned in CallReply mode.
Connection Failure	Whenever a Connector Interface (CI) operation results in a connection-related error, control is passed to the On Connection Failure Hook. If Auto-Reconnect is enabled, then it is engaged after the Hook completes. If Auto-Reconnect is not enabled, error flow is initiated and the mode-specific Error Hook is called.
Mode-specific	Each Connector mode has its own mode-specific Error Hook. Regardless of whether this Hook is enabled or not, error flow continues to Default On Error.
Default On Error	This Hook is shared between Connector modes and is also found in Function Components. Any errors that occur during AL cycling (i.e. not during Prolog or Epilog processing) will end up in the Default On Error Hook, unless control is explicitly passed elsewhere.

Each type of Error Hook listed above serves a specific purpose.

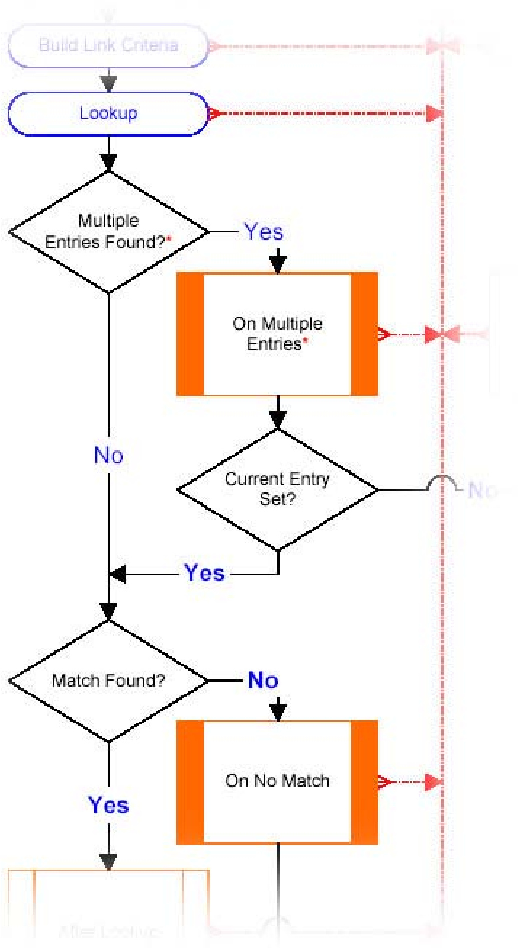
4.1 Mandatory

Usually, if a Hook is not enabled then it is quietly “ignored” during the execution of the Connector. Mandatory Hooks differ in that if the execution flow of the Connector ever reaches one of them, an error occurs if the Hook is not enabled. As noted in the table above, Mandatory Hooks are invoked based on the number of results returned by either a *lookup* (either zero or multiple returned) or *callreply* operation (only for zero returned).

Several Connector modes perform a *lookup*: Lookup, Delete, Update and Delta[5]. Note that for Update mode, whether or not data is found by the lookup is how this mode differentiates between doing an *add* or *modify* operation. As a result, there is no On No Match Hook for Update (or Delta) mode.

[5] Note that Delta mode is listed here for completeness sake. Delta mode will only perform a lookup if the underlying data source does not offer incremental modify operations. Since Delta mode is only available in the LDAP Connector and LDAP directories support incremental modifies, lookup is never done.

These Hooks appear in the Flow Diagrams as boxes with solid orange bars on both sides, as shown in this fragment of the Lookup Mode flow:



Although all Mandatory Hooks will result in an error if control is passed to them and they are not enabled, On Multiple Entries will also throw an error if no *current* Entry is set during execution of the Hook.

This behavior is based on the assumption that you are always expecting a single item returned from a lookup or callreply operation. Any other result needs be specifically dealt with by coding the appropriate Hook. In some case, it is impossible to continue with the Connector mode flow – for example, if you are in Delete mode but the initial lookup did not locate the entry to delete, or if multiple entries are found matching your Link Criteria. These situations require conscious intervention, and are sometimes a signal that you should be using a different mechanism to solve the problem.

As an example, let's go back to the situation where our Delete mode Connector found more than a single matching entry. If we allow the Connector flow to continue, the delete operation will be applied using the same Link Criteria that caused the On Multiple Entries exception. Some data sources will allow this (e.g. deleting multiple rows in an RDBMS table), while others won't. So we can either put script in this Hooks to handle this situation, or we could choose to change the Link Criteria to help ensure a single match[6].

[6] Or we can use a Loop Component to do the On Multiple Found handling for us.

4.2 Connection Failure

The On Connection Failure Hook is called whenever a Connector Interface operation (like *getNext*, *add* or *delete*) fails with a connection-related error. This could be due to any number of reasons, like a firewall timing out an open connection, or because the data source itself has gone offline.

These types of problems are often referred to as *infrastructural errors*. They are not specific to data content or the state and behavior of your AssemblyLine. Instead, they occur in the environment where your solution resides. The purpose of the On Connection Failure Hook is to give you the chance to deal with this type of exception differently than you would with standard errors.

Note that if the Auto-Reconnect feature is enabled for the Connector, then it is engaged immediately following this Hook. This means that you have the option of changing Connector parameters in this Hook before (re)connect is attempted, switching to a backup server if desired.

If the (re)connect is successful, then the CI operation that failed is reattempted and flow continues as if no error had occurred. Note that this can have unwanted side-effects in some situations. For example, if the *getNext* operation fails for an Iterator then performing a re-connect will also reinitialize the result set for iteration. This not only means that processing will resume *at the start* of the result set again, but that collection of entries returned for iteration may be different than it was initially.

Another potentially dangerous situation is when you are writing to a JDBC data source with the Commit parameter set to On Connector close. Here you run the risk that all writes performed before the Connection Failure are aborted (rollback) by the underlying RDBMS. If re-connect is successful, the AL will continue as though nothing was wrong. However, only updates done after the re-connect will be committed when the Connector is finally closed at AL shutdown. You either need to disable Reconnect, or use a different setting for the JDBC Commit parameter.

If for some reason you do not want to continue to Auto-Reconnect then you must redirect the flow using a command like `system.skipEntry()` or `system.ignoreEntry()`. The flowchart detailing this behavior is found in the Flow Diagrams page titled Connector Reconnect [2].

4.3 Mode-specific On Error

If an error occurs during a Connector Interface operation, then the On Error Hook for the Connector mode is invoked[7]. This Hook is not mandatory, so having it not enabled will not cause additional exceptions to be thrown.

After the mode-specific On Error Hook is executed (or skipped, if not enabled) flow continues to the Default On Error Hook

4.4 Default On Error

Although other error Hooks can precede it – like On Connection Failure or mode-specific On Error described in the previous sections – this is the error Hook that is ultimately called for any error-exception thrown during Connector or Function operation.

[7] This is also true of connection failures as well, although the On Connection Failure (and possibly Reconnect feature, if enabled) is executed first.

If this Hook is not enabled then the AssemblyLine aborts with the error and the AL's On Failure Hook is invoked. However, if Default On Error is enabled, the exception is effectively "swallowed" and control is passed to the next AL component.

5 Logging

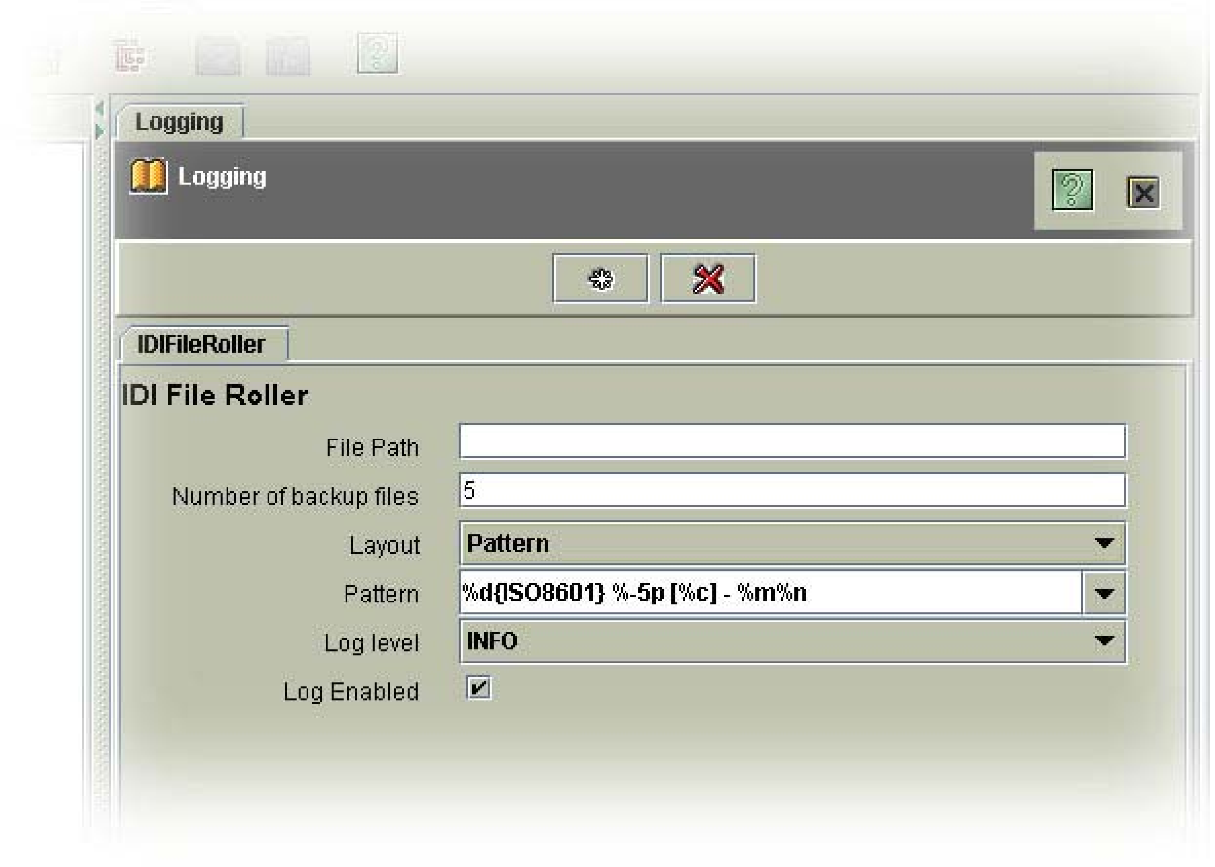
An important part of any error handling scheme is logging. Of course, logging has other uses as well; like passing data to other applications, or writing an AssemblyLine audit trail. As a result, this chapter will deal with logging in more general terms than just error handling.

TDI uses a Java API for logging called *Log4j*. This flexible framework provides a rich set of features that TDI leverages in such a way as to pass this flexibility on to you. It is not necessary that you know how Log4j works to do logging in TDI. However, any knowledge you do have can be directly applied to your solutions. So without going into the gritty details, logging in log4j can be thought of in three parts: *logger*, *appender* and *layout*.

The first part (*logger*) refers to the mechanism that enables logging, and this bit handled for you by TDI.

The second term (*appender*) is also a job for TDI. This work is carried out a logging component called, not surprisingly, an Appender. TDI provides a range of Appenders[3] that each supports a specific log system or mechanism.

Finally, *layout* defines the format in which your log messages are written. You define the layout for an Appender by setting its parameters.

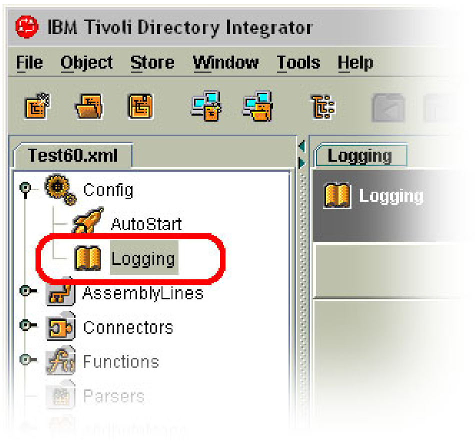


IDI File Roller

File Path	
Number of backup files	5
Layout	Pattern
Pattern	%d{ISO8601} %-5p [%c] - %m%n
Log level	INFO
Log Enabled	✓

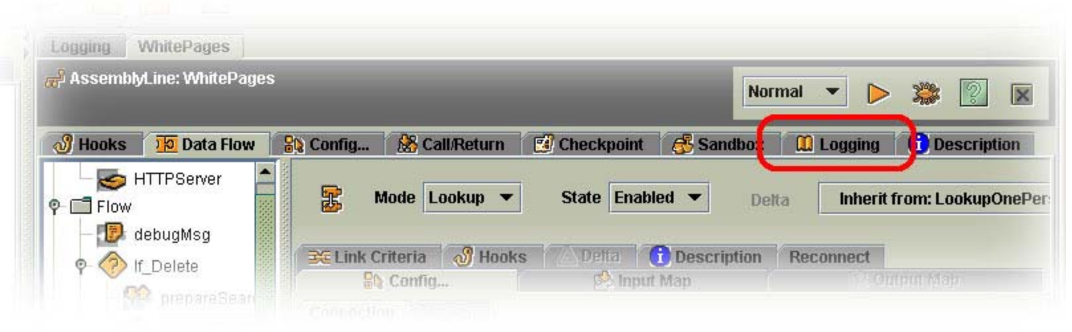
The above screenshot is of the IDI File Roller Appender, and the top two parameters are specific to this component. Here you tell the Appender what file to use as well as how many backup copies it should maintain. The next two parameters – Layout and Pattern – are how the log messages are to be written. Finally, the Log Level parameter instructs the *logger* feature in TDI which message priority levels to enable for this Appender. There are five levels to choose from: DEBUG, INFO, WARN, ERROR, and FATAL; in ascending order of priority. This parameter controls how verbose the Appender will be and setting one level will enable that priority plus all those that are higher. For example, if you set the log level to FATAL, then only this level of messages will be written by the Appender. However, setting it to WARN means that it will handle ERROR and FATAL as well.

Logging can be defined at the Config level, as well as for specific AssemblyLines. Setting up how all AssemblyLines in a Config will do their logging is done under the Config folder of the Config Browser. Here you will see an Item called Logging.



Selecting this item brings up a Logging Details window where you can add and remove Appenders that will be applied to all ALs.

In addition to Config-level settings, each AssemblyLine offers a Logging tab where you can specify further Appenders for this task.



References

1. Getting Started. Part of the official TDI documentation. See http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?topic=/com.ibm.IBMDI.doc/gettingstarted.htm
2. AssemblyLine and Connector mode flowcharts. Part of the official TDI documentation. See http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/topic/com.ibm.IBMDI.doc/referenceguide392.htm.
3. Details on Log4j in TDI can be found in the TDI Administrator Guide. See http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/topic/com.ibm.IBMDI.doc/adminguide48.htm