Directory Integrator
Version 7.1.1

Users Guide

SC27-2706-01

Directory Integrator
Version 7.1.1

Users Guide

SC27-2706-01

Notices

Before using this information and the product it supports, read the general information under "Notices" on page 261.

Edition notice

This edition applies to version 7.1.1 of the IBM Tivoli Directory Integrator and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 8. Tivoli Directory Integrator Dashboard 227

Appendix A. What's new for IBM Tivoli Directory Integrator version 7.1.1 . . . 241

Appendix B. IBM Tivoli Directory Integrator terms 247

Notices 261

Index 265

Figures

Tables

Preface

This document contains the information that you need to develop solutions using components that are part of the IBM® Tivoli® Directory Integrator.

Tivoli Directory Integrator components are designed for network administrators who are responsible for maintaining user directories and other resources. This document assumes that you have practical experience installing and using both Tivoli Directory Integrator and IBM Tivoli Directory Server.

Publications

Read the descriptions of the IBM Tivoli Directory Integrator V7.1.1 library and the related publications to determine which publications you might find helpful. After you determine the publications you need, refer to the instructions for accessing publications online.

IBM Tivoli Directory Integrator library

Use these short descriptions of publications and of external sources that can help you understand methodology and components.

IBM Tivoli Directory Integrator V7.1.1 Getting Started
Contains a brief tutorial and introduction to Tivoli Directory Integrator. Includes examples to create interaction and hands-on learning of Tivoli Directory Integrator.

IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide
Includes complete information about installing, migrating from a previous version, configuring the logging functionality, and the security model underlying the Remote Server API of Tivoli Directory Integrator. Contains information on how to deploy and manage solutions.

IBM Tivoli Directory Integrator V7.1.1 Users Guide
Contains information about using Tivoli Directory Integrator. Contains instructions for designing solutions using the Directory Integrator designer tool (the Configuration Editor) or running the ready-made solutions from the command line. Also provides information about interfaces, concepts and AssemblyLine creation.

IBM Tivoli Directory Integrator V7.1.1 Reference Guide
Contains detailed information about the individual components of Tivoli Directory Integrator: Connectors, Function Components, Parsers, Objects and so forth – the building blocks of the AssemblyLine.

IBM Tivoli Directory Integrator V7.1.1 Problem Determination Guide
Provides information about Tivoli Directory Integrator tools, resources, and techniques that can aid in the identification and resolution of problems.

IBM Tivoli Directory Integrator V7.1.1 Messages Guide
Provides a list of all informational, warning and error messages associated with the Tivoli Directory Integrator.

IBM Tivoli Directory Integrator V7.1.1 Password Synchronization Plug-ins Guide
Includes complete information for installing and configuring each of the five IBM Password Synchronization Plug-ins: Windows Password Synchronizer, Sun Directory Server Password Synchronizer, IBM Tivoli Directory Server Password Synchronizer, Domino® Password Synchronizer and Password Synchronizer for UNIX and Linux. Also provides configuration instructions for the LDAP Password Store and JMS Password Store.

IBM Tivoli Directory Integrator V7.1.1 Release Notes
Describes new features and late-breaking information about Tivoli Directory Integrator that did not get included in the documentation.

Related Publications

Information related to the IBM Tivoli Directory Integrator is available in the following publications:

- IBM Tivoli Directory Integrator V7.1.1 uses the JNDI client from Oracle. For information about the JNDI client, refer to the *Java Naming and Directory Interface™ Specification* at http://download.oracle.com/javase/6/docs/technotes/guides/jndi/index.html .

- The Tivoli Software Library provides a variety of Tivoli publications such as white papers, datasheets, demonstrations, redbooks, and announcement letters. The Tivoli Software Library is available on the Web at: http://www.ibm.com/software/tivoli/library/
- The *Tivoli Software Glossary* includes definitions for many of the technical terms related to Tivoli software. The Tivoli Software Glossary is available on the Web, in English only, at http://publib.boulder.ibm.com/tividd/glossary/tivoliglossarymst.htm
- A list of most requested documents as well as those identified as valuable in helping answer your questions related to IBM Tivoli Directory Integrator can be found at http://www.ibm.com/support/docview.wss?rs=697&context=SSCQGF&uid=swg27010509.

Accessing publications online

The publications for this product are available online in Portable Document Format (PDF) or Hypertext Markup Language (HTML) format, or both in the Tivoli software library: http://www.ibm.com/software/tivoli/library.

To locate product publications in the library, click **Product manuals** on the left side of the Library page. Then, locate and click the name of the product on the Tivoli software information center page.

A list of most requested documents as well as those identified as valuable in helping answer your questions related toIBM Tivoli Directory Integrator can be found at http://www-01.ibm.com/support/docview.wss?rs=697&uid=swg27009673.

Information is organized by product and includes readme files, installation guides, user's guides, administrator's guides, and developer's references as necessary.

Note: To ensure proper printing of PDF publications, select the **Fit to page** check box in the Adobe Acrobat Print window. The Acrobat Print window is available when you select **File>Print**.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully. With IBM Tivoli Directory Integrator (Tivoli Directory Integrator), you can use assistive technologies to hear and navigate the interface. After installation you also can use the keyboard instead of the mouse to operate all features of the graphical user interface.

Accessibility features

The following list includes major accessibility features of IBM Tivoli Directory Integrator:

- Supports keyboard-only operation.
- Supports interfaces commonly used by screen readers.
- Discerns keys as tactually separate, and does not activate keys just by touching them.
- Avoids the use of color as the only way to communicate status and information.
- Provides accessible documentation.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys for common Windows actions such as access to the File menu, and to the copy, paste, and delete actions. Actions that are unique use keyboard shortcuts. Keyboard shortcuts have been provided wherever needed for all actions.

Interface information

The accessibility features of the user interface and documentation include:

- Steps for changing fonts, colors, and contrast settings in the Configuration Editor:
  1. Type `Alt-W` to access the Configuration Editor **Window** menu. Using the downward arrow, select **Preferences...** and press `Enter`.
  2. Under the **Appearance** tab, select **Colors and Fonts** settings to change the fonts for any of the functional areas in the Configuration Editor.
  3. Under **View and Editor Folders**, select the colors for the Configuration Editor, and by selecting colors, you can also change the contrast.
- Steps for customizing keyboard shortcuts, specific to IBM Tivoli Directory Integrator:
  1. Type `Alt-W` to access the Configuration Editor **Window** menu. Using the downward arrow, select **Preferences...** .
  2. Using the downward arrow, select the General category; right arrow to open this, and type downward arrow until you reach the entry **Keys**.

     Underneath the **Scheme** selector, there is a field, the contents of which say "type filter text." Type `tivoli directory integrator` in the filter text field. All specific Tivoli Directory Integrator shortcuts are now shown.
  3. Assign a keybinding to any Tivoli Directory Integrator command of your choosing.
  4. Click **Apply** to make the change permanent.

The Configuration Editor is a specialized instance of an Eclipse workbench. More detailed information about accessibility features of applications built using Eclipse can be found at http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.user/concepts/accessibility/accessmain.htm

- The information center and its related publications are accessibility-enabled for the JAWS screen reader and the IBM Home Page Reader. You can operate all documentation features using the keyboard instead of the mouse.

Vendor software

The installer uses the InstallAnywhere 2010 (IA) installer technology.

Related accessibility information

Visit the *IBM Accessibility Center* at http://www.ibm.com/able for more information about IBM's commitment to accessibility.

Chapter 1. General Concepts

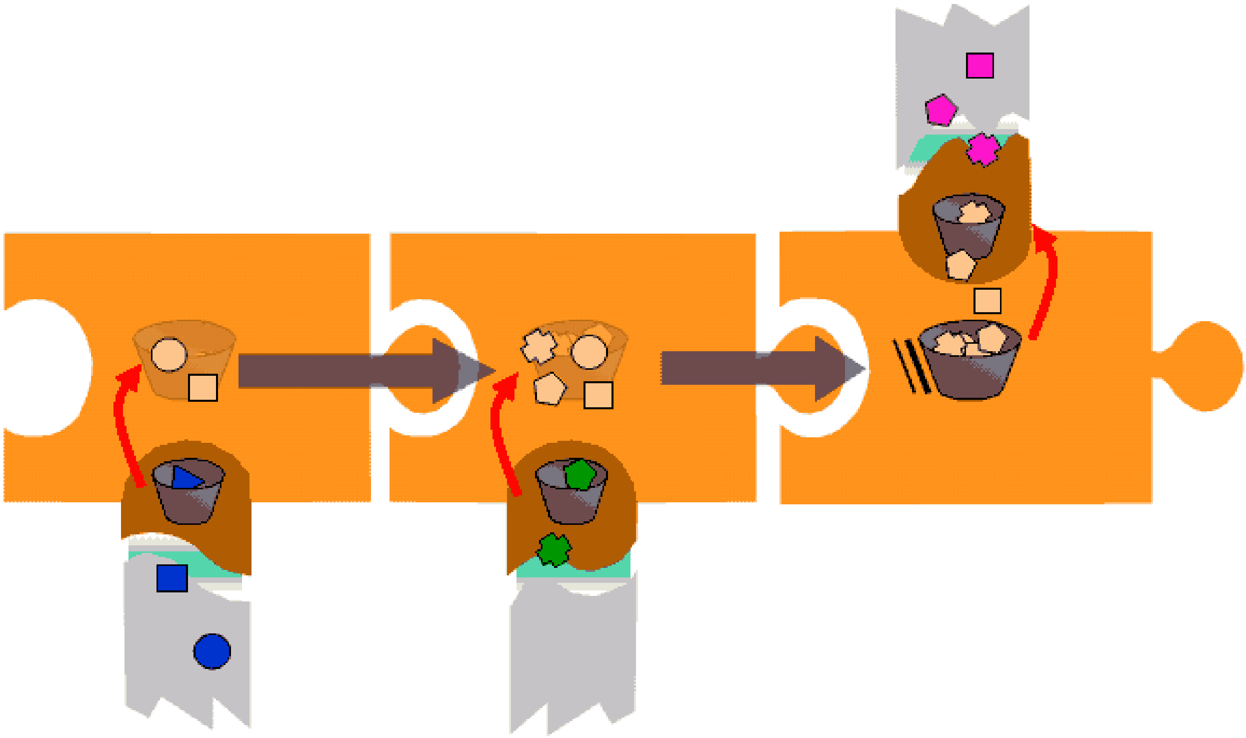
This chapter introduces some of the basic concepts in IBM Tivoli Directory Integrator, along with those elements of the architecture that allow you to build your solutions, their characteristics and behaviors.

The AssemblyLine

An AssemblyLine (AL) is a set of components strung together to move and transform data; it describes the "route" along which the data will pass. The data that is been handled through that journey is represented as an *Entry* object. The AssemblyLine works with a single entry at a time on each cycle of the AssemblyLine. It is the unit of work in TDI and typically represents a flow of information from one or more data sources to one or more targets.

Overview

Some of the components that comprise the AssemblyLine retrieve data from one or more *connected systems*—data obtained this way is said to "feed" the AL. Data to be processed is fed into the AL one *Entry* at a time, where these Entries carry Attributes with *values* coming from directory entries, database rows, e-mails, *Lotus*® *Notes*® documents, records or similar data objects. Each entry carries *Attributes* that hold the data values read from fields or columns in the source system. These Attributes are renamed, reformatted or computed as processing flows from one component to the next in the AL. New information can be "joined" from other sources and all or parts of the transformed data can be written to target stores or sent to target systems as desired. This can be illustrated thus:



In this diagram, picture the collection of large jigsaw puzzle pieces as the AssemblyLine, the leftmost blue dots and squares in the grey stream entering from below as raw data from an input stream, and the purple bits on the top right as data output on an output stream. The darker orange element intersecting a jigsaw piece with the bucket in it denotes a *Parser*, turning raw data into structured data, which then can start travelling down the AssemblyLine (as lighter-colored elements in a bucket). The middle jigsaw piece pictures a Connector reading already-structured data from for example a database.

Data enters the AssemblyLine from connected systems using “Connectors” on page 3 in some sort of input *Mode*, and is output later to one or more connected systems using Connectors in some output Mode.

Data can either be read from record-oriented systems like a database or a message queue: in this case the various columns in the input are readily mapped into Attributes in the resulting work Entry, which is depicted as a "bucket" in the puzzle piece on the left. Or, data can be read from a data stream, like a text file in a filesystem, a network connection, and so forth. In this case, a Parser can be prefixed to the Connector, in order to make sense of the input stream, and cut it up into pieces after which it can be assigned to Attributes in the work Entry.

Once the first Connector has done its work, the bucket of information (the "work Entry", called, appropriately, *work*) is passed along the AssemblyLine to the next Component—in the illustration, another Connector. Since the data from the first Connector is available, it can now be used as key information to retrieve, or lookup data in the second connected system. Once the relevant data is found, it can be merged into *work*, complementing the data that is still around from the first Connector.

Finally, the merged data is passed along the AssemblyLine to the third puzzle piece or Connector this time in some output Mode, which takes care of outputting the data to the connected system. If the connected system is record-oriented the various Attributes in *work* are just mapped to columns in the record; if the connected system is stream-oriented, a Parser can do the necessary formatting.

Other components, like “Script Components” on page 18 and “Functions” on page 18, can be inserted at will in the AssemblyLine to perform operations on the data in *work*.

It is important to keep in mind that the AssemblyLine is designed and optimized for working with one item at a time. However, if you want to do multiple updates or multiple deletes (for example, processing more than a single item at the time) then you must write AssemblyLine scripts to do this. If necessary, this kind of processing can be implemented using JavaScript, Java libraries and standard IBM Tivoli Directory Integrator functionality, such as pooling the data to a sorted data store, for example with the JDBC Connector, and then reading it back and processing it with a second AssemblyLine.

AssemblyLines are built, configured and tested using the IBM Tivoli Directory Integrator Config Editor (CE), see Chapter 3, “The Configuration Editor,” on page 67 for more information. The AssemblyLine has a Data Flow tab in the Config Editor. This is where the list of components that make up this AL are kept.

All components in an AL are automatically registered as script variables. So if you have a Connector called ReadHRdump then you can access it and its methods directly from script using the *ReadHRdump* variable. As a result, you will want to name your AL components as you would script variables: Use alphanumeric characters only, do not start the name with a number, and do not use special national characters (for example, å, ä), separators (apart from underscore '_'), white space, and so forth.

There is always an alternative method for accessing an AL component (for example, the `task.getConnector()` function) but a conscious naming convention is always advisable.

Starting an AssemblyLine in TDI is a fairly costly operation, as it involves the creation of a new Java thread and usually sets up connections to one or more data sources. Consider carefully if your solution design could be made to work with fewer, rather than more, distinct AssemblyLines, where each AssemblyLine does more work; for example, by using Branches or Switches to define multiple operations handled by a single AL. Note that each operation can still be implemented as a separate AssemblyLine, but these can be embedded "hot-and-ready" into a single AL that dispatches work to them by using the AL Connector or AL Function. This also allows you to leverage features like Global Connector Pools to manage resource usage and boost performance and scalability.

Components

AssemblyLines can include the following components:

- "Connectors"
- "Functions" on page 18
- "Script Components" on page 18
- "AttributeMaps" on page 20
- "Branch Components" on page 22

Additionally, Connectors can have "Parsers" on page 25 configured; also, at System, Config, AssemblyLine, Attribute map and Attribute level there are options to configure"Null Behavior" on page 20.

Accessing AL components inside the AssemblyLine

Each AL component is available as a pre-registered script variable with the name you chose for the component.

Note that you can dynamically load components with scripted calls to functions like `system.getConnector()`, although this is not for inexperienced users.[1]

AssemblyLine parameter passing

There are three ways for data to get into an AssemblyLine:

- Generating your own initial entry inside the AssemblyLine; for example, in a Prolog script.
- Fed from one or more Iterators[2].
- Starting the AssemblyLine with parameters from another AssemblyLine using the AL Connector or AL Function Component, or using an API call.

If you want to start an AssemblyLine with parameters from another AssemblyLine, then you have a couple of options:

- Use the *Task Call Block* (TCB), which is the preferred method. See "Task Call Block (TCB)" on page 57 for more information. This section also discusses techniques for dynamically disabling and enabling AssemblyLine components.
- Provide an Initial Work Entry directly; refer to "Providing an Initial Work Entry (IWE)" on page 6 for details.

  Note: These options are provided for compatibility with earlier versions.

Connectors

Connectors are used to access and update information sources. The job of a Connector is to level the playing field so that you do not have to deal with the technical details of working with various data stores, systems, services or transports. As such, each type of Connector is designed to use a specific protocol or API, handling the details of data source access so that you can concentrate on the data manipulations and relationships, as well as custom processing like filtering and consistency control.

[1] The Connector object you get from this call is a *Connector Interface* object, and is the data source specific part of an AssemblyLine Connector. When you change the type of any Connector, you are actually swapping out its data source intelligence (the Connector Interface) which provides the functionality for accessing data on a specific system, service or data store. Most of the functionality of an AssemblyLine Connector, including the attribute maps, Link Criteria and Hooks, is provided by the kernel and is kept intact when you switch Connector types.

[2] An *Iterator* is a shorthand notation of a Connector in Iterator Mode.

Overview

Connectors are used to abstract away the details of some system or store, giving you the same set of access features. This lets you work with a broad range of disparate technologies and formats in a consistent and predictable way. A typical AssemblyLine (AL) has one Connector providing input and at least one Connector writing out data.

There are two categories of Connectors:

- The first category is where both the transport and the structure of data content is known to the Connector; that is, the schema of the data source can be queried or detected using a well known API such as JDBC or LDAP.
- The second category is where the transport mechanism is known, but not the content structuring—which typically looks like a stream of data. This category requires a *Parser* (see “Parsers” on page 25 and the "Parsers" chapter in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*) to interpret or generate the content structure in order for the AssemblyLine to function properly.

A rich Connector library is one of the strengths of IBM Tivoli Directory Integrator. The list of all Connectors included with TDI can be found in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*. But you can also write your own Connector in JavaScript or even Java; see "Implementing your own Components", *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Each Connector is designed for a specific protocol, API or transport and handles marshalling data between the native type of the connected system and Java objects. Unlike the other components, Connectors have a Mode setting that determines how this Connector accesses its connected system; see “Connector modes” on page 5 for more information. Each Connector supports only a subset of modes that are suited for its connected system. For example, the File System Connector supports only a single output mode, AddOnly, and not Update, Delete or CallReply. When you use a Connector, you must first consult the documentation for this component for a list of supported modes.

Note: AssemblyLines can consist of as many, or as few, Connectors (and other Components) as required to implement your specific Data Flow. There is no limitation in the system. However, best practice is to keep an AssemblyLine as simple as possible in order to maximize maintainability.

When you select a Connector for your AssemblyLine, a dialog box is displayed enabling you to choose the type of Connector you want to *inherit* from. Inheritance is an important concept when working with IBM Tivoli Directory Integrator because all the components you include in solutions inherit some or all of their characteristics from another component—either from one of the basic types, or from your library of pre-configured components: Connectors, Parsers, Functions and so forth in the Resources section of your workspace.

When used in an AL, Connectors provide an Initialize option to control when the component is set up; for example, connections made, resources bound, and so forth. By default, all Connectors initialize when the AssemblyLine starts up: the *AL Startup Phase*.

Connectors in Server or Iterator mode feed the AssemblyLine, and are responsible for feeding the AL with a new Work Entry for each cycle that the AL makes. The Work Entry is passed from component to component in the Flow section, following any Branching logic you've implemented, until the end of the Flow is reached. At this point, end-of-cycle behavior begins, such as the Iterator getting the next entry from its source and passing it to the Flow section for a new cycle.

While an Iterator in the Feeds section will actually drive the Flow, an Iterator in the Flow section will simply get the next entry and offer its data Attributes for Input Mapping into the Work Entry.

Note: You can put a Connector in Iterator Mode into the Flow section. As such, the Iterator works 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 retrieves one entry (`getNextEntry`) on each cycle of the AL. However, an Iterator in the Flow section does not drive the AL itself, as it would in Feeds.

Feeds section behavior is different for Server and Iterator modes: An Iterator expects to be the first component running in the AL, and it will only read its next entry if the Work Entry does not already exist. If the AL is passed an Initial Work Entry, then Iterators do not read any data for this first cycle. It also means that Iterators run in a series, with the second one starting to return Entries once the first one has reached end-of-data and returned nothing (null).

Server Mode, on the other hand, causes the Connector to launch a server *listener* thread, for example, to an IP port or event-notification callback, and then pass control to the next Feeds Connector.

When you call an AL using the AssemblyLine Function component (AL FC), if you use the manual cycle mode then only the Flow section components are used each time the FC performs the call.

There is a **Connectors** folder in your workspace in the TDI Navigator where you can maintain your library of configured Connectors. This is also where Connector Pools are defined.

Connector modes

The mode of an AssemblyLine Connector defines what role that Connector plays in the data flow, and controls how the automated behavior of the AssemblyLine drives the Component. It determines whether to read from an input source, write to it or both.

Connectors can be set to one of these standard modes:

- Iterator
- Lookup
- AddOnly
- Update
- Delete
- CallReply
- Server
- Delta

These modes are discussed in the sections below. For a detailed description of Connector mode behavior, as well as that of the AssemblyLine in general, see "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Iterator mode:

Connectors in Iterator mode are used to scan a data source and extract its data. The Connector in Iterator mode actually iterates through the data source entries, reads their attribute values, and delivers each entry to the AssemblyLine Flow section components, one at a time. A Connector in Iterator mode is commonly referred to as an Iterator Connector, or just Iterator.

AssemblyLines (except those called with an IWE; see “Providing an Initial Work Entry (IWE)” on page 6) typically contain at least one Connector in Iterator mode. Iterators supply the AssemblyLine with data by building Work Entries and passing these to the AL Flow section.

Flow section components are powered in order, starting at the top of the Flow list. When Flow processing completes, control is passed back to the Iterator in order to retrieve the next entry.

Multiple Iterators in an AssemblyLine: If you have more than one Connector in Iterator mode, these Connectors are stacked in the order in which they appear in the Config (and the Connector List in the Config Editor, in the Feeds section) and are processed one at a time. So, if you are using two Iterators, the first one reads from its data source, passing the resulting Work Entry to the first non-Iterator, until it reaches the end of its data set. When the first Iterator has exhausted its input source, the second Iterator starts reading in data.

An initial Work Entry is treated as coming from an invisible Iterator processed before any other Iterators. This means an IWE is passed to the first non-Iterator in the AssemblyLine, skipping all Iterators during the first cycle. This behavior is visible on the AssemblyLine Flow page in "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Assume you have an AssemblyLine with two Iterators, **a** preceding **b**. The first Iterator, **a**, is used until **a** returns no more entries. Then the AssemblyLine switches to **b**, ignoring **a**. If an Initial Work Entry (IWE) is passed to this AssemblyLine, then both Iterators are ignored for the first cycle, after which the AssemblyLine starts calling **a**.

Sometimes the IWE is used to pass configuration parameters into an AssemblyLine, but not data. However, the presence of an IWE causes Iterators in the AssemblyLine to be skipped during the first cycle. If you do not want this to happen, you must empty out the Work Entry object by calling the `task.setWork(null)` function in a Prolog script. This causes the first Iterator to operate normally.

Using the Iterator mode:

It is very common to have an Iterator drive the AssemblyLine.

The most common pattern for using a Connector in Iterator mode is:

1. Using the Config Editor, add a Connector in Iterator mode to your workspace. See “Creating a Connector” on page 112.
2. Set the mode (Iterator) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
3. Set up the Attribute Map; see “Input Attribute mapping” on page 110.

These mapped Attributes are retrieved from the data source, placed in the *Work* entry, and passed to the Connectors in the Flow section in the AssemblyLine.

If you did not create the Connector directly inside an AssemblyLine, then in order to use this Connector in an AssemblyLine, drag the Connector from its location in `<workspace>/Resources/Connectors` to the **Feed** section of an AssemblyLine.

Providing an Initial Work Entry (IWE): This is an alternative way of passing parameters using a TCB and is supported for compatibility with earlier versions.

When an AssemblyLine is started with the `system.startAL()` call from a script, the AssemblyLine can still be passed parameters by setting attribute or property values in the Initial Work entry, which is accessed through the *work* variable. It is then your job to apply these values to set Connector parameters; for example, in the AssemblyLine **Prolog – Init** Hook using the *connectorName*.`setParam()` function.

Note: You must clear the Work entry with the `task.setWork(null)` call, otherwise Iterators in the AssemblyLine pass through on the first cycle.

You can examine the result of the AssemblyLine, which is the **Work Entry** when the AssemblyLine stops, by using the `getResult()` function. See also "Runtime provided Connector" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Below is an example of passing in a Connector parameter value with an IWE:

```
var entry = system.newEntry();
entry.setAttribute ("userNameForLookup", "John Doe");

// Here we start the AssemblyLine
var al = main.startAL ( "EmailLookupAL", entry );

// wait for al to finish
al.join();

var result = al.getResult();

// assume al sets the mail attribute in its working entry
task.logmsg ("Returned email = " + result.getString("mail"));
```

Lookup mode:

Lookup mode enables you to join data from different data sources using the relationship between attributes in these systems. A Connector in Lookup mode is often referred to as a Lookup Connector.

In order to set up a Lookup Connector in the Config Editor, you must tell the Connector how you define a match between data already in the AssemblyLine and that found in the connected system. This is called the Connector's *Link Criteria*, and each Lookup Connector has an associated **Link Criteria** tab where you define the rules for finding matching entries. See "Link Criteria" on page 16 for more information.

Using the Lookup mode: The most common pattern for using a Connector in Lookup mode is:

1. Using the Config Editor, add a Connector in Lookup mode to your workspace. See "Creating a Connector" on page 112.
2. Set the mode (Lookup) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
3. Set up the Attribute Map; see "Input Attribute mapping" on page 110.
4. Open the **Link Criteria** tab on the Connector configuration window and set up the rules for attribute matching. The outcome of this process will determine which entries are retrieved from the connected system, and here you have a couple of choices:
   a. Click **Add** to add a new Link Criterion and select an attribute from the connected system, the matching operator (for example, Equals, Begins With, and so forth) and then the Work Entry attribute to be matched. When the Connector performs the Lookup, it creates the underlying API or protocol syntax based on the Link Criteria you have specified, keeping your solution independent of the type of system used. You can add multiple Link Criteria, which are connected by the Boolean operator AND, together to build the search call.
   b. You can also select **Build criteria with custom script**, which opens a script editor window where you can create your own search string, passing this back to the Connector using the `ret.filter` object. For example:

      ```
      ret.filter = "uid=" + work.getString("uid");
      ```

   Note that Expressions can also be used to dynamically specify the Attribute or Value to use for any Link Criteria. See "Expressions" on page 30 for information. Also see "Link Criteria" on page 16 for more details about Link Criteria.

The attributes that you read (and compute) in the Input Map are available to other downstream Connectors and script logic using the Work entry object.

If you did not create the Connector directly inside an AssemblyLine, then to use the Connector in an AssemblyLine, drag it from its location in `<workspace>/Resources/Connectors` to the **Flow** section of the AssemblyLine.

AddOnly mode:

Connectors in AddOnly mode, commonly referred to as AddOnly Connectors, are used for adding new data entries to a data source.

This Connector mode requires almost no configuration. Set the connection parameters and then select (map) the attributes to write from the Work Entry.

Using the AddOnly mode: The most common and simple pattern for using a Connector in AddOnly mode is:

1. Using the Config Editor, add a Connector in AddOnly mode to your workspace. See “Creating a Connector” on page 112.
2. Set the mode (AddOnly) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
3. Set up the Attribute Map; see “Output Attribute mapping” on page 111.

If you did not create the Connector directly inside an AssemblyLine, then to use the Connector in an AssemblyLine, drag it from its location in `<workspace>/Resources/Connectors` to the **Flow** section of the AssemblyLine.

Work entry attributes you have mapped are output to the connected system when the Connector is called by the AssemblyLine.

Update mode:

Connectors in Update mode, usually referred to as Update Connectors, are used for adding and modifying data in a data source. For each entry passed from the AssemblyLine, the Update Connector tries to locate a matching entry from the data source to modify with the values of the entry attributes received. If no match is found, the Update mode Connector will add a new entry.

As with Lookup Connectors, you must tell the Connector how you define a match between data already in the AssemblyLine and that found in the connected system. This is called the Connector's Link Criteria, and each Update Connector has an associated Link Criteria (see “Link Criteria” on page 16) tab where you define the rules for finding matching entries. If no such entry is found, a new entry is added to the data source. However, if a matching entry is found, it is modified. If more than one entry matches the Link Criteria, the Multiple Entries Found Hook is called. Furthermore, the Output Map can be configured to specify which attributes are to be used during an Add or Modify operation.

When doing a Modify operation, only those attributes that are marked as Modify (Mod) in the Output Map are changed in the data source. If the entry passed from the AssemblyLine does not have a value for one attribute, the Null Behavior for that attribute becomes significant. If it is set to Delete, the attribute does not exist in the modifying entry, thus the attribute cannot be changed in the data source. If it is set to *NULL*, the attribute exists in the modifying entry, but with a null value, which means that the attribute is deleted in the data source.

An important feature that Update Connectors offer is the Compute Changes option. When turned on, the Connector first checks the new values against the old ones and updates only if and where needed. Thus you can skip unnecessary updates which can be valuable if the update operation is a heavy one for the particular data source you are updating.

Some Update Connectors offer the option to skip unneccessary lookups when doing Updates. If the Connector supports it, you will see a **Skip Lookup** check box next to the **Compute Changes** check box. When selected, it changes the behavior of the Connector so that no lookup is performed to find an entry that corresponds to the link criteria. For this reason no Before/After Lookup hooks are invoked. Also the On Multiple Entries hook cannot be invoked. With this option turned on only search criteria is built and modify is called directly. This differs from the default connector behavior in Update mode because if no entry is found with the link criteria it is not added as a new one.

Using the Update mode: The most common and simple pattern for using a Connector in Update mode is:

1. Using the Config Editor, add a Connector in Update mode to your workspace. See “Creating a Connector” on page 112.
2. Set the mode (Update) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
3. Set up the Attribute Map; see “Output Attribute mapping” on page 111.
4. Open the **Link Criteria** tab on the Connector configuration window and set up the rules for attribute matching. Here you have a couple of choices:
   a. Click **Add** to add a new Link Criterion and select an attribute from the connected system, the matching operator (for example, Equals, Begins With, and so forth) and then the Work entry attribute to be matched. When the Connector performs the Lookup, it creates the underlying API or protocol syntax based on the Link Criteria you have specified, keeping your solution independent of the type of system used. You can add multiple Link Criteria, which are connected by the Boolean operator AND, together to build the search call.
   b. You can also select **Build criteria with custom script**, which opens a script editor window where you can create your own search string, passing this back to the Connector using the `ret.filter` object. For example:

      ```
      ret.filter = "uid=" + work.getString("uid");
      ```

   Note that Expressions can also be used to dynamically specify the Attribute or Value to use for any Link Criteria. See “Expressions” on page 30 for information. Also see “Link Criteria” on page 16 for more details about Link Criteria.

Entries with the Attributes you have selected to map on input are added in the data source during the AssemblyLine's execution.

To use the Connector in an AssemblyLine, drag it from its location in `<workspace>/Resources/Connectors` to the **Flow** section of an AssemblyLine. You can now map Work entry attributes to the output by dragging attributes that were mapped in previously, onto the Update Connector in the **Attribute Maps** window of the AssemblyLine.

You can also create entirely new Attributes by right-clicking on the Connector in this window, and selecting **Add attribute map item**.

Note: In Update mode, multiple entries can be updated. See "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Delete mode:

Connectors in Delete mode, often referred to as Delete Connectors are used for deleting data from a data source.

For each entry passed to the Delete Connector, it tries to locate matching data in the connected system. If a single matching entry is found, it is deleted, otherwise the On No Match Hook is called if none were found, or the On Multiple Entries hook is more than a single match was found. As with Lookup and Update modes, Delete mode requires you to define rules for finding the matching entry for deletion. This is configured in the Link Criteria tab of the Connector.

Some Delete Connectors offer the option to skip unneccessary lookups when doing Deletes. If the Connector supports it, you will see a Skip Lookup check box next to the Compute Changes check box. When selected, it changes the behavior of the Connector so that no lookup is performed to find an entry that corresponds to the link criteria. For this reason no Before/After Lookup hooks are invoked. Also the On Multiple Entries hook cannot be invoked. With this option turned on only search criteria are built and delete is called directly.

Using the Delete mode: The most common and simple pattern for using a Connector in Delete mode is:

1. Using the Config Editor, add a Connector in Delete mode to your workspace. See "Creating a Connector" on page 112.
2. Set the mode (Delete) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
3. Set up the Attribute Map; see "Input Attribute mapping" on page 110.
4. In the Input Attribute Map tab you can select attributes from the Connector Attribute list and then drag them into your Input Map. You can also add and remove attributes manually.

   Note: The Input Map is used in Delete mode for reading the matching entry found in the data source into the *conn* entry object, which can then be used in your scripts (for example, to determine if the entry actually is to be deleted).
5. Open the **Link Criteria** tab on the Connector configuration window and set up the rules for attribute matching. The outcome of this process will determine which entries are retrieved from the connected system, and here you have a couple of choices:
   a. Click **Add** to add a new Link Criterion and select an attribute from the connected system, the matching operator (for example, Equals, Begins With, and so forth) and then the Work Entry attribute to be matched. When the Connector performs the Lookup, it creates the underlying API or protocol syntax based on the Link Criteria you have specified, keeping your solution independent of the type of system used. You can add multiple Link Criteria, which are connected by the Boolean operator AND, together to build the search call.
   b. You can also select **Build criteria with custom script**, which opens a script editor window where you can create your own search string, passing this back to the Connector using the `ret.filter` object. For example:

      ```
      ret.filter = "uid=" + work.getString("uid");
      ```

   See "Link Criteria" on page 16 for more information about Link Criteria.

If you did not create the Connector directly inside an AssemblyLine, then to use the Connector in an AssemblyLine, drag it from its location in `<workspace>/Resources/Connectors` to the **Flow** section of the AssemblyLine.

CallReply mode:

CallReply mode is used to make requests to data source services, such as Web services, that require you to send input parameters and receive a reply with return values.

Unlike the other modes, the CallReply mode gives access to both Input and Output Attribute Maps.

The Connector first performs an Output map operation, and in doing so calls an external system, with parameters provided by the Output map operation. This is immediately followed by an Input map operation, thus picking up the reply from the external system.

Using the CallReply mode: The most common and simple pattern for using a Connector in CallReply mode is:

1. Using the Config Editor, add a Connector in CallReply mode to your workspace. See "Creating a Connector" on page 112. Very few Connectors support this mode.
2. Set the mode (CallReply) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.

3. Set up the Output Attribute Map; see “Output Attribute mapping” on page 111.
4. Set up the Input Attribute Map; see “Input Attribute mapping” on page 110.

Keep in mind that Attributes from the Output Map are supplied to the connected system as input parameters; and Attributes mapped through the Input Attribute map contain the reply from the connected system.

Server mode:

The Server mode, available in a select number of Connectors, is designed to provide the functionality of waiting for an incoming event, dispatch a thread dealing with the event, and send a reply back to the originator.

Currently, all Server Mode connectors are connection-based. As a result, any AssemblyLine (AL) that uses a Server Mode Connector in its Feeds section will initialize and then wait for an incoming connection (by means of a TCP, HTTP, LDAP, Web Services, SNMP connection); when a connection is initiated, the Server Mode connector clones the AL it is part of, and resumes waiting for the next event (that is, a new connection initiation). In the cloned worker AL, meanwhile, the Server Mode Connector places itself in Iterator mode, and starts reading data from the connection. The data obtained from the connection is then fed to the rest of the AL in normal Iterator fashion, including following the standard Iterator Hook flow, reading the event entries one at a time and passing them to the other Flow components for processing until there is no more data to read. At the end of each cycle (often there will only be one) the AL headed by the Server Mode connector sends a reply back to the client—unless you decide to skip the reply phase with, for example, `system.skipEntry();`.

Once the AL it feeds is complete (that is, the data source is exhausted) that thread terminates; at this time, the worker AL is cleared away, and if necessary, the Pool Manager is informed that this AL instance is available again.

The original Server Mode connector, meanwhile, is still actively listening for more connection initiations.

Note: Under certain rare conditions, such as when you issue more than 5 client requests to the server in parallel, primarily on the zOS operating system, SNMP clients exit, giving bad Protocol Data Unit (PDU) exceptions. However, in a more realistic real-life situation an agent like the SNMP Server Connector is rarely queried intensely by multiple managers, like the SNMP Connector.

The SNMP Connector has an undocumented configuration parameter named *snmpWalkTimeout*. You can override the default for this parameter, which is 5000 ms. The parameter is not accessible using the Config Editor. You can set the override value for this parameter using JavaScript. Set the value you want for the snmpWalkTimeout parameter in the following format:

```
thisConnector.connector.setParam("snmpWalkTimeout", "100000");
```

Server Mode and the ALPool: The process of creating a clone AL can be optimized by using the AssemblyLine Pool (ALPool). On event detection, the Server mode Connector either proceeds with the Flow section of this AL; or if an ALPool is configured for this AL, then it contacts the Pool Manager process to request an available AL instance to handle this event.

When an AssemblyLine with a Server mode Connector uses the ALPool, the ALPool will execute AL instances from beginning to end. Before the AL instance in the ALPool closes the Flow Connectors, the ALPool retrieves those Connectors into a pooled connector set that will be reused in the next AL instance created by the ALPool. In essence, the ALPool uses the `tcb.setRuntimeConnector()` method.

There are two system properties that govern the behavior of Connector pooling:

com.ibm.di.server.connectorpooltimeout
This property defines the timeout in seconds before a pooled connector set is released.

Table 1. Connector Pool Timeout property Value table

Value	Significance
< 0	Disable Connector pooling
0	Timeout disabled; pool Connectors never timeout
> 0	Number of seconds before pooled Connectors timeout

com.ibm.di.server.connectorpoolexclude
This property defines the Connector types that are excluded from pooling. If a Connector's class name appears in this comma separated list it is not included in the Connector pool set.

When a new AssemblyLine (AL) instance is created by the ALPool, it will look for an available pooled connector set, which, if present, is provided to the new AL Instance as runtime-provided connectors. This ensures proper flow of the AL in general in terms of Attribute Mapping, Hook execution, and so forth.

Connectors are never shared. They are only assigned to a single AL instance when used.

Using the Server Mode: The most common pattern for using a Connector in Server Mode is:

1. Using the Config Editor, add a Connector in Server mode to your workspace. See “Creating a Connector” on page 112.
2. Set the mode (Server) and other connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
3. Set up the Attribute Map; see “Input Attribute mapping” on page 110.

These mapped Attributes are retrieved from the data source, placed in the *Work* entry, and passed to the Connectors in the Flow section in the AssemblyLine.

Note:

1. Server mode Connectors are special in that they usually need to return some information to the client that connects to it. Due to this nature, setting up Attribute Maps by clicking **Discover Attributes** will in many cases not result in any meaningful Schema; therefore, you will in most cases need to set up attribute maps completely by hand by selecting **Add new attribute**; alternatively delete unneccessary ones by selecting a mapping and deleting it. The data mapped out in this manner is sent back to the client, for each cycle of the AL; though as mentioned before, in a typical request or response way of operation there will often be only one cycle.
2. Server Mode Connectors based on TCP protocols have a parameter called Connection Backlog that controls the queue length for incoming connections.
3. In the AssemblyLine component list you can observe sections called Feeds and Flow. Server Mode Connectors observe a third phase in AssemblyLine processing, the *Response* phase. The Output Map and Response Hooks are part of the Server Mode Connector itself on screen, but the execution of response behavior is done after Flow section processing is finished.

   Note that a `system.skipEntry();` call will instruct the AssemblyLine 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.

If you did not create the Connector directly inside an AssemblyLine, then in order to use this Connector in an AssemblyLine, drag the Connector from its location in `<workspace>/Resources/Connectors` to the **Feed** section of an AssemblyLine.

Delta mode:

The Delta mode is designed to simplify the application of changes to data by providing incremental modifications to the connected system, based on delta operation codes.

Delta operation codes are set by either the Iterator Delta engine feature (**Delta** tab for Iterators), or Change Detection Connectors like the IBM Tivoli Directory Server, LDAP, or Active Directory (AD) Connectors, or the ones for RDBMS and Lotus/Domino Changes; or by parsing this delta information with the Lightweight Directory Interchange Format (LDIF) or Directory Services Markup Language (DSML) Parsers.

In versions earlier than IBM Tivoli Directory Integrator V6.1, 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. This limitation is addressed through the *Commit* parameter on the Connector Delta tab. The setting of this parameter controls when the Delta engine commits snapshots taken of incoming data to the System Store.

Delta mode is only available for LDAP and JDBC Connectors.

Note: A Connector in Delta mode must be paired with another Connector that provides Delta information, otherwise the Delta mode has no delta operation codes to work with.

The Delta features in Tivoli Directory Integrator (see Chapter 7, "Deltas," on page 213) are designed to facilitate synchronization solutions. You can look at the Delta capabilities of the system as divided into two sections: *Detection* and *Application*.

Delta Detection: IBM Tivoli Directory Integrator provides a number of change, or delta, detection mechanisms and tools:

Delta Engine

This is a feature available to Connectors in Iterator mode. If enabled from the Iterator's Delta tab, the Delta engine feature uses the System Store to take a snapshot of data being iterated. Then on successive runs, each entry iterated is compared with the snapshot database (the Delta Store) to see what has changed.

Change Detection Connector

These components leverage information in the connected system to detect changes, and are either used in Iterator or Server Mode, depending on the Connector. For example, Iterator mode is used for many of the Change Detection Connectors, like those for LDAP, IBM Tivoli Directory Server Changelog, as well as the RDBMS, Active Directory and Notes/Domino Change Detection Connectors.

The Change Detection Connectors have been designed to make them behave in a common way, as well as to provide the same parameter labels for common settings. The Connectors are:

- IBM Tivoli Directory Server (TDS) Changelog
- AD Change Detection (Active Directory)
- Domino Change Detection
- Sun Directory Change Detection (openLDAP, SunOne, iPlanet, and so forth)
- RDBMS Change Detection (DB2®, Oracle, SQL server, and so forth)
- z/OS® LDAP Changelog

See the "Connectors" chapter in the*IBM Tivoli Directory Integrator V7.1.1 Reference Guide* for more information about these Connectors.

The Delta engine feature reports specific changes all the way down to the individual values of attributes. This fine degree of change detection is also available when parsing LDIF files. Others components are limited to simply reporting if an entire entry is added, modified or deleted.

By selecting the **Allow Duplicate Delta keys** check box in the Iterator's Delta tab, you indicate that you allow duplicate delta keys, for those cases of long running AssemblyLines which need to process the same entries more than once. This means that duplicate entries can be handled for AssemblyLines that use both Changelog or Change Detection Connectors, Delta mode connectors and the Delta mode stores, when an entry has already been updated.

Attention: There is a possibility to have, for example, an AssemblyLine with a number of Changelog and Delta mode Connectors. In this case, if the Delta mode Connector is pointing to the same underlying system as the Changelog Connector, the Delta operation could trigger the Changelog again. As there is no way to differentiate between newly-received changes and those triggered by the Delta engine, you should carefully consider your scenario in order not to enter into an endless loop.

The delta information computed by the Delta engine is stored in the Work Entry object, and depending on the Change Detection component or feature used can be stored as an *Entry-Level operation code*, at the *Attribute-Level* or even at the *Attribute Value-Level*.

As an example, set up a File System Connector with the Delta engine feature enabled. Have it iterate over a simple XML document that you can easily modify in a text editor. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3333</ValueTag>
        </Telephone>
        <Birthdate>1958-12-24</Birthdate>
        <Title>Full-Time TDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
    </Entry>
</DocRoot>
```

Be sure to use the special map-all attribute map character, the asterisk (*). This is the only Attribute you need in your map to ensure that all Attributes returned are mapped in to the Work entry object.

Now add a Script Component with the following code:

```
// Get the names of all Attributes in work as a String array
var attName = work.getAttributeNames();
// Print the Entry-level delta op code
task.logmsg(" Entry (" +
    work.getString( "FullName" ) + ") : " +
    work.getOperation() );
// Loop through all the Attributes in work
for (i = 0; i < attName.length; i++) {
  // Grab an Attribute and print the Attribute-level op code
 att = work.getAttribute( attName[ i ] );
 task.logmsg("    Att ( " + attName[i] + ") : " + att.getOperation() );
  // Now loop through all the Attribute's values and print their op codes
 for (j = 0; j < att.size(); j++) {
  task.logmsg( "      Val (" +
                   att.getValue( j ) + ") : " +
                   att.getValueOperation( j ) );
 }
}
```

The first time you run this AL, your Script Component code will create this log output:

```
12:46:31   Entry (John Doe) : add
12:46:31      Att ( Telephone) : replace
12:46:31        Val (111-1111) :
12:46:31        Val (222-2222) :
```

```
12:46:31        Val (333-3333) :
12:46:31      Att ( Birthdate) : replace
12:46:31        Val (1958-12-24) :
12:46:31      Att ( Title) : replace
12:46:31        Val (Full-Time TDI Specialist) :
12:46:31      Att ( uid) : replace
12:46:31        Val (jdoe) :
12:46:31      Att ( FullName) : replace
12:46:31        Val (John Doe) :
```

Since this entry was not found in the previously empty Delta Store, it is tagged at the entry-level as *new*. Furthermore, each of its Attributes has a *replace* code, meaning that all values have changed (which makes sense because the Delta is telling us that this is new data).

Make the following changes to your XML file:

1. Change the last Telephone number value to 333-3334.
2. Delete Birthdate.
3. Add a new Address Attribute.

Your resulting Config should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3334</ValueTag>
        </Telephone>
        <Title>Full-Time TDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
        <Address>123 Willowby Lane</Address>
    </Entry>
</DocRoot>
```

Run your AL again. This time your log output should look like this:

```
13:53:22   Entry (John Doe) : modify
13:53:22      Att ( Telephone) : modify
13:53:22        Val (111-1111) : unchanged
13:53:22        Val (222-2222) : unchanged
13:53:22        Val (333-3334) : add
13:53:22        Val (333-3333) : delete
13:53:22      Att ( Birthdate) : delete
13:53:22        Val (1958-12-24) : delete
13:53:22      Att ( uid) : unchanged
13:53:22        Val (jdoe) : unchanged
13:53:22      Att ( Title) : unchanged
13:53:22        Val (Full-Time TDI Specialist) : unchanged
13:53:22      Att ( Address) : add
13:53:22        Val (123 Willowby Lane) : add
13:53:22      Att ( FullName) : unchanged
13:53:22        Val (John Doe) : unchanged
```

Now the entry is tagged as *modify* and the Attributes reflect what the modifications for each of them. As you can see, the Birthdate Attribute is marked as *delete* and Address as *add*. That's the reason you used the special map-all character for our Input Map. If you had mapped only the Attributes that existed in the first version of this XML document, we would not have retrieved Address when it appeared in the input.

Note especially the last two value entries under the Telephone Attribute, marked as *modify*. The change to one of the values of this Attribute resulted in two Delta items: a value *delete* and then an *add*.

To build a data synchronization AssemblyLine in earlier versions of TDI, you had to script in order to handle flow control. Although you could be receiving *adds*, *modifies* and *deletes* from your change component or feature, a Connector could only be set to one of the two required output modes: Update or Delete. So either you had two Connectors pointing to the same target system and you put script in the Before Execute Hook of each to ignore the entry if its operation code did not match the mode of this component; or you could have a single Connector (either Update or Delete mode) in *Passive* state, and then control its execution from script code where you checked the operation code. This still meant that even though you knew what had changed in the case of a modified entry, your Update Mode Connector would still read in the original data before writing the changes back to the data source. This can lead to unwanted network or datasource traffic when you are only changing a single value in a multi-valued group-related Attribute containing thousands of values.

Enter the Connector *Delta* mode.

Delta Application (Connector Delta Mode): The Delta mode is designed to simplify the application of delta information; that is, make the actual changes in a number of ways.

Firstly, Delta mode handles all types of deltas: adds, modifies and deletes. This reduces the number of data synchronization ALs to two Connectors: One Delta Detection Connector in the Feeds section to pick up the changes, and a second one in Delta mode to apply these changes to a target system.

Furthermore, Delta mode will apply the delta information at the lowest level supported by the target system itself. This is done by first checking the Connector Interface to see what level of incremental modification is supported by the data source[3]. If you are working with an LDAP directory, then Delta mode will perform Attribute value adds and deletes. In the context of a traditional RDBMS (JDBC), then doing a delete and then an add of a column value does not make sense, so this is handled as a value replacement for that Attribute.

This is dealt with automatically by the Delta mode for those data sources that support this functionality[4]. If the data source offers optimized calls to handle incremental modifications, and these are supported by the Connector Interface, then Delta mode will use these. On the other hand, if the connected system does not offer "intelligent" delta update mechanisms, Delta mode will simulate these as much as possible, performing pre-update lookups (like Update mode), change computations and subsequent application of the detected changes.

Link Criteria:

The Link Criteria is used to tell a Connector in Update, Lookup and Delete modes how you define a match between data attributes in the AssemblyLine and those found in the connected system.

The Link Criteria is accessible in the Config Editor through the **Link Criteria** tab, which is only enabled for Update, Lookup and Delete Connector modes.

There are two types of Link Criteria, **Simple** and **Advanced**.

Simple Link Criteria: For each simple Link Criteria, specify the Connector Attribute (those attributes defined in the Connector Schema), the Operator to use (for example, Contains, Equals, and so forth), and the Value to use with the operation. The value you use can be entered directly, or it can refer to the value of an attribute in the Work entry that is available at this point in the AssemblyLine flow. When the Connector performs the Lookup operation (for Lookup, Update and Delete modes) it converts the Link Criteria to the data source-specific call, enabling you to keep your solution independent of the underlying technology.

[3] Note that the only Connectors that support incremental modifications are the LDAP and JDBC Connectors, since LDAP directories provide this functionality.

[4] Also, you can control these built-in behaviors through configuration parameters and Hook code.

If you want to build a Link Criteria using the value of an attribute in the Work entry, simply use the name of the attribute in the Value field of the Link Criteria, preceded by the dollar sign ($). So, if you want to match the attribute named *cn* with an attribute in the Work entry called FullName, your Link Criteria is specified as:

```
cn EQUALS $FullName
```

If you want to find a specific person directly, set the Link Criteria with a literal constant value:

```
cn EQUALS Joe Smith
```

Note: The dollar sign ($) matches the first value of a multi-valued attribute only. If you want to match an attribute in the data source with any of the multiple values stored in a Work entry attribute instead, then use the *at* symbol (@). For example:

```
dn EQUALS @members
```

This example tries to match the dn attribute in the connected system to any one of the values of the multi-valued attribute in the Work entry named *members*.

A Connector can have multiple Link Criteria defined, and these are normally connected together, by use of the Boolean operator AND, to find the match.

However, if you click **Match Any**, just one of the Link Criteria needs to match, the equivalent of an OR operation.

Note that the name of the Attribute to match can be specified as an Expression (See “Expressions” on page 30 for details). The possible formats for the Value field of a Simple Link Criteria are:

A text string
Mapped to a constant with that value.

$Name
Corresponds to `work.getString("Name")`, that is the first value of the attribute *Name*.

@Name
Matches one of the values of the multi-valued attribute *Name*.

A TDI Expression
Described in detail here: “Expressions” on page 30.

Advanced Link Criteria: You can also create your own custom search criteria by checking the **Build criteria with custom script** check box. This presents you with a script editor to write your own Link Criteria expression. Not all Connectors support the Advanced Link Criteria, and the Connector documentation states whether Advanced Link Criteria is supported. See "Connectors" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

The search expression that you build must comply with the syntax expected by the underlying system. In order to pass your search expression to the Connector, you must populate the *ret.filter* object with your string expression.

A simple JavaScript example for an SQL Connector is:

```
ret.filter = " ID LIKE '" + work.getString("Name") + "'";
```

This custom Link Criteria assumes an example where the data source has an attribute called *ID* (typically a column name) that we want to match with the *Name* attribute in the Work entry.

Note:

1. The first part of the SQL expression, `Select * from Table Where`, is provided by IBM Tivoli Directory Integrator.

2. Single quotation marks have been added because `work.getString()` returns a string, while SQL syntax asks for single quotation marks around strings constants.
3. The special syntax with **$** and **@** is not used here.

Link Criteria errors

The most common error you get when using Link Criteria is:

```
ERROR> AssemblyLine x failed because
No criteria can be built from input (no link criteria specified)
```

This error occurs when you have a Link Criteria that refers to an attribute that cannot be found during the Lookup. For example, with the following Link Criteria:

```
uid equals $w_uid
```

Link Criteria setup fails if *w_uid* is not present in the Work entry. This might be because it is not read from the input sources (for example, not in an Input Map, or missing from the input source) or is removed from the Work entry in a script. In other words, the function call `work.getAttribute("w_uid")` returns NULL.

One way to avoid this is to write code in the Before Execute Hook of the Lookup, Delete, or Update mode Connector that skips its operation when the Link Criteria cannot be resolved due to missing attributes. For example:

```
if (work.getAttribute("w_uid") == null)
  system.ignoreEntry();
```

Your business rules might require other processing, such as a `skipEntry()` call instead of `ignoreEntry()`, which causes the AssemblyLine to stop processing the current entry and begin from the top on a new iteration. The ignoreEntry() function simply skips the current Connector and continues with the rest of the AssemblyLine.

Functions

A *Function*, often referred to as a Function Component (FC), is a component much like a Connector, except that it does not have a mode setting. Whereas Connectors provide standard access verbs for connected systems (Lookup, Delete, Update, and so forth), Functions on the other hand only perform a single operation, like pushing data through a Parser, dispatching work to another AssemblyLine or making a Web service call.

Overview

Functions can appear anywhere in the Flow section of an AL. The Functions library folder in the Config browser can be used to manage your library of Function Components.

Like Connectors, Functions in AssemblyLines provide an Initialize option to determine when this component starts. By default, Functions initialize during AL startup.

Script Components

The Script Component (SC) is a user-defined block of JavaScript code that you can drop any place in the AssemblyLine data Flow list, alongside your Connectors and Function Components, causing the script code within to be executed for each cycle at this point in the AL workflow.

Overview

Unlike Hooks, Script Components are easily moved around in the AssemblyLine flow, making them very powerful tools for everything from debugging to prototyping and implementing your own flow logic. Also, unlike the other types of AL components, the Script does not have any predefined behavior or mode; you can implement behavior and mode with your script. A library of Scripts can be stored under the Scripts library folder in the Config browser.

Usage

For example, if you want to test and debug only part of an AssemblyLine, you could put the following code in an SC to limit and control AL flow.

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

When placed at the appropriate point in the flow of an AssemblyLine, this SC then displays the contents of the work object by writing it to log output and then skip the rest of the AL for this cycle. By moving the SC up and down the component list, you control how much of the AL is actually executed. If you swap out the `system.skipEntry()` call with `system.skipTo("ALComponentName")`, you directly pass control to a specific AL Component.

You can also use SCs to drive other components. A typical scenario when doing directory or database synchronization is having to handle both updated and deleted information. Since Connectors powered by the built-in AL workflow can only operate in one mode at a time (Update or Delete) you will need to extend this logic a bit with your own code. One method is to add two Connectors, one in Update mode and one in Delete mode, and then put code in the Before Execute Hook in each Connector's to tell it to skip change operations that it should not handle. For example, in the Before Execute Hook of the Update Connector you would write something like this:

```
// The LDAP change log contains an attribute called "changeType"
if (work.getString("changeType").equals("delete"))
 system.ignoreEntry();
```

This will cause your Update mode Connector to skip deleted Entries. You would have complementary code in the Before Execute Hook of your Delete mode Connector, skipping everything but deletes.

However, if you are synchronizing updates to multiple targets, this would require you to have two Connectors per data source. Another approach is to have a single Connector in Passive state that you power from script. As an example, let's say you have a Passive AL Connector called *synchIDS*. You can then add an SC with the following code to run it:[5]

```
if (work.getString("changeType").equals("delete"))
 synchIDS.deleteEntry( work )
else
   synchIDS.update( work );
```

As long as you label your SC clearly, indicating that it is running Passive Connectors, this approach will result in shorter AssemblyLines that will be easier to read and maintain. This is an example of having to choose between two best practices: keeping the AL short, and using built-in logic versus custom script. However, in this case the goals of legibility and simplicity are best served by writing a little script.

The Script Component also comes in handy when you want to test logic and script code. Just create an AssemblyLine with a single Script Component in the data Flow list, put in your code and run it.

See Also

Chapter 2, “Scripting in TDI,” on page 37.

[5] Since Passive Connectors are not run by the AL logic, it does not matter where they appear in the Data Flow list.

AttributeMaps

Attribute Maps are pathways for data to flow into or out of the AssemblyLine. Attribute Maps appear in Connectors and Functions as Input and Output Maps, and are also available as stand-alone components in the AssemblyLine.

Overview

The diagram at the start of the section entitled "The AssemblyLine" on page 1 depicts three Attribute Maps as curved arrows: two Input Maps bringing data into the AssemblyLine, as well as an Output Map passing data to the Connector's cache (the *Conn* Entry) so it can be written.

Each Attribute Map holds a list of rules that create Attributes in either the Work Entry or the Conn Entry. A mapping rule specifies two things:

1. The name of the Attribute to be created (or overwritten) in the target Entry. This is the Work Entry in the case of Input Maps and free-standing Attribute Map components, while Output Maps target the Conn Entry.
2. The assignment used to populate the Attribute with one or more values. And assignment can be either assignment script, for example:

   ```
   work.Title
   ```

   or it can be a literal text (including newline characters) with optional token substitution:

   ```
   <html>
      <header>
         <title>{work.Title}</title>
      </header>
      <body>
         <p>Look for the "Title" Attribute value in the title of this page</p>
      </body>
   </html>
   ```

Attribute Maps are created using the Config Editor; see "Attribute Mapping and Schema" on page 104.

Attribute Maps support inheritance, both at the Map level and for individual mapping rules. Note that you can drag a Script onto an Attribute Map to set up an inherited JavaScript mapping rule.

Attribute Maps also provide a functionality called "Null Behavior," which is used to control how missing data is handled.

See "Internal data model: Entries, Attributes and Values" on page 38 for some more information on Attribute Maps.

Null Behavior

Occasionally, the system tries to map an attribute that is missing. For example, if an optional telephone number is not present in the input data source, or an attribute in an Output Map is removed from the Work entry. Other times although an attribute is present, it has no values - like a nullable column in an database table.

Different data sources treat missing values in different ways (null value, empty string) and the feature described in this section provides a way of customizing how missing attributes - or attribute values - are treated. This feature is called *Null Behavior* and with it you can define both what a "null value" is as well as how it is to be handled.

Note: The JDBC Connector has the `jdbcExposeNullValues` parameter setting, enabling you to map null values to missing Attributes (see "JDBC Connector" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*).

Null behavior can be specified at a number of levels: system, Config, AssemblyLine, AttributeMap and Attribute. However, since Null Behavior is highly data source-specific, it makes most sense to set this system property at the Attribute map level (for example, for all Attributes handled by a Connector's Input or Output Maps). These possible levels are described here in more detail:

System level
Specifying system level Null Behavior is done in the global.properties file by setting the `rsadmin.attribute.nullBehavior` and `rsadmin.attribute.nullDefinition` properties, one of the values listed later in this section.

Config level
This overrides System level Null Behavior, and is configured by setting the `rsadmin.attribute.nullBehavior` or `rsadmin.attribute.nullDefinition` properties in a Property Store to one of the following values listed later in this section.

AssemblyLine level
AssemblyLine Null Behavior is specified in by clicking the **Options...** button in the AssemblyLine toolbar, select **AssemblyLine settings**, and click **Null Value Behavior** in the dialog box that comes up.

Attribute map level
Defining Null Behavior for all the Attributes in a map is done by clicking the **More...** button above the attribute map list, and selecting **Null behavior**.

Attribute level
Null Behavior can be configured for a specific attribute by right-clicking on it in an attribute map and selecting **Null behavior**. When this has been defined for an Attribute, then this is indicated by the presence of a blue bullet symbol in the mapped item.

Null Behavior supports five different settings for defining what a null value is, as shown below (Note that the text in parenthesis for each setting is the value used to set the System-level Null Behavior definition, and is typically defined in the Solution or Global Property Store). Each setting shows the actual property value in parenthesis. Note that these definitions are listed in inclusive order, so that the second case also includes the first one; the third one includes the first two; and so forth:

Attribute is missing (`AbsentAttribute`)
The Attribute referenced as the source of value(s) in an attribute map is missing.

Attribute with no values (`EmptyAttribute`)
The Attribute used as the source of value(s) in an Attribute is found, but has no values. The previous case is also checked for.

Attribute contains an empty string value (`EmptyString`)
The Attribute is found, but has only a single string value.

Value (`value`)
The Attribute contains a specified value. For AssemblyLine, attribute map and Attribute-level null value definition, this value is set in the **Value** field of the Null Behavior dialog. Here you can specify multiple attribute values if desired by placing values on separate lines. If you use `rsadmin.attribute.nullDefinition` for system and Config level setting then you must also set the `rsadmin.attribute.nullDefinitionValue` property.

Note: Several enhancements have been made to the HTTP server connector. TCP-based components, like the HTTP server Connector, have a switch in their Config screens for returning TCP headers as Attribute values. When this flag is cleared, TCP headers are stored as properties in the returned entry object.

Default Behavior (`Default Behavior`)
The null value definition must be inherited from a higher level. For example, an Attribute inherits its null value definition from the attribute map setting, which in turn inherits it from the AssemblyLine.

Note: Config level Null Behavior overrides any system level settings. Furthermore, the Default Behavior setting at system level is the same as specifying **delete**, while at Config level it is equivalent to **value**.

The Null Behavior feature also lets you define the action to be taken in case a null value is detected:

Empty String (`empty string`)
Missing attributes are mapped with a single value which has an empty String value ("").

Null (`null`)
Missing attributes are mapped with no values, meaning that the `att.getValue()` call returns **null**.

Delete (`delete`)
The attribute is removed from the map.

Value (`value`)
Missing attributes are mapped with a specified value. For AssemblyLine, attribute map and Attribute-level Null Behavior, the values are set in the **Value** edit of the Null Behavior Dialog. Here you can specify multiple attribute values if desired by placing values on separate lines. If you use `rsadmin.attribute.nullBehavior` for system and Config level settings then you must also set the `rsadmin.attribute.nullBehaviorValue` property.

Default Behavior (`Default Behavior`)
Null Behavior must be inherited from a higher level. For example, Attribute level inherits from the AttributeMap, which in turn inherits from the AssemblyLine setting.

Note: Config level Null Behavior overrides any system level settings. Furthermore, the Default Behavior setting at system level is the same as specifying **delete**, while at Config level this is equivalent to **value**.

Branch Components

Branch components affect the order in which the other components, such as, Connectors, Scripts, Functions, AttributeMaps and other Branch components, are executed in the Flow of the AssemblyLine.

Overview

Branch components come in three varieties:

- Simple (also called just Branch)
- Loops (Branches that loop)
- Switches (Branches that share the same expression)

Branches can appear anywhere in the Flow section, but there is no library folder for them in the Resources section of your workspace.

A Branch does not have to run to completion; the same script call is used to programmatically exit any type of Branch: `system.exitBranch()`. See “Exiting a Branch (or Loop or the AL Flow)” on page 24 for more information.

The three Branch component types are:

Branch

While each type of Branch lets you define alternate routes for AssemblyLine processing, this simplest form determines the action in case: "If this situation occurs, then take this action." You define what "situation occurs" means by setting up Conditions that must be met; for example, by comparing data values or checking the result of some operation. If the conditions are true, the components attached under this Branch are executed.

The Branch allows you to define Conditions based on any data in the TDI server: Attribute values, parameters settings, externally accessible properties, and any information available using JavaScript, such as operating system calls for disk or memory usage. Multiple Conditions are ANDed or ORed, depending on the **Match Any** check box setting.

This simplest form of Branch component also supports three subtype settings, IF, ELSE-IF and ELSE, that you can select.

IF
Can appear anywhere within the AssemblyLine Flow, the IF Branch provides an alternative track for process to follow if Conditions are true. Once the components under the Branch are executed, control passes to the first component *after* this Branch. If you do not want this to happen, you must either add an ELSE or ELSE IF Branch, or exit the Branch with a scripted call to `system.exitBranch()`.

ELSE-IF
Identical to the IF Branch, except it can only appear immediately following an IF or ELSE-IF Branch.

ELSE
Can only appear immediately after an IF or ELSE-IF Branch, the ELSE Branch has no Conditions. Its components are processed only if no preceding IF or ELSE-IF Branch was true. Furthermore, the ELSE instance always evaluate to true so there are no conditions evaluated during the cycle.

As mentioned above, you can prematurely exit a Branch by means of scripting, using `system.exitBranch()`.

Loop

The Loop component provides functionality for adding cyclic logic within an AssemblyLine. Loops can be configured for three modes of operation: based on Conditions, based on a Connector or based on the values of an Attribute:

Conditional
Just as with a simple Branch, you can define Conditions that control Loop behavior. The Loop will continue to cycle as long as the Conditions are met, and will stop as soon as they fail. The details window for Loops is the same as for the simple Branches described in the previous section.

Connector
This method lets you set up a Connector in either Iterator or Lookup mode, and will cycle through your Loop flow for each entry returned. The details window of this type of Loop contains the Connector tabs necessary to configure it, connect and discover attributes and set up the Input Map.

Note that you have a parameter called **Init Options** with which you can instruct the AL to either:

- **Do Nothing** means that the Connector will not be prepared in any way between AL cycles.
- **Initialize and Select/Lookup** causes the Connector to be re-initialized for each AL cycle.
- **Select/Lookup Only** keeps the Connector initialized, but redoes either the Iterator select or the Lookup, depending on the Mode setting.

Note also that there is a **Connector Parameters** tab, which functions similarly to an Output Map in that you can select which Connector parameters are to be set from **work** Attribute values.

This brings us to the topic of how Looping with an Iterator differs from doing so based on Lookup mode. Both options perform *searches* that create a result set returned for looping. For Iterator mode, the result set is controlled exclusively by the parameter settings of this component. Lookup mode, on the other hand, uses Link Criteria to define search or match rules. Since it frees you from having to code Hooks like On No Match or On Multiple Found, this is the preferred way of doing searches that may not always return one (and just one) matched entry.

Attribute Value

By selecting any Attribute available in the work Entry, the Loop flow will be executed for each of its values. Each value is passed into the Loop in a new Work Entry attribute named in the second parameter. This option allows you to easily work with multi-valued attributes, like group membership lists or e-mail.

You can prematurely exit a Loop by means of scripting, using `system.exitBranch()`.

Switch

Unlike expressions used in the Conditions of Branches and Loops, the Switch expression can result in more values than just `true` or `false`. For example, you could Switch on the value of an Attribute, or the operation requested when this AL was called from another AssemblyLine or process. Under the Switch component, you add a Case for each constant value of the Switch expression that you want to handle. So for example, if you set up the Switch to use the delta operation code in the Work Entry, your Cases would be for values like "add", "delete" and "modify."

In an AL Switch-Case construct, multiple cases can be active at the same time. TDI checks each case, just as it would a series of standard IF Branches. The following example shows how multiple cases work:

```
work.setAttribute("test","abc");

Switch work.test
  Case startsWith("a"): this is true
  Case contains ("bc"): this is true
  Case length=3: this is true
```

The three Switch work.test expressions that are true will trigger Switch execution.

You can prematurely exit a Switch-Case by means of scripting, using `system.exitBranch();`.

Exiting a Branch (or Loop or the AL Flow)

If you want to exit a Branch, Loop, or Switch, or even built-in Branches like the AL Flow section, you use the `system.exitBranch()` method from a place where you can script, for example, a Hook, or even a Script Component. Calling `system.exitBranch()` with no parameters (or with an empty string) will cause the containing Branch to exit, and flow continues with the first component after the Branch.

You can also provide the method with a string parameter containing either:

One of the reserved keywords: Branch, Loop, Flow, Cycle or AssemblyLine (case insensitive)

This will break the first Branch of this type, tracing backwards up the AssemblyLine. So if your script code is in a Branch within a Loop, and you execute the call `system.exitBranch("Loop")`, you will exit both the Branch and the Loop containing it. Using the reserved word `Flow` causes the 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, while the `AssemblyLine` keyword will cause the AL to stop and shutdown.

All other values used in the `system.exitBranch()` call cause a break out of the branch/loop having the specified name. So, for example, the call `system.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, `system.exitBranch()` will cause processing to continue after the specified Loop/Branch.

The name of a Branch or Loop (case sensitive)

If you pass the name of a Branch or Loop in which your script call is nested, then control will pass to the component following it in the AL. If no Branch or Loop with this name is found, tracing backwards from the point of the call, then an error results.

There is also a *continue* functionality in Loop Components. The following methods are available in the system object:

```
system.continueLoop();
system.continueLoop(name);
```

where *name* is a case-sensitive string, indicating a Loop name. In the case where a Loop name is provided, the program flow is transferred to the Loop Component with that name.

Parsers

Parsers are used in conjunction with a byte stream component, for example, a File System Connector, to interpret or generate the structure of content being read or written.

Note that when the byte stream you are trying to parse is not in harmony with the chosen Parser, you get a `sun.io.MalformedInputException`. For example, this error message can show up when using the **Input Map** tab to browse a file.

The Config Editor provides two places where you can select Parsers:

1. In the **Parser** tab of a byte stream Connector.
2. From your own scripts; for example, Hooks and script components.

For more information about individual Parsers, see "Parsers" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Character Encoding conversion

Java2 uses Unicode as its internal character encoding. Unicode is a double byte character set. When you work with strings and characters in AssemblyLines and Connectors, they are always assumed to be in Unicode. Most Connectors provide some means of character encoding conversion. When you read from text files on the local system, Java2 has already established a default character encoding conversion which is dependent on the platform you are running.

The TDI server has the **-n** command-line option, which specifies the character set of Config files it will use when writing new ones; it also embeds this character set designator in the file so that it can correctly interpret the file when reading it back in later.

However, occasionally you read or write data from or to text files in which information is encoded in different character encodings. For example, Connectors that require a Parser usually accept a **Character Set** parameter in the Parser configuration. This parameter must be set to one of the accepted conversion tables as specified by the IANA Charset Registry (http://www.iana.org/assignments/character-sets).

Some files, when UTF-8, UTF-16 or UTF-32 encoded, may contain a Byte Order Marker (BOM) at the beginning of the file. A BOM is the encoding of the characters `0xFEFF`. This can be used as a signature for the encoding used. However, the TDI File Connector does not recognize a BOM.

If you try to read a file with a BOM, you should add this code to for example, the Before Selection Hook of the connector:

```
    var bom = thisConnector.connector.getParser().getReader().read(); // skip the BOM = 65279
```

This code will read and skip the BOM, assuming that you have specified the correct character set for the parser.

Some care must be taken with the HTTP protocol; see *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*, in the section about character sets encoding in the description of the HTTP Parser for more details.

Accessing your own Java classes

You can access your own custom Java classes from inside the IBM Tivoli Directory Integrator framework as long as the these are *public* classes and methods. These libraries must be packaged into a .jar or .zip file, and then be placed in the `TDI_install/jars` directory, preferably in your own subdirectory. You can also use the CLASSPATH environment variable or the Java runtime environment extension folder, but both of these methods are discouraged. These methods let you call classes from within your own classes only if the loader happens to load the classes before your own.

If you are running the server from the Config Editor, you must restart the Config Editor before it detects new classes in the `TDI_install/jars` directory and subdirectories.

After putting the .jar files in the jars subdirectory, you can create an instance of the class to refer to within the IBM Tivoli Directory Integrator. Note that the Java Function Component allows you to open .jar files, browse objects contained in them as well as their methods. Once you have chosen the function to call, the FC prepares your Input and Output schema to match the parameters needed by the Java function.

For more information about calling Java classes from script, see “Instantiating a Java class” on page 64.

Instantiating the classes using the Config Editor

Use the Java Libraries folder of the **Solution Logging and Settings** window in the Config Editor to declare your classes. This works only if your class has a no-argument constructor, which is usually but not always the default constructor.

When adding a class object click **Add...** and specify two parameters: the script object name, that is, the name of the scripting variable that is an instance of your Java class, and the Java class name. For example, you can have a Script Object Name *mycls* while the Java Class might be *my.java.classname*. The *mycls* object will be available for any AssemblyLines defined before the Global Prologs execute.

Note: Note that this causes your object to be instantiated for every AssemblyLine run. If this is not desirable, and if you prefer to instantiate on demand, then see the next section.

Runtime instantiation of the classes

If you want to instantiate your class at a specific point of execution or for the classes without no-argument constructors, you need to instantiate the class during run time. For example:

```
crytoLib = new com.acme.myCryptoLib();
```

AssemblyLine flow and Hooks

AssemblyLines provide built-in automated behavior that helps you rapidly build and deploy your data flows. This automated behavior is detailed in the Flow Diagrams in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*. In addition, Connectors and Functions have their own behaviors and these are also shown in the Flow Diagrams. Note that Connector behavior depends on the Mode setting.

Throughout these built-in logic flows are numerous *waypoints* where you can add your own scripted logic to extend built-in behavior, or to override it completely. These waypoints are called "Hooks" and are available for customizing under the Hooks tab of all Connectors and Functions, as well as of the AssemblyLine itself.

You can enable and disable Hooks according to whether a particular Hook is applicable to the AssemblyLine you are running. When you disable a Hook, you do not break its inheritance in the connector of which it is a part.

When an AssemblyLine is launched, it goes through three phases: Startup, Data flow and Shutdown. During Startup, Prolog Hooks are available for reconfiguring components before they are initialized. In the Data flow phase, each Work Entry fed into the AssemblyLine is passed down the Flow components for processing.

Finally, during Shutdown, Epilog Hooks can be used to carry out end-of-job work, like checking and reporting on error status, or storing state data for the next time the AL is started.

Startup Phase

At this point the server is instructed to load and run an AssemblyLine. The server uses the blueprint stored in the Config file to set up the AL. If a TaskCallBlock (TCB) is passed into the AssemblyLine, then its contents are evaluated (which can result in changes to AL component parameters). At this point, Prolog Hook flows are initiated.

Global Prologs

First, if any Global Prologs are defined then these are evaluated. Global Prologs are Scripts in the Resources folder in the Project that have been included (in the **Solution Logging and Settings** window) in the AssemblyLine. This is typically done in the AssemblyLine Settings window of the AssemblyLine by selecting the Scripts to run at AL startup. After all Global Prologs are finished, the AL Prolog Hooks are called.

AL Prolog Hooks (Before Initialize)

First the AssemblyLine Prolog - Before Initialize Hook is invoked. After this, all Connectors and Functions configured to Initialize "at Startup" go through their initialization phase, which also invokes their Prolog Hooks, as seen in the next point.

Connector/Function Initialization

The initialization sequence is performed for each Connector and Function with Initialization set to "at Startup." These are started in turn as defined by their order in the AssemblyLine. For each Connector or Function the flow is as follows:

1. The **Prolog – Before Initialize** Hook of the component is called.
2. The component is started; for example, connecting to its underlying data source, target or API.
3. For Connectors in Iterator mode, the **Prolog – Before Selection** Hook is processed, and the Connector performs the entry selection; this triggers a data source specific call, like performing an SQL SELECT or an LDAP search.
4. For Iterators, the **Prolog – After Selection** Hook is evaluated.
5. The **Prolog – After Initialize** Hook is called, ending the sequence.

If initialization fails for a Connector, then AssemblyLine flow passes to the **Prolog – On Error** Hook where you can deal with this error.

The Reconnect feature allows you to configure a Connector to automatically attempt to re-establish its connection if an error occurs during setup or data access. These settings are found under the Connector's **Connection Failure** tab.

Note: Script Connectors, that is, Connectors implemented using JavaScript, are evaluated at this stage so that required Connector functions are registered and initialization code is executed.

AL Prolog Hooks (After Initialize)

The **AssemblyLine Prolog - After Initialize** Hook is executed. Completion of this Hook signals the end of Start up Phase, and the beginning of Data flow Phase.

Data flow Phase

AssemblyLine Start of Cycle Hook

This Hook is invoked at the start of every cycle before Feeds or Flow components.

AssemblyLine Cycle

Control is passed to the first component in the flow, typically a Server or Iterator mode Connector in the Feeds section.

If you have one or more Iterators in the AssemblyLine, then the first one starts the cycle by retrieving the next entry from its result set and mapping Attributes into the Work Entry. The resulting Work Entry is passed to Flow section components, starting at the top of the list as seen in the CE.

For Server mode Connectors, a listener process is launched that waits for incoming client connections. When a connection request is detected, the Connector clones itself, accepts the connection and then switches itself to Iterator mode in order to feed data from the client into the Flow section for processing. Either way, you get an Iterator driving Work Entries to the Flow components. (Meanwhile, the original Server mode Connector waits for additional incoming connection requests.)

If your AssemblyLine neither has an Iterator Mode or Server Mode Connector, then you have a one-shot AssemblyLine typically used to process an Initial Work Entry (IWE) fed by another calling process.

End-of-Cycle

When the last Flow component is executed, one of three things can happen:

- If the current Work Entry came from an Iterator, control is passed back to the Iterator to get the next entry from its source.
- In the case of a server mode Connector, a reply is made to the client.
- For an AL that is called in manual cycle mode, the thread is passed back to the caller so that results can be accessed.

There is no specific Hook at this point, although this can be added to your AssemblyLine by inserting a Script at the end.

End-of-Data

End-of-data is an Iterator mode Hook that is called when the end of the input data set is reached. At this point, control is either passed to the next Feeds Connector, or the AssemblyLine goes into Shutdown Phase.

Shutdown Phase

At this point, AL processing has either completed normally, or aborted due to an error.

AssemblyLine Epilog - Before Close Hook

The AssemblyLine Hook called **Epilog – Before Close** is processed.

Connectors/Function Close flow

The Epilog Hooks of each Connectors and Function are called in the order that they appear in the Config Editor:

1. The **Before Close** Hook.
2. The close operation is carried out; for example, closing a connection or release an API callback.
3. The **After Close** Hook.

AssemblyLine Epilog - After Close Hook

Finally, the AssemblyLine **Epilog – After Close** Hook is run.

Server mode Connector Setup

When a Connector in server mode starts, it goes into event listening mode. Once an event is received, it clones the AL and resumes waiting for more events. In the clone, meanwhile, the Connector switches itself to Iterator mode and passes control to the next component in the Feeds list. This process allows you to have multiple server mode Connectors active and feeding data into the flow at the same time—one example would be to have several HTTP server Connectors in server mode listening to different ports, but feeding the same AL. Although server mode Connectors are part of an AssemblyLine configuration, they run as separate processes, as threads.

There is an additional set of Hooks that is evaluated for Connectors in this mode. The Hooks specific to server mode functionality for dealing with incoming connections are:

Before Accepting connection
This Hook is called before the Connector goes into listening mode.

After Accepting connection
Once a connection is received, this Hook is invoked. Note that no data is available at this time. In order to examine incoming event information, use the Iterator Hooks like **After GetNext** or **GetNext Successful**.

Error on Accepting connection
This Hook is executed if an error occurs in any of the server mode Hooks, or received from the data source during event listening.

- As mentioned previously, if you have more than one Connector in Iterator mode (see "Multiple Iterators in an AssemblyLine" on page 5), these Connectors are stacked in the order in which they are displayed in the configuration, from top to bottom. For example, if you have two Iterators, **a** and **b**, then **a** is called until it returns no more entries before the AssemblyLine switches to **b**.
- If you have no Connectors in Iterator mode, and no Initial Work Entry (IWE) is provided to the AssemblyLine when it is started; for example, by a calling from another AssemblyLine, and if no Work entry is created in an AssemblyLine or Connector Prolog Hook, then the AssemblyLine still performs a single pass.

Finally, there is a **Shutdown Request** Hook where you can put code that is processed if the AssemblyLine is closed down properly due to an external request to shut down (as opposed to one that crashes), enabling you to make it perform a graceful shutdown.

Special functions are available from the *system* object to skip or retry the current Work entry, as well as to skip over a Connector, and so forth. See "Controlling the flow of an AssemblyLine" on page 30 for more details.

Handle termination and cleanup for critical errors

There are a number of methods that allow detecting and handling of Tivoli Directory Integrator internal errors, as well as errors occurring in TDI Connectors, Parsers, Function Components and so on. These methods include:

- Error Hooks, in which JavaScript code can be written to handle an error. This method is accessible to Tivoli Directory Integrator users. Also see "Controlling the flow of an AssemblyLine" on page 30.
- Java try-catch-finally blocks, which make sure that a minor failure does not break the Server as well as that all errors are handled appropriately. Such blocks are already in place in the core Tivoli Directory Integrator server classes.

The JVM shutdown Hook feature improves the reliability of the Server. Java shutdown Hooks allow a piece of code to perform some processing after Control-C is pressed, or when the JVM is shutting down for some other reason, even System.exit.

You 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 using an optional property in the `global.properties` or `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: for example, sending a message that the Tivoli Directory Integrator server has terminated, carrying out clean up operations, or even restarting a new server if so desired.

Controlling the flow of an AssemblyLine

Hooks are programmable waypoints in the built-in automated behavior of IBM Tivoli Directory Integrator, where you can impose your own logic.

Hooks are found in AssemblyLines, Connectors and Function Components. For example, if you want to skip or restart parts of the AssemblyLine entirely, you typically do this from within a Hook in a Connector:

Note: The constructs below can be used to exit a Branch Component or Loop, too.

system.ignoreEntry()
Ignore the current Connector and continue processing your existing data entry with the next Connector.

system.skipEntry()
Skip (drop) the entry completely, aborting the current cycle, return control to the start of the AssemblyLine and get the next entry from the current Iterator.

system.exitFlow()
Drop further processing of the current entry, execute end-of-cycle logic; for example, save the Iterator State Key (if the Connector is configured for this), return control to the start of the AssemblyLine and get the next entry from the current Iterator.

system.restartEntry()
Restart from the beginning of the AssemblyLine, forcing the current Iterator to reuse the current entry.

system.skipTo(String name)
Skip to the named Connector.

system.abortAssemblyLine(String reason)
Abort the entire AssemblyLine with the specified error message.

Note: If you put any code in an **Error Hook** and do not terminate the current AssemblyLine or EventHandler, then processing continues regardless of how you got to the Error Hook. This means that even syntax errors in your script are ignored. So be sure to check the *error* object if you want to know what caused the error.

The methods described in the previous list can be regarded as goto-statements, in that no further code in this Hook is run. For example:

```
system.skipEntry(); // Causes the flow to change
// This next line is never executed.
task.logmsg("This will never be reached");
```

Note: There is a difference between an error Hook that is absent, and one that is empty – even though this may not always be easy to spot in the Configuration Editor. An *empty* error Hook causes the system to reset the error condition that caused the Hook to be called, after which the server continues processing, whereas an *absent* or undefined Hook causes the system to perform default error handling (typically aborting the AssemblyLine).

Expressions

IBM Tivoli Directory Integrator provides the v.6-compatible Expressions feature that allows you to compute parameters and other settings at run time, making your solutions dynamically configurable. This feature expands on the Properties handling found in previous versions.

In addition to support for simple External Properties references (fully compatible with earlier versions), Expressions provide more power in manipulating AssemblyLine and component configuration settings during AL or 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. IBM Tivoli Directory Integrator provides an Expression Editor to facilitate building these expressions.

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://docs.oracle.com/javase/1.6.0/docs/api/java/text/MessageFormat.html.

Note: The MessageFormat based Expressions shown in this section were the cornerstone of parameter substitution in Tivoli Directory Integrator v.6; in v.7 best practice is to use Advanced (JavaScript) expressions instead.

In addition to features described in the above class, Tivoli Directory Integrator provides a number of runtime objects that can be used in expressions—although the availability of some objects will depend on runtime state; for example, whether *conn* or *current* defined, or the *error* entry. The Expressions syntax provides a shorthand notation for accessing the information in these objects, like Attributes in a named entry object, or a specific parameter of a component.

Table 2. Script objects, their usage and availability.

TDI reference	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.
	This Advanced attribute map:
	`ret.value = work.getString`
	`("givenName") +`
	`" " +`
	`work.getString("sn");`
	can be expressed simply as:
	`{work.givenName} {work.sn}`
		AssemblyLine

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 current component AL. Furthermore, if *config* is used in the parameter of a Connector, Parser or Function, then it refers to the Config object *Interface* of that component, for example, 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

Table 2. Script objects, their usage and availability. (continued)

TDI reference	Value	Availability

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, set 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 script code ... // Must contain "return" EOF **Note:** v.6 syntax; use Advanced (JavaScipt) option in the Expression Editor instead	*Embedded* script code used to generate a value for the Expression. This script must return a value. The "EOF" text used here is an arbitrary string that terminates the JavaScript snippet. The JavaScript is collected 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 script engine instance of the AssemblyLine, so you have access to all variables otherwise present for scripting. **Note:** There is a shorthand 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.givenName + " " + work.surName}`	Always

Embedded JavaScript in Expressions have access to the script engine of the AssemblyLine. 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 script engine of the AL to see if it is defined there.

Expressions in component parameters

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

Table 3. Special objects usable in Expressions

Object	Value
config	The Interface configuration object of the component.
mc	The MetamergeConfig object of the Config instance (`config.getMetamergeConfig()`).
work	The Work entry of the AssemblyLine.
task	The AssemblyLine object.

As an example, take a JDBC Connector with the Table Name parameter set to "Accounts". You could then click on the **`SQL Select`** parameter label or the **Open parameter value dialog** button adjacent to the field, choose **Text w/substitution**, and then enter this into the large text 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:

```
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 a *uniqueID* Attribute in the Work entry with the value "42", the final result will be:

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

This evaluated result is not displayed in the CE. Simply entering curly braces will not cause Expression evaluation to be done for the parameter value. Instead, you have two choices when tying Expressions to parameters:

1. Press the **Open parameter value dialog** button adjacent to the field (or click on the Parameter label) and select **Text w/substitution** to open the Expressions dialog while in the parameter input field. You may enter your Expression in the large text field in this dialog. Click **OK** to enter your Expression.
2. Type the special preamble, @SUBSTITUTE, manually into the parameter input field, followed by the Expression. For example:

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

   Note: This last method of entering expressions directly is not recommended; use the Expression Editor instead.

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.

Table 4. Pre-defined objects for use in Expressions in LinkCriteria

Object	Significance
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 component itself, or a named component
alcomponent	The Connector or Function 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 Expression used in Link Criteria:

```
{work.matchAtt}  EQUALS {work.uid}
```

is equivalent to this:

```
uid  EQUALS $uid
```

Expressions in Branches, Loops and Switch/Case

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

Table 5. Pre-defined objects for use in Expressions in Branch Components

Object	Significance
config	The Interface configuration object of the component
mc	The MetamergeConfig object of the Config instance (`config.getMetamergeConfig()`)
work	The Work Entry of the AssemblyLine
task	The AssemblyLine
alcomponent	The Connector or Function Component

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.

Scripting with Expressions

You can also use Expressions 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));
```

The expression that results from the JavaScript code issues the following log messages when run for several iterations in the test AssemblyLine:

```
14:35:29  Patty S Duggan -> duggan
14:35:29  Nicholas P Butler -> butler
14:35:29  Henri T Deutch -> deutch
14:35:29  Ivan L Rodriguez -> rodriguez
14:35:29  Akhbar S Kahn -> sahmad
14:35:29  Manoj M Gupta -> gupta
```

The Entry object

One of the cornerstones of understanding Tivoli Directory Integrator is knowing how data is stored and transported within the system. This is done using an object called an *entry*. The entry object can be thought of as a "Java bucket" that can hold any number of Attributes: none, one or many.

Attributes are also bucket-like objects in Tivoli Directory Integrator. Each Attribute can contain zero or more *values*, these being the actual data values that are read from, and written to, connected systems. Attribute values are Java objects as well; they can be strings, integers and timestamps; whatever is needed to match the native type of this data value. A single Attribute can readily hold values of different types. However, the values of a single Attribute will tend to be of the same type in most data sources.

Although this *entry-attribute-value* paradigm matches nicely to the concept of Lightweight Directory Access Protocol (LDAP) directory entries, this is also how rows in databases are represented inside Tivoli Directory Integrator, as are records in files, IBM Lotus Notes documents and HTTP pages received over the network. All data, from any source that Tivoli Directory Integrator works with, are stored internally as entry objects with Attributes and their values.

Contrary to earlier versions of Tivoli Directory Integrator, from v7.0 hierarchical Entry objects are supported, in the AssemblyLine and by some of the components that can be part of an AssemblyLine. The Entry object is extended to provide several convenient methods for dealing with hierarchical data, although by default, this is hidden and only comes into play if you explicitly enable it, or use it with components that require the hierarchical features. It also implements org.w3c.dom.Document, which makes it the top level Node in the hierarchy. For more information on this, see “Working with hierarchical Entry objects” on page 40.

There are a handful of entry objects that are created and maintained by Tivoli Directory Integrator. The most visible instance is called the *Work entry*, and it serves as the main data carrier in an AssemblyLine (AL). This is the bucket used to transport data down the AL, passing from one component to the next.

The Work Entry is available for use in scripting through the pre-registered variable *work*, giving you direct access to the Attributes being handled by an AssemblyLine (and their values). Furthermore, all Attributes carried by the Work entry are displayed in the Config Editor, under the **Work Attribute** header in the Attribute Maps area in the AssemblyLine Editor window of an AssemblyLine.

Entry types

There are a number of data objects that reside in AssemblyLines that follow the Entry data model. These are:

Work This is the aforementioned Entry that travels from component to component in the AssemblyLine, and carries data between them. Its pre-registered variable name is *work*, and is available for use in scripting almost everywhere[6].

Conn This is the Entry-like object that a Connector uses as an intermediary between the connected system and the AssemblyLine, before you make the data, or a subset thereof, available in the

[6] When the AssemblyLine you are working with is not called with an Initial Work Entry, then the *work* object is not available until *after* the Prolog hooks. In the Prolog hooks you can have code as follows:

```
if (work != null) {
 // An Initial work Entry has been provided, we can get values from there
 .... some code
} else {
 // No initial work Entry has been provided
 ... some other code
}
```

Work Entry. The process of moving data between Conn and Work is called Attribute Mapping. Its pre-registered variable name is *conn*, and is available for use in scripting inside many of the Hooks in Connectors and Function Components.

Current
This Entry-like object is available inside certain Hooks in Connectors in Update mode, and holds the data from the connected system before any updates have been applied to it. Its pre-registered variable name is *current*.

Error This Entry-like object only exists in certain Hooks in Components, when some error condition has occurred and the relevant Error Hook has been invoked. It contains information about the actual exception that was thrown, with possibly some additional variables and data, enabling you to pinpoint what exactly caused the error. Its pre-registered variable name is *error*.

The Connector Flow diagrams in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide* will show you which of these objects are available under which circumstances.

See also

"Internal data model: Entries, Attributes and Values" on page 38.

Chapter 2. Scripting in TDI

IBM Tivoli Directory Integrator provides its users with a highly-flexible engine that can be customized both from the user interface controls of the Configuration Editor, as well as through scripting of custom logic. While the user interface controls provide a means of controlling the data flow at a higher level, scripting provides you with the ability to control almost any aspect of the data flow at any level, including overriding standard Tivoli Directory Integrator processing. Special functions are available in the *system* object to reiterate on an AssemblyLine entry, skip a Connector and start new AssemblyLines. The scripting language used for implementing this custom logic is JavaScript.

Ready-to-use, Tivoli Directory Integrator provides the tools to quickly snap together the framework of an integration solution. However, for all but the most trivial migration jobs, you will need to customize and extend the built-in behavior of the product by writing JavaScript.

Tivoli Directory Integrator is pure Java. Whenever you issue a command to Directory Integrator, work with components and objects, or manipulate data in your flow, you are working with Java objects. IBM Tivoli Directory Integrator v7.1.1 uses IBM Java version 1.6.

Your customization on the other hand is done in JavaScript, and this marriage of two ostensibly similar, yet fundamentally different, programming languages warrants closer examination.

Experience with JavaScript will be very helpful. The examples provided may add to your experience. However, this manual does not teach JavaScript itself , merely its application in TDI. You will need to secure your JavaScript reference materials elsewhere.

There are a number of commercially available reference guides to JavaScript, as well as documentation, tutorials and examples on the net. Note however that much of the JavaScript content out on the Web is related to beautifying and automating HTML content. You need only concern yourself with the core language itself, as it is described at the following link: http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.5/guide/index.html

There is also a handy link on this site for downloading the reference in HTML format for installation locally. An excellent handbook on JavaScript is *The Definitive JavaScript Guide*, 4th Edition by David Flanagan (O'Reilly).

You will also want the Javadocs for Java as well, since all TDI objects, as well as the data values inside your solution, are in the form of Java objects. These documents are located online at this URL: http://docs.oracle.com/javase/1.6.0/docs/api/index.html

The J2SE documentation itself can be found here: http://docs.oracle.com/javase/1.6.0/docs/index.html

Scripting is necessary when you need to add custom processing to your AssemblyLine. Examples of where scripting can be helpful include the following tasks:

Attribute manipulation or computation
You need to calculate the value of an output attribute based on one or more input attributes.

Data filtering
You want to process only entries that match a particular set of criteria.

Data consistency or validity checking
You need to report or correct invalid data values.

Flow control
You want to override the update operation of the Connector you are using.

Initialization

You want to run some initializing procedures before your AssemblyLine starts.

Each of these cases mentioned, and many others not mentioned, usually require scripting.

Examples

Go to the `examples/scripting` subdirectory of your IBM Tivoli Directory Integrator installation.

Internal data model: Entries, Attributes and Values

When Tivoli Directory Integrator components access information from connected systems, they convert the data from system-specific types to an internal representation using Java objects. On output, components convert the other way, going from this internal data model to the native types of the target system. This same internal representation is used when you wish to pass data to and from AssemblyLines. It is therefore vital that you understand how the Tivoli Directory Integrator internal data model works.

Looking in detail at when a data value is received by a component, a corresponding Tivoli Directory Integrator *Attribute* object is created using the name of the attribute being read. The data value itself (or values, if it is a multi-valued attribute) are converted to appropriate Java objects—like `Java.lang.String` or `java.sql.Timestamp` —and assigned to the Attribute. If you take a look in the Tivoli Directory Integrator API documentation, you will see that the Attribute object provides a number of useful methods, like `getValue()`, `addValue()` and `size()`. This allows you to create, enumerate and manipulate the values of an Attribute directly from script. You can also instantiate new Attribute objects as needed, as shown in this Attribute Map example for advanced mapping the objectClass attribute of a directory:

```
var oc = system.newAttribute( "objectClass" );

oc.addValue( "top" );
oc.addValue( "person" );
oc.addValue( "organizationalPerson" );
oc.addValue( "inetOrgPerson" );

ret.value = oc;
```

Attributes themselves are collected in a data storage object called an *entry object*. The *entry* is the primary data carrier object in the system and Tivoli Directory Integrator gives you access to important entry objects by registering them as script variables. A prime example is the Work entry object in the AssemblyLine, used to pass data between AL components (as well as between AssemblyLines). This entry object is local to each AssemblyLine and available as the script variable *work*.

Tivoli Directory Integrator provides some shortcuts and convenience features when working in JavaScript, so the above specific advanced mapping can be simply coded as follows:

```
ret.value = [ "top", "person", "organizationalPerson", "inetOrgPerson" ];
```

The advanced mapping feature supports JavaScript arrays and Entries for passing multiple attribute values.

For example, in an Input Attribute Map (which causes mapped Attributes to show up in the *work* Entry on return), suppose you have the assignment

```
ret.val = anentry;
```

for the Attribute called "last". Let us further assume that *work* is empty to start with, and *anentry* contains the Attributes "cn", "sn" and "mail".

After attribute mapping *work* will contain "cn", "sn" and "mail" attributtes, **not** a single Attribute called "last" with "anentry" as value. In essence, what happens in Attribute mapping is that when an attribute map returns an Entry object, it is merged with the receiving Entry – either *work* or *conn*, depending on what map it is (Input or Output).[7]

Looking at the Javadocs, you will see that the entry object offers various functions for working with Entries and their Attributes and values, including `getAttributeNames()`, `getAttribute()` and `setAttribute()`. If you wanted to create and add an Attribute to the AssemblyLine Work entry, you could use the following script, for example, in a Hook or a Script Component:

```
var oc = system.newAttribute( "objectClass" );

oc.addValue( "top" );
oc.addValue( "organizationalUnit" )

work.setAttribute( oc );
```

Note that in this case we do **not** have the option of using a JavaScript array to set the value:

```
oc.addValue( ["top", "organizationalUnit"] ); // Does not work like Advanced Mapping
```

This code will result in the `oc` attribute getting a single value, which in turn is an array of strings.

Entry objects can also contain *properties*. Properties are data containers like Attributes, except that they are only single-valued. While Attributes are used to store data content, properties hold parametric information, allowing you to keep this information separated. Properties do not show up for attribute map selection or in the Work entry list, but can be accessed much like Attributes from script. entry functions like `getProperty()` and `setProperty()` are used for this, and these work directly with Property values, which can be any type of Java object, just like Attribute values. There is no intermediate Property object as there is when you work with Attributes.

In many cases, you can restrict the data model to an entry containing zero or more Attributes, each with zero or more values—a flat schema.

This is one of the strengths of Tivoli Directory Integrator: simplifying and harmonizing data representations and schema. It also represents a challenge when you need to handle information with a more complex structure. However, since an Attribute value can be any type of Java object, including another entry object (with its own Attributes and values), Tivoli Directory Integrator allows you to work with hierarchically structured data.

This more elaborate and structured way of handling hierarchical objects is described in “Working with hierarchical Entry objects” on page 40.

[7] In Tivoli Directory Integrator V7.1.1, taking advantage of hierarchical objects, you can circumvent this behavior by first encapsulating an Entry in an Attribute before Attribute Mapping takes place. For example,

```
// this is the entry to return
e = system.newEntry();
e.setAttribute("some", "value");

// Create an Attribute object. We don't need to provide a name since the mapping will use current map's name.
attr = system.newAttribute(null);

// add the entry to the Attribute object and return that instead of the Entry object
attr.addValue(e);

return attr;
```

If this was entered as the Advanced Attribute Map for the "last" Attribute, then after Attribute Mapping, the *work* Entry will now contain an Attribute called "last". This Attribute is an Entry, in turn comprised of two attributes called "some" and "value".

Working with hierarchical Entry objects

An alternative way of working with hierarchically structured data is to take advantage of the support for hierarchical objects in the Tivoli Directory Integrator entry object.

Different from earlier versions, Tivoli Directory Integrator v7.1.1 supports the concept of the hierarchical Entry object. The Entry object represents the root of the hierarchy and each Attribute represents a node in that hierarchy. Following this logic the values of every Attribute are leafs in the hierarchy. An API for traversing the hierarchy exists as well. This API is a partial implementation of the DOM 3 specification. Only a few classes from that specification have been implemented:

- `Org.w3c.dom.Document` – implemented by the Entry class.
- `Org.w3c.Element` – implemented by the Attribute class.
- `Org.w3c.Attr` – implemented by the Property class.
- `Org.w3c.Text` and `org.w3c.CDATASection` – implemented by the AttributeValue class.

These classes are the minimum set of classes provided by the DOM specification that are needed to represent a hierarchical data. The pre-v7.0 API is not hierarchy aware (for example, it cannot access/modify/remove child Elements) for backward compatibility reasons. This is why only the DOM API can manipulate a hierarchical structure.

To keep the Entry structure backward compatible by default the Entry always uses flat Attributes. The Entry only becomes hierarchical on demand – after you call one of the newly provided DOM APIs. This allows only components aware of the hierarchical nature of the Entry to make use of it, the rest of the components don't need to be changed in order to keep running. Starting from Tivoli Directory Integrator v7.0, a new name notation is introduced in order for users to have an easier way of creating hierarchical trees. Every name containing a dot in it is thought to be a composite name compound of simple names; these names are separated by dots. When such a composite name is passed to a hierarchical Entry, it breaks it down to simple names and builds the hierarchy that is described by that composite name.

For example if you run the following JavaScript code:

```
// create a new empty Entry object
var entry = new com.ibm.di.entry.Entry(true);
// create a new branch of 2 levels
entry.setAttribute("firstLevelChild.secondLevelChild", "level2Value");
// finds the already existing branch and creates a new node on level 3
entry.setAttribute("firstLevelChild.secondLevelChild.thirdLevelChild", "level3Value");
```

the following structure will be created:

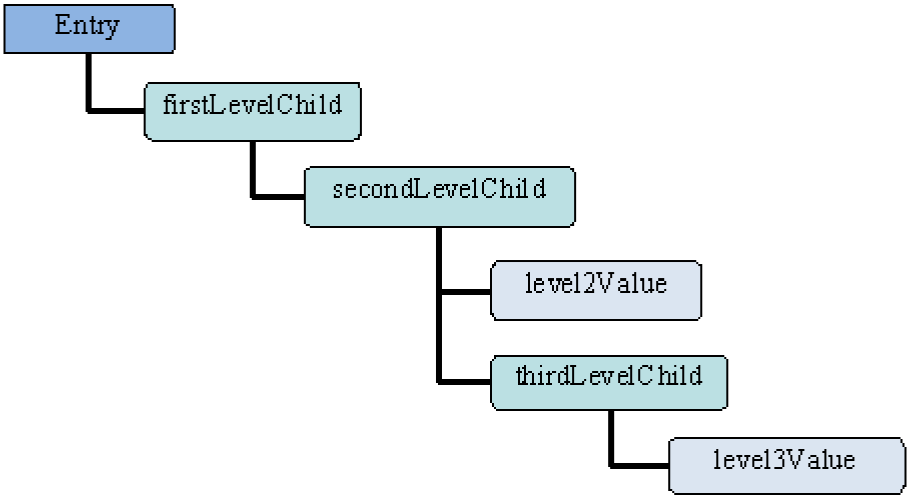


Figure 1. Simple hierarchical entry

Note: It is important to know that the names are not broken down to simple names until the Entry structure is not converted to a hierarchical one. For example if the entry is a flat one and only the old methods are used that would still create the same old flat structure:

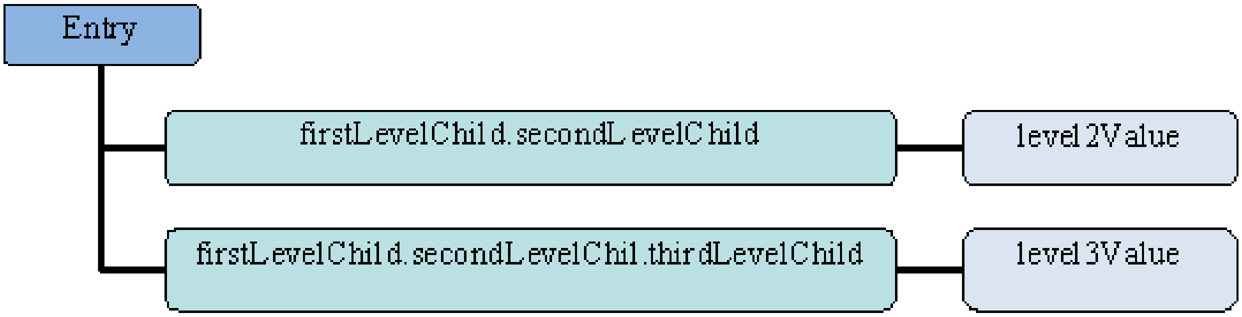

Entry
firstLevelChild.secondLevelChild
level2Value
firstLevelChild.secondLevelChil.thirdLevelChild
level3Value

Figure 2. Traditional, flat entry

In some cases it may be necessary to have dots in the names of the Entry's attributes, so the Entry object also understands escape characters (currently only \\ and \. are supported).

Note: When working from script the backslash should be properly escaped with another backlash! For example if the following hierarchy is needed:

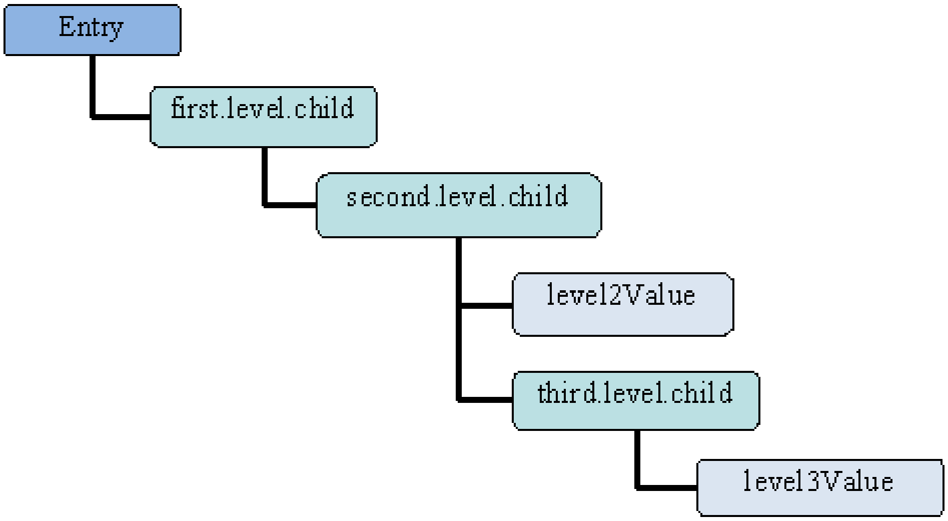


Figure 3. Another simple hierarchical entry

the following script would create it:

```
var entry = new com.ibm.di.entry.Entry(true);
entry.setAttribute("first\\.level\\.child.second\\.level\\.child", "level2Value");
entry.setAttribute("first\\.level\\.child.second\\.level\\.child.third\\.level\\.child", "level3Value");
```

In order to maintain backward compatibility with previous releases of Tivoli Directory Integrator when implicitly working with hierarchical Entries, all the old methods exposed by both the Attribute and the Entry classes have changed slightly. For example, the `Entry.getAttributeNames()` method will return an array of full paths to the leafs that are available in the tree. In reference to the above structure the getAttributeNames method returns the array:

```
["first\.level\.child.second\.level\.child", "first\.level\.child.second\.level\.child.third\.level\.child"]
```

The `Entry.size()` method returns the total number of elements in the array returned by `getAttributeNames()` method. In our case the size() method would return 2 (the number of leafs in the whole tree) instead of 1 (as you might expect, considering that the Entry object has only one attribute as a child).

This is because both getAttributeNames() and size() methods only work with flat structures. In order to get the real size of the children you would need to use the DOM API as follows:
`Entry.getChildNodes().getLength();`

If the entry is a flat one the old API will behave as in previous releases.

The Attribute object

The Attribute object is enhanced with the ability to create hierarchical structures. These structures follow the DOM specification and that is why the Attribute object is aware of XML Namespace concepts.

The Attribute class also had to be expanded in order to provide new methods for the hierarchical functionality. The getValue/setValue/addValue methods are also backward compatible and will return only the values the particular element has. The difference here is when each value is accessed through the DOM API it will be wrapped in an AttributeValue class and will be treated as a Text node. In order to get the child elements of an Attribute (for example, Attributes and AttributeValues) the DOM API has to be used.

The Attribute child of an Entry is also able to switch the Entry's structure to hierarchical one when any of its DOM methods are accessed. Unlike the Entry class the Attribute class does this implicitly and does not provide a way to do the switching explicitly.

Tivoli Directory Integrator v7.0 supports extensions to scripting capabilities of the Server to easily access complex structures. Please see the section “Navigation within scripts” on page 44 for more details.

The AttributeValue object

This class represents a value in the hierarchical tree. As per DOM specification the values in the tree are always Strings. The AttributeValue class however, has held objects of any kind for the last few releases. This is valid in the current version as well. The only difference for the AttributeValue object is that when accessed through DOM it will return a String representation of the contained object. In order for the AttributeValue class to represent a Node and have a value at the same time in the terms that the DOM specification defines, it must implement either the org.w3c.dom.Text or org.w3c.dom.CDATASection interfaces. The AttributeValue implements them both, and can represent either of these Nodes depending on your needs.

The Property object

Attributes can have zero, one or more Property objects. The Property class implements the org.w3c.dom.Attr interface and thus represents the attributes in terms of DOM concepts. Using properties you may declare prefixes/namespaces in terms of XML concepts.

Transferring Objects

Mapping an Attribute from one entry to another will always copy the source Attribute.

For example:

```
entry.appendChild(conn.getFirstChild());
// or
entry.setAttribute("name", conn.getFirstChild());
```

Even when the Attribute is not a first-level child of the Entry it is still copied. This can also be accomplished by the script:

```
entry.a.b.c.d.appendChild(conn.e.f.g);
```

In order to move an Attribute object between entries without cloning it you will need to first detach it from its old parent and then attach it to a new parent.

For example:

```
var src = entry1.b.source;
entry1.b.removeChild(src);
entry2.a.target.appendChild(src);
```

When moving an Attribute object from one parent to another parent in the same Entry the Attribute is automatically moved. No cloning is done.

For example:

```
entry.a.target.appendChild(entry.b.source);
```

In this example the "source" Attribute is detached from its parent ("entry.b") and then attached to the "entry.a.target" Attribute. No cloning is done.

If you do not want to remove the Attribute object from the source then you can append a copy of the Attribute like this:

```
entry.a.target.appendChild(entry.b.source.clone());
```

Navigation within scripts

The Tivoli Directory Integrator ScriptEngine enables you to easily access the attributes of an entry just by referring to them by name; for example, `entry.attrName` returns the attribute with name *attrName*.

1. The JavaScript Engine resolves names based on the context object the name was requested on. For example, if the call `entry.a` is performed, the *entry* name is the context object and *a* is the name of the child object to resolve. The JavaScript Engine uses left-to-right interpretation to evaluate each context object until the final one is resolved. Based on the diagram below the following call, `entry.a.b.c`, is resolved using this procedure: Find the *entry* object to use it as the context object for the first step.
2. Search the context object for a name *a*. The *entry* object has only one child with name *a*. Consider that child to be the next context object for the next step.
3. Search the context object for a name *b*. The context object has two children named *b*. Put them in a list and return that list.
4. The final operation searches the list returned in the previous operation for the name *c*. Each element in the list has at least one such child. Get them all and put them in a list, which is the actual result of resolving the full expression.

The following entry object example diagram illustrates this:

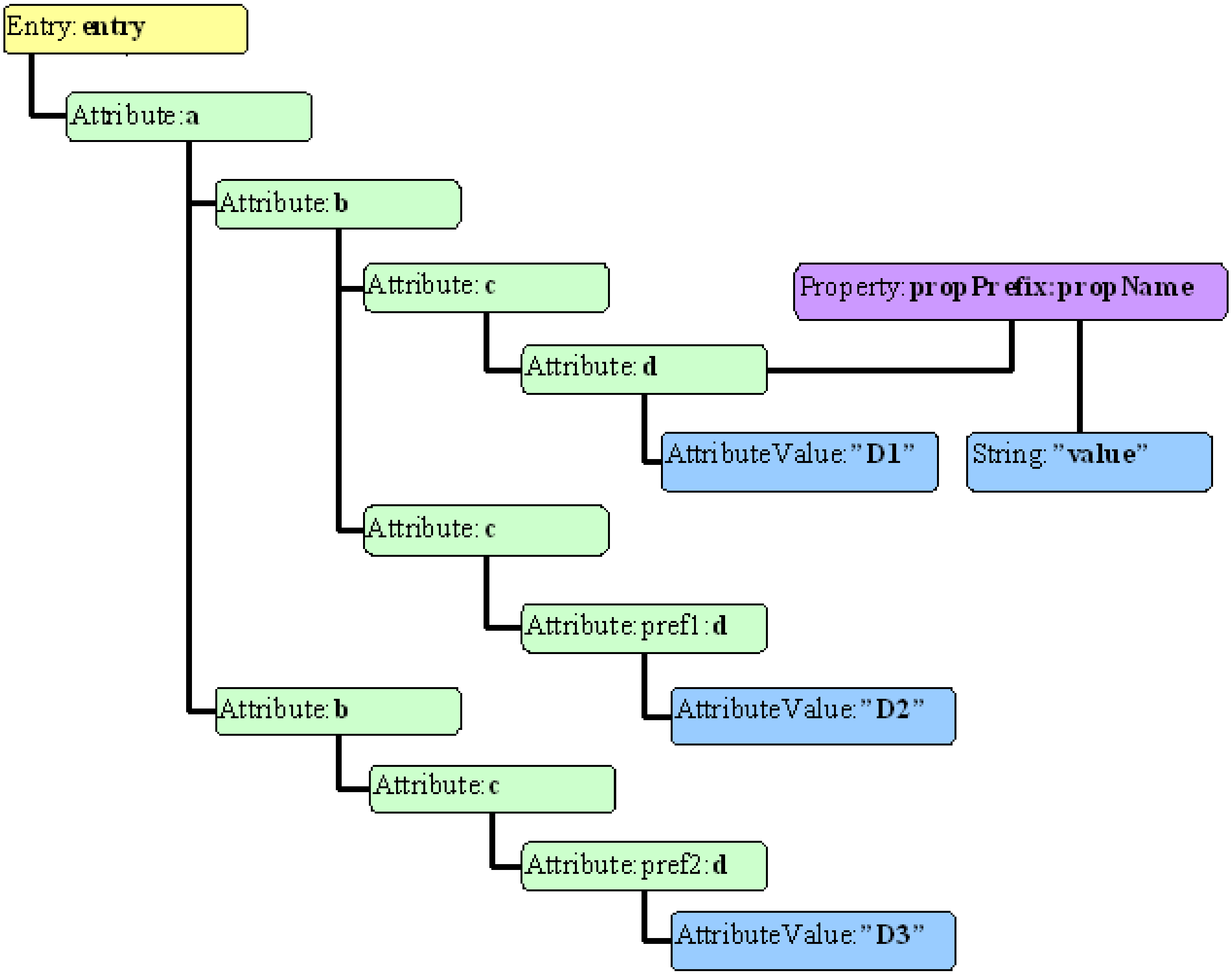


Figure 4. Hierarchical Entry object example

The script engine provided by IBM Tivoli Directory Integrator allows more arbitrary names to be used in child resolving process. For example if the name of the child contains dots then you can refer to it using the square-bracket syntax as shown below:

```
work["{namespace}name:Containing\.invalid\.charaters"]
```

Notice that the dots are being escaped to denote that they are part of the local name and that they must not be treated as path separators by the script engine.

Depending on the current context object on which an operation is performed the end result might differ. Tivoli Directory Integrator V7.1.1 adheres to the standard object manipulation the JavaScript Engine provides by implementing several enhancements to the following objects:

Entry When the context object is an instance of this type, the name resolving mechanism will look for the following syntax:

- `@<name>` – searches the Entry object for a property with the specified name. The resolved object is either null or the object mapped to that property.
- `<prefix>:<localName>` – searches the Entry object for a child that have a prefix equal to <prefix> and a Local Name equal to <localName>. The resolved object could be either null or an existing Attribute object.
- `<localName>` – searches the Entry object for for the first child that has a Local Name equal to <localName>. The resolved object could be either null, or an Attribute object.

- `{namespaceURI}<localName>` – searches the Entry for the first child Attribute that belongs to the specified namespaceURI and have the same Local Name as the specified name.

  Note: If a prefix is provided, it will be ignored and the name resolving mechanism will only look for the specified namespaceURI and localName. The resolved object could be either null or an Attribute object.

Attribute

When the context object is an instance of this type, the name resolving mechanism will look for the following syntax:

- `@<prefix>:<localName>` and `@<localName>` – searches the Attribute for Property objects that have the same prefix and/or local name or just the specified local name. Returns either null if no properties match the specified name, or a single Property object.
- `@{namespaceURI}<localName>` – searches the Attribute for a Property that belongs to the specified namespaceURI and have the same Local Name as the specified name.

  Note: If a prefix is provided it will be ignored and the name resolving mechanism will only look for the specified namespaceURI and localName. The resolved object could be either null or a Property object.
- `[<index>]` – specifies the position of the value in the Attribute to retrieve. Using this notation you cannot access a child of this Attribute. Returns either null, or the Object at the specified position.
- `<prefix>:<localName>` and `<localName>` – searches the Attribute for a child(ren) with the specified prefix and/or local name. Returns either null or the child with the specified name (if only one), or a NodeList with all the child Attributes that match the criteria.
- `{namespaceURI}<localName>` – searches the Attribute for all the children Attributes that belong to the specified namespaceURI and have the same Local Name as the specified name.

  Note: If a prefix is provided it will be ignored and the name resolving mechanism will only look for the specified namespaceURI and localName. The resolved object could be null, a single Attribute object or a NodeList containing all of the Attributes that match the search name.

NodeList

When the context object is an instance of this type the name resolving mechanism will look for the following syntax:

- `[<index>]` – specifies the position of the element in the NodeList to retrieve. Returns either null, or the Object at the specified position. Throws exception if the index is out of the bounds.
- `@<prefix>:<localName>` and `@<localName>` – searches each of the elements of the NodeList for a property that has the same prefix and/or local name. Returns either null, a Property object (if only one found), or a List of all the Property objects found in the NodeList.
- `@{namespaceURI}<localName>` – searches each of the Attributes for a Property that belongs to the specified namespaceURI and have the same Local Name as the specified name.

  Note: If a prefix is provided it will be ignored and the name resolving mechanism will only look for the specified namespaceURI and localName. The resolved object could be either null or a Property object (if only one found) or a NodeList of all the Property objects found in the NodeList.
- `<prefix>:<localName>` and `<localName>` – searches each of the elements of the NodeList for a child that has the same prefix and/or local name. Returns either null, an Attribute object (if only one found), or a NodeList of all the Attribute objects found in the NodeList.
- `{namespaceURI}<localName>` – searches each of the Attributes for all the children Attributes that belong to the specified namespaceURI and have the same Local Name as the specified name.

Note: If a prefix is provided it will be ignored and the name resolving mechanism will only look for the specified namespaceURI and localName. The resolved object could be null, a single Attribute object or a NodeList containing all of the Attributes that match the search name.

You now have the following options:

1. Ability to access all the *d* elements by referring to them starting from the top; for example, `entry.a.b.c.d` – this returns an object of type NodeList with all the *d* attributes that match that path. In our example this will return all the three *d* elements.
2. Ability to access attributes by specifying both prefix and local name; for example, `entry["a.b.c.pref1:d"]` – this will return a single Attribute only, the one with a prefix of "pref1".
3. Ability to access each Attribute of an NodeList using the [ ] notation, for example, `entry.a.b.c.d[0]` – this returns a single Attribute, namely the first *d* element of the structure above.
4. Ability to navigate through elements using the [ ] notation; for example, `entry.a.b[0].c.d` – this returns a List of Attributes, but this time it contains all the *d* attributes form the first *b* branch.
5. Ability to get the property of an attribute (that is, an Attribute of an Element using the DOM naming convention) using the @ notation; for example, `entry.a.b.c.d[0]["@propPrefix:propName"]` – this returns us a String object containing the value of that property (that is, the Attribute's value according to DOM). Note here that the propPrefix and the propName are separated by the colon sign (":"), and that if the property has a prefix, it is mandatory that both the prefix and the name be specified in order for the property to be found. Alternatively the namespace and the propertyName can be used to find a property that has a prefix.
6. Ability to call methods of an Attribute before searching for child with the name of the method that the user wants to execute; for example, `entry.a.b.[0].getChildNodes()` – this returns a NodeList object that holds all the Attribute values that the first *b* Attribute contains.
7. Ability to access entry attributes by name; for example, `entry.attrName` – this will return the Attribute mapped to the *attrName* key.
8. Ability to access entry properties using the .@ notation; for example, `entry.@propName` – this will return the Object mapped as property to the *propName* key.
9. Ability to access child nodes by specifying the namespace those children belongs to. For example `work.a.b[{someNamespace}c]` – this will return all the *c* objects that belong to the "someNamespace" namespace.

Note:

1. If the name of an attribute is the same as the name of a method of an Entry/Attribute then the method will be called. If you want to access the attribute, you must use the object's methods like before (that is, `Entry.getAttribute("getAttribute");`.)
2. If a flat entry is used the script engine will not convert the entry to a hierarchical unless the namespace of the attribute to find is specified. For example: `entry["{ns}element"]`. The script engine will automatically convert the entry to a hierarchical one if the context object is of type Attribute. For example in this case: `entry.attr.child`.
3. If the attribute to find contains dots in its name and the entry is flat the you must use the bracket notation instead of the dot notation, or force the entry to become hierarchical before resolving the child node. For example if the entry is flat you cannot access the attribute `http.body` using the call `entry.http.body`. For this to happen you will have to use this instead: `entry["http.body"]` or call `entry.enableDOM()` prior to calling `entry.http.body`.
4. If you intend to use the reference retrieved from the expression, for example, `a.b.c.d` more than once, then it is good practice to assign that reference to a local variable since each repeating evaluation of the same expression will result in additional overhead for retrieving the same reference.
5. If, for example, the expression `entry.a.b.c[0].d` is used, then this will refer to an Attribute object; but if `entry.a.b[0].c.d` is used, then this will refer to a NodeList object. In order to recognize the referenced object you can check the name of the object, for example:

```
var obj = a.b.c.d;
if (obj.getClass().getSimpleName().equals("Attribute") ) {
  // handle this as Attribute
} else {
  // handle this as a list of Attributes
}
```

If, for example, you need to handle each element returned by an expression, even if only one element is returned then you can use a for/in loop structure like this:

```
for (obj in entry.a.b.c.d) {
  // the obj will be an object of type Attribute
}
```

The same usage is now available with the Entry object, for example

```
for (obj in work) {
  // the obj will be an Entry's attribute
}
```

6. ScriptEngineOptions also allows assigning values. For example the following expressions are valid:

   entry.a.b.c.d[0]["@propPrefix:propName"] = "new value";
   This changes the value of the property. If the property does not exist then it will be created.

   entry.a.b.c.d[0][0] = "new value";
   This replaces the attribute value on position 0.

   entry.a.b.c.d[0][1] = "another value";
   This adds another value to the attribute (if the index is the same as the size of the array of values.)

   entry.a.b.c.d[0][a.b.c.d[0].size()] = "appended value";
   If you need to append a new value to the list of objects, but do not know the last element position you can use `Attribute#size()` to get the number of children this element has.

   new com.ibm.di.entry.Entry().@propName = "someValue";
   This will create a new property in the Entry object with name propName if it does not exist, and will set its value to the String "someValue".

   entry.a.b.c[1] = "value";
   This will resolve *entry.a.b.c* as a NodeList and will automatically add as a value the new String object to the second element of the resolved NodeList object. If, for example, *entry.a.b.c* resolves to an Attribute, then the new String value will replace the second value of the resolved *c* Attribute.

   entry.a.b.c[2]["pref3:d"] = "value";
   This will add a new *pref3:d* child to the third *c* attribute from the *entry.a.b.c* list. Note that *entry.a.b.c[2]* resolves to an Attribute and to a list.

   entry.a.b.c["{namespaceURI}:d"] = "myTextValue";
   This sets a value to the first child Attribute of *c* that has a local name equals to *d* and belongs to the namespace `namespaceURI`. If no such attribute is found, a new one is created and the value is assigned to it. When creating an attribute, you might as well associate the namespace with a prefix. In our case this could be done with this:
   `entry.a.b.c["{namespaceURI}prefix:d"] = "myTextValue";`.

   entry["{http://ibm.com/xmlns/}first.second.third"] = null;
   This will first try to find the element with local name "first" from the namespace "http://ibm.com/xmlns/". When it fails it will create the element and then will try to resolve its child element "second". When it fails it will create it and set its namespace to the one of the first element. Finally an attempt to resolve the third element will be made. When it fails the last element will be created with no values. This is equivalent to `entry["{http://ibm.com/xmlns/}first.{http://ibm.com/xmlns/}second.{http://ibm.com/xmlns/}third"] = null;`

Creating the above structure with script

```
// create a new entry that will hold the structure
var entry = new com.ibm.di.entry.Entry();
// create the first branch
var d = entry.newAttribute("a.b.c.d");
// create a new property and assign it a value
d["@propPrefix:PropName"] = "value";
// create a new value of the d Attribute
d[0] = "D1";
// append a new value for the entry.a.b attribute
entry.a.b.appendChild(entry.createElement("c"));
// create a new Attribute d with a prefix on the second c attribute,
// and assign the string "D2" as a first value
entry.a.b.c[1]["pref1:d"] = "D2";
// create a new child of the a Attribute
entry.a.appendChild(entry.createElement("b"));
// choose the second attribute from the entry.a.b NodeList and create a new child named c
entry.a.b[1].appendChild(entry.createElement("c"));
// create the d child of the c Attribute
entry.a.b[1].c["pref2:d"] = "D3";
```

Navigation using XPath

The Entry class provides convenient methods for navigating and retrieving data based on XPath expressions. Using XPath for querying data from an Entry is much more advanced than using any simple navigation within scripts. It is much easier to implement a searching and/or a value matching logic in a single expression than writing multi-lined scripts in order to achieve the same results.

The Entry class provides the following methods:

- `NodeList getNodeList (String xPath);`
- `Attribute getFirstAttribute(String xPath);`
- `String getStringValue(String xPath);`
- `Number getNumberValue(String xPath);`
- `Boolean getBooleanValue(String xPath);`

Flattening hierarchical structures

When working with hierarchical data it is sometimes necessary to flatten the nodes of the hierarchy. To do that you can either write a script for traversing the tree, or use one of the methods:

- `getElementsByTagName(String namespace, String localName);`
- `getElementsByTagName(String tagName);`

These methods traverse the tree and search for elements with the specified name and namespace. The DOM API allows these methods to accept the character "*" for both names and namespaces. That character represents a wildcard, which is used to match any element name/namespace. Using that character for name flattens the tree by returning all the Attribute nodes in a NodeList. It should be noted that the structure of the hierarchy has not been changed. The returned NodeList is just a container for the Element nodes that were found in the tree.

If you run the following script on the entry from the structure in section "Navigation within scripts" on page 44:

```
var list = entry.getElementsByTagName("*");
```

then the *list* variable will hold the following structure:

```
list
   |
  + a
   |
  + b
   |
  + c
   |
  + d
   |
  + c
   |
  + pref:d
   |
  + b
   |
  + c
   |
  + pref2:d
```

Exceptions

An exception is thrown in the following cases:

- If the method `entry.appendChild(newAttr)` is called and the entry already contains an Attribute with the name of the newAttr object passed to the method.
- If for any of the methods defined by the Document/Node/Element and implemented by the Entry/Attribute classes a unexpected parameter is passed.
- An ArrayIndexOutOfBoundsException is thrown if the supplied script refers to an index of an Attribute/NodeList that does not exist.

Integrating scripting into your solution

As already explained, you use script whenever you need custom processing in your integration solution. Best practices with IBM Tivoli Directory Integrator divide this custom processing into two categories: attribute transformation and flow control.

Note: This is convention, and not a limitation or rule enforced by the system. The need for custom data processing inevitably comes at some identifiable point in the flow of data (for example, before any processing begins, before processing a particular entry, after a failure, and so forth), so by placing your code as close to this point as possible, you can make solutions that are easier to understand and maintain.

The logical place to do the attribute transformations is in your **Attribute Maps**, both Input and Output. If you need to compute a new attribute that is required by scripted logic or other Connectors downstream in the AssemblyLine, best practice is to do this in an **Input Map** if possible. Alternatively, if you must transform attributes for the sake of a single output source, then you can avoid cluttering the **work** entry object with output-specific transformations by putting these in the **Output Map**. of the relevant Connector.

The other category of custom logic, flow control, is best implemented in the Hooks that are invoked at that point in the automated workflow where the logic is needed. These control points are easily accessed from the Config Editor. Implementing custom processing is simply a matter of identifying the correct control point and adding your script in the appropriate edit window.

AssemblyLine **Script Components**, independent blocks of scripted code, also provide you with a place to create your own custom processing, and then enable you to reposition your code within the AssemblyLine. Although Script Components are frequently used during test and debugging, they can also serve an important role in a production Config. Just remember to name your components clearly and to include some documentation in the script itself to explain [8] why you implemented this logic in a Script Component, and not in an **Attribute Map** or **Hook**.

While it is important to both correctly identify the appropriate control point where you input your script, it is equally important to limit the scope of your script to cover just the single goal associated with the control point. If you keep your units of logic independent of each other, then there is a greater chance that they will be reusable and less of a chance that they might break when components are reordered or reused in other contexts. One way to build reusable code is by creating your own functions in your **Script Library** (or a **Prolog** Hook) to implement frequently used logic, instead of copying and pasting the same code into multiple places.

To sum up some of the best practices that you want to keep in mind while building solutions:

- Do attribute manipulation in Attribute Maps.
- Put flow control (filtering, validation, branching, and so forth) in Hooks, and where necessary, AssemblyLine script components.
- Use the automated behavior as much as possible; for example, AssemblyLine workflow and Connector modes.
- Simplify your solution by keeping AssemblyLines short and focused.
- Put often used logic in discrete blocks; for example, Scripts in your Resources section.
- Think reuse.

It is worth mentioning again that although the methods outlined previously are best practices, you might encounter situations where you have to deviate from established convention. If this is the case, your solution documentation is vital for maintaining and enhancing your work over time.

Controlling execution with scripting

The engine exposes a number of classes and objects that can be accessed, read and modified from user-created scripts in an AssemblyLine. These objects represent the state of the AssemblyLine and the whole Tivoli Directory Integrator environment at any moment. By modifying any of these objects, you modify the Directory Integrator environment and thus affect the execution of the integration process.

Note: Changes can be applied to either instances of a component or AssemblyLine. Changes can also be made to operational parameters, such as system or Java parameters. Changes can also be made to the configuration file, or Config. In this case, new instances of Config objects reflect these changes.

For more information about global objects, see the Javadocs included as part of the Tivoli Directory Integrator product by selecting **Help** > **Javadocs** in the Config Editor.

A description of all classes and instances available can be found in the installation package.

By understanding the classes and interfaces exposed, you can better understand the elements of the Directory Integrator engine as well as the relations between them.

Using variables

It is important to distinguish between the standard container object for data being processed (the Entry object) and other generic variables and data types provided to you by JavaScript, as well as those that you create yourself. Your creativity and the capabilities of the scripting language are your only

[8] To others, as well as yourself when you have to revisit your code some time later!

restrictions in terms of what can be placed in scripts inside your Tivoli Directory Integrator solutions. However, when you manipulate data in the context of the data flow, you must be aware of and use the structure of the Entry object.

Entry objects carry attributes, which are themselves the container for data values. Attribute values are themselves objects (`java.lang.String`, `java.util.Date` and more complex structures). An attribute value can even be another entry object with its own set of attributes and values. It is the job of Tivoli Directory Integrator to understand how data is stored in the connected system, as well as how to convert these native types to and from the data representation of the system, which is in Java objects.

If you know the class of the attribute value, you can successfully access and interpret this value. For example, if a `java.lang.String` attribute contains a floating point value that you want to use as a floating point, you must first manually transform this value, by means of the scripting language, to some numeric data type.

When creating variables or processes not directly related to the data flowing in the integration process and the global objects available, the following principle applies: You can declare and use any variables (objects) enabled by the scripting language. The purpose of these variables is to help you achieve the specific goal associated with the control point in which you script. The variables must serve only as temporary buffers and not attempt to affect the state of the Directory Integrator environment.

Using properties

During the lifetime of the AssemblyLine, the TDI Server makes available a number of component Properties, related to the execution environment of the AssemblyLine, that you can query in your scripts: either from Script Components or Component Hooks. One property (*lastCallStatus*) can even be set.

You access the properties by using an AssemblyLine object, for example, a Connector and call its method `get(property_name)` to extract the property_name value; alternatively, use the `put(property_name, property_value)` method to set the property to the desired value. When setting properties, the property name or property value cannot be null; if one of them is null an Exception with an appropriate message will be thrown.

The set of properties available is as follows:

Table 6. Component properties available during AssemblyLine Execution

Property	**Usage**
numErrors	The number of errors occurred.
numAdd	Total number of entries the AssemblyLine has added (performed by Connectors in AddOnly mode).
numModify	Total number of entries the AssemblyLine has modified (performed by Connectors in Update mode).
numDelete	Total number of entries the AssemblyLine has deleted (performed by Connectors in Delete mode).
numGet	Total number of entries the AssemblyLine has retrieved (performed by Connectors in Iterator mode).
numGetTries	Total number of times the AssemblyLine has attempted to retrieve an entry (performed by Connectors in Iterator mode).
numGetClient	Total number of accepted Clients (available for Connectors in Server mode).
numGetClientTries	Total number of times the AssemblyLine has attempted to get the next connected client (performed by Connectors in Server mode).
numCallreply	Total number of Call/Reply operations the AssemblyLine has executed (performed by Connectors in CallReply mode).

Table 6. Component properties available during AssemblyLine Execution (continued)

Property	Usage
numLookup	Total number of Lookup operations the AssemblyLine has executed (performed by Connectors in Update/Delete/Lookup mode).
numNoChange	Total number of entries the AssemblyLine processed but left unchanged.
numSkipped	Total number of entries the AssemblyLine has skipped.
numIgnored	Total number of entries the AssemblyLine has ignored (performed by Connectors in Update/Delta mode).
lastCallStatus	Contains the status for the AL execution. It is not just a read-only property and can be modified by you. The value of this property is "fail" or "success" dependant on the AL execution.
lastConn	The Conn entry from the last Connector operation. Before the first Connector operation, lastConn has a value of null.
lastError	The last error as a Java object.
hooksInvoked	A java.util.List of the names of the hooks invoked the last time the Component was invoked. The names are internal names.
success	This property is set to true if the last operation was a success, and false otherwise.
endOfData	True when the Iterator Component has reached End of Data, false otherwise. Changing this property has no effect.

Note: If an attempt is made to change a read-only property then an Exception with appropriate message will be thrown.

Example

To illustrate the use of these Component Properties, let's assume you have a FileSystem Connector called *FS*, and some Script Components. The following JavaScript Code is in the "GetNext Successful" hook of FS:

```
if(work.getString("ID") == null)
throw new java.lang.Exception("Missing ID");
//for the AL Cycle to execute properly I need an ID, so throw an Exception
```

And this JavaScript Code is in the "DefaultOnError" hook of FS:

```
if(FS.get("lastError").getMessage().equals("Missing ID")) {
//I could fix this by adding an ID which would help AL execution
 work.setAttribute("ID", "SomeID"); //add the ID
 if(FS.get("fixErrors") == null) {
  var vector = new java.util.Vector();
  vector.add(FS.get("lastError"));
  FS.put("fixErrors", vector);//save all fixed errors in my custom property
 } else {//I have previously fixed similar error
  var vector = FS.get("fixErrors");
  vector.add(FS.get("lastError"));
  FS.put("fixErrors", vector); //save all fixed errors in my custom property
 }
 FS.put("lastCallStatus", "success");
} else {//I could not fix this error
 if(FS.get("notFixErrors") == null) {
  var vector = new java.util.Vector();
  vector.add(FS.get("lastError"));
  FS.put("notFixErrors", vector); //save all not fixed errors in my custom property
 } else {
  var vector = FS.get("notFixErrors");
  vector.add(FS.get("lastError"));
  FS.put("notFixErrors", vector); //save all not fixed errors in my custom property
 }
 FS.put("lastCallStatus", "fail");
}
```

Finally, in a Script Component, consider the following code:

```
main.logmsg("AL Cycle status: " + FS.get("lastCallStatus"));
//print the AL status for this AL Cycle
//I can also report all errors which have occurred
// during the AL Execution through my custom property, "vector"
```

Control points for scripting

Scripting in an AssemblyLine

AssemblyLines provide for a standard, pre-programmed behavior. If you want to deviate from this standard behavior, you can do so by implementing your own business logic by means of scripting.

Script Component

You can add Script Components to your AssemblyLine in addition to Connectors by right-clicking **Insert Components...** > **Scripts** > **Script** on the appropriate Component or section in the AssemblyLine Components window in the AssemblyLine Editor. The Script Component is started once for each entry processed by the AssemblyLine, and can be placed anywhere in the AssemblyLine.

Note: Iterators are still processed first, even if you place your Script Component before them in the AssemblyLine.

For more information, see “Script Components” on page 18.

AssemblyLine Hooks

AssemblyLine Hooks (that is, Hooks that apply to the AssemblyLine as a whole, not any individual Component) are found in the **Hooks** tab of the AssemblyLine. These Hooks are all executed only once per AssemblyLine run, or, in the case of **Shutdown Request**, whenever the AssemblyLine is told to shut down by some external process. However, if you start your AssemblyLine multiple times (for example, by using the AssemblyLine Connector), then you start the Hooks multiple times as well.

Hooks inside a Connector are only evaluated and executed (where defined and non-empty) when the Connector in which they are defined, is run. See “Scripting in a Connector” on page 60 for more information.

Server Hooks

Server Hooks allow you to write JavaScript code to respond to events and errors that occur at the server level. Unlike AssemblyLine and component Hooks, Server Hooks are stored in separate script files. These files are kept in the *serverhooks* folder in the current solution directory and must contain specifically named script functions. 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.

In addition to these Hooks being called by the Server when specific events occur, they can also be invoked from your scripts. Calls to these Hooks are synchronized to avoid potential multi-threading issues.

Upon startup, Tivoli Directory Integrator loads and executes all user scripts in the *serverhooks* subdirectory. Scripts may or may not contain function declarations. A script that has no function declarations is executed once at startup before any configuration instances are started. Code that defines standard TDI Server Hook functions are prefixed with "TDI_"., and these are executed at various points during operation.

All Tivoli Directory Integrator Server Hook functions have the following JavaScript signature:

```
/**
   * @param NameOfFunction The configuration instance invoking the function
  * @param source The component invoking the function
  * @param user Arbitrary parameter information from the source
  */
 function TDI_functionName(Name_of_function, source, user) {
 }
```

The "NameOfFunction" 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.

The following standard function names are invoked by various Tivoli Directory Integrator components:

Function Name	Called by (source)	User Parameter and Expected Value

TDI_ALStarted	Config Instance	Called when an AssemblyLine is started.
		user = The AssemblyLine that started
		return value ignored

TDI_ALStopped	Config Instance	Called when an AssemblyLine stopped.
		user = The AssemblyLine that stopped
		return value ignored

TDI_ConfigStarted	Server	Called when a Config instance started.
		user = The configuration instance
		return value ignored

TDI_ConfigStopped	Server	Called after a Config instanced stopped.
		user = The configuration instance
		return value ignored

TDI_Shutdown	Server/Config Instance	Called immediately before the TDI server is terminating the JVM (for example, System.exit()).
		user = Exit status (integer)
		return value 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 mentioned previously, 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 available for encrypting script files. Note that Tivoli Directory Integrator automatically tries to decrypt files with extension .jse, hence encrypted files should preferably have that extension.

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 Server Hook use are:

- A custom object that you always want loaded in Tivoli Directory Integrator for use for 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.
- One or more custom ALs that you start creates an audit log for these events or propagates these events to other systems using some transport (SNMP, HTTP, JMS, and so forth).
- A corporate security policy you implement that is invoked every time a Config is loaded or AL started.

Calling Server Hooks from script: The `com.ibm.di.server.RS` class (script variable "main") has a 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;
```

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.

In a script in an AssemblyLine you may do this:

```
main.invokeServerHook("MyCustomHook", this, "custom information");
```

Accessing AL components inside the AssemblyLine

Each AL component is available as a pre-registered script variable with the name you chose for the component.

Note that you can dynamically load components with scripted calls to functions like `system.getConnector()`, although this is for experienced users.[9]

AssemblyLine parameter passing

There are three ways to get data into an AssemblyLine:

- Generating your own initial entry inside the AssemblyLine, for example, in a Prolog script.

[9] The Connector object you get from this call is a *Connector Interface* object, and is the data source specific part of an AssemblyLine Connector. When you change the *type* of any Connector, you are actually swapping out its data source intelligence (the Connector Interface) which provides the functionality for accessing data on a specific system, service or data store. Most of the functionality of an AssemblyLine Connector, including the attribute maps, Link Criteria and Hooks, is provided by the IBM Tivoli Directory Integrator kernel and is kept intact when you switch Connector types.

- Fed from one or more Iterators.
- Starting the AssemblyLine with parameters from another AssemblyLine using the AL Connector or AL Function Component, or using an API call.

If you want to start an AssemblyLine with parameters from another AssemblyLine, then you have a couple of options:

- Use the **Task Call Block** (TCB), which is the preferred method. The TCB is detailed below.
- Provide an Initial Work Entry directly.

Note: These two options are provided for compatibility with earlier versions.

Task Call Block (TCB)

The Task Call Block (TCB) is a special kind of Entry object used by a caller to set a number of parameters for an AssemblyLine.

Basic Use: The Task Call Block (TCB) is a special kind of Entry object used by a caller to set a number of parameters for an AssemblyLine. The TCB can provide you with a list of input or output parameters specified by an AssemblyLine, including operation codes defined in the **Operations** tab of the AssemblyLine, as well as enabling the caller to set parameters for the AssemblyLine's Connectors. The TCB consists of the following logical sections:

- The Initial Work Entry passed to the AssemblyLine: `tcb.setInitalWorkEntry()`
- The Connector parameters: `tcb.setConnectorParameter()`
- The input or output mapping rules for the AssemblyLine, which are set in the Config Editor under the **Operations** tab
- An optional user-provided *accumulator* object that receives all work entries from the AssemblyLine: `tcb.setAccumulator()`

For example, starting an AssemblyLine with an Initial Work Entry and setting the *filePath* parameter of a Connector called MyInput to d:\myinput.txt is accomplished with the following code:

```
var tcb = system.newTCB();                      // Create a new TCB
var myIWE = system.newEntry();                  // Create a new entry object
myIWE.setAttribute("name","John Doe");          // Add an attribute to myIWE
tcb.setInitialWorkEntry ( myIWE );              // Set the IWE and parameters
// Note that since this is a JavaScript string, we must "escape" the forward slash
//      or use a backslash (Windows syntax)
tcb.setConnectorParameter ( "MyInput", "filePath", "d:\\myinput.txt" );

var al = main.startAL ( "MyAssemblyLine", tcb ); // Start the AL with the tcb
al.join();                                      // Wait for AL to finish
```

Starting an AssemblyLine with operations: AssemblyLines can be defined with *Operations*; a concept whereby a number of Input Maps are defined for the AssemblyLine. Depending on how the AssemblyLine is invoked, a different Input Map is activated. Inside the AssemblyLine you will need to check the op-entry to find out which operation is active, and use Branch Components to tailor the flow inside the AssemblyLine to the relevant operation.

One of the ways to start an AssemblyLine with an Operation is by means of a TCB, and script code.

If an AssemblyLine named "al1" has the following operations: "Default", "Op1" and "Op2", then this script will start the AL with operation set to "Op1":

```
var tcb = system.newTCB("al1");
tcb.setALOperation("Op1");
main.startAL(tcb);
```

Not specifying any operation will start the AL with operation set to "Default":

```
var tcb = system.newTCB("al1");
main.startAL(tcb);
```

In case the AL doesn't have a Default operation (for example, only "Op1" and "Op2" operations) the second script will throw an exception.

More information about the AssemblyLine Operations concept is available in the section entitled "Creating new components using Adapters" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Using an accumulator: As noted previously, you can also pass in an accumulator object to an AssemblyLine with the TCB. An accumulator can be any one of the following classes or interfaces:

java.util.Collection
All work entries are cloned and added to the collection; for example, ArrayList, Vector, and so forth.

com.ibm.di.connector.ConnectorInterface (Connector Interface)
The putEntry() method for this Connector Interface is called with the Work entry at the end of each AssemblyLine cycle.

com.ibm.di.parser.ParserInterface (Parser)
The writeEntry() method is called for this Parser with the Work entry at the end of each AssemblyLine cycle.

com.ibm.di.server.AssemblyLine Component (AssemblyLine Connector)
The add() method is called for this AssemblyLine Connector with the Work entry at the end of each AssemblyLine cycle.

If the accumulator is not one of these classes or interfaces, an exception is returned.

For example, to accumulate all work entries of an AssemblyLine into an XML file you can use the following script:

```
var parser = system.getParser ( "example_name.XML" ); // Get a Parser
// Set it up to write to file
parser.setOutputStream ( new java.io.FileOutputStream ( "d:/accum.xml" ));
parser.initParser();                              // Initialize it.
tcb.setAccumulator ( parser );                    // Set Parser to tcb

var al = main.startAL ( "MyAssemblyLine", tcb );      // Start AL with tcb
al.join();                                        // Wait for AL to finish

parser.closeParser(); // Close the parser - this flushes and
              // closes the output file
```

As well, you can configure a Connector instead of programming the Parser manually as in the following script:

```
var connector = system.getConnector("myFileSysConnWithXMLParser");
tcb.setAccumulator ( connector );

var al = main.startAL( "MyAssemblyLine", tcb);
al.join();

connector.terminate();
```

Typically, you initialize the TCB and then the TCB is used by the AssemblyLine. If the AssemblyLine has an Operations specification, the TCB remaps input attributes to the Initial Work Entry as expected by the AssemblyLine, and likewise for setting the result object. This is done so that the external call interface to an AssemblyLine can remain the same even though the internal Work entry names change in the AssemblyLine. Once the TCB is passed to an AssemblyLine, you must not expect anything more from the TCB. Use the AssemblyLine's `getResult()` and `getStats()` to retrieve the result object and statistics.

The TCB result mapping is performed before the Epilog so you can still access the final result before the AssemblyLine caller gets to it.

Disabling AssemblyLine components: In IBM Tivoli Directory Integrator V7.1.1, you can specify that certain AssemblyLine components must not be created or initialized on AssemblyLine initialization. This is done by disabling those components using the TCB.

AssemblyLine components are enabled by default.

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

The actual enabling or disabling of the AssemblyLine components happens in the `com.ibm.di.server.TaskCallBlock.applyALSettings(AssemblyLineConfig alc)` method. This method is invoked upon 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 is disabled from the Config Editor, it can be enabled using this script call:

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

Providing an Initial Work Entry (IWE)

Providing an Initial Work Entry (IWE) is an alternative way of passing parameters using a TCB and is supported for compatibility with earlier versions..

When an AssemblyLine is started with the **system.startAL()** call from a script, the AssemblyLine can still be passed parameters by setting attribute or property values in the Initial Work Entry, which is accessed through the *work* variable. It is then your job to apply these values to set Connector parameters; for example, in the AssemblyLine **Prolog – Init** Hook using the *connectorName*.`setParam()` function.

Note: You must clear the Work entry with the `task.setWork(null)` call, otherwise Iterators in the AssemblyLine pass through on the first cycle.

You can examine the result of the AssemblyLine, which is the Work entry when the AssemblyLine stops, by using the `getResult()` function. See also "runtime provided Connector" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Below is an example of passing in a Connector parameter value with an IWE:

```
var entry = system.newEntry();
entry.setAttribute ("userNameForLookup", "John Doe");

// Here we start the AssemblyLine
var al = main.startAL ( "EmailLookupAL", entry );

// wait for al to finish
al.join();

var result = al.getResult();

// assume al sets the mail attribute in its working entry
task.logmsg ("Returned email = " + result.getString("mail"));
```

Scripting in a Connector

Input Map and Output Map

Custom attribute mapping is performed in these tabs. When the attribute is selected, you must select the **Advanced Mapping** check box, and input your script in the edit window. Remember that after you have done all processing necessary, you must assign the result value achieved to `ret.value`, for example:

```
   ...
ret.value = myResultValue;
```

Alternatively, in Tivoli Directory Integrator V7.1.1 you may use the keyword *return* followed by the simple value you would like to pass as result:

```
return "mystring";
```

Connector Hooks

Hooks give you the means to respond to certain events that occur, and override the basic functionality of a Connector. You have access to the global objects when scripting Hooks, although some of the standard objects might not be available in every Hook. For details on temporary object availability, see "AssemblyLine and Connector mode flowcharts" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*. You also have full control over the environment, the AssemblyLine, the Connector, entries and attributes. Hooks give you a diversity of control points for customizing the process flow. See "AssemblyLine flow and Hooks" on page 26.

Setting internal parameters by scripting

It is possible to set the connection parameters for a Connector using the following script:

```
myConnector.setParam ( "filePath", "examples/scripting/sample.csv" );
```

This is typically something you do in the Prolog, but it can be very useful while the AssemblyLine is running as well, provided that you stop and reinitialize the Connector.

```
myConnector.terminate();
myConnector.setParam ( "filePath", "examples/scripting/sample.csv" );
myConnector.initialize(null);
```

Scripting in a Parser

Scripting in a Parser actually refers to implementing your own Parser by scripting. A description of this process is included in the section called "Script Parser" in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Java + Script ≠ JavaScript

JavaScript is **not** Java. It may look like Java, but it is actually just close enough to really cause confusion. JavaScript was originally called *Live!Script* back when Netscape first created it. Although there is broad support for JavaScript, you will learn that *dialects* exist; for example, Microsoft's version, called *JScript*. There is a standard definition, which is known as *ECMAScript*, and you can find its specification at this URL: http://www.ecma-international.org/

Although the syntax is similar, Java and JavaScript deal with data and data types differently. This is one of the main sources of confusion, and with that, errors when working with JavaScript.

Data Representation

Java supports something called *primitives*, which are simple values like signed integers, decimal values and single bytes. Primitives do not provide any functionality, only non-complex data content. You can use them in computations and expressions, assign them to variables and pass them around in function calls. Java also provides a wealth of objects that not only carry data content (even highly complex data), but also provide intelligence in the form of object functions, also known as methods.

When you make the script call to `task.logmsg( "Hello, World" )`, you are calling the task object's `logmsg()` method.

Many Java primitives have corresponding objects. One example is integers, which can used in their primitive form *(int)* or by manipulating `java.lang.Integer` objects.

JavaScript does not employ the concept of primitives. Instead, all data is represented as JavaScript objects. Furthermore, JavaScript has only a handful of native objects as compared to Java's rich data vocabulary. So while Java differentiates between non-fractional numeric objects and their decimal counterparts – even distinguishing between signed and unsigned types, as well as offering similar objects for different levels of precision – JavaScript lumps all numeric values into a single object type called a *Number*.

As a result, you can get seemingly erroneous results when comparing numeric values in JavaScript:

```
if (myVal == 3) {
 // do something here if myVal equals 3
}
```

If `myVal` was set by an arithmetic operation, or references a Java decimal object, the object's value could be 3.00001 or 2.99999. Although this is very close to 3, it will not pass the above equivalency test. To avoid this particular problem, you can convert the operand to a Java Integer object to ensure a signed, non-fractional value. Then your Boolean expression will behave as expected.

```
if (java.lang.Integer( myVal ) == 3) { ...
```

Or you can make sure that your variables reference appropriate Java objects to begin with. In general, you will want to be conscious of the types of objects you are using.

Ambiguous Function Calls

Java also provides a primitive type called a *char*, which can contain a single character value. A collection of characters could be represented in Java as an array of character primitives, or it could be handled as a `java.lang.String` object. As mentioned before, JavaScript does not savvy primitives. Character data must be dealt with using the JavaScript *string* object. Even if you specify a single character in JavaScript ("a"), this is considered a *string*.

Now consider that when you call a Java function from your script, this is matched up to the actual method using the name of the function as well as the number and types of the parameters used in the call. This matching is carried out by the LiveConnect extension to JavaScript. LiveConnect does its best to figure out which function signature you are referring to, which is no small task given that Java and JavaScript represent parameter types in different ways. But JavaScript and LiveConnect do some internal conversions for you, trying to match up Java and JavaScript data types.

A problem arises when you have multiple versions of a single method, each taking a different set of parameters; in particular, if two functions have the same number of parameters, but of different types, **and** if these types are not differentiated in JavaScript. Let's take a look at an example script that will be performing an MD5 encryption of a string.

```
// Create MessageDigest object for MD5 hash
var md = new java.security.MessageDigest.getInstance( "MD5" );

// Get the EID attribute value as byte array.
var ab = java.lang.String( "message to encrypt" ).getBytes();

md.update( ab );

var retHash = md.digest();
```

The above `update()` call will fail with an evaluation exception stating that the function call is ambiguous. This is because the `MessageDigest` object has multiple versions of this function with similar signatures: one accepting a single byte and one expecting a byte array (`byte[]`). You can get around this if you can find another variant of the same method taking a different number of parameters, thus giving it a uniquely identifiable signature. Fortunately, MessageDigest provides something suitable: a version of update() that takes a byte array plus a couple of numeric values (offset and length parameters). So you can change your code to use this call instead:

```
md.update( ab, 0, ab.length );
```

Finally, you can always specify the exact signature of the Java method you want to use by quoting it inside brackets after the object:

```
md["update(byte[])"](ab);
```

Here we are calling for the version of the update() function declared with a single byte array parameter.

Char/String data in Java versus JavaScript Strings

Both Java and JavaScript provide a String object. Although these two types of String objects behave in a similar fashion and offer a number of analogous functions, they differ in significant ways. For example, each object type provides a different mechanism for returning the length of a string. With Java Strings you use the length() method. JavaScript Strings on the other hand have a length *variable*.

```
var jStr_1 = new java.lang.String( "Hello, World" ); // Java String
task.logmsg( "the length of jStr_1 is " + jStr_1.length() );

var jsStr_A = "Hello, World"; // JavaScript String
task.logmsg( "the length of jsStr_A is " + jsStr_A.length );
```

This subtle difference can lead to baffling syntax errors. Trying to call jsStr_A.length() will result in a runtime error, since this object has no length() method.

Even more confounding mistakes can occur with string comparisons.

```
var jsStr_A = "Hello, World"; // JavaScript String
var jsStr_B = "Hello, World"; // JavaScript String

if ( jsStr_A == jsStr_B )
 task.logmsg( "TRUE" );
else
 task.logmsg( "FALSE" );
```

As expected, you will get a result of "TRUE" from the above snippet. However, things work a little differently with Java Strings.

```
var jStr_1 = java.lang.String( "Hello, World" ); // Java String
var jStr_2 = java.lang.String( "Hello, World" ); // Java String

if ( jStr_1 == jStr_2 )
 task.logmsg( "TRUE" );
else
 task.logmsg( "FALSE" );
```

This will result in "FALSE", since the equivalency operator above will be comparing to see if both variables reference the *same object* in memory, instead of matching their values. To compare Java String values, you must use the appropriate String method:

```
if ( jStr_1.equals( jStr_2 ) ) ...
```

But wait, there's more. This next snippet of code will get you a result of "TRUE":

```
var jsStr_A = "Hello, World"; // JavaScript String
var jStr_1 = java.lang.String( "Hello, World" ); // Java String

if ( jsStr_A == jStr_1 )
 task.logmsg( "TRUE" );
else
 task.logmsg( "FALSE" );
```

Since JavaScript cannot operate on an unknown type like a Java String object, it first converts jStr_1 to an equivalent JavaScript String in order to perform the evaluation.

In summary, be aware of the types of objects you are working with. And remember that TDI functions always return Java Objects. Keeping these factors in mind will help minimize errors in your script code.

Variable scope and naming

JavaScript is a relatively informal language and does not require you to define variables before assigning values to them. Neither does it enforce strict type checking, or signal when a variable is redefined. This makes JavaScript fast and easy to work with, but can quickly lead to illegible code and confounding errors, especially since you can create variables that overwrite built-in ones.

One bug that can defy debugging is when you declare a variable with the same name as a built-in one, like work, conn and current, so you will need to familiarize yourself with the reserved names used by IBM Tivoli Directory Integrator.

Another common problem occurs when you create new variables that redefine existing ones, perhaps used in included Configs or Script Libraries. These mistakes can be avoided if you are conscious about the naming of variables and their *scope*. Scope defines the sphere of influence of a variable, and in Directory Integrator we talk about global variables; those which are available in all Hooks, Script Components and attribute maps, and those that are local to a function.

To get a better understanding of scope you must first understand that every AL has its own Script Engine, and therefore runs in its own script context. Any variable not specifically defined as local inside a function declaration is global for that Script Engine. So the following code will create a global variable:

```
myVar = "Know thyself";
```

This variable will be available from this point on for the life the AL. Making this variable local requires two steps: using the var keyword when declaring the variable, and putting the declaration inside a function:

```
function myFunc() {
 var myVar = "Know thyself";
}
```

Now myVar as defined above will cease to exist after the closing curly brace. Note that placing the variable inside a function is not enough; you have to use var as well to indicate that you are declaring a new, local, variable.

```
var glbVar = "This variable has global scope";
glbVar2 = "Another global variable";

function myFunc() {
    var lclVar = "Locally scoped within this block";
    glbVar3 = "This is global, since "var" was not used";
};
```

As long as you declare a local variable within a function then you can call it what you like. As soon as it goes out of scope, any previous type and value are restored

Even though the var keyword is not required for defining global variables, it is best practice to do so. And it is recommended that you define your variables at the top of your script, including enough comments to give the reader an understanding of what they will be used for. This approach not only improves the legibility of your code, it also forces you to make conscious decisions about variable naming and scope.[10]

Instantiating a Java class

You can call external Java classes ; that is, those you have added to the Tivoli Directory Integrator runtime environment, or the CLASSPATH, using script code.

For example, assuming you want to use the standard `java.io.FileReader`, use the following script:

```
var javafile = new java.io.FileReader( "myfile" );
```

You now have an object called *javafile*; using it you can call all of the methods of the object.

The same technique is used to instantiate your own objects:

```
var myfile = new my.FileReader("myfile");
```

Using binary values in scripting

Binary values can be retrieved from Attributes by using the `getObject()` function of the entry. The binary Attribute value itself is returned as a byte array. Here is a JavaScript example:

```
var x = conn.getObject("objectGUID");
for ( i = 0; i < x.length; i++ )
{
   task.logmsg ("GUID[" + i + "]: " + x[i]);
}
```

This example writes some numbers varying between -128 and 127 into the log file. You might want to do something else with your data. If you have read a password from a Connector, which stored it as a ByteArray, you can convert it to a string with this code:

```
password = system.arrayToString(conn.getObject("userpassword"));
```

Using date values in scripting

When working with dates in IBM Tivoli Directory Integrator, this implies using instances of `java.util.Date`. Armed with any of the available scripting languages, you can implement your own mechanism for handling dates; however, this is not a common practice.

The Directory Integrator scripting engine provides you with a mechanism for parsing dates. The *system* object has a `parseDate(date, format)` method accessible at any time.

Note: When you get an instance of `java.util.Date`, you can use the standard Java libraries and classes to extend your processing.

Here is a simple JavaScript example that handles dates. This code can be placed and started from any scripting control point:

```
var string_date1 = "07.09.1978";
var date1 = system.parseDate(string_date1, "dd.MM.yyyy");

var string_date2 = "1977.02.01";
```

[10] Function naming works a little differently. Programming languages like Java identify a function by the combination of its name plus the number and types of parameters. JavaScript just uses the name. So if you have multiple definitions of the same function, JavaScript will only "remember" the last one — regardless of whether that definition has a different number of parameters.

```
var date2 = system.parseDate(string_date2, "yyyy.dd.MM");

task.logmsg(date1 + " is before " + date2 + ": " +
  date1.before(date2));
```

The script code first parses two date values (in different formats) into `java.util.Date`. It then uses the standard `java.util.Date`**`.before()`** method to determine whether the first date instance comes before the second one. The output of this script is then printed to the log file.

Using floating point values in scripting

The following examples demonstrate how floating point values can be used within the scripting code you create. All of these examples are implemented in JavaScript. While the same examples might be repeated using several other scripting languages, the syntax might be different. The following simple script assigns floating point values to two variables in order to find their average. This code can be started from any scripting control point. The log file output is `" r = 3.85 "`.

```
var a = 5.5;
var b = 2.2;
var r = (a + b) / 2;
task.logmsg("r = " + r);
```

The next example extends this simple script. Consider that in your input Connector is a multi-valued attribute called "Marks" containing string values (java.lang.String) representing floating point values; a common situation. This attribute is mapped to an attribute in your output Connector called "AverageMark", which holds the average value of all values of the "Marks" attribute. The following code is used in the Advanced Mapping of the "AverageMark" attribute:

```
// First return the values of the "Marks" attribute
var values = work.getAttribute("Marks").getValues();

// Zero out counter and sum variables
var sum = 0;
var count = 0;

// Loop through the values, counting and summing them
for (i=0; i<values.length; i++)
{
   // use the Double() function to convert value to number
   sum = sum + new Number(java.lang.Double(values[i]));
   count++;
}

// If count > 0, compute the average
var average = (count > 0) ? (sum / count) : 0;

// Return the computed average
ret.value = average;
```

The central call in this example is the `java.lang.Double(values[i])` used to convert the currently indexed value of "Marks" into a numeric value that can then be used in the average computation.

Chapter 3. The Configuration Editor

The IBM Tivoli Directory Integrator Eclipse Configuration Editor (CE) is the primary tool Tivoli Directory Integrator (TDI) offers to develop TDI solutions. It lets you create, maintain, test and debug Config files; it builds on the Eclipse platform to provide a development environment that is both comprehensive and extensible.

In order to understand the concepts presented here more easily, you should be familiar with common Eclipse concepts such as Editors, Views and other common Eclipse extension points.

The Project Model

With the advent of the Eclipse-based Configuration Editor (CE), development of solutions in Tivoli Directory Integrator (TDI) is not, as in pre-V7.0 versions, based upon the development of plain Config files, but upon a Project model, and *Workspaces*. Using the Project model and the Workspace, you can do your development work; once you are ready to deploy the solution, you extract a Config file from the Workspace and send it to a suitable TDI Server: the runtime environment. This yields several benefits, some of which are:

- Cleaner Config files; unnecessary and unused Config elements will not be exported from the Workspace;
- More efficient editing of TDI configurations;
- Greater re-use of common components;
- The possibility of using source code management systems, for example CVS.

The Workspace

When you start the CE you will be prompted to select a workspace directory. The workspace directory is where the CE stores all your TDI projects and files. This is different from the solution directory, though the workspace directory may very well be contained inside the solution directory. From the workspace, the CE gathers files to form a runtime configuration file (`rs.xml`) that can be executed on a TDI server. Files and projects in the workspace can be thought of as the source for various runtime configuration files.

Install Directory, Solution Directory and the Working directory

Some confusion might arise from having three different directories in play. As stated in the previous section, the workspace directory belongs to the CE where the project files are stored. The install and solution directories belong to the TDI server when it executes. However, when you configure a connector in the CE for example, you will often use the discover attributes function; this will cause the CE to open the connector and execute operations on the connector. Since some connectors use file names for multiple purposes, it can be confusing when relative paths are used. For this reason, the default working directory for the CE should always be the same as the solution directory you are primarily working with[11]. Primarily, since it is possible to create several servers and solution directories inside the CE. You can still work with those and run AssemblyLines on those servers, but relative paths in configurations are no longer the same when you use connectors in the CE as opposed to using them when you run the AssemblyLine.

[11] The working directory should not be confused with the workspace. The workspace is the area where the project files are stored; the working directory is the position in the file system that defines all non-fully qualified filenames. For portability reasons, this should be the solution directory.

Consider a configuration with a file connector using `abc.txt` as its input file. When you use discover attributes in the CE it will look for a file in the working directory of the CE (the preferred solution directory specified at install time). When you run the AssemblyLine with this connector everything looks fine. Then you create a new server with a solution directory pointing somewhere else than the current working directory. You configure the connector and the connector resolves the file name to the correct place (that is, the abc file resides in your current working directory). However, when you run the AssemblyLine it fails, since it can't find the file. This is because the new server runs in a different working directory (the solution directory) than the CE.

By default, the working directory of the CE is set to the solution directory of the default server that is used to execute AssemblyLines. So if you don't change anything you should not see any of these problems except when you create new solution directories or change the solution directory of the default server.

The Workbench

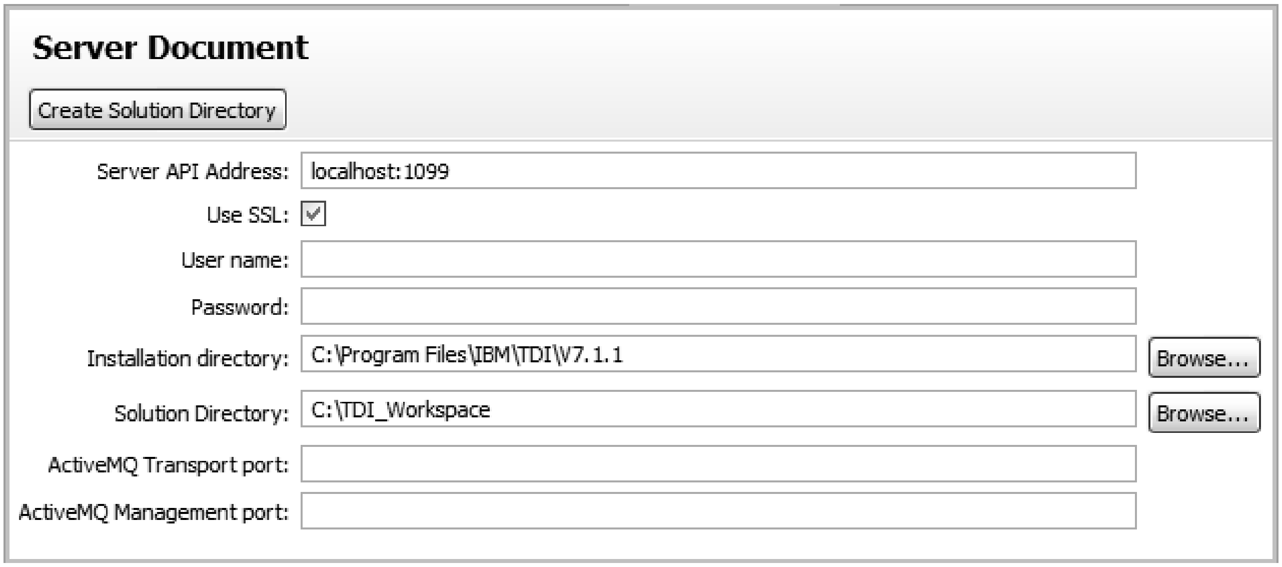
The workbench is the main user interface application in which you perform all configuration and development of TDI solutions. The workbench consists of a number of views and editors that let you do this.

The TDI Servers view

A Tivoli Directory Integrator (TDI) server instance is not started every time you run an AssemblyLine like earlier versions (pre-7.0) did. Instead, a TDI server is automatically started when you start the CE, which is used to test and run AssemblyLines that you develop. Contrary to earlier versions, when the object of your test runs finishes, this server does not terminate but stays around, waiting for your next test run.

In the Servers View you can manage server definitions. Servers defined here can be local servers or remote servers and is used by the various TDI projects you create. Servers that have an installation path defined are considered “local” servers that can be launched by the CE.

One server is automatically defined and is named *Default*. This server is the default server used by new TDI projects. The default server is created with values from the global properties file using the solution directory as defined by the user (for example from the `TDI_SOLDIR` variable, which in turn is setup by the Installer).



Server Document

Server API Address: localhost:1099

Use SSL: ☑

User name:

Password:

Installation directory: C:\Program Files\IBM\TDI\V7.1.1

Solution Directory: C:\TDI_Workspace

ActiveMQ Transport port:

ActiveMQ Management port:

Figure 5. The default TDI Server definition

The TDI Project

To develop a TDI solution you must first create a TDI Project. A project in Eclipse is a collection of related files and resources.

When the project is created, it is populated with a few folders where common configuration objects are located. The layout of a new TDI project is shown below.

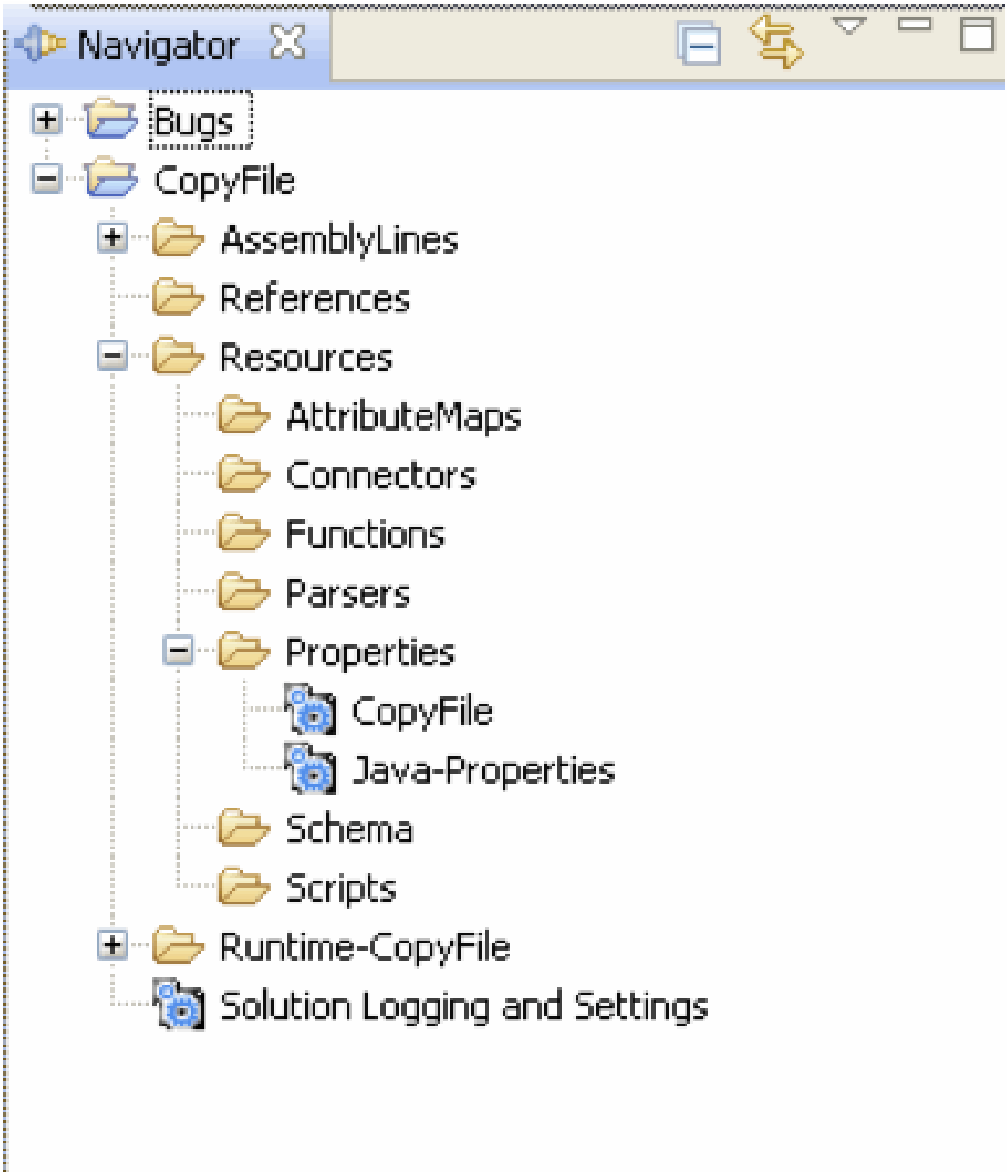


Figure 6. The TDI Project tree

The AssemblyLines folder contains the AssemblyLines of the project; whereas the resources folder contains all components that are shared or used by the AssemblyLines or apply to the solution in general: properties, logging parameters and so forth. To create new resources either use the **File/New...** wizard from the main menu or use the context menus on each folder (that is, right-click AssemblyLines folder to create a new AssemblyLine).

The Runtime directory (Runtime-*ProjectName*) is a special directory that contains the runtime configuration file as well as custom properties files. Whenever a component has changed in the project (connector, AssemblyLine, property file and so forth) the relevant files in this directory are also updated. All generated files are tagged as derived files, which means that you get a warning if you attempt to modify the contents. Be aware that if you do change any of these files they will be overwritten. The intention with the runtime directory is to create a directory of runtime files for the project that can be copied, or checked out if you use source control, and used by a TDI server.

Configuration Files

The configuration file (Config) run by a server is a composite XML document in which AssemblyLines, connectors and so forth are all part of the same document. In the IBM Tivoli Directory Integrator Eclipse CE each of these components are allocated their own physical file. These files only contain one configuration object.

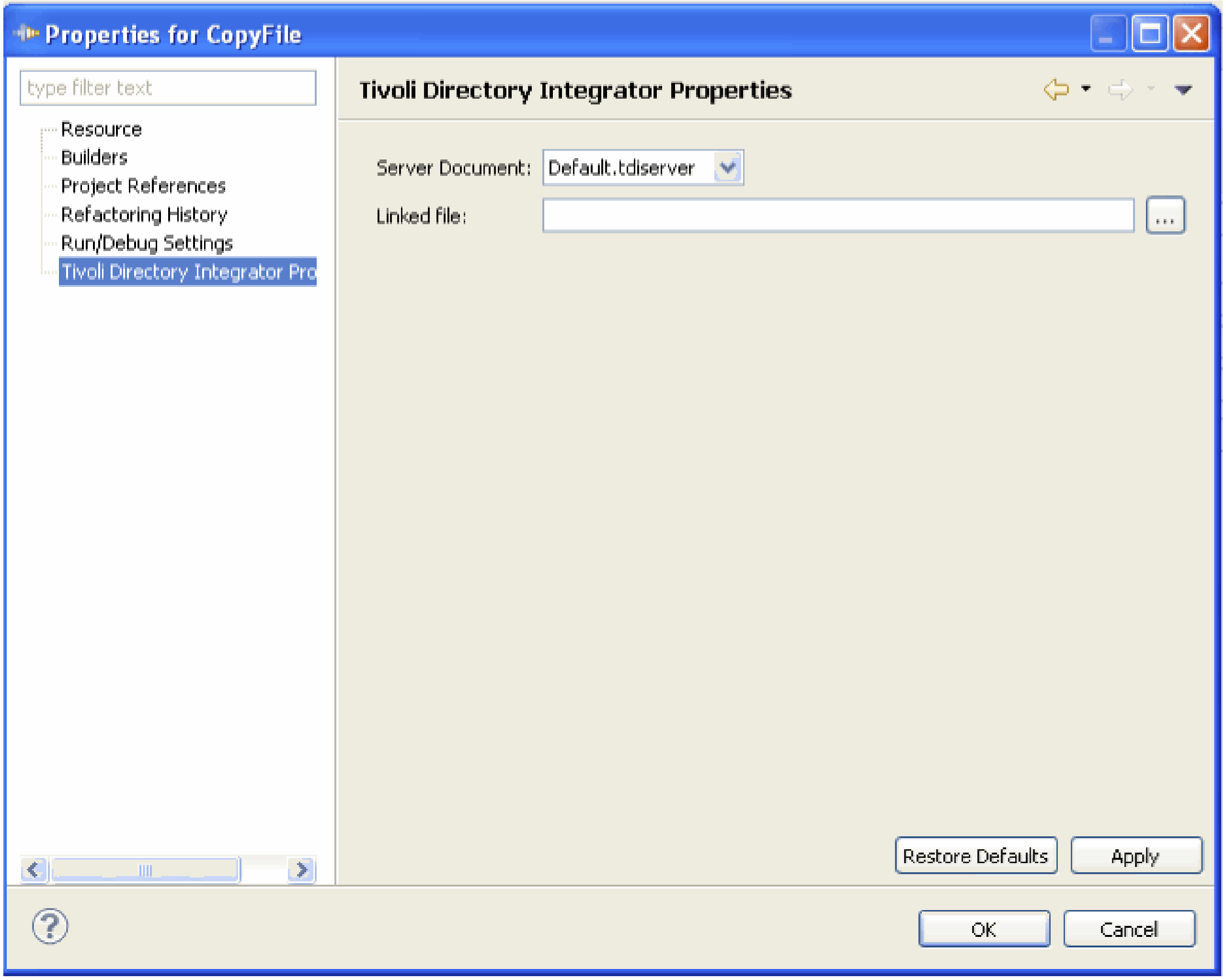
One of the reasons for splitting the configuration file into separate files during solution development was to make sharing of components easier. Also, having each component in its own file lends itself nicely to source control systems like CVS and also to multiuser development where different people work on different parts of the solution at the same time.

Configurations prior to this version can be imported using an import wizard. The imported configuration splits into individual configuration files as a result of this process. Another way is to use the **File** > **Open Tivoli Directory Integrator Configuration File** option, which will import the pre-7.0 configuration into a new project. Be aware though that this option also set the auto update back to the source file. See the next section for more information on the linked file feature.

Runtime Configuration File

Hidden from normal view is the runtime configuration file. This is the file that TDI servers use to run the solution and also the file that previous versions (pre-7.0) would work directly on. Whenever you save a configuration file in the CE, it will cause an update to the runtime configuration file as well. This file is also the one that is transferred to a TDI server for execution. This file is maintained by the Project Builder and should not be modified by the end user.

You can configure your project to export the runtime configuration file automatically when it changes. Use the project properties panel to configure the file the project is linked to.



Server Document: Default.tdiserver

Linked file:

Figure 7. TDI Project Properties window

Whenever the runtime configuration is updated it is also copied to the file specified in the TDI properties section of the project. The linked file is automatically set when you use the **File** > **Open Directory Integrator Configuration** command.

The Project Builder

A custom project builder is associated with the project whose purpose is to assemble all artifacts into a *runnable configuration file*.

This builder can be run automatically whenever a resource changes, or manually through the standard **Project** > **Build** menu item. Either way, the TDI project builder maintains a configuration file that is updated when it is invoked. When a resource has been changed (modified, added or removed) the builder will update the runnable configuration file with that resource update. Only recognized configuration files will be acted upon; that is, AssemblyLines but not .gif files for example. The runnable configuration file is usually hidden since the file name starts with a dot (.). The file is named `.rs.xml`, located under the project folder. This is the compiled configuration file that is sent to a TDI server for execution.

The builder will relocate and rename components in the target configuration. If the Resources folder contains both properties and connectors, they will be relocated to their standard folders in the target configuration (that is, Properties and Connectors folder).

The project builder also checks all modified components for obvious errors and potential problems. These problems are logged to the standard Eclipse **Problems** view. Each problem item contains a description of the problem as well as the location so you can double-click and activate the editor where the problem was identified.

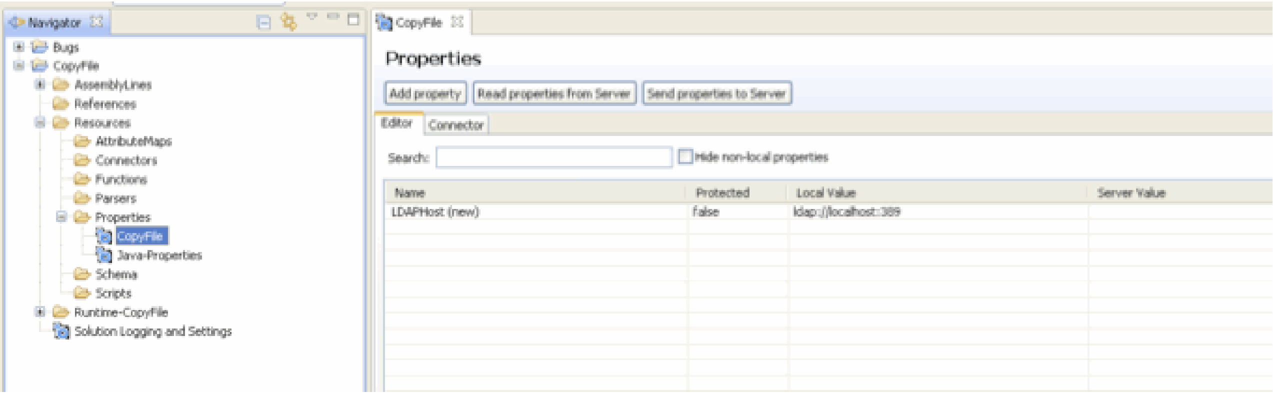
Properties and substitution

Every TDI project is associated with a TDI server. The associated server is the server that will be used to execute the AssemblyLines in the project. Since the server may be located on a different machine, the project model must provide access to properties through local copies of chosen property stores.

Property stores can be downloaded from the server as needed where the local copy contains two values for each property. One value is what was downloaded from the server (remote value), and the other is the value set by the user (local value). This is done so as to be able to see which properties are in possible conflict, as well as being able to extract the set of properties in use by the solution. There is no requirement to download a property store before adding properties to it though; however, when you run the solution any properties with a local value will be checked against the value on the server to prevent unintentional overwrites of existing properties.

Editing the property files can be done by opening (or creating) the property files in the Resources folder. After you have created a property file you can modify its contents and perform upload and download. Upload and Download is what you do to synchronize the properties locally with those on the server.

Note: TDI currently uses the equal sign "=" or colon ":" as the separator in key/value pairs property files, whichever is first. Using equal signs or colons in property names and property values is therefore not supported. The property file key/value separator in TDI V6.0 and earlier was only the ":" character; therefore, property files migrated from V6.0 and earlier may require editing.



Name	Protected	Local Value	Server Value
LDAPHost (new)	false	ldap://localhost:389	

Figure 8. Properties view

Note that custom property stores with a relative path (for example, the Connector configuration uses a relative path) will have a corresponding file generated in the run time, *Project* directory. When you create new custom properties the default path will be set to "{config.$directory}/*Filename*.properties" where *Filename* is the name you give your new property store. The "{config.$directory}" resolves to the location of the run time file.

By default, the shared property stores are not added to a project (for example, Global, Solution and System store). You can still add those to your project if you want to maintain changes to those files. To view the shared property stores, you should use the Servers view or use **Browse System Stores** on the main toolbar.

The User Interface Model

The user interface (UI) is provided by a set of views, editors and other UI-related resources. These are provided through standard extension point mechanisms as defined by the Eclipse platform.

While there are many extension point contributions in the CE, only the most important ones are listed here.

Table 7. Eclipse CE extension point contributions

Contribution	Description
Editors	Editors for all major configuration files are provided: • AssemblyLine (`.assemblyline` files) • Connector (`.connector` files) • Function (`.function` files) • Scripts (`.script` files) • Properties (`.tdiproperties` files) • AttributeMap (`.attributemap` files)
Views	Several views are provided to aid in visualizing various aspects of configuration files.
Menus, Toolbars	All actions in the CE that operate on configuration objects are defined as standard actions.
Wizards	Several wizards are provided to aid in creating new projects and configuration files.

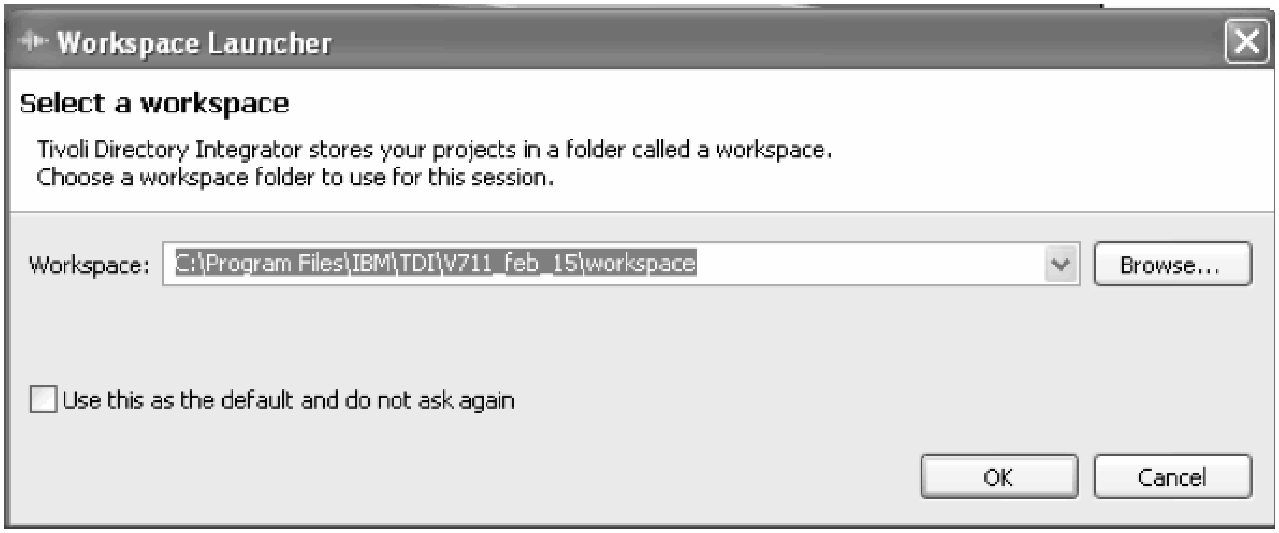
The CE follows the MVC (model, view, controller) paradigm. The editor for a configuration file creates the widgets that show the contents of the configuration file. The widgets register toolbars and menus they use with the Eclipse framework so contributions can be made to these. All actions that affect the configuration file are implemented and contributed using standard Eclipse mechanisms.

The User Interface

The Application Window

The first time you start the Config Editor (CE), it will prompt you for a workspace directory. The workspace directory is the filesystem location where your projects are stored. You can change this directory later from the **File** menu.

This dialog will show up every time you start the CE unless you select the check box to make the specified workspace directory the default location.



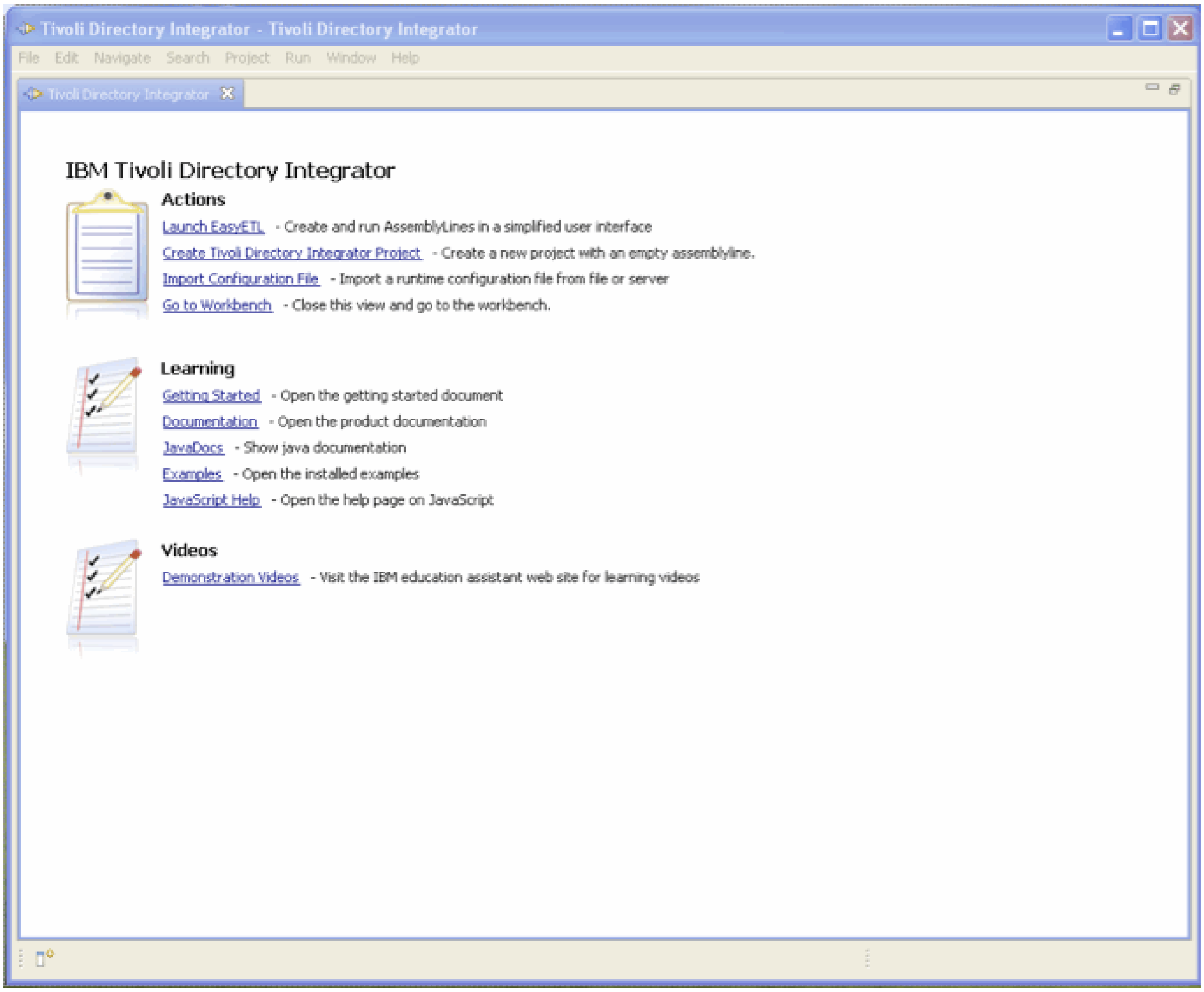
Select a workspace

Tivoli Directory Integrator stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Program Files\IBM\TDI\V711_feb_15\workspace

□ Use this as the default and do not ask again

After this dialog the main workspace window will appear. If this is the first time you start the CE, it will show the welcome screen:


Tivoli Directory Integrator - Tivoli Directory Integrator
File Edit Navigate Search Project Run Window Help
Tivoli Directory Integrator

IBM Tivoli Directory Integrator

Actions

Launch EasyETL - Create and run AssemblyLines in a simplified user interface
Create Tivoli Directory Integrator Project - Create a new project with an empty assemblyline.
Import Configuration File - Import a runtime configuration file from file or server
Go to Workbench - Close this view and go to the workbench.

Learning

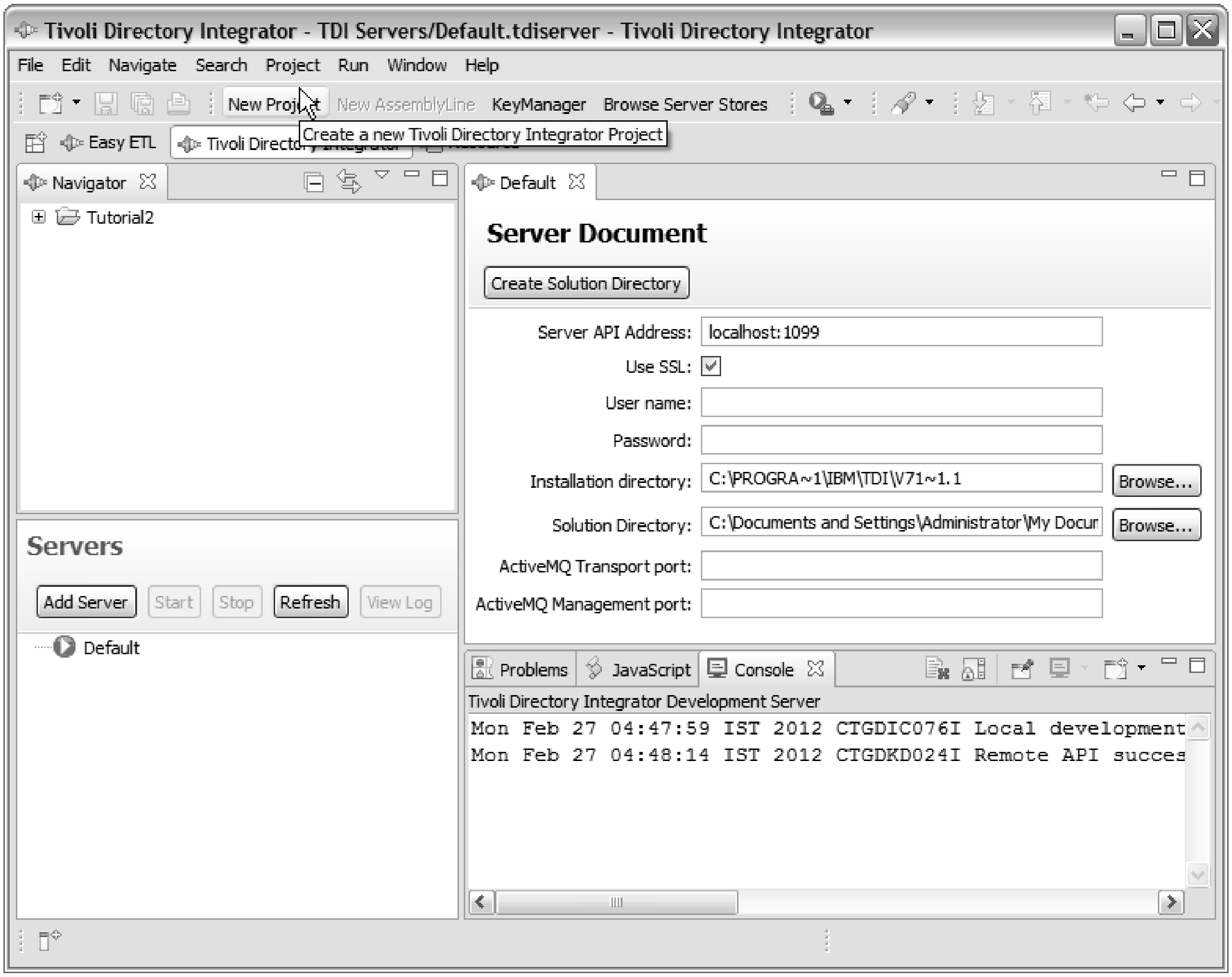
Getting Started - Open the getting started document
Documentation - Open the product documentation
JavaDocs - Show java documentation
Examples - Open the installed examples
JavaScript Help - Open the help page on JavaScript

Videos

Demonstration Videos - Visit the IBM education assistant web site for learning videos

This welcome screen provides a few quick links to common tasks and information sites. The documentation link takes you to the configured product documentation (system property *com.ibm.tdi.helpLoc*).

The welcome screen can be reopened later by choosing **Help** > **Welcome**. When the welcome screen is closed you will see the workspace window:


Tivoli Directory Integrator - TDI Servers/Default.tdiserver - Tivoli Directory Integrator
File Edit Navigate Search Project Run Window Help
New Project
New AssemblyLine
KeyManager
Browse Server Stores
Create a new Tivoli Directory Integrator Project
Easy ETL
Tivoli Directory Integrator
Navigator
Tutorial2
Default
Server Document
Create Solution Directory
Browse...
Browse...
Servers
Add Server
Start
Stop
Refresh
View Log
Default
Problems
JavaScript
Console
Tivoli Directory Integrator Development Server

Server API Address: localhost:1099
Use SSL: ☑
User name:
Password:
Installation directory: C:\PROGRA~1\IBM\TDI\V71~1.1
Solution Directory: C:\Documents and Settings\Administrator\My Docur
ActiveMQ Transport port:
ActiveMQ Management port:

```
Mon Feb 27 04:47:59 IST 2012 CTGDIC076I Local development
Mon Feb 27 04:48:14 IST 2012 CTGDKD024I Remote API succes
```

This is the main window where you manage your projects and configurations.

In this picture there is one open editor (for the Default.server document) and a number of views. The most important views are the ones you see in this picture.

The *Navigator* (upper left) contains all the projects and source files for server configurations and TDI solutions. The navigator can also contain other files and projects such as text files and so forth. The CE will treat TDI projects specifically, so other files and projects remain unaffected by the CE. You will see how in the section about the project builder.

The *Servers* view (lower left) shows the status for each of the servers that are defined in the "TDI Servers" project. You can have as many servers defined as you like. The view provides a number of functions to operate on servers and their configurations. The refresh button will refresh status for all servers in the view.

The *editor area* (upper right) is where all editors show up. When you open a document, such as an AssemblyLine configuration, it ends up in this area. This area is split vertically with an area (bottom right) that contains various views to provide other relevant information. Among the most important are the Problems view that shows potential problems with a TDI component, the Error Log that shows errors that occur while developing solutions and finally the Console view that shows the console log for running TDI servers (for example, those that are started by the CE).

Servers view

The Servers View contains a number of useful functions for managing server instances. Apart from adding and removing server instances there are many commands associated with the servers and their active config instances and AssemblyLines.

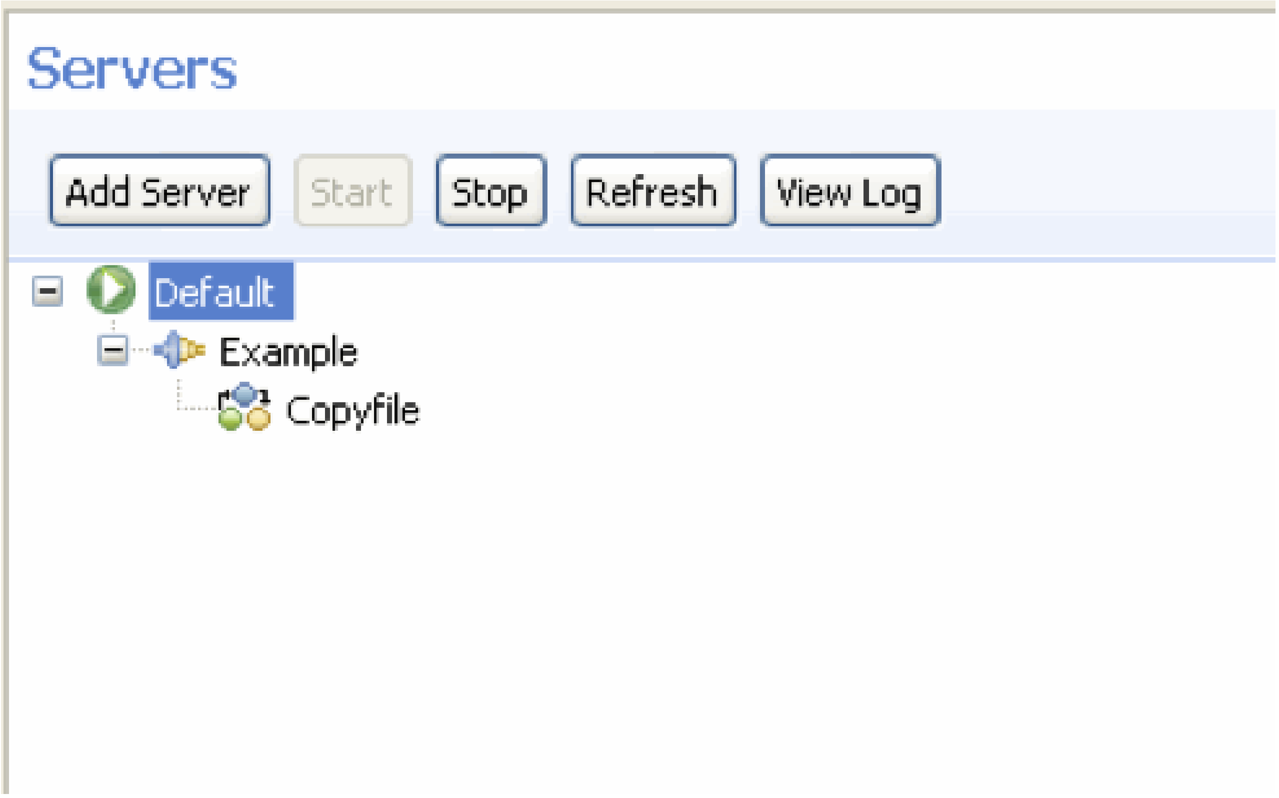


Figure 9. Servers view in the Configuration Editor

The main toolbar shows the following buttons:

Add Server
Use this to add another server

Start Use this to start a "local" server (for example, one that is accessible in your file system).

If you have a config instance selected you can start one or more of its AssemblyLines.

Stop Use this to send a stop request to the server, config instance or AssemblyLine.

Refresh
Use this to refresh the contents of the Servers view

View Log
Use this to view the standard log file, "ibmdi.log" file on a "local" server

The pop-up menu for each item in the view shows additional commands you can execute based on the selection.

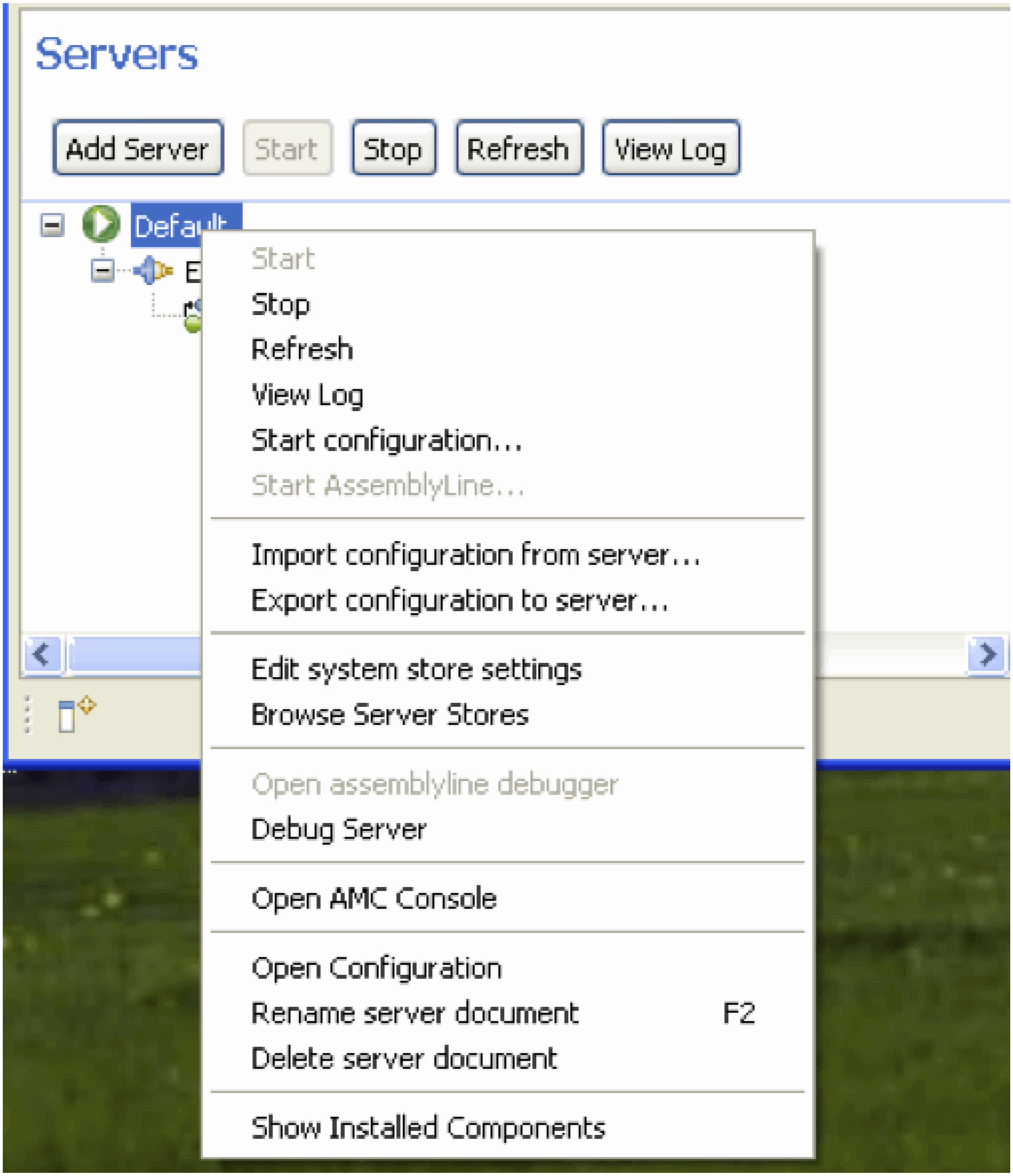


Figure 10. Servers view; pop-up menu

The possible commands are:

Start Start server

Stop Stop server, config instance or AssemblyLine.

Refresh
Refresh servers view.

View Log
Opens the ibmdi.log file in a text editor

Start configuration...
When a server is selected you can start a configuration instance on that server.

Start AssemblyLine....
When a configuration instance is selected you can start an AssemblyLine from that configuration instance.

Import configuration from server...
Lets you import a configuration from the server to a CE project.

Export configuration to server...
Lets you export a CE project to a runtime configuration on the selected server.

Edit system store settings
Opens the system store settings for the selected server

Browse System Stores
Opens the system store data browser to view/edit system store tables.

Open AssemblyLine debugger
Attaches a debugger to the selected AssemblyLine. This will let you debug an AssemblyLine that is already running.

Debug Server
Opens a server debug session for the selected server

Show Installed Components
Shows a list of installed components and their versions for the selected server

Open AMC console
Opens the AMC console for the selected server. If the server was installed without AMC this option is grayed out.

Delete server document
Deletes the server from the servers view

Rename server document
Renames the selected server. This has no effect on the server itself; it is only a local representation of the server.

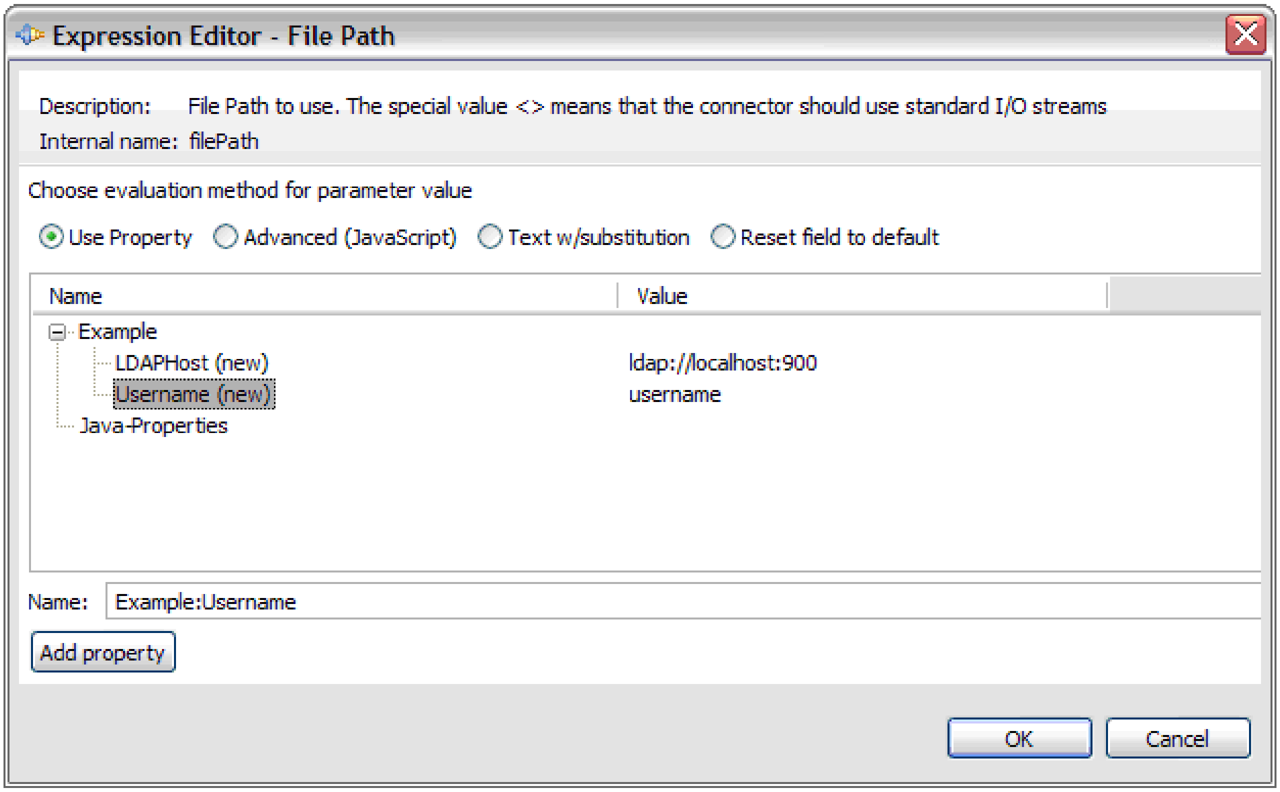
Menu options are disabled and enabled based on the selection.

The Expression Editor

The expression editor is available in many different contexts. Often, when specifying parameter values such as connection parameters, link criteria and so forth you can use the expression editor instead of typing a simple value for the parameter.

Use Property

This option lets you choose an existing property from your property stores or create a new property/value pair.



Description: File Path to use. The special value <> means that the connector should use standard I/O streams

Internal name: filePath

Choose evaluation method for parameter value

● Use Property ○ Advanced (JavaScript) ○ Text w/substitution ○ Reset field to default

Name	Value
Example	
LDAPHost (new)	ldap://localhost:900
Username (new)	username
Java-Properties	

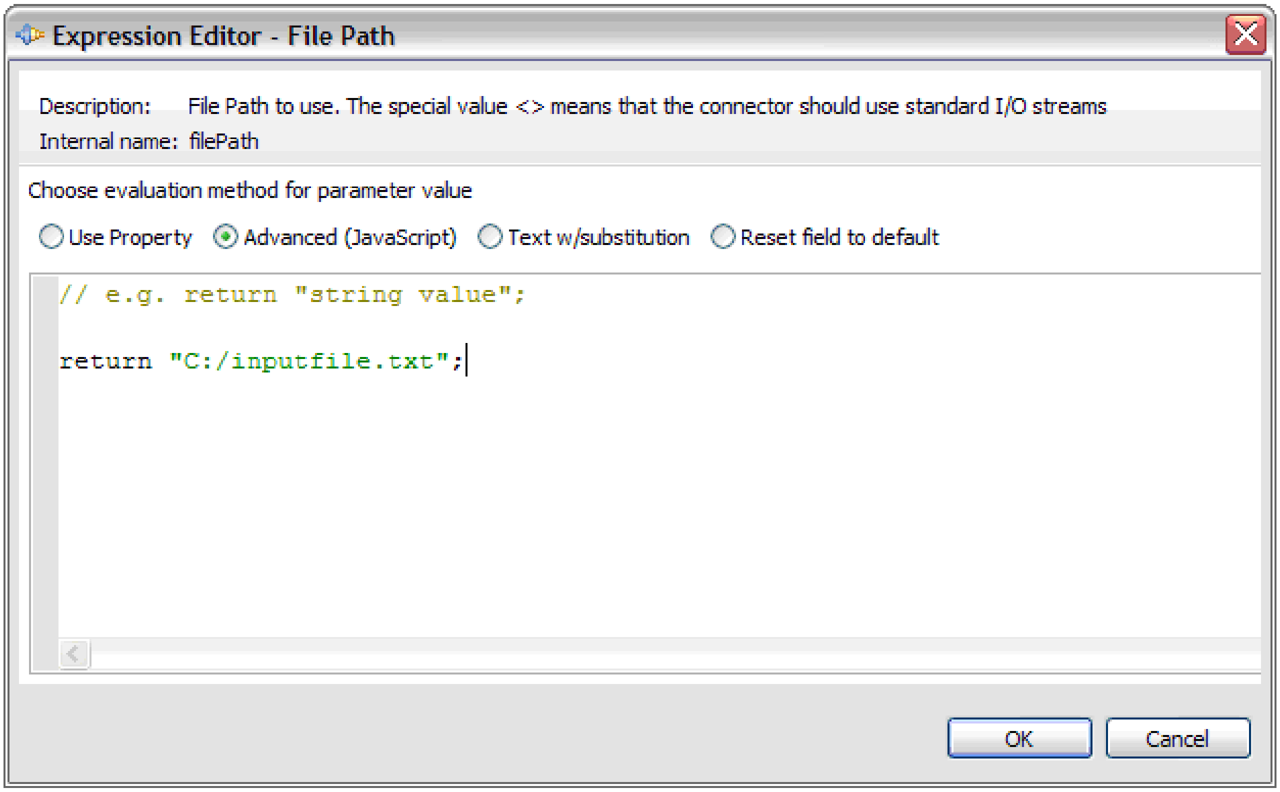
Name: Example:Username

Figure 11. Expression Editor: simple property

When you select a property, the *Name* text field below the tree is updated with the expression for that property. By default, the expression includes the store name ("Example" in this case) followed by a colon and the property name ("Username" in this case). When you click **OK**, the expression in the text field is used for the parameter. This also means that you can type the expression directly in this field without going through the tree of properties. If you for example know that there is a property called "FilePath" on the server this solution will run on, you can remove the store name if you don't know that (that is, type "FilePath" without the "store:" prefix).

Advanced (JavaScript)

With this option the value is computed as the result of JavaScript code you enter in the editor.



Description: File Path to use. The special value <> means that the connector should use standard I/O streams

Internal name: filePath

Choose evaluation method for parameter value

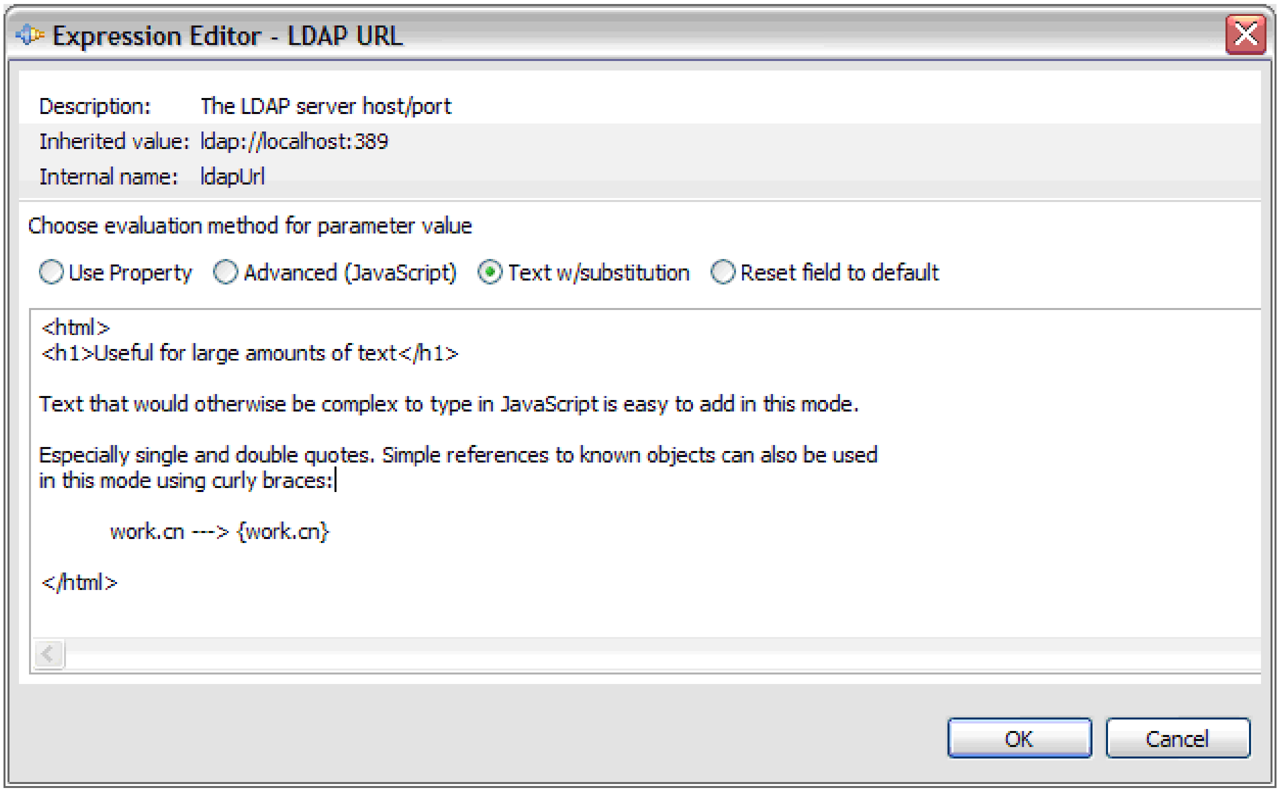
○ Use Property ● Advanced (JavaScript) ○ Text w/substitution ○ Reset field to default

```
// e.g. return "string value";

return "C:/inputfile.txt";
```

Figure 12. Expression Editor: Advanced (JavaScript)

Text with substitution

This is the "TDI Expression" from version 6.x. From version 7 onwards, it is recommended to use JavaScript for complex evaluation of variables and properties. This option however is very useful when you want to enter large amounts of text. The substitution options are compatible with version 6 expressions.



Description: The LDAP server host/port

Inherited value: ldap://localhost:389

Internal name: ldapUrl

Choose evaluation method for parameter value

○ Use Property ○ Advanced (JavaScript) ● Text w/substitution ○ Reset field to default

```
<html>
<h1>Useful for large amounts of text</h1>

Text that would otherwise be complex to type in JavaScript is easy to add in this mode.

Especially single and double quotes. Simple references to known objects can also be used
in this mode using curly braces:

          work.cn ---> {work.cn}

</html>
```

Figure 13. Expression Editor: text with v.6-style substitution

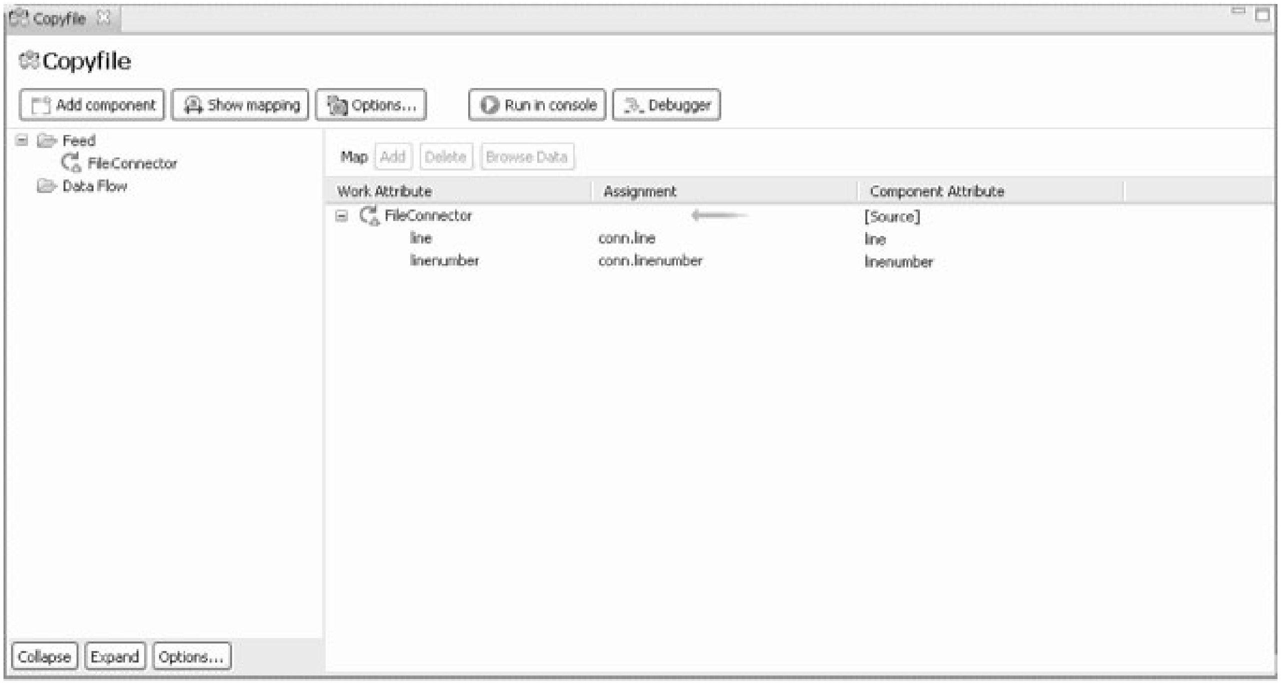
Reset field to default

The last option is used to reset the parameter value to its default value (also known as the inherited value). Select this option and press **OK** to reset the parameter value.

The AssemblyLine Editor

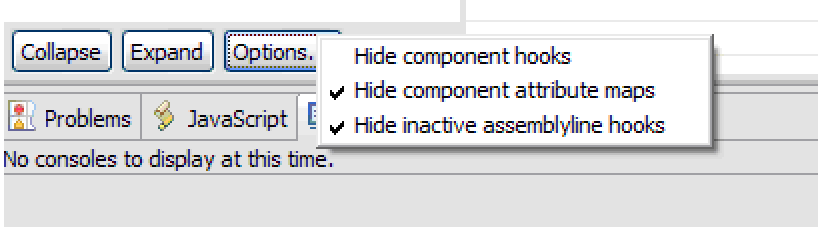
The AssemblyLine editor is the principal editor used when developing Tivoli Directory Integrator solutions.

In this editor you build the AssemblyLine by adding and configuring components. As you go along you can run the AssemblyLine to see the effects of the added components.



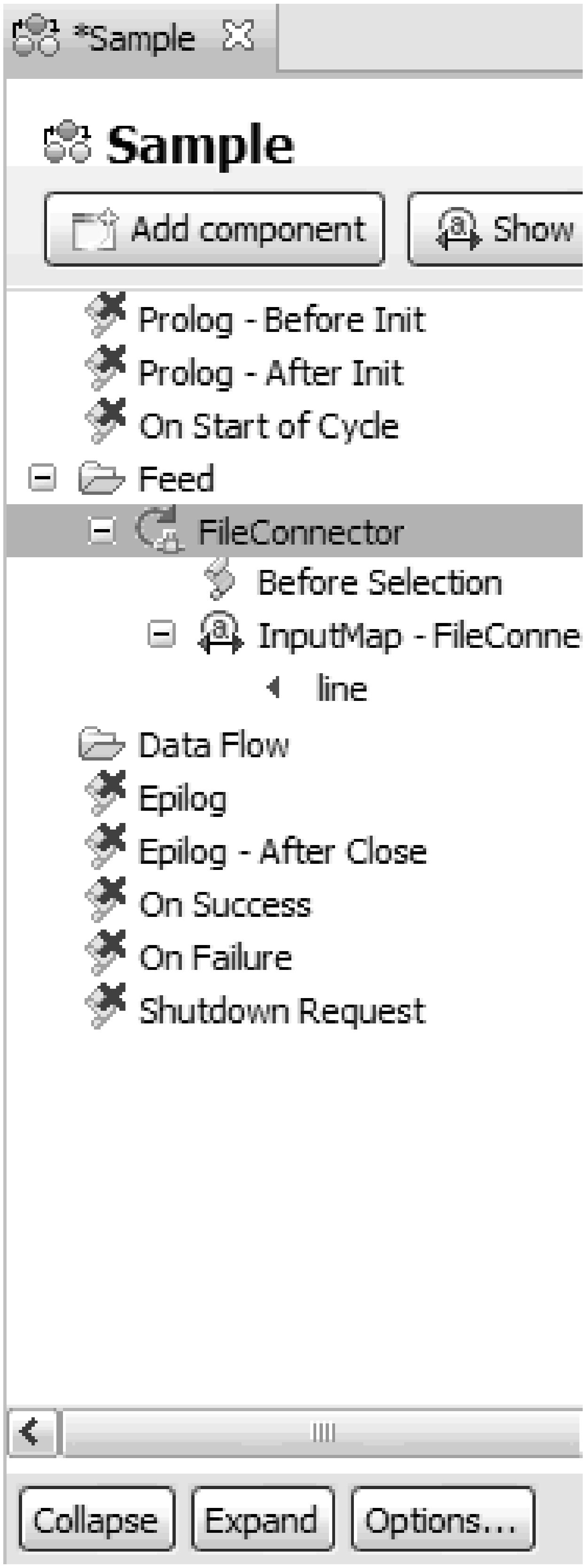
Work Attribute	Assignment	Component Attribute
FileConnector	←	[Source]
line	conn.line	line
linenumber	conn.linenumber	linenumber

The AssemblyLine editor shows two main sections. The section on the left shows the AssemblyLine components and hooks. In the toolbar you can choose the level of detail you want to see in the tree.

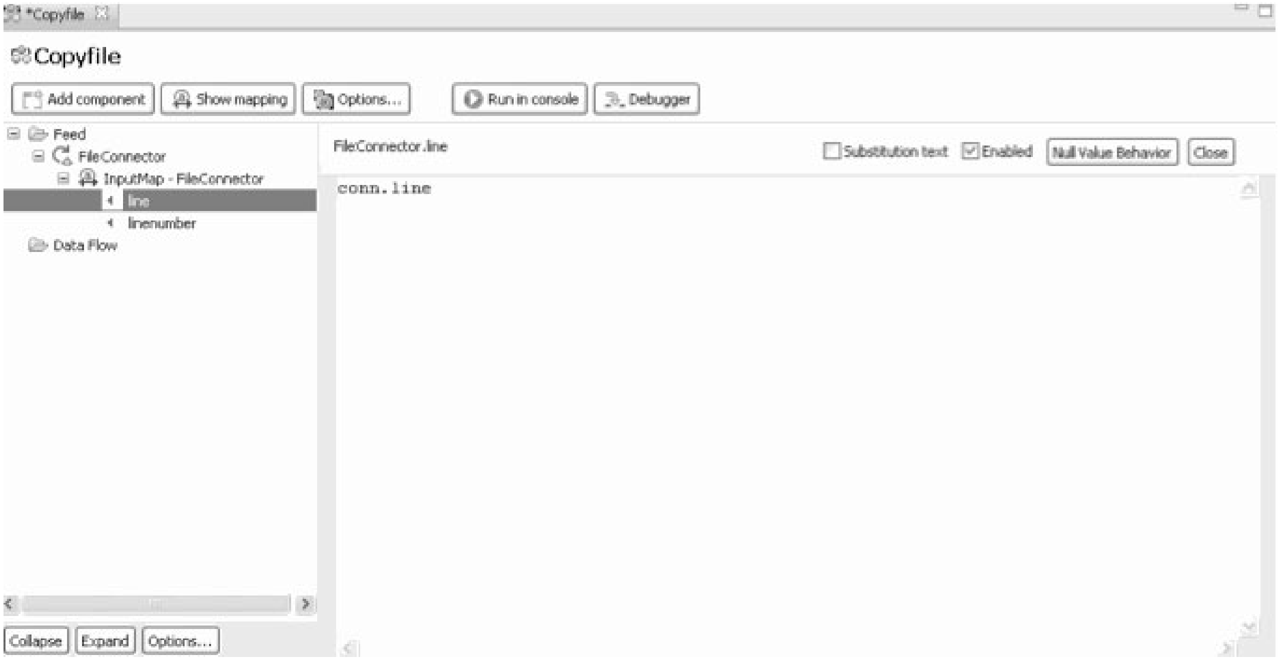

Collapse
Expand
Options...
Hide component hooks
✓ Hide component attribute maps
✓ Hide inactive assemblyline hooks
Problems
JavaScript

No consoles to display at this time.

The **Options...** button lets you choose how much of the AssemblyLine is revealed in the component tree view.



This picture shows all AssemblyLine hooks as well as component attribute maps in the tree. When you select hooks or attribute map items in this tree, the right hand side will provide a larger view of the item than you would inside the component editor itself:



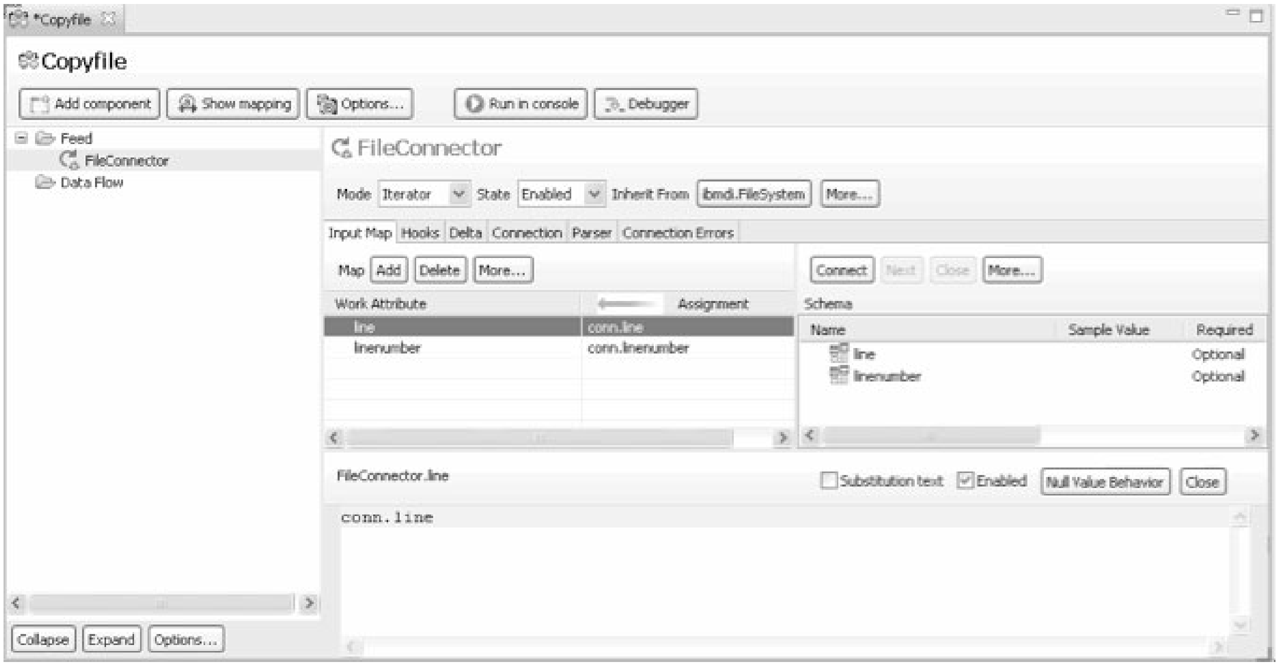
FileConnector.line

☐ Substitution text ☑ Enabled

```
conn.line
```

The mapping section on the right shows the maps for all components with an attribute map. The first level in this tree is the component name with the individual attribute map items below them. Selecting such an item brings up the details editor for that item. If you show hooks in the AssemblyLine components view you can double click those as well to bring up the script editor.

The picture below shows how the quick editor appears with the script for the attribute map.



Mode Iterator State Enabled Inherit From ibmdi.FileSystem

Work Attribute	Assignment
line	conn.line
linenumber	conn.linenumber

Name	Sample Value	Required
line		Optional
linenumber		Optional

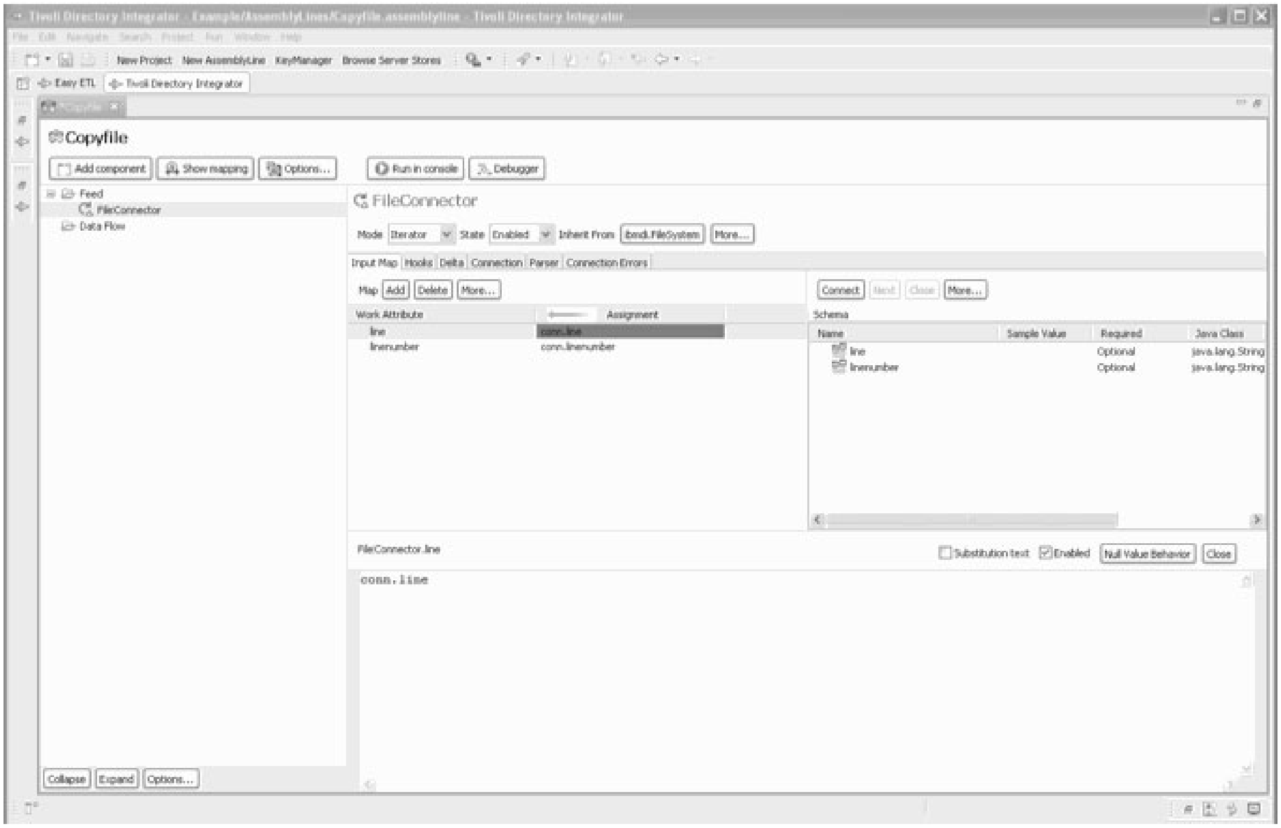
FileConnector.line

☐ Substitution text ☑ Enabled

```
conn.line
```

The quick editor also shows some additional options. The "Substitution text" is the version 6 "TDI Expression" format where you can enter text and some simple expansion macros. From version 7, this format is primarily intended for large or complex text values since JavaScript provides more powerful expression syntax.

The component flow section shows all components in the AssemblyLine. When you select a component the right hand side of the editor is replaced with the configuration screen for that component. One useful short cut in the CE is the Ctrl-M short cut that maximizes the current editor or view to fill the application screen. Ctrl-M will toggle between maximized and normal size.



Mode Iterator State Enabled Inherit From ibmdi.FileSystem More...

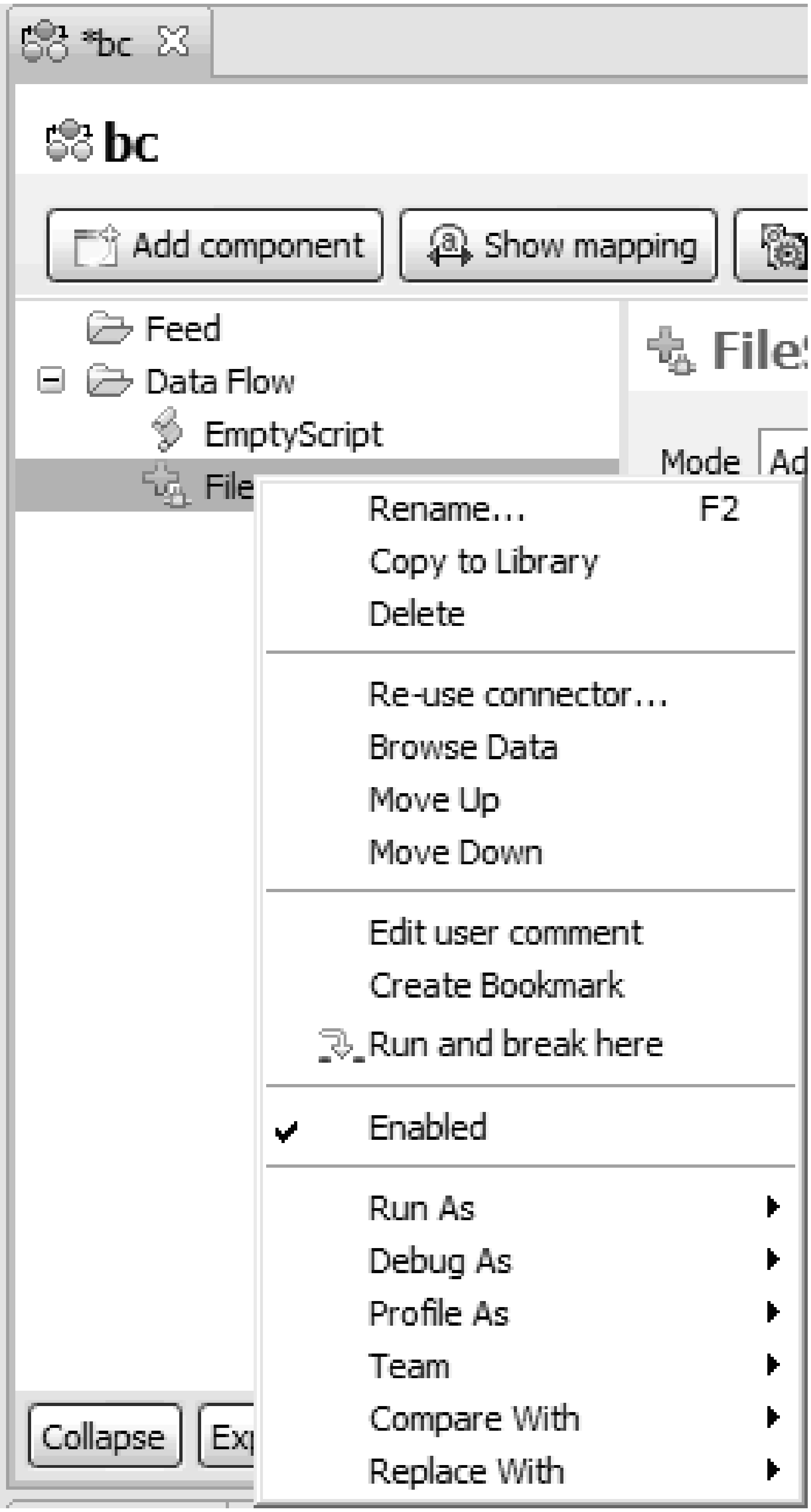
Work Attribute	Assignment
line	conn.line
linenumber	conn.linenumber

Name	Sample Value	Required	Java Class
line		Optional	java.lang.String
linenumber		Optional	java.lang.String

FileConnector.line ☐ Substitution text ☑ Enabled

```
conn.line
```

When you click a component, a configuration screen for that component replaces the overall attribute mapping view. You can also right-click the component to access the pop-up menu for the component.



AssemblyLine Options

From the **Settings** drop down button you can select a number of options.

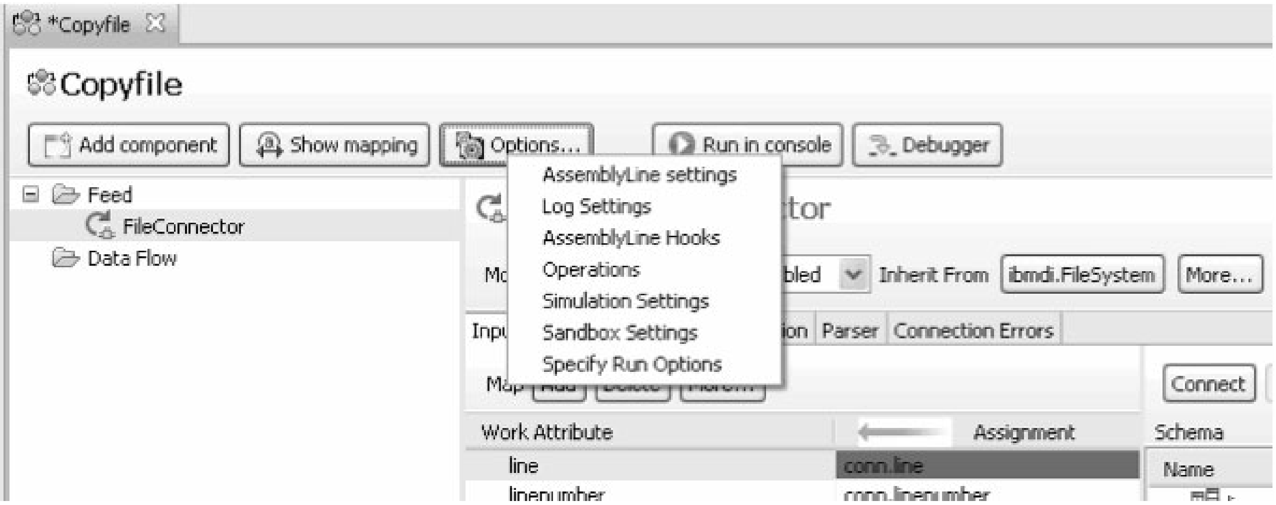
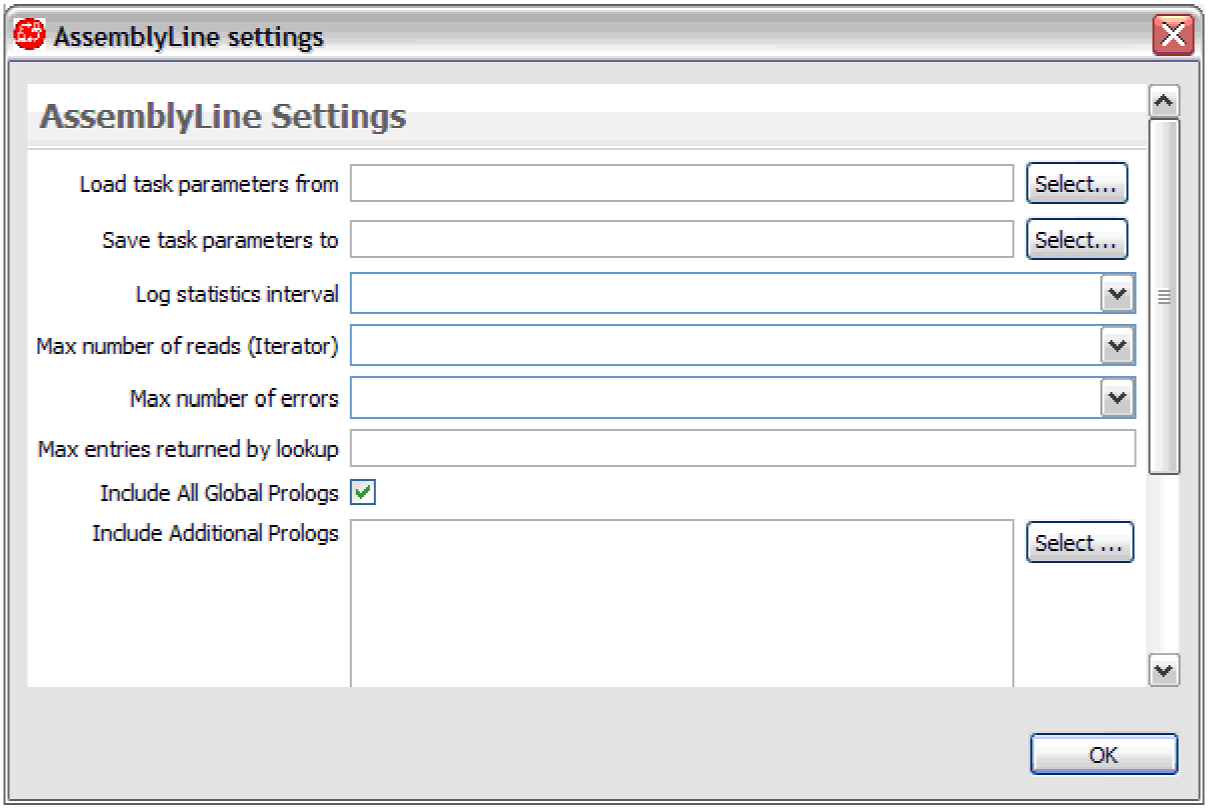


Figure 14. AssemblyLine options menu

Available from the drop-down menu are a number of options screens, governing various aspects of the AssemblyLine, both in terms of design as well as run-time options. These screens are:

- "AssemblyLine Settings" on page 89
- "Log Settings" on page 90
- "AssemblyLine Hooks" on page 91
- "AssemblyLine Operations" on page 92
- "Simulation Settings" on page 93
- "Sandbox Settings" on page 95

AssemblyLine Settings

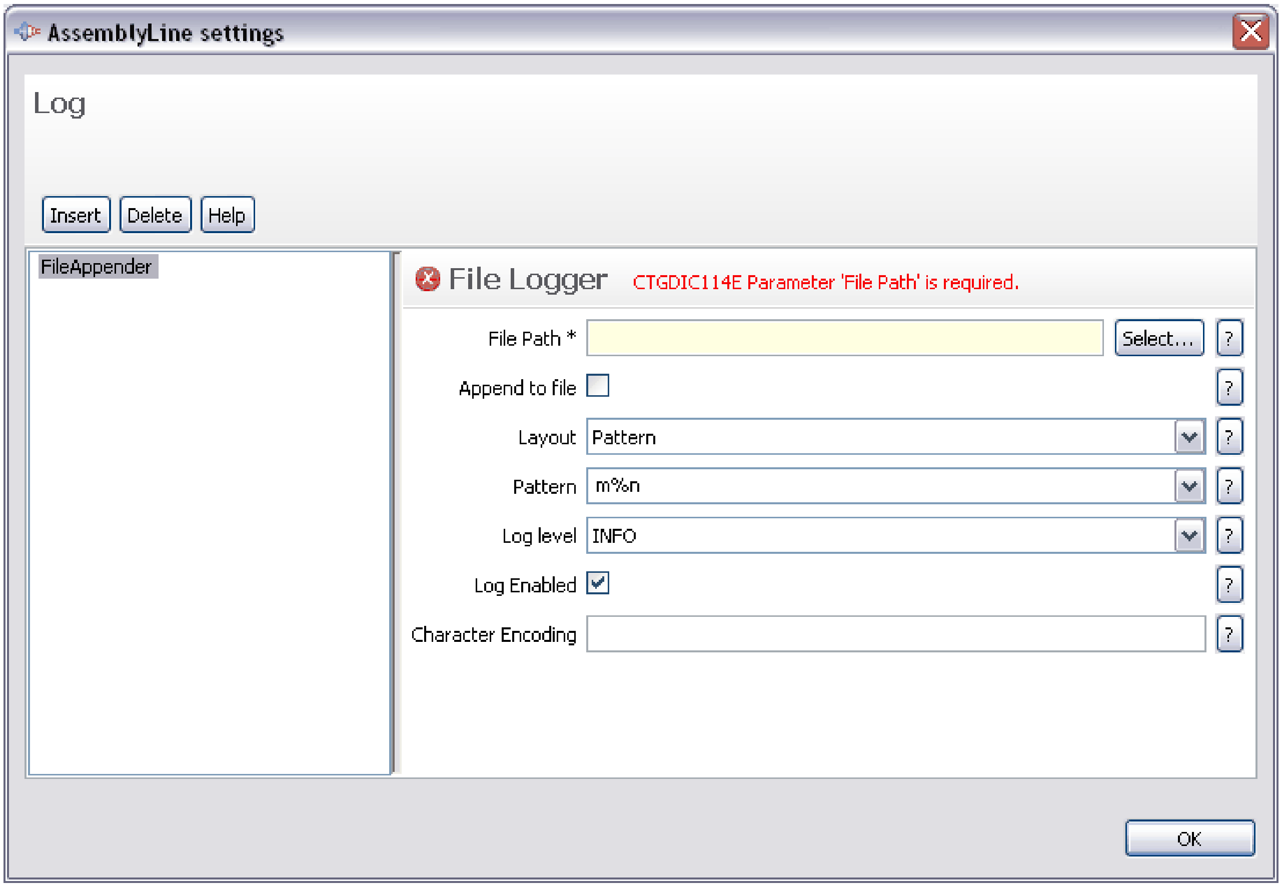


Include All Global Prologs ☑

Figure 15. AssemblyLine Settings

In this window you can specify options that affect how the AssemblyLine executes.

Log Settings



CTGDIC114E Parameter 'File Path' is required.

File Path *

Append to file □

Layout: Pattern

Pattern: m%n

Log level: INFO

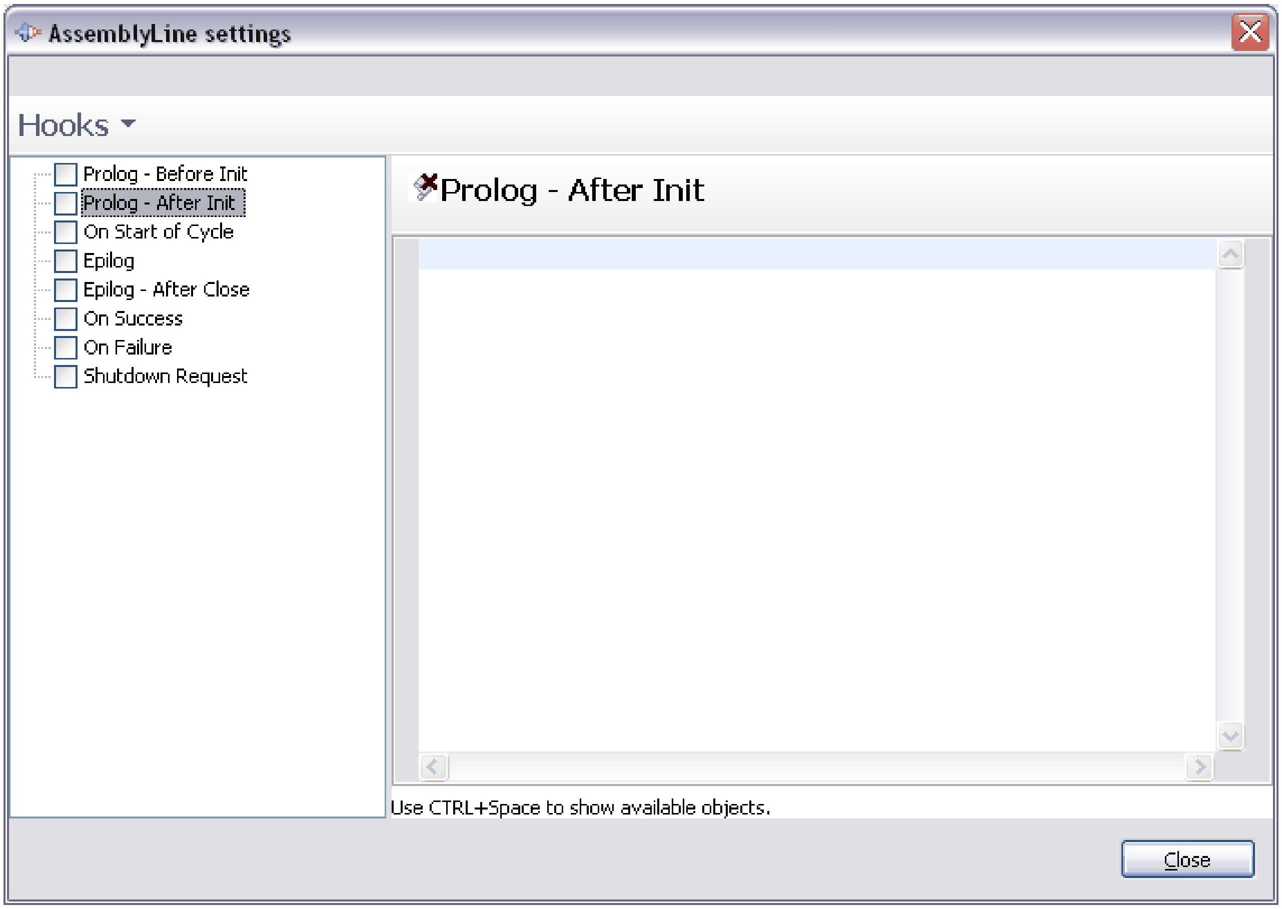
Log Enabled ☑

Character Encoding

Figure 16. AssemblyLine Log settings

In this window you can add loggers for this AssemblyLine. The loggers are not global, and only activated for the AssemblyLine.

AssemblyLine Hooks



□ Prolog - Before Init
□ Prolog - After Init
□ On Start of Cycle
□ Epilog
□ Epilog - After Close
□ On Success
□ On Failure
□ Shutdown Request

Prolog - After Init

Use CTRL+Space to show available objects.

Figure 17. AssemblyLine Hooks

In this window you can enable/disable AssemblyLine level hooks. Enabled hooks will also show up in the components panel in the AssemblyLine editor.

AssemblyLine Operations

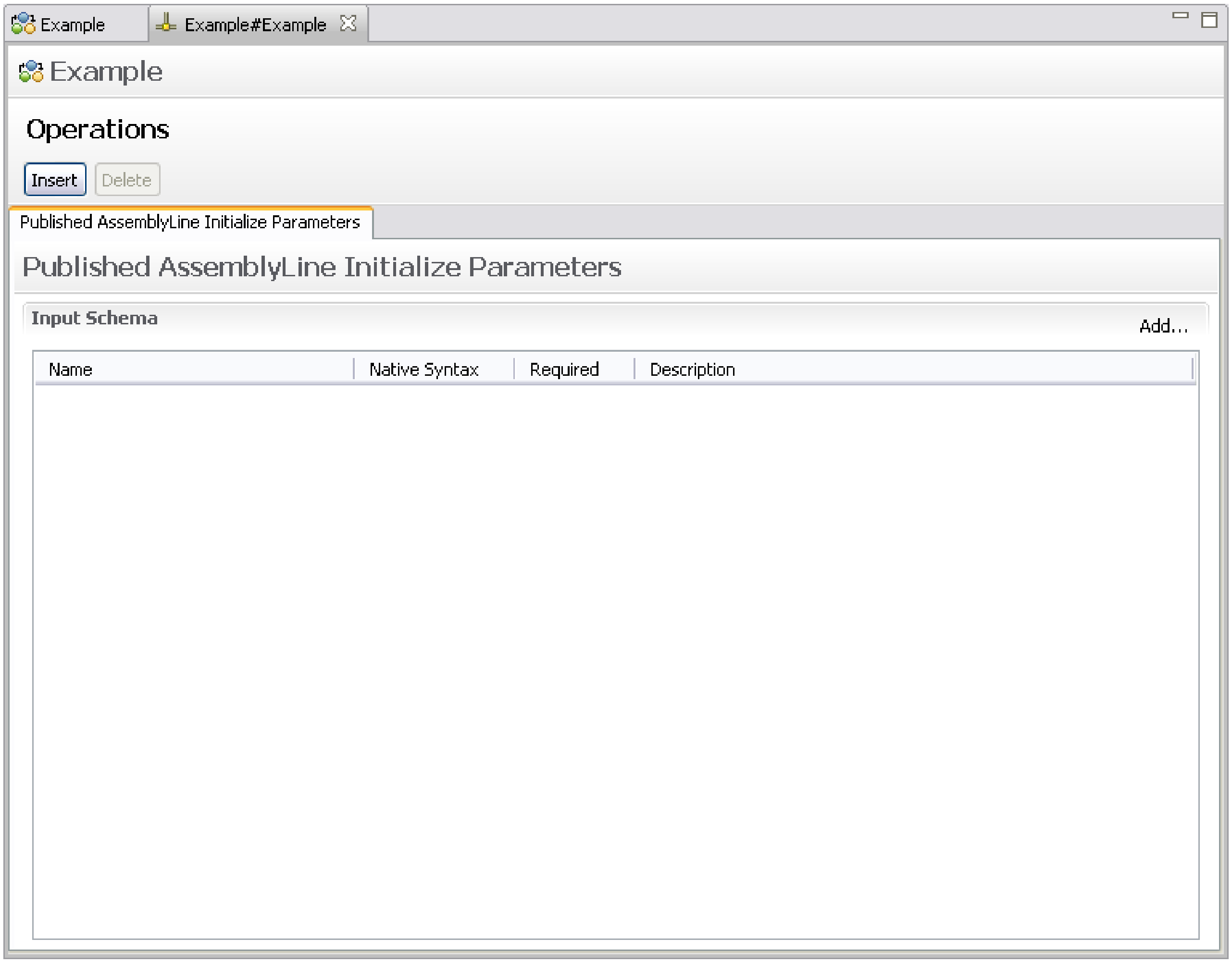
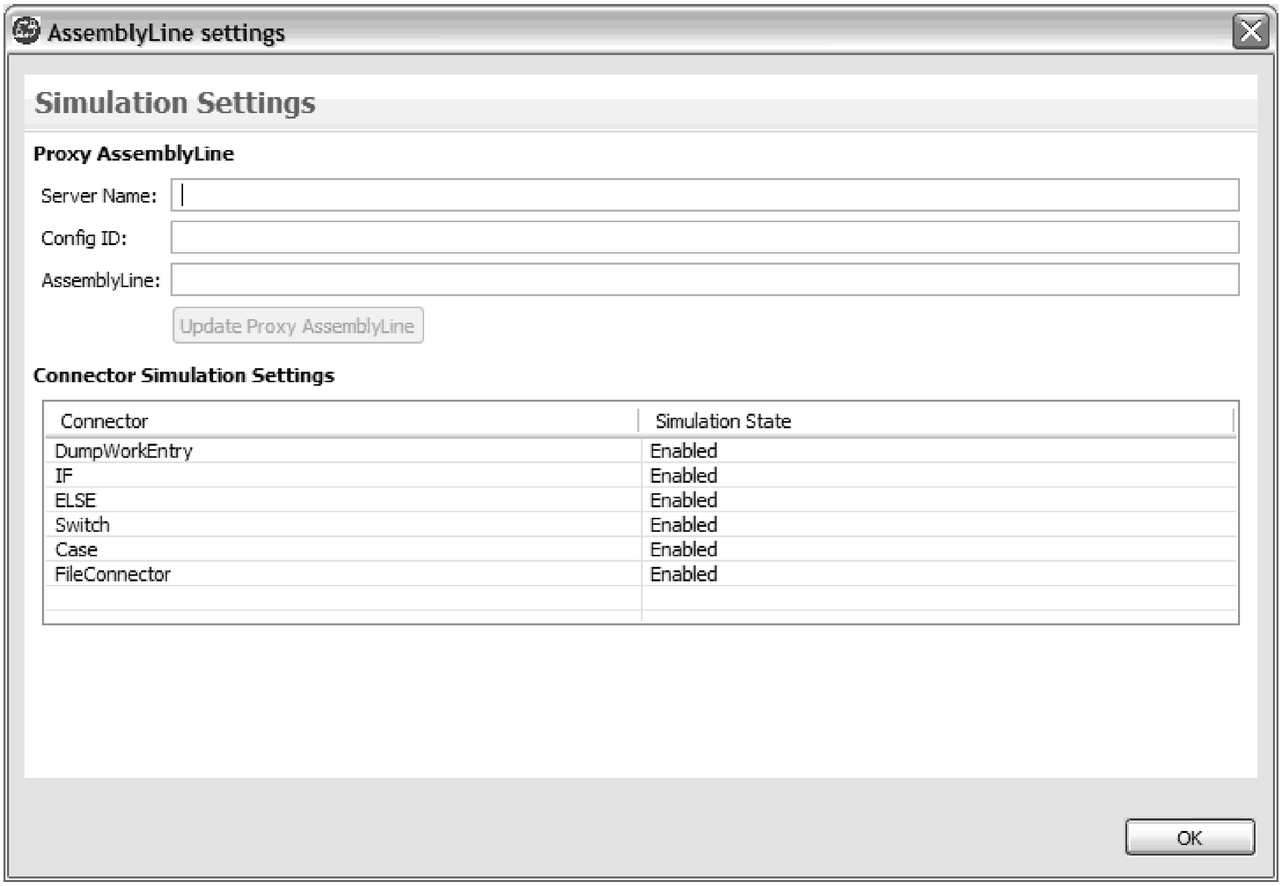


Figure 18. AssemblyLine Operations

This window lets you define AssemblyLine operations. See section AL Operations, in "Creating new components using Adapters" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide* for a complete description of operations. The Insert button lets you add a new operation. The **Published AssemblyLine Initialize Parameters**, which is available by default, is a special operation that is used to provide values to an AssemblyLine before components are initialized.

Simulation Settings



AssemblyLine settings

Simulation Settings

Proxy AssemblyLine

Server Name:

Config ID:

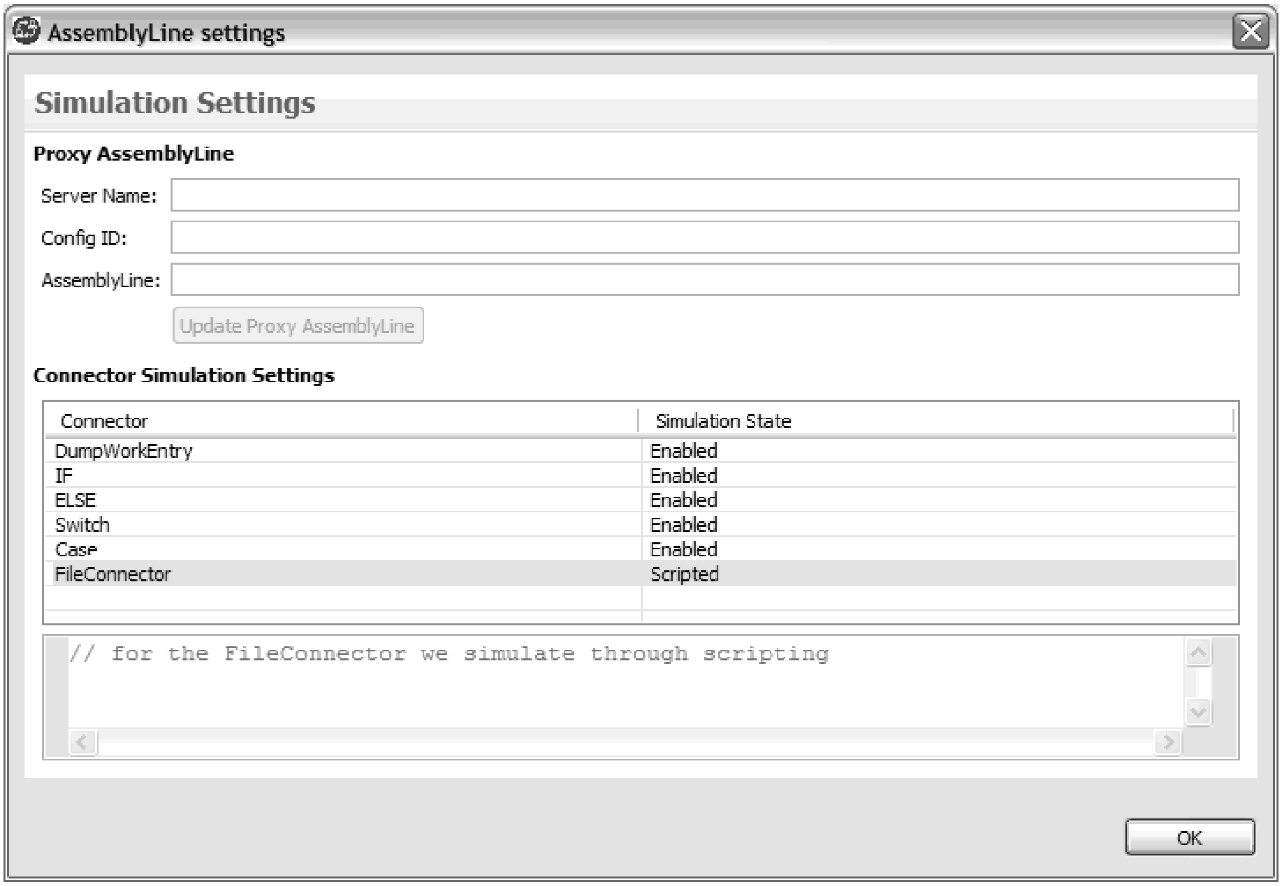
AssemblyLine:

Connector Simulation Settings

Connector	Simulation State
DumpWorkEntry	Enabled
IF	Enabled
ELSE	Enabled
Switch	Enabled
Case	Enabled
FileConnector	Enabled

Figure 19. AssemblyLine simulation settings

This window lets you configure the simulation settings per component. See “AssemblyLine Simulation Mode” on page 194 for more information. The panel will show a script editor at the bottom when you choose scripting for simulation:



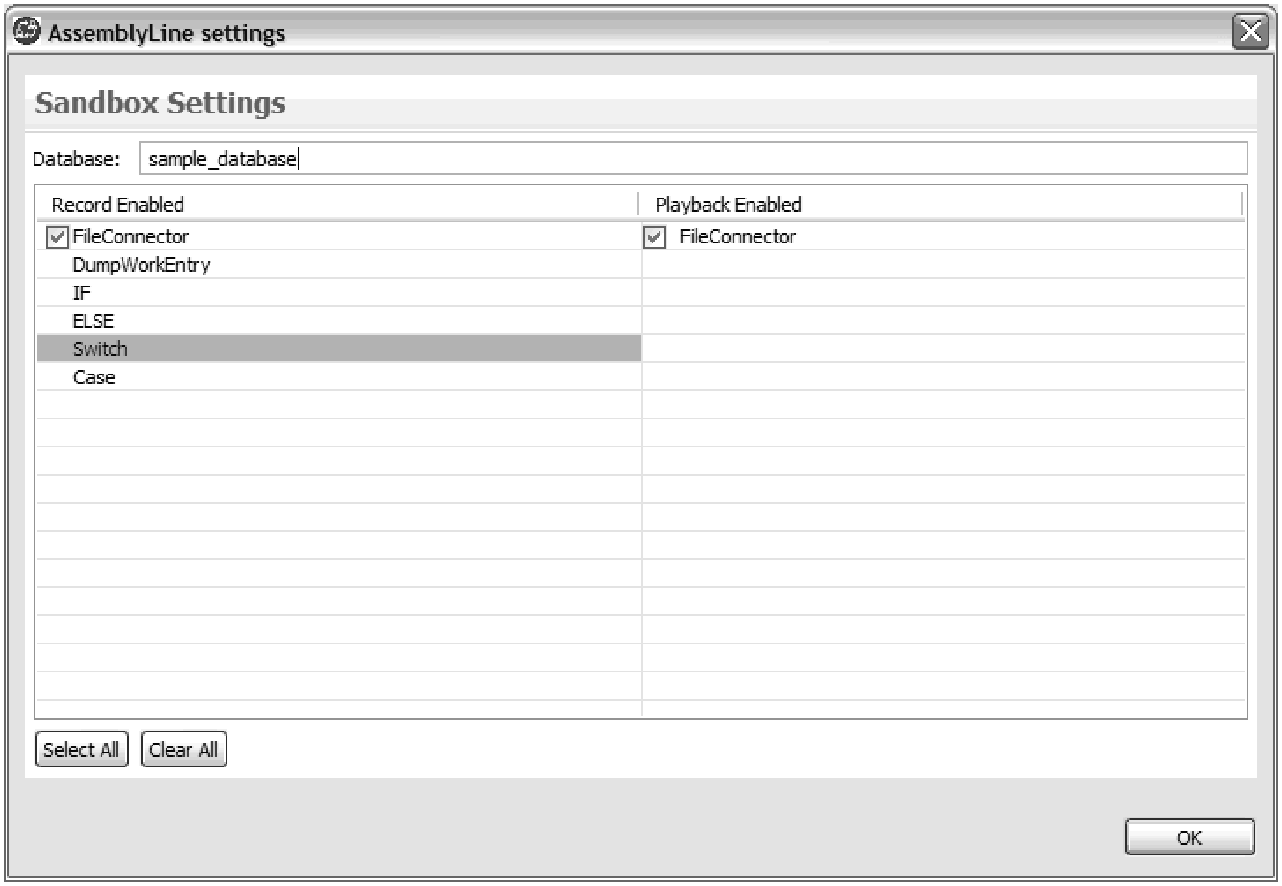
Connector	Simulation State
DumpWorkEntry	Enabled
IF	Enabled
ELSE	Enabled
Switch	Enabled
Case	Enabled
FileConnector	Scripted

```
// for the FileConnector we simulate through scripting
```

Figure 20. AssemblyLine simulation settings window, with script editor

You can also create/update the proxy AssemblyLine used by the simulation code with the **Update Proxy AssemblyLine** button.

Sandbox Settings



Database: sample_database

Record Enabled	Playback Enabled
☑ FileConnector	☑ FileConnector
DumpWorkEntry	
IF	
ELSE	
Switch	
Case	

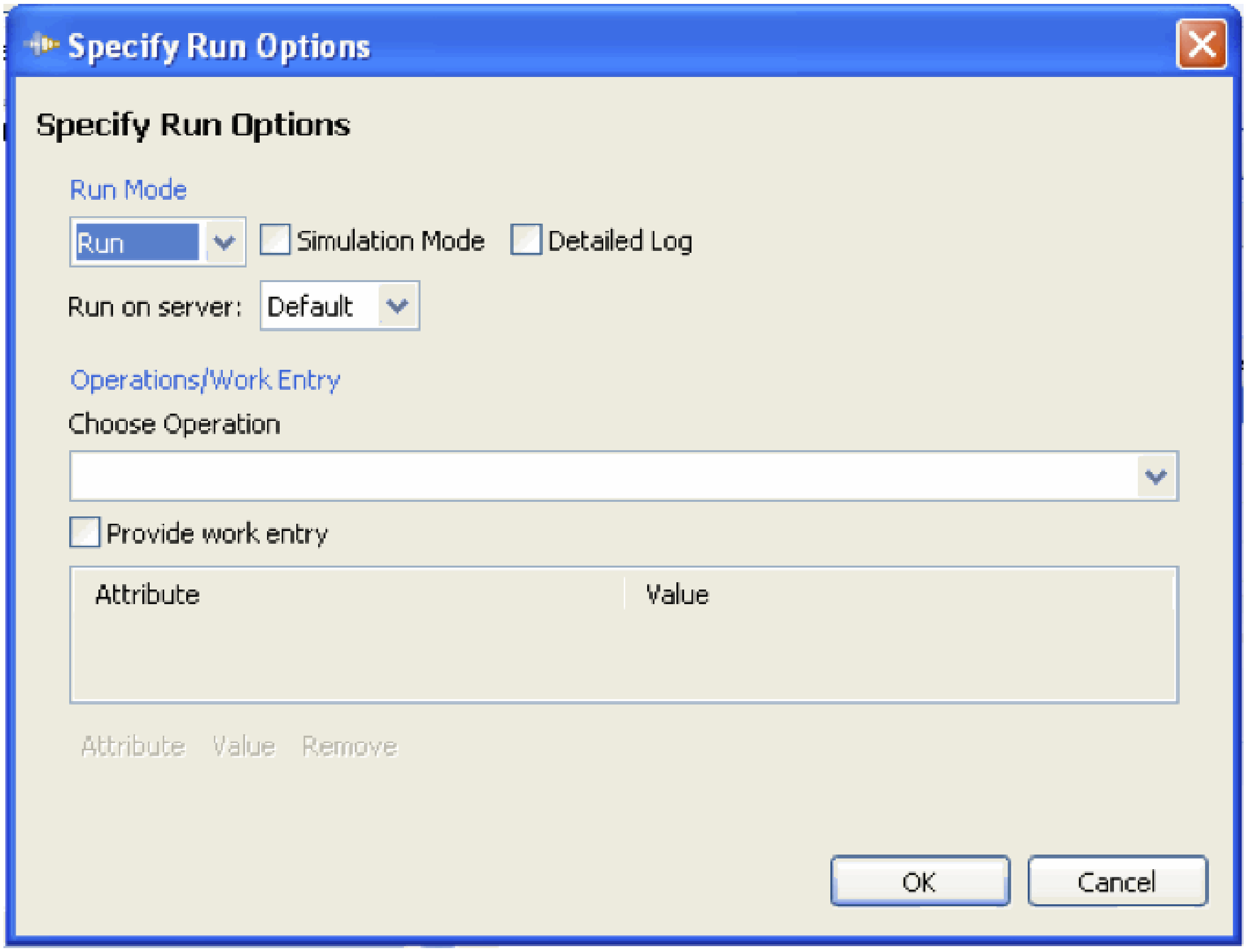
Figure 21. AssemblyLine Sandbox settings

The sandbox settings lets you configure which components are record/playback enabled when you run your AssemblyLine in either Record or Playback mode.

For more information about the Sandbox functionality in Tivoli Directory Integrator, see “Sandbox” on page 193.

Specify Run Options

The run options dialog lets you configure how the AssemblyLine is run.



Specify Run Options

Specify Run Options

Run Mode

Run □ Simulation Mode □ Detailed Log

Run on server: Default

Operations/Work Entry

Choose Operation

□ Provide work entry

Attribute	Value

Figure 22. Specify Run Options dialog

When you select **Provide work entry**, you can build a static Entry that is fed into the AssemblyLine when it starts.

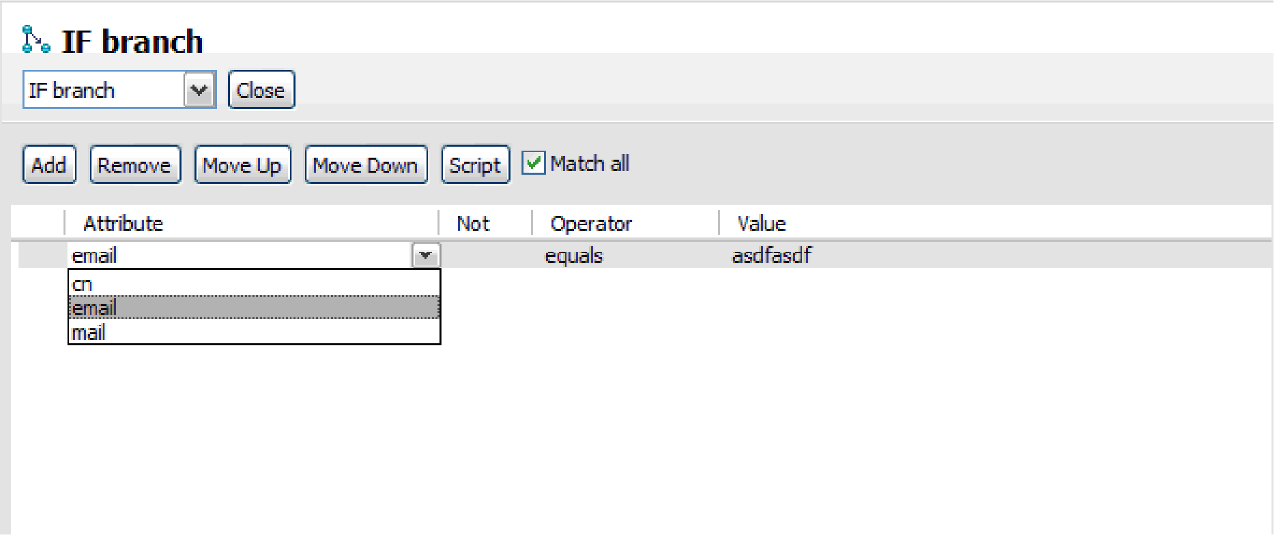
Component panels

When you open a component in the AssemblyLine you will get a quick editor panel in the lower part of the AssemblyLine window.

You will get this for the following components:

- "IF/ELSE/ELSE-IF Branch" on page 97
- "Switch/Case Branch" on page 97
- "For-Each Attribute Value" on page 99
- "Conditional Loop" on page 99
- "Connector Loop" on page 100
- "Attribute Map" on page 101

IF/ELSE/ELSE-IF Branch



Attribute	Not	Operator	Value
email		equals	asdfasdf

Figure 23. Quick Editor for IF/ELSE/ELSE-IF Branch

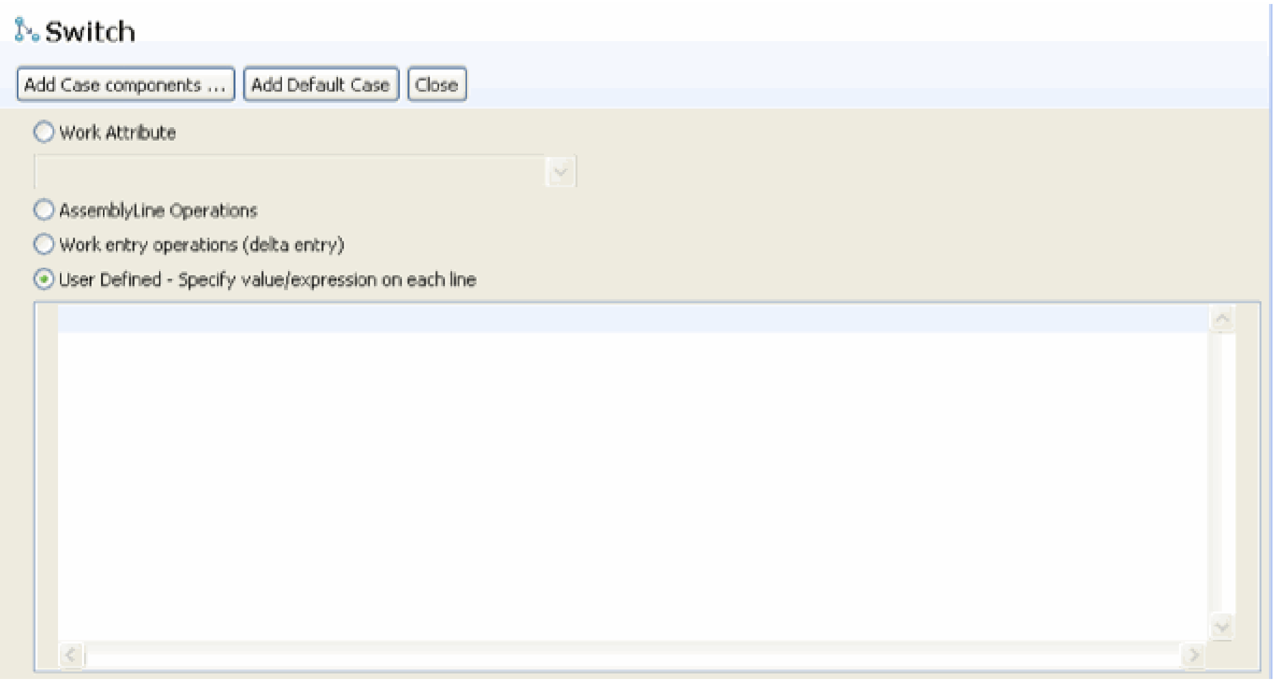
The IF and ELSE-IF branch lets you specify simple expressions and a custom script that returns *true* or *false*. Use the **Match all** checkbox to return true only if all conditions evaluate to *true* (*AND* mode). If unchecked only one of the conditions must match (*OR* mode).

The ELSE branch has no parameters.

Use the **Add** button to add new rows of Attribute/Operator/Value controls. You can remove these rows using the **Delete** button. The value can be a constant or an expression. Use the expression editor to configure the expression by clicking the button following the value text field. The move buttons are used to reorder the expressions. The expressions are evaluated from top to bottom. You can also change the branch type (that is, IF, ELSE and so forth) using the drop down below the title.

Switch/Case Branch

The Switch configuration has a number of options to select a value. This value is used to match the values in the contained Case branches. When they are equal the Case branch is executed. The switch branch is always executed.



○ Work Attribute

○ AssemblyLine Operations

○ Work entry operations (delta entry)

● User Defined - Specify value/expression on each line

Figure 24. Switch/Case Branch

Selection options you can choose for your Switch/Case branch are:

Work Attribute
Select from the list of known work attributes

AssemblyLine Operations
Select from the list of know AssemblyLine operation names.

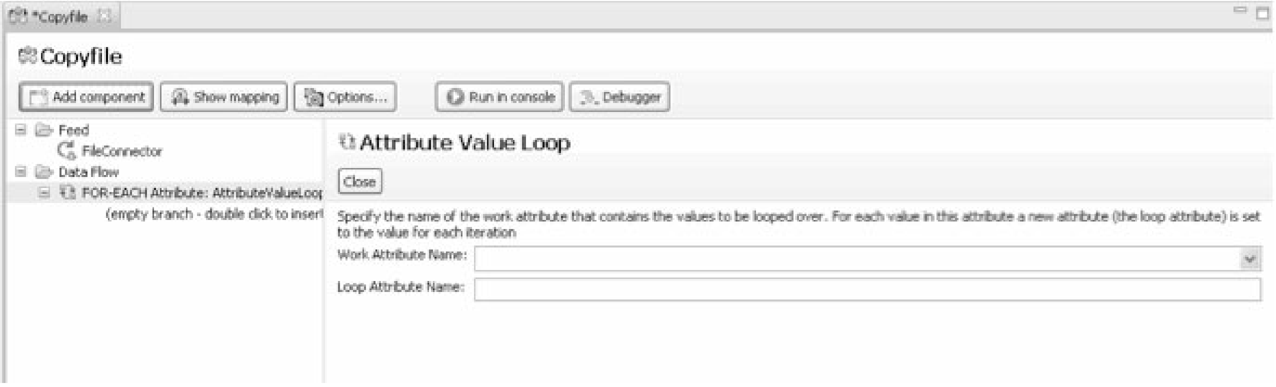
Work Entry Operations
Uses the operation value from the work entry (for example, `work.getOperation()` method).

User Defined
Here you can specify your own value.

Use the **Add Case Components...** dialog to generate the Case branches inside this Switch branch. Based on your selection it will automatically suggest those values that makes sense for the selection. If you choose work entry operations it will suggest all known operation values (for example, add, modify).

Use the **Add Default Case** button to add a default case. The default case will execute in the event that none of the other case branches matched the value from the Switch component.

For-Each Attribute Value



Specify the name of the work attribute that contains the values to be looped over. For each value in this attribute a new attribute (the loop attribute) is set to the value for each iteration

Work Attribute Name:

Loop Attribute Name:

Figure 25. Attribute Value loop

This component loops over values in an attribute. Specify the work entry attribute name to loop over (work attribute name) and the loop attribute name. The loop attribute name is set to the current loop value from the work entry attribute (for example, `foreach f in work.attr.values; set loopattr = f`).

Conditional Loop

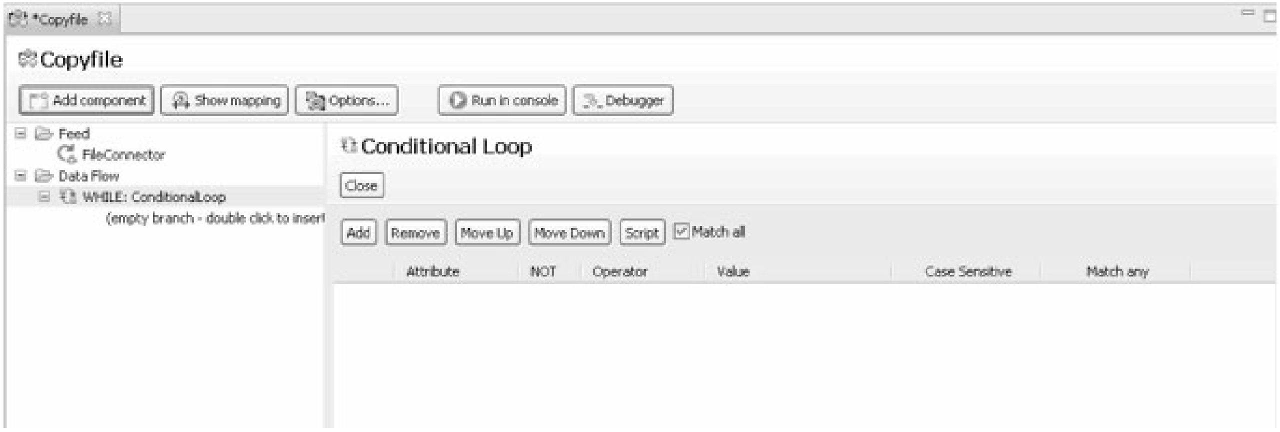


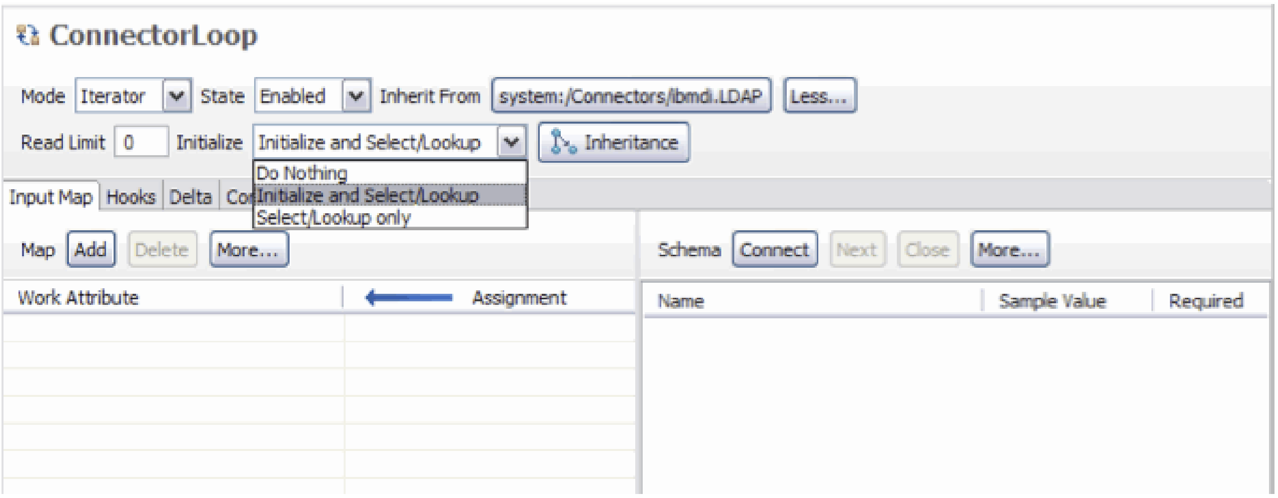
Figure 26. Conditional loop

The conditional loop lets you specify simple expressions and a custom script that return true/false.

Use the **Add**button to add new rows of Attribute/Operator/Value controls. You can remove these rows using the Delete button. The value can be a constant or an expression. Use the expression editor to configure the expression by clicking the button following the value text field.

The **Match All** checkbox determines whether all lines must match (**Match all** checked) or if only one needs to be true for the branch to execute.

Connector Loop


ConnectorLoop

Mode Iterator

State Enabled

Inherit From system:/Connectors/ibmdi.LDAP

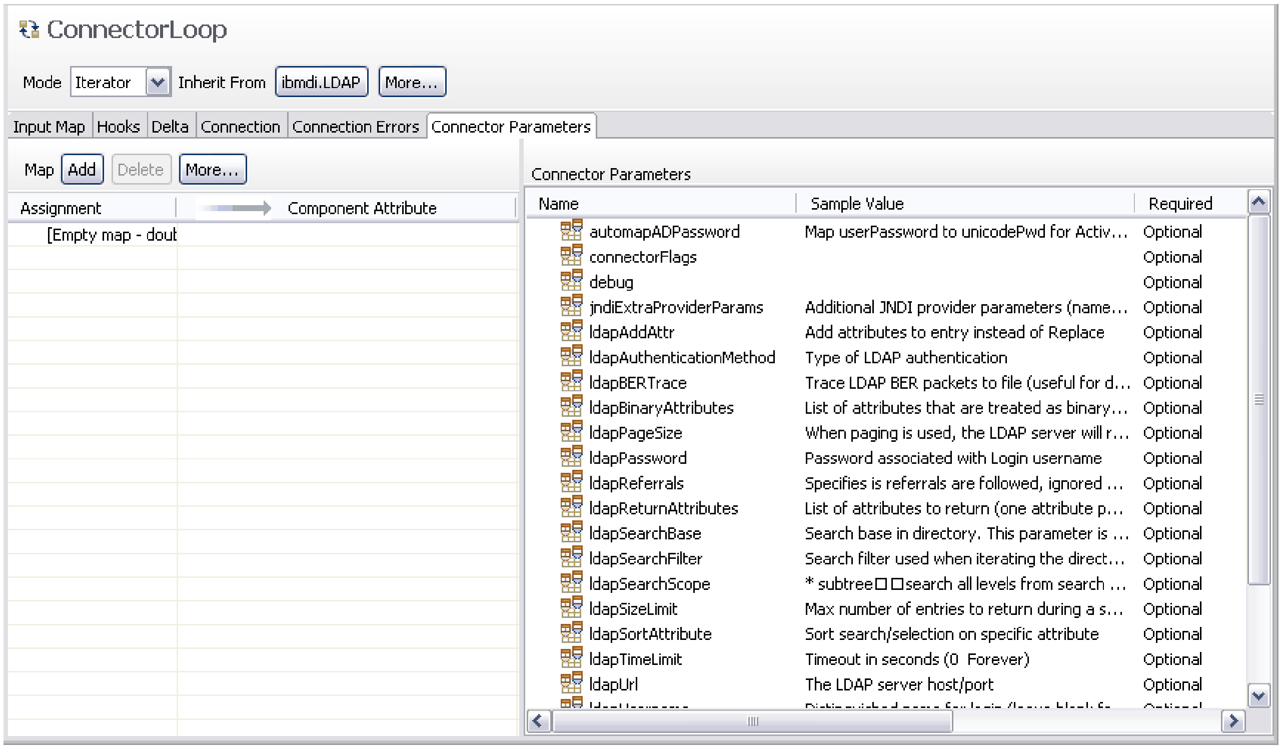
Read Limit 0

Initialize Initialize and Select/Lookup

Figure 27. Connector loop

The Connector loop uses the Connector editor to configure a connector. There are a few differences though as shown in the above picture. The initialize option now shows those options relevant for a connector loop instead of the normal initialize options. Also, only Iterator and Lookup can be selected from the Mode dropdown.

Apart from the usual tabs you see for a connector, there is also an output attribute map that lets you configure dynamic assignments of connector parameters in the **Connector Parameters** tab. This is slightly different from the ordinary output map in that it shows a fixed schema, which is the list of parameters for the chosen connector. The schema part also has no Connect/Next buttons to affect the schema.



ConnectorLoop

Mode Iterator | Inherit From ibmdi.LDAP | More...

Input Map | Hooks | Delta | Connection | Connection Errors | Connector Parameters

Map | Add | Delete | More...

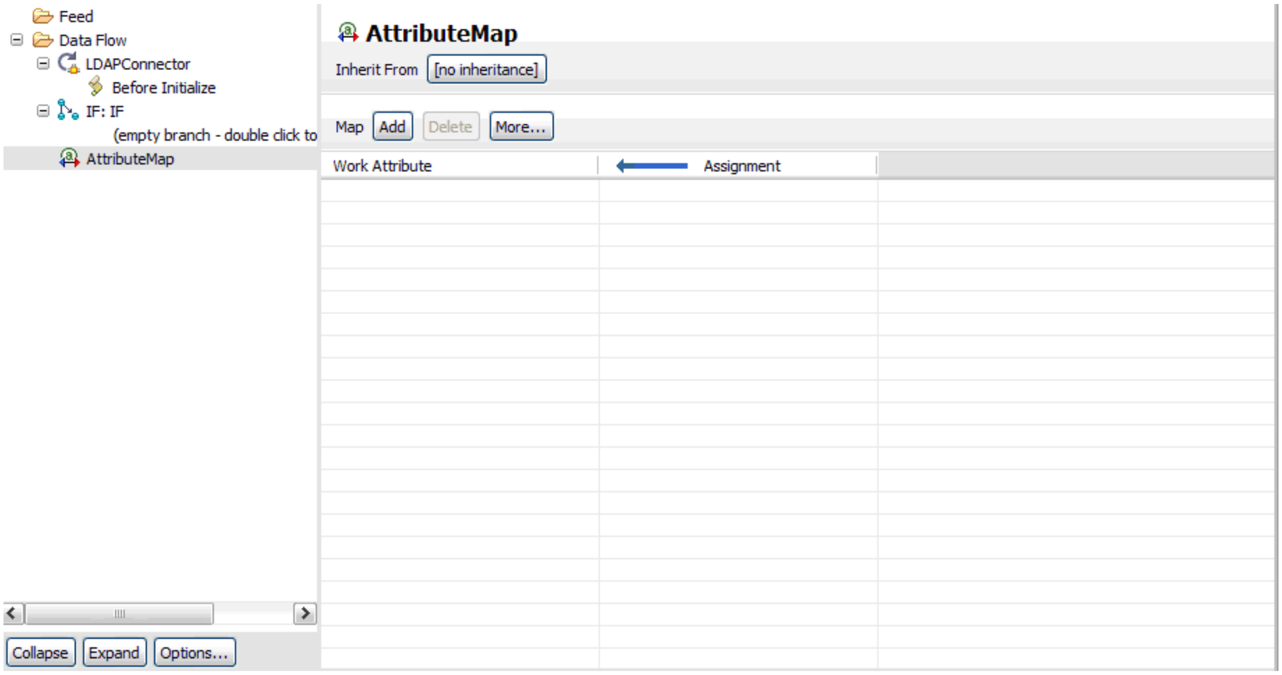
Assignment	→ Component Attribute
[Empty map - dout	

Connector Parameters

Name	Sample Value	Required
automapADPassword	Map userPassword to unicodePwd for Activ...	Optional
connectorFlags		Optional
debug		Optional
jndiExtraProviderParams	Additional JNDI provider parameters (name...	Optional
ldapAddAttr	Add attributes to entry instead of Replace	Optional
ldapAuthenticationMethod	Type of LDAP authentication	Optional
ldapBERTrace	Trace LDAP BER packets to file (useful for d...	Optional
ldapBinaryAttributes	List of attributes that are treated as binary...	Optional
ldapPageSize	When paging is used, the LDAP server will r...	Optional
ldapPassword	Password associated with Login username	Optional
ldapReferrals	Specifies is referrals are followed, ignored ...	Optional
ldapReturnAttributes	List of attributes to return (one attribute p...	Optional
ldapSearchBase	Search base in directory. This parameter is ...	Optional
ldapSearchFilter	Search filter used when iterating the direct...	Optional
ldapSearchScope	* subtree□ □search all levels from search ...	Optional
ldapSizeLimit	Max number of entries to return during a s...	Optional
ldapSortAttribute	Sort search/selection on specific attribute	Optional
ldapTimeLimit	Timeout in seconds (0 Forever)	Optional
ldapUrl	The LDAP server host/port	Optional

Figure 28. Connector Parameters in Connector Loop

Attribute Map



Feed
Data Flow
LDAPConnector
Before Initialize
IF: IF
(empty branch - double click to
AttributeMap

Collapse | Expand | Options...

AttributeMap

Inherit From [no inheritance]

Map | Add | Delete | More...

Work Attribute	← Assignment

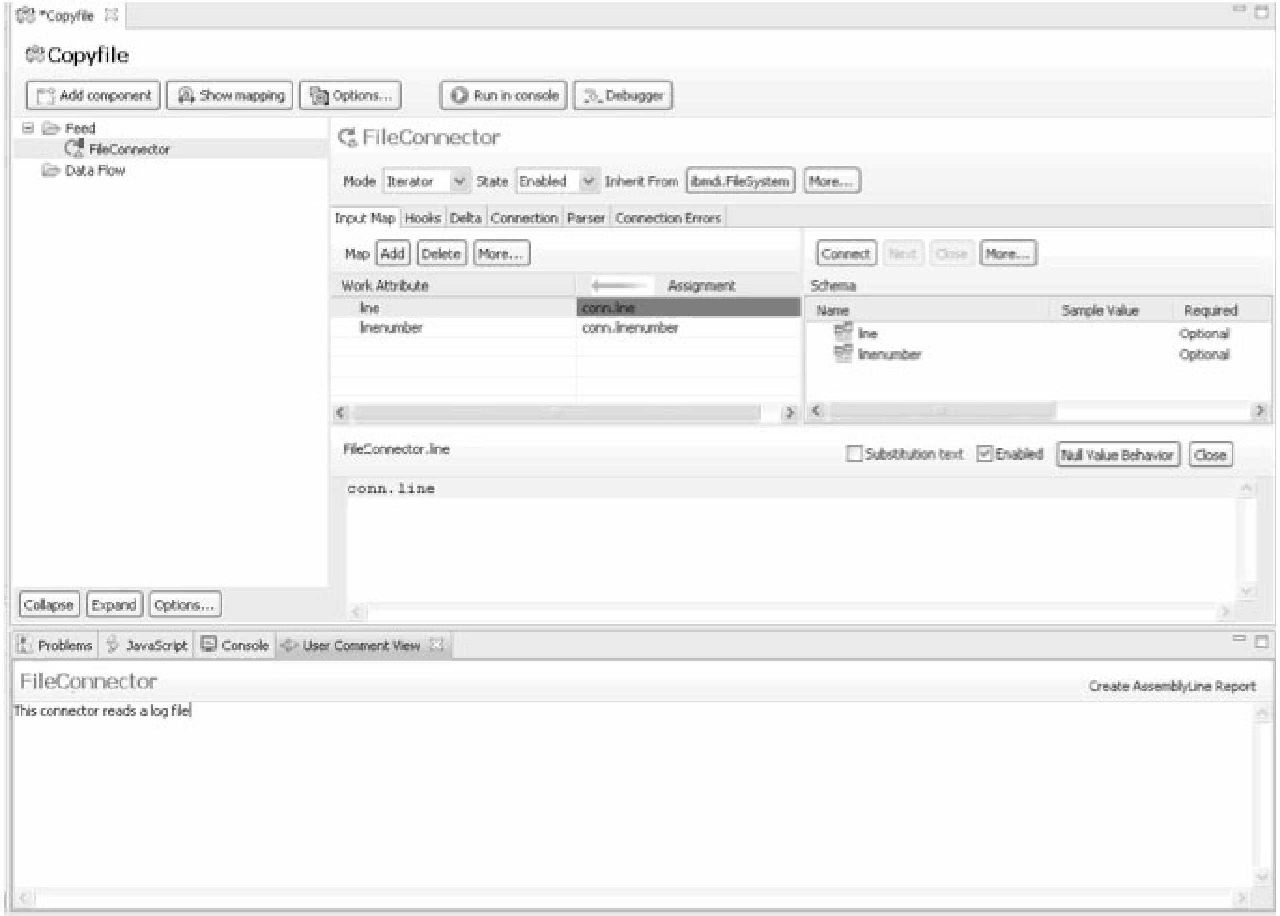
Figure 29. Independent Attribute Map Component

The attribute map component shows a single panel where you can map attributes to the work entry, independent from attribute mapping inside other components like a connector. The component also offers to reuse attribute maps from other components such as connectors, functions and other attribute map components in your library. For more information, see “Attribute Mapping and Schema” on page 104.

User Documentation View

Sometimes an AssemblyLine turn out to be quite complex. To aid another in reading the configuration you can document parts of the AssemblyLine using the documentation view.

In the AssemblyLine outline you can right-click a component and choose **Edit user comment** to bring the documentation view to front:



Mode Iterator State Enabled Inherit From ibmdi.FileSystem

Work Attribute	Assignment
line	conn.line
linenumber	conn.linenumber

Name	Sample Value	Required
line		Optional
linenumber		Optional

FileConnector.line ☐ Substitution text ☑ Enabled

```
conn.line
```

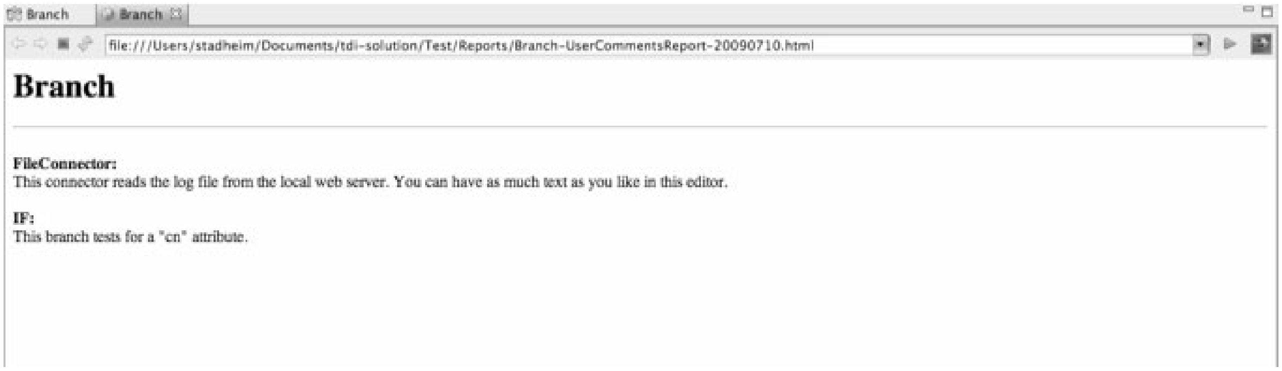
FileConnector

This connector reads a log file

Figure 30. User Documentation View

As the selection changes in the AssemblyLine editor the documentation view reflects the current selection. The text you type in the view is saved when you save the AssemblyLine. Components with a comment is decorated in the upper left corner of the component icon.

The **Create AssemblyLine Report** button will create a report with all user comments entered in the AssemblyLine. The report template used is called `UserCommentsReport.xsl` in the *TDI_Install_dir*/`XSLT/ConfigReports` directory.



Branch
Branch
file:///Users/stadheim/Documents/tdi-solution/Test/Reports/Branch-UserCommentsReport-20090710.html

Branch

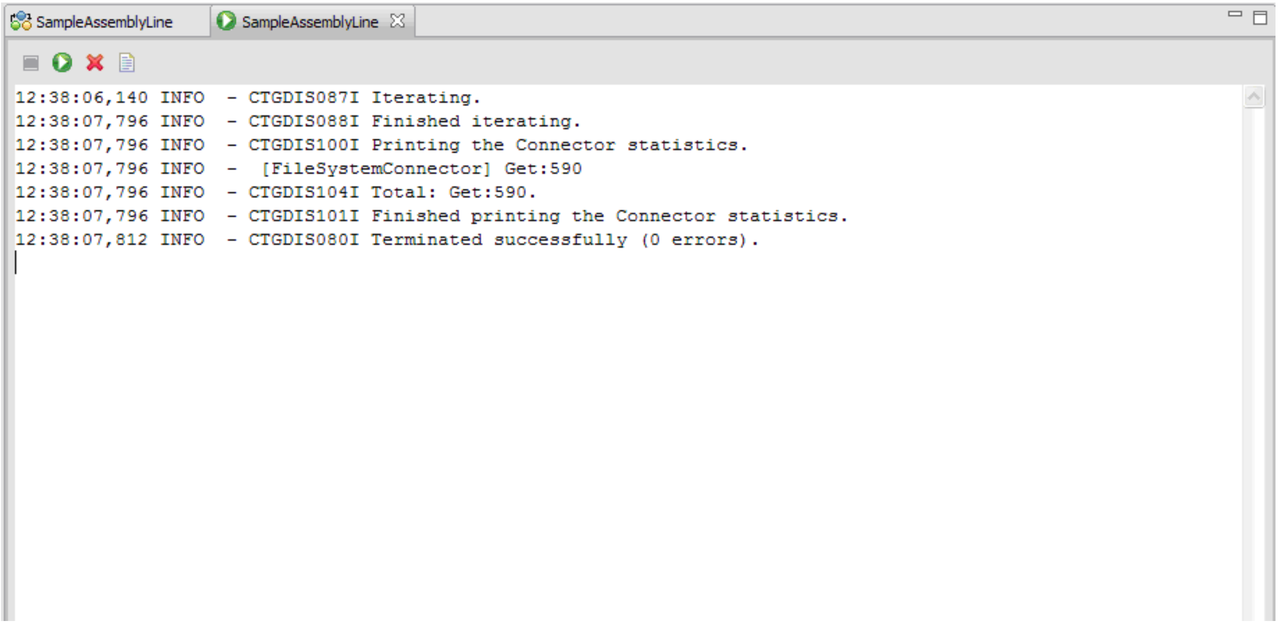
FileConnector:
This connector reads the log file from the local web server. You can have as much text as you like in this editor.

IF:
This branch tests for a "cn" attribute.

Figure 31. Sample AssemblyLine report

Run AssemblyLine window

When you run the AssemblyLine you will get a window with the log output shown.



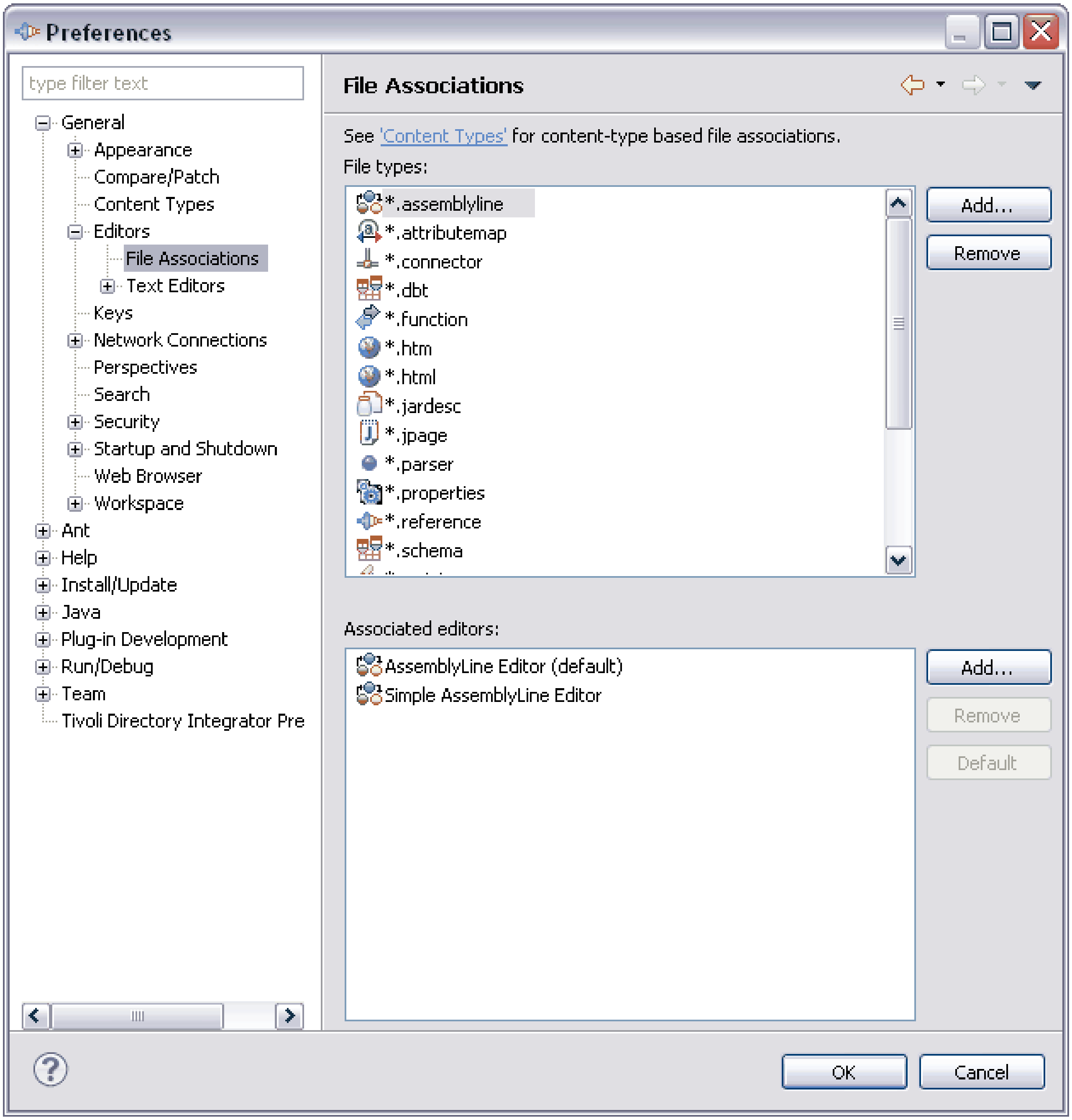
```
12:38:06,140 INFO  - CTGDIS087I Iterating.
12:38:07,796 INFO  - CTGDIS088I Finished iterating.
12:38:07,796 INFO  - CTGDIS100I Printing the Connector statistics.
12:38:07,796 INFO  -  [FileSystemConnector] Get:590
12:38:07,796 INFO  - CTGDIS104I Total: Get:590.
12:38:07,796 INFO  - CTGDIS101I Finished printing the Connector statistics.
12:38:07,812 INFO  - CTGDIS080I Terminated successfully (0 errors).
```

Figure 32. Console log

In this screen you can also stop a running AssemblyLine and restart it after it has terminated.

Note: Stopping an AssemblyLine means that the Config Editor sends a stop notification to the Config Instance (usually the default local Server) that is running the AL; it does not immediately kill the thread, but stops execution as soon as the server regains control. This differs from previous versions where pressing the **Stop** button would cause the entire server process that was running the AssemblyLine to be killed.
The two other buttons are to clear the log window and open the log file in a separate editor window. The log window only shows the last few hundred lines to avoid out-of-memory problems.

The log is written to a temporary file with a prefix of "tdi_ce_al_log" and an extension ".log". The file is placed in the platform specific temporary directory, which is often defined by the TEMP/TMP environment variable. The log file is automatically deleted when you close the Run AssemblyLine window, but in case the application or machine crashes you may have to manually remove these log files. The editor to use for this file defaults to the simple text editor, but can be changed by mapping the ".log" extension to a different editor (including external editors). Use the **Windows** > **Preferences** menu option to open the following dialog:



File Associations

See 'Content Types' for content-type based file associations.

File types:

- *.assemblyline
- *.attributemap
- *.connector
- *.dbt
- *.function
- *.htm
- *.html
- *.jardesc
- *.jpage
- *.parser
- *.properties
- *.reference
- *.schema

Associated editors:

- AssemblyLine Editor (default)
- Simple AssemblyLine Editor

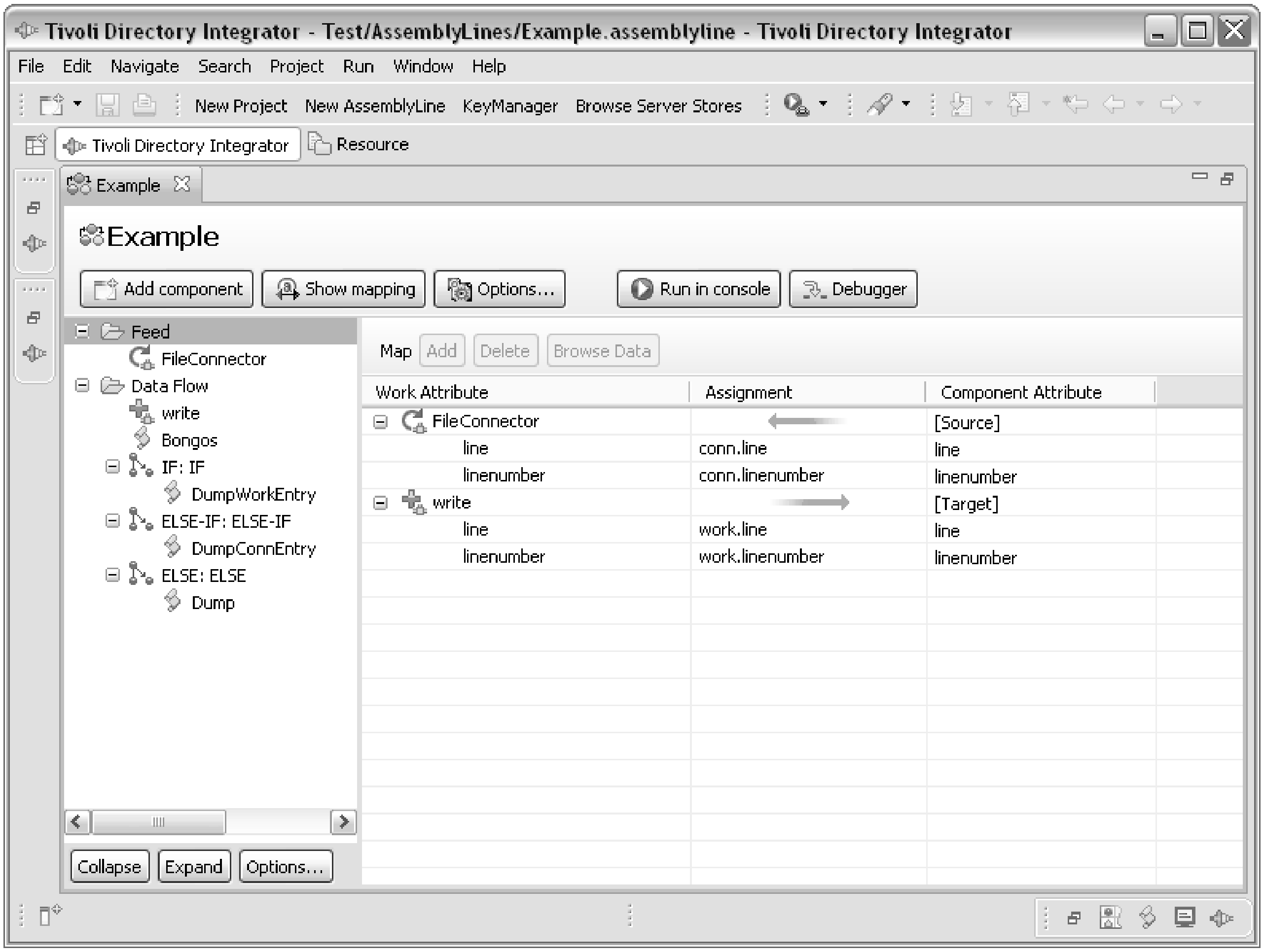
Figure 33. Configuration Editor File associations preferences

Here you can add the “.log” extension and associate it with an editor.

Attribute Mapping and Schema

Attribute mapping is done using either the attribute map panel in the AssemblyLine or in the component editor.

In the AssemblyLine editor you can add attributes either by right clicking in the attribute maps section and choosing add attribute, or use the **Add** button in the toolbar as shown below.


Tivoli Directory Integrator - Test/AssemblyLines/Example.assemblyline - Tivoli Directory Integrator
File Edit Navigate Search Project Run Window Help
New Project New AssemblyLine KeyManager Browse Server Stores
Tivoli Directory Integrator
Resource
Example
Example
Add component
Show mapping
Options...
Run in console
Debugger
Feed
FileConnector
Data Flow
write
Bongos
IF: IF
DumpWorkEntry
ELSE-IF: ELSE-IF
DumpConnEntry
ELSE: ELSE
Dump
Collapse
Expand
Options...
Map
Add
Delete
Browse Data

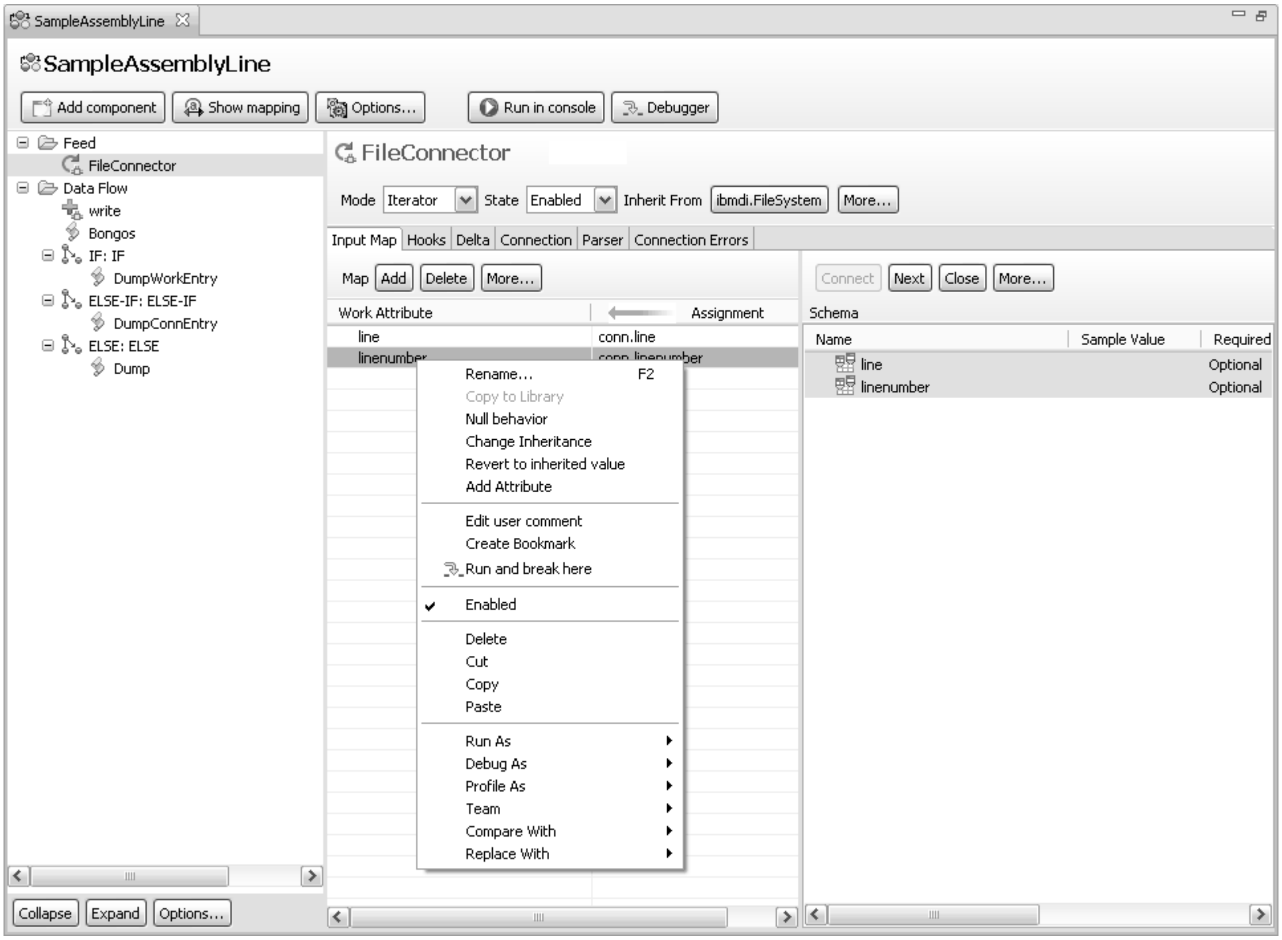
Work Attribute	Assignment	Component Attribute
FileConnector	←	[Source]
line	conn.line	line
linenumber	conn.linenumber	linenumber
write	→	[Target]
line	work.line	line
linenumber	work.linenumber	linenumber

Figure 34. Attribute Mapping

In this window you don't see the schema for the components in the AssemblyLine. To work with the schema you open the editor for the component by selecting it in the left tree.

The typical scenario for attribute mapping is to first discover the schema for the component. When you do a discover schema, the CE will run a background job that executes the query schema method of the component. If no schema is returned the CE will ask if you would like to read an entry to attempt to derive the schema from that. The result is then populated back into the schema for the component you are editing.

The picture below shows the contents of the input schema for a component after discovering attributes. If a component for some reason doesn't provide you with a schema you can add schema items manually using the **Add...** button on the toolbar or reuse a schema from another component configuration with the **Change Inheritance** option.



Mode Iterator State Enabled Inherit From ibmdi.FileSystem

Work Attribute	Assignment
line	conn.line
linenumber	conn.linenumber

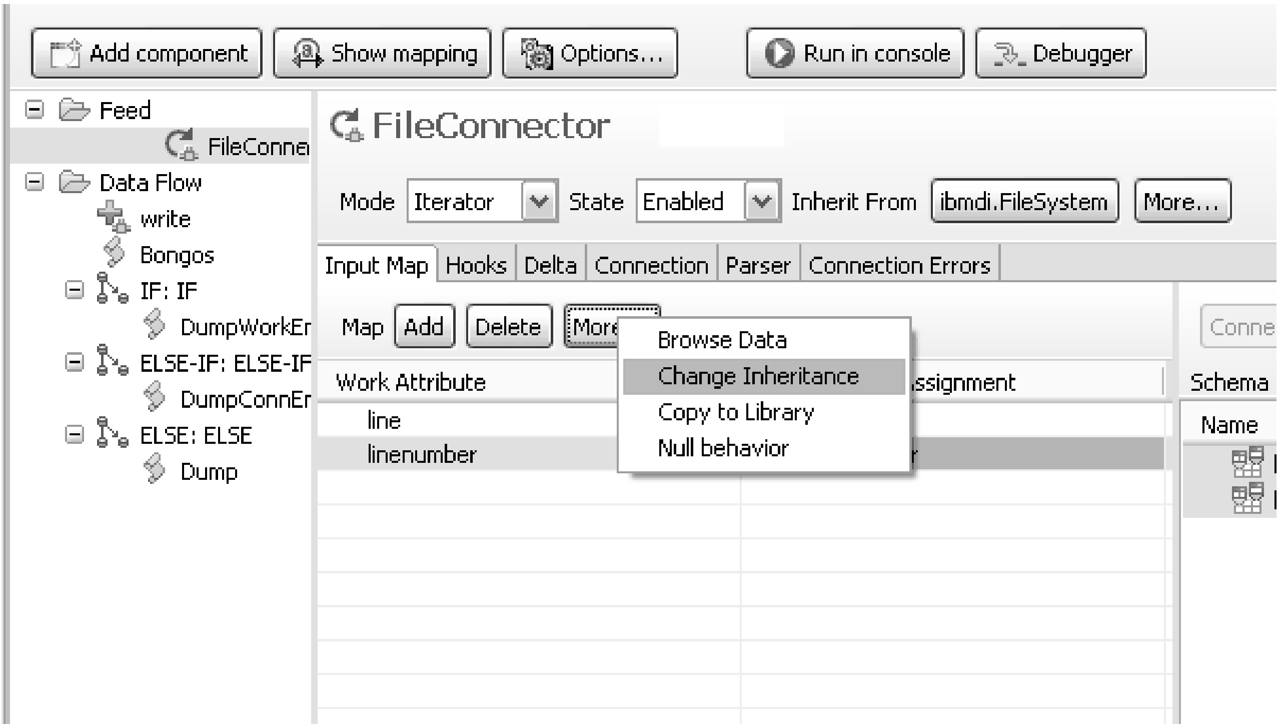
Schema

Name	Sample Value	Required
line		Optional
linenumber		Optional

Figure 35. Attribute Mapping, with discovered Attributes

You can also use the drop down menu on the title bar to change the inheritance for the schema configuration.

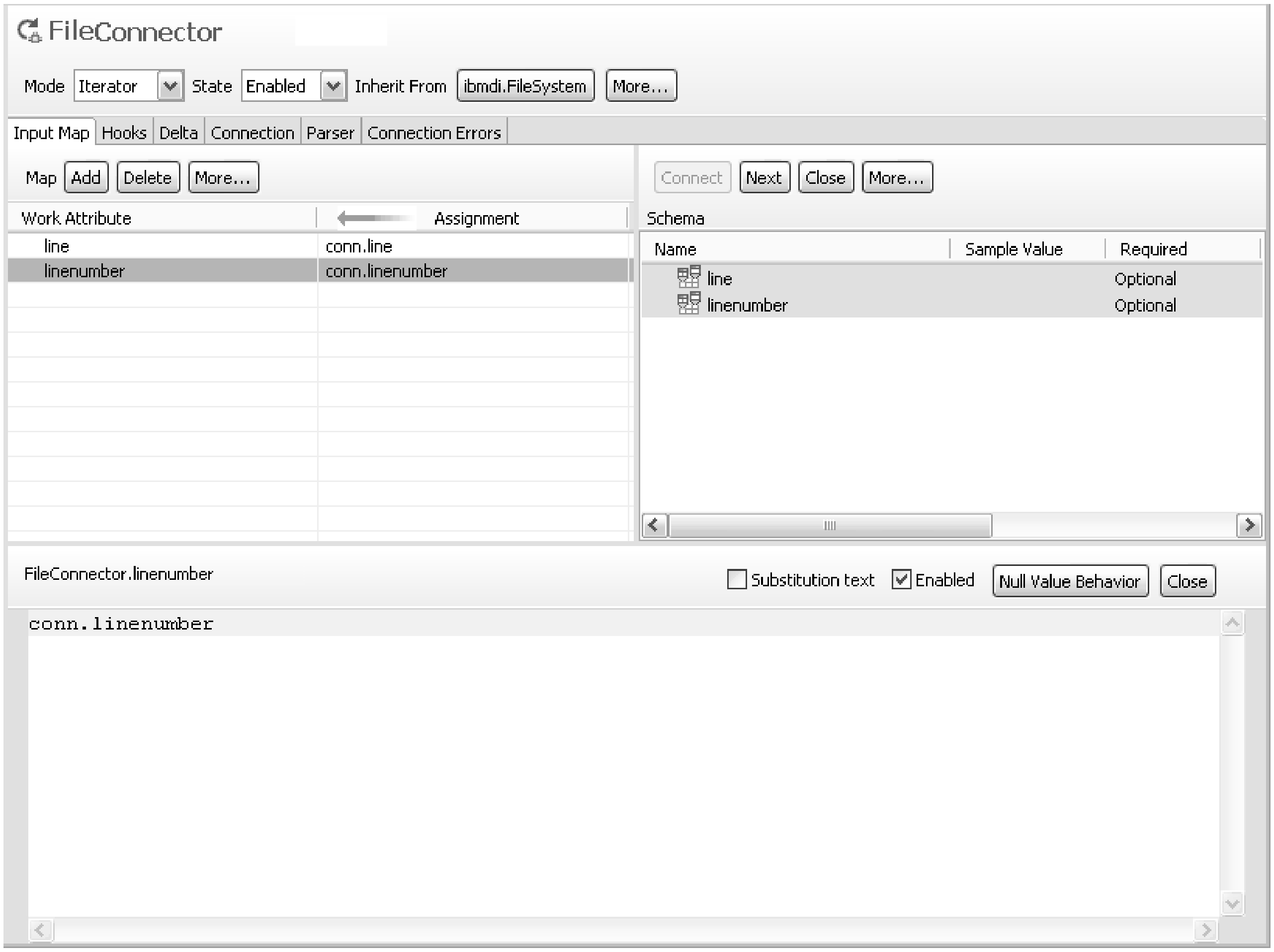
Having a schema, you can drag and drop individual items into to the attribute map or use the **Map Attribute** function from the context menu and modify the mapping if necessary.



Mode Iterator State Enabled Inherit From ibmdi.FileSystem More...

Figure 36. Changing Attribute Map inheritance

Note: Drag and drop functionality depends to a certain extent on your windowing environment. In particular, on UNIX systems, the Common Desktop Environment (CDE) does not provide this, so in order to set up mapping you will need to use the **Map Attribute** function from the context menu.



Mode: Iterator
State: Enabled
Inherit From: ibmdi.FileSystem

Work Attribute	Assignment
line	conn.line
linenumber	conn.linenumber

Schema

Name	Sample Value	Required
line		Optional
linenumber		Optional

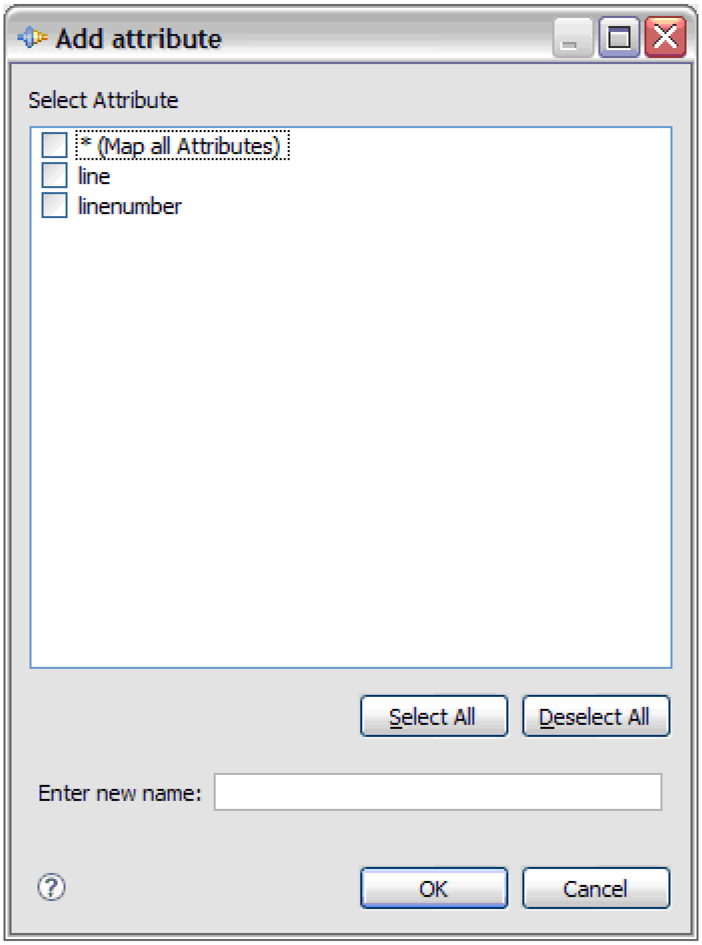
FileConnector.linenumber

☐ Substitution text ☑ Enabled

```
conn.linenumber
```

Figure 37. Attribute Mapping, with JavaScript editing window for individual Attribute

If you have no schema or want to add attributes independent of the schema you can of course do so. Use the **Add** button to add a new attribute to the map. You name the attribute, and an expression of either "conn.*attribute-name*" or "work.*attribute-name*" is assigned to the new attribute. This can be done in both the AssemblyLine editor and in the Connector editor windows.



Select Attribute

□ * (Map all Attributes)
□ line
□ linenumber

Enter new name:

Figure 38. Add Attribute dialog

A dialog appears with an editable text field where you can type the name of the new attribute. The list above contains all known attribute names from the schema; you can select those you want added to the attribute map.

As you add more components to the AssemblyLine you can drag attributes between them where it makes sense. Dragging a component onto another component will map all mapped attributes to the target component. You can also drag attributes from the attribute map onto components in the left panel showing all components in the AssemblyLine. This will perform a simple map of all those items you drag over. This is similar to dropping them onto the component in the attribute map panel.

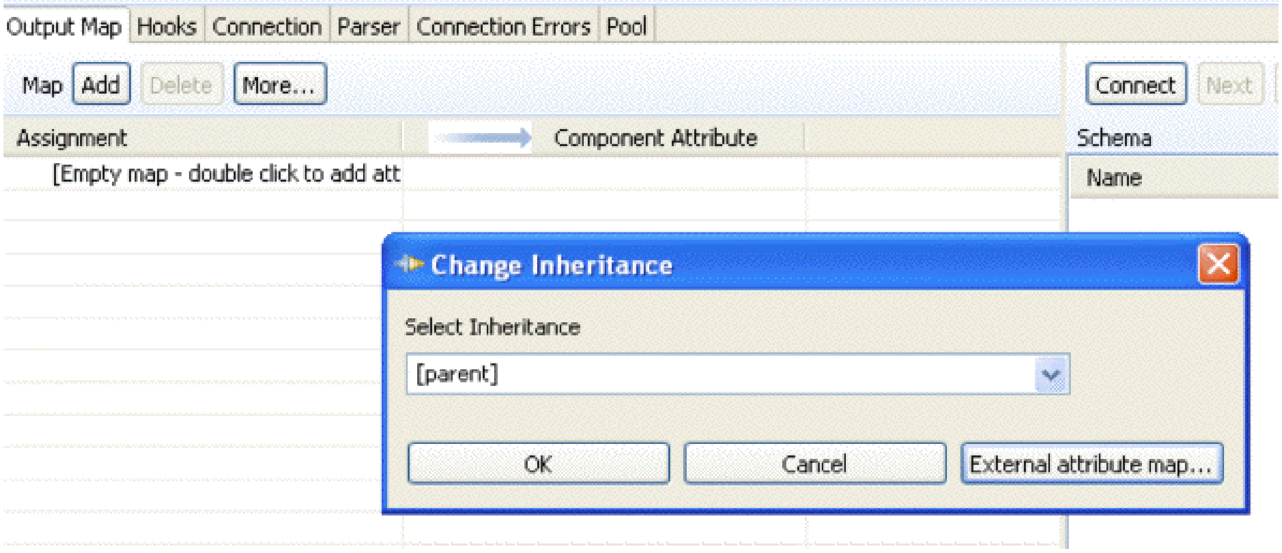
The concept of Attribute Mapping is treated fairly extensively, replete with examples, in *IBM Tivoli Directory Integrator V7.1.1 Getting Started*.

Depending on the Connector, and the mode it is configured in, there will be different tabs in the Connector configuration window.

- Connectors in a mode which supports input from a connected system, will have a section called **Input Attributes**.
- Connectors in a mode which supports output to a connected system, will have a section called **Output Attributes**.
- Some Connectors support modes that can do both Input and Output. If configured that way, you will see an **Input Attributes** section as well as an **Output Attributes** section.

External attribute maps

Attribute maps can inherit from external attribute map files. An external attribute map file is a text file that contains attribute map items just like you have it in the actual mapping screen. The difference is that the external file uses a different format than the internal XML structure. This makes it easier for you to configure the attribute map for any connector without even going into the CE. The CE provides this option in the inheritance dialog for attribute maps:



Select Inheritance

[parent]

Figure 39. Attribute map: inheritance dialog

Click the **External attribute map...** button to choose an existing file, or type "`file:`" followed by the full path to the attribute map file. If you want to use relative path names, prefix the filename with a dot+slash (./).

Input Attribute mapping:

Input Attribute mapping is the process that accomplishes the moving of data from the input source to the Work entry in the AssemblyLine. Input Attribute maps are shown in the Attribute Maps window of the Connector, when brought up in the Connector Editor, with an arrow pointing to the Connector from an entity referred to as "[Source]". They are also shown in the Schema window, under Input Attribute map.

Before you begin

In order to be able to set up the Input Attribute map, the Connector must be set to a mode which supports input, in the toolbar of the Connector. Modes that support input are typically Iterator, Lookup and Server.

Then, in the **Input Map** section, you select those Attributes from the input source that you wish to process in the AssemblyLine.

About this task

Connectors to be set up for Input Attribute mapping can either reside in the `<workspace>/Resources/Connectors`, or in their designated position in the AssemblyLine.

Procedure

1. Click **Input Map**.

2. Click **Connect**, followed by **Next** to get the schema for many datasources. Some Connectors or Connector-Parser combinations have pre-defined schemas, whereas others prompt you to read a sample entry from the data source and examine it to discover attributes.
3. Finally, select Attributes from the **Schema** list and then drag them into the Attribute Map, or add these manually with the Add and Delete buttons. The Attribute Map controls which Attributes are brought into your AL for processing, as well as any transformations you specify.

What to do next

These mapped Attributes are retrieved from the data source, placed in the *Work* entry, and passed to subsequent Connectors in the Flow section in the AssemblyLine.

If you did not create the Connector directly in an AssemblyLine, then in order to use this Connector in an AssemblyLine, drag the Connector from its location in `<workspace>/Resources/Connectors` to the **Feed** section of an AssemblyLine.

Output Attribute mapping:

Output Attribute mapping is the process that accomplishes the moving of data from the Work entry in the AssemblyLine to the output destination in the connected system. Output Attribute maps are shown in the Attribute Maps window of the Connector, when brought up in the Connector Editor, with an arrow pointing from the Connector to an entity referred to as "[Target]". They are also shown in the Schema window, under Output Attribute map.

Before you begin

In order to be able to set up the Output Attribute map, the Connector must be set to a mode which supports output, toolbar of the Connector. The typical mode for output is AddOnly. Some modes, like CallReply, support both input and output.

Then, in the **Output Map** in the Output Attributes section, you select those Attributes from the Work entry in the AssemblyLine that you wish to output to the connected system.

About this task

Connectors to be set up for Output Attribute mapping can either reside in the `<workspace>/Resources/Connectors`, or in their designated position in the AssemblyLine. However, when the Connector is in `<workspace>/Resources/Connectors` only, that is, not a member of an AssemblyLine, you cannot easily drag Work entry attributes into the Output Attribute map. In this case, either drag the Connector into an AssemblyLine, or create the mappings manually, by clicking **Add** in the Attribute Maps window, alternatively by right-clicking on the Connector in the Attribute Maps window, and select **Add attribute map item**.

Procedure

1. Click **Output Map**.
2. Click **Connect**, to get the schema for the datasource. Some Connectors or Connector-Parser combinations have pre-defined schemas, which will be displayed. Many Connectors, however, do not.
3. If your Connector is in an AssemblyLine, drag Work entry attributes mapped in previously onto the Connector in the Attribute Maps window of the AssemblyLine editor. Alternatively, create Attributes manually—name matching occurs at run time. For example, an Output Map Attribute map item created as `some_attribute` causes a Work entry attribute named some_attribute to be mapped to a connected system-attribute of the same name.

What to do next

These mapped Attributes are retrieved from the *Work* entry when this Connector is called in the Flow of the AssemblyLine, and are output to the connected system.

If you did not create the Connector directly in an AssemblyLine, then in order to use this Connector in an AssemblyLine, drag the Connector from its location in `<workspace>/Resources/Connectors` to the **Flow** section of an AssemblyLine.

The Connector Editor

The connector editor is used when you edit connector files or use the **Edit** function on a Connector inside the AssemblyLine.

Creating a Connector is outlined in the section “Creating a Connector.”

The editor uses the same widgets you find in wizards and popup dialogs for connectors. The editor consists of six tabs in which configuration panels for various aspects of the connector are shown. At the top are the main attributes of the connector such as its mode, state and other general connector options.

The tabs are:

1. “Input and Output Attribute Maps” on page 113
2. “Hooks” on page 113
3. “Connection” on page 115
4. “Parser” on page 115
5. “Link Criteria” on page 116
6. “Connection Errors” on page 119
7. “Delta” on page 121
8. “Pool” on page 122
9. “Connector Inheritance” on page 123

Creating a Connector

Creating a Connector involves deciding where the Connector is to reside, and which initial parameters to assign to it.

Before you begin

Decide upon where the Connector should reside; this is in either of two places:

1. Connectors meant for re-use and resource sharing reside under the `<workspace>/Resources/Connectors` directory. This is generally the best place to create and maintain your Connectors. Connectors defined this way are added to AssemblyLines by dragging them into the appropriate place.

   After this, you can alter those few settings of the Connector such that it plays its designated role in the AssemblyLine, but leaving the majority of the settings inherited, and therefore unchanged from their definitions in the Resources section.
2. You can also create a Connector directly in an AssemblyLine; Connectors defined this way are ad-hoc definitions that are only valid in the context of that particular AssemblyLine.

About this task

Connectors form the backbone of any solution created with IBM Tivoli Directory Integrator, they establish the connection to the systems you wish to exchange data with.

Procedure

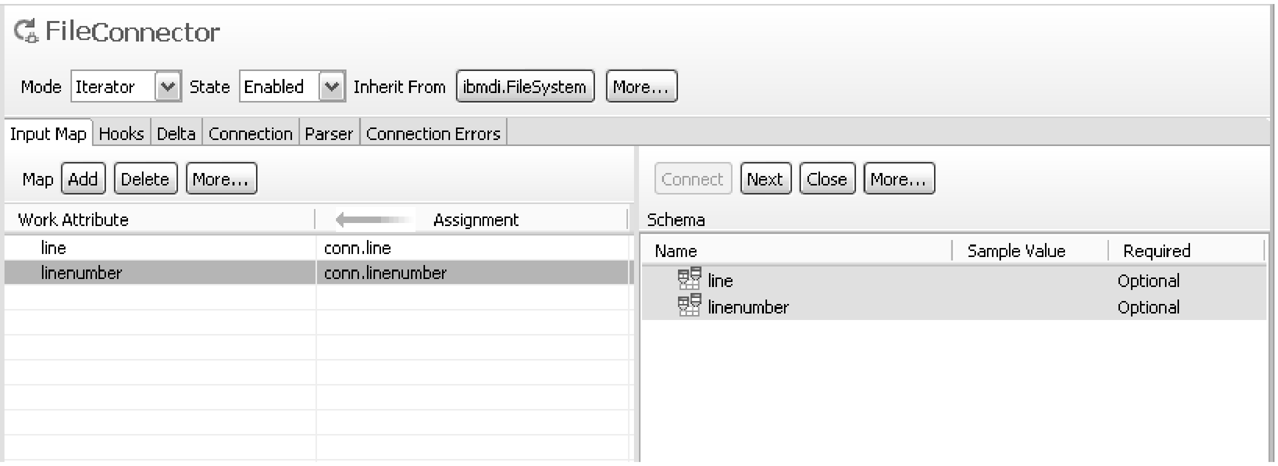
1. Right-click and select **Resources** > **Connectors** > **New Connector...** in your workspace, or select **File** > **New** > **Connector**
2. Navigate to the location where you want your new Connector, and name your new Connector.
   a. The recommended location is `<workspace>/Resources/Connectors`.
   b. Alternatively, you can create the new Connector directly in your AssemblyLine. Navigate to the location in the target AssemblyLine; either the Feed section for Iterator and Server mode Connectors, or Flow for all other modes.
3. Click **Finish** to create the Connector.
4. In the **Connection** tab, set the mode of your new Connector to your desired mode.
5. Set the connection parameters for this Connector in the **Connection** tab; required parameters are marked with an asterisk (*). Some Connectors require you to configure a Parser as well in the **Parser** tab.
6. Go to the**Attribute Map** window in the Connector configuration window to discover or define the schema for this data source: click **Discover Attributes** to get the schema for the datasource. Some Connectors or Connector-Parser combinations have pre-defined schemas. Those that do not, prompt you to read a sample entry from the data source and examine it to discover attributes.

What to do next

Once the Connector has been defined, you are ready to set up which pieces of information known as *Attributes* are to flow to and from the AssemblyLine. This process is called *Attribute Mapping*, and the point of view is from the AssemblyLine. Hence, the definition of mapping Attributes from the connected system, through the input Connector to the Work entry in the AssemblyLine is done in the *Input Attribute Map*; and the reverse, mapping Attributes from the Work entry, through the output Connector to the connected system is done in the *Output Attribute Map*.

Input and Output Attribute Maps

The Attribute Map tabs show the input and output attribute maps and schemas for a component.



Mode Iterator State Enabled Inherit From ibmdi.FileSystem

Work Attribute	Assignment
line	conn.line
linenumber	conn.linenumber

Name	Sample Value	Required
line		Optional
linenumber		Optional

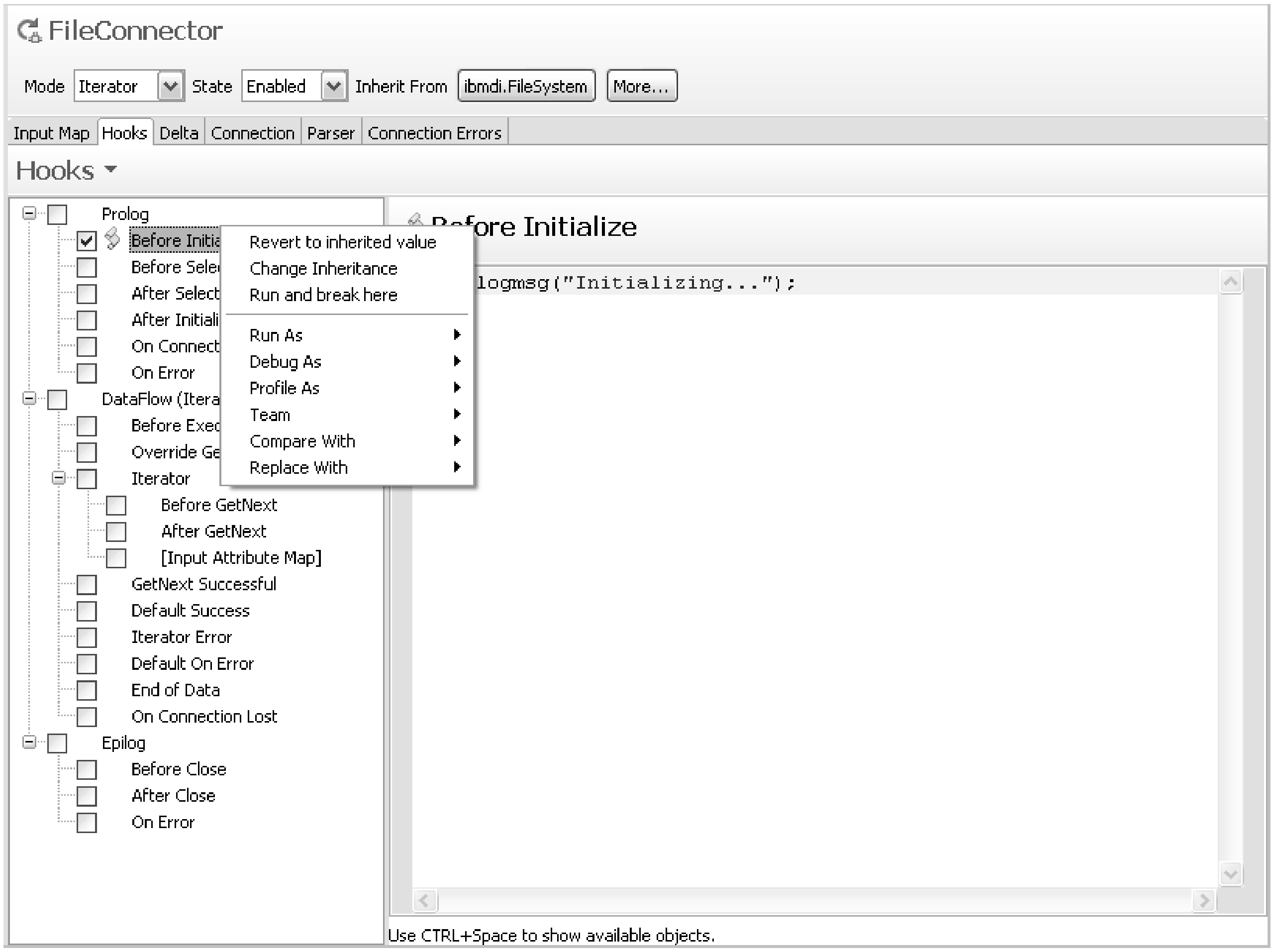
Figure 40. Attribute Map window

See section "Attribute Mapping and Schema" on page 104 in the AssemblyLine editor for a description of this window.

Hooks

The Hooks tab shows all hooks for the connector.

Use the checkbox to enable/disable a hook and select the hook to edit its contents. When you modify the contents of a hook it is automatically enabled. Hooks that contains script code will have a script icon in the tree view so you can quickly identify if a hook has contents or not. Note that if a hook is enabled, it will be executed when it is reached in the execution flow, whether it contains any script or not.



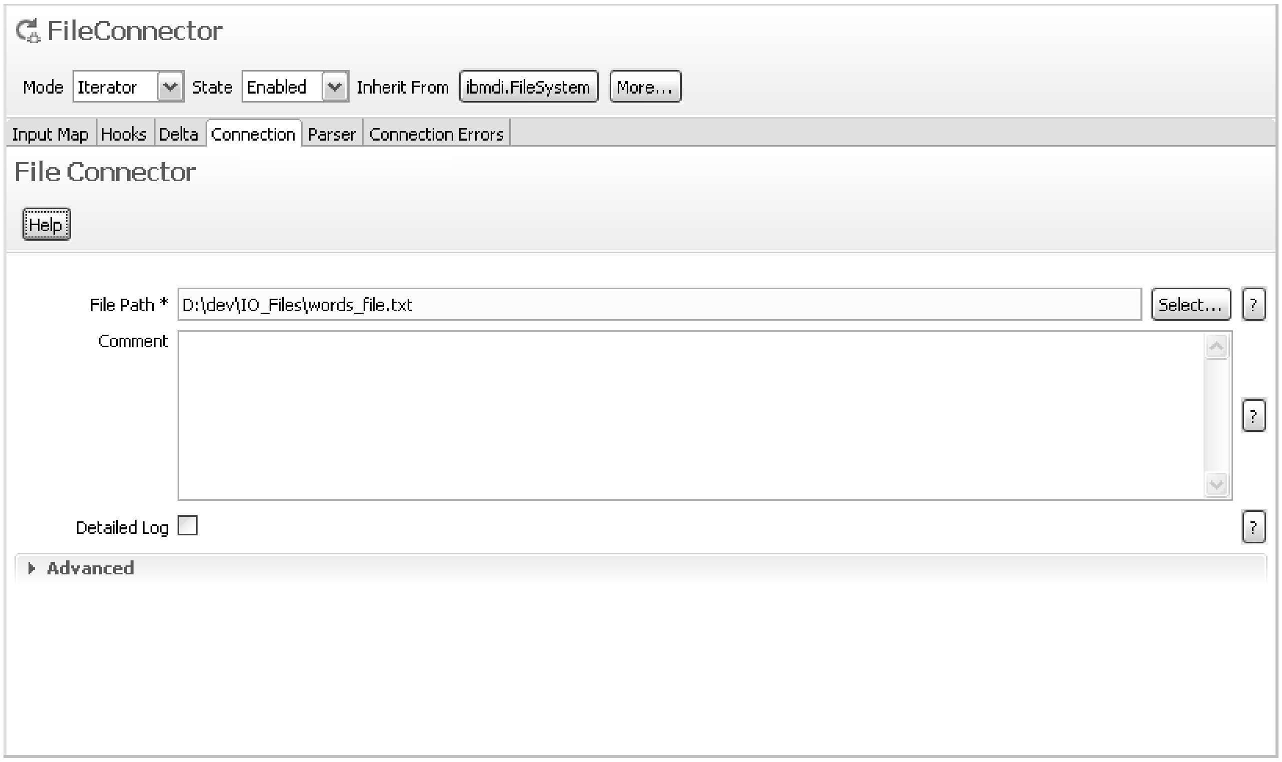
Mode Iterator State Enabled Inherit From ibmdi.FileSystem

```
logmsg("Initializing...");
```

Use CTRL+Space to show available objects.

See the section "AssemblyLine flow and Hooks" on page 26 for a discussion of the various hooks, both at the AssemblyLine level as well as the individual component level.

Connection



Mode: Iterator

State: Enabled

Inherit From: ibmdi.FileSystem

File Path * D:\dev\IO_Files\words_file.txt

Comment

Detailed Log □

Figure 41. Connection tab

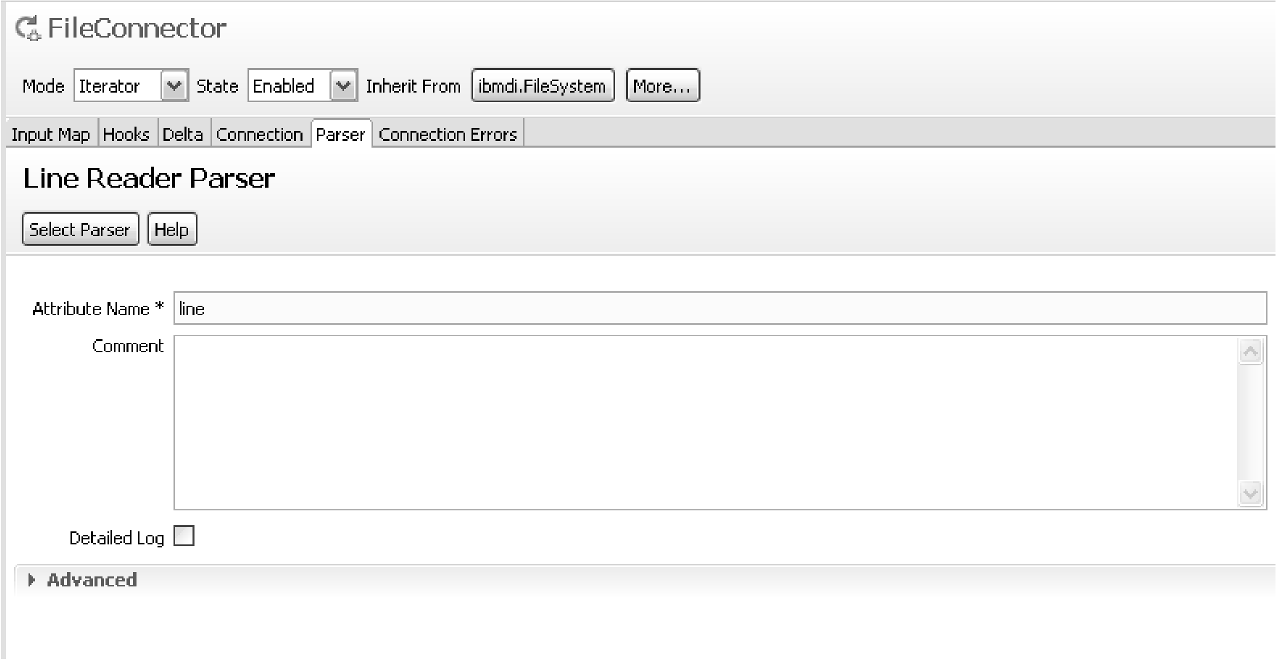
Parameters in the Connection tab are highly specific to the component you want to configure. Refer to the individual specification of the component in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Parser

If a connector can use (or requires) a parser you will also have a tab for this.

Use the **Select Parser** toolbar button to change the parser for the connector.

For example, the Line Reader Parser has a configuration screen like the one shown below:



FileConnector

Mode: Iterator State: Enabled Inherit From: ibmdi.FileSystem

Line Reader Parser

Attribute Name * line

Comment

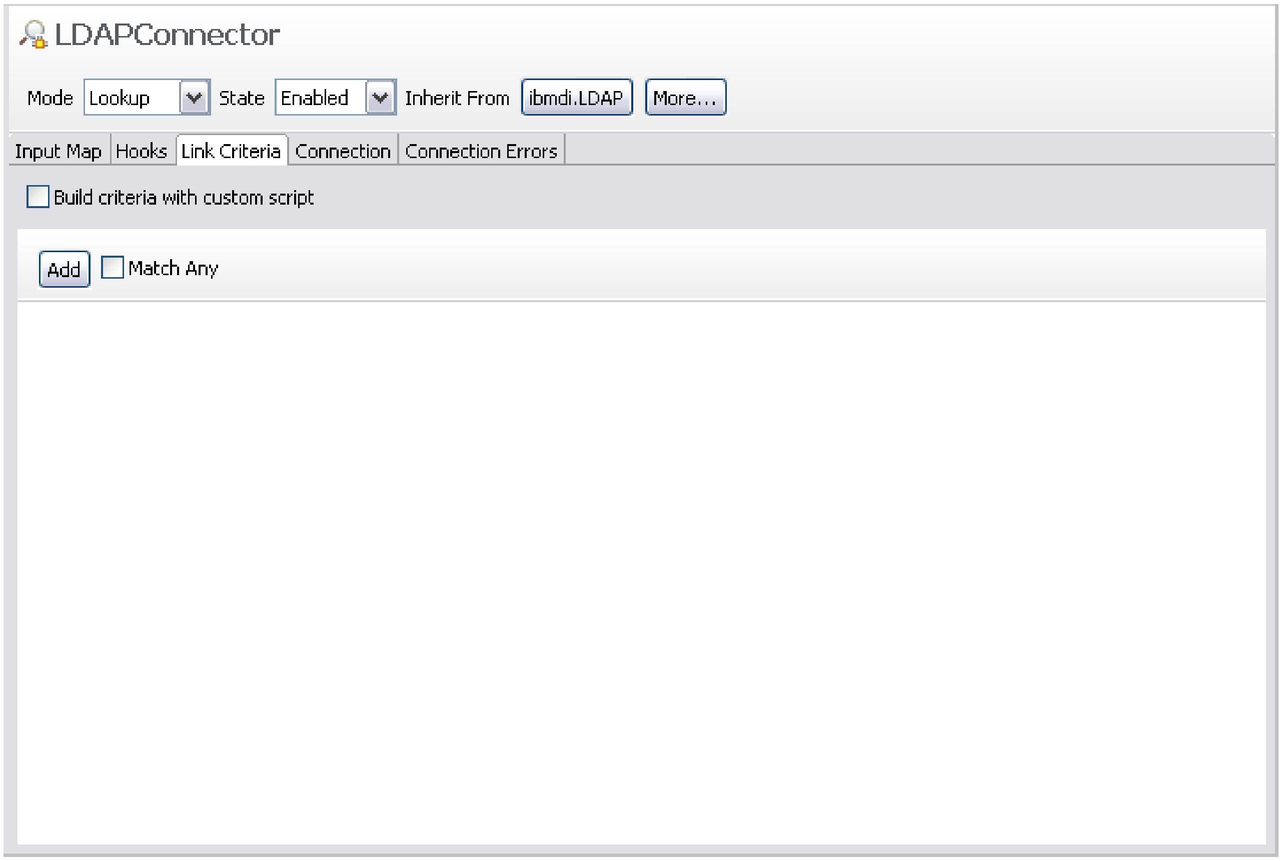
Detailed Log □

Figure 42. Line Reader parser

Link Criteria

When a component requires a link criteria you will see the Link Criteria tab.

Whether a component actually will exhibit a Link Criteria tab depends not only on the component type, but also on its mode. See “Link Criteria” on page 16 for more information.



LDAPConnector

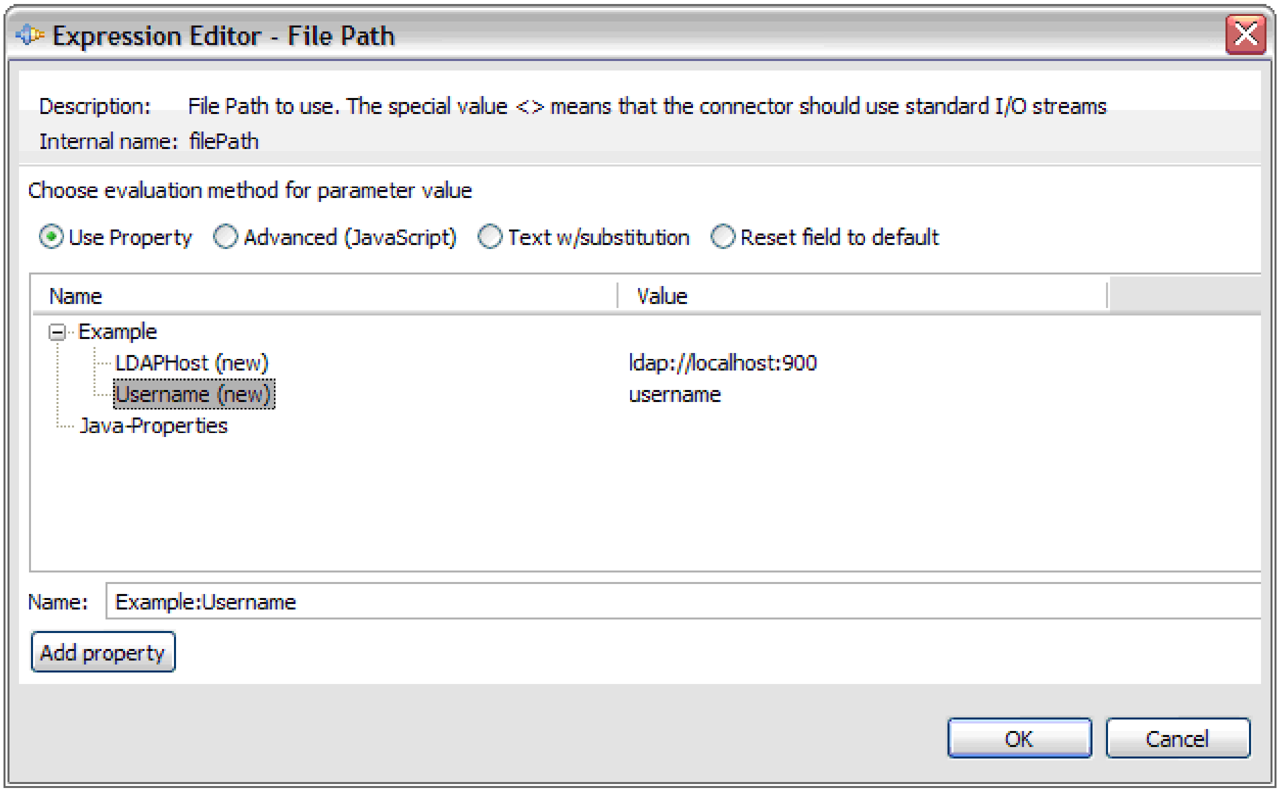
Mode Lookup State Enabled Inherit From ibmdi.LDAP

□ Build criteria with custom script

□ Match Any

Figure 43. Link Criteria tab

Use the **Add** button to add a new row of controls to the view. Use the **Delete** button to remove individual rows. In the view you specify the attribute name, the operand (for example, equals, contains) and the match value. The match value can be a constant or an expression. Use the button to bring up the expression editor:



Expression Editor - File Path

Description: File Path to use. The special value <> means that the connector should use standard I/O streams
Internal name: filePath

Choose evaluation method for parameter value

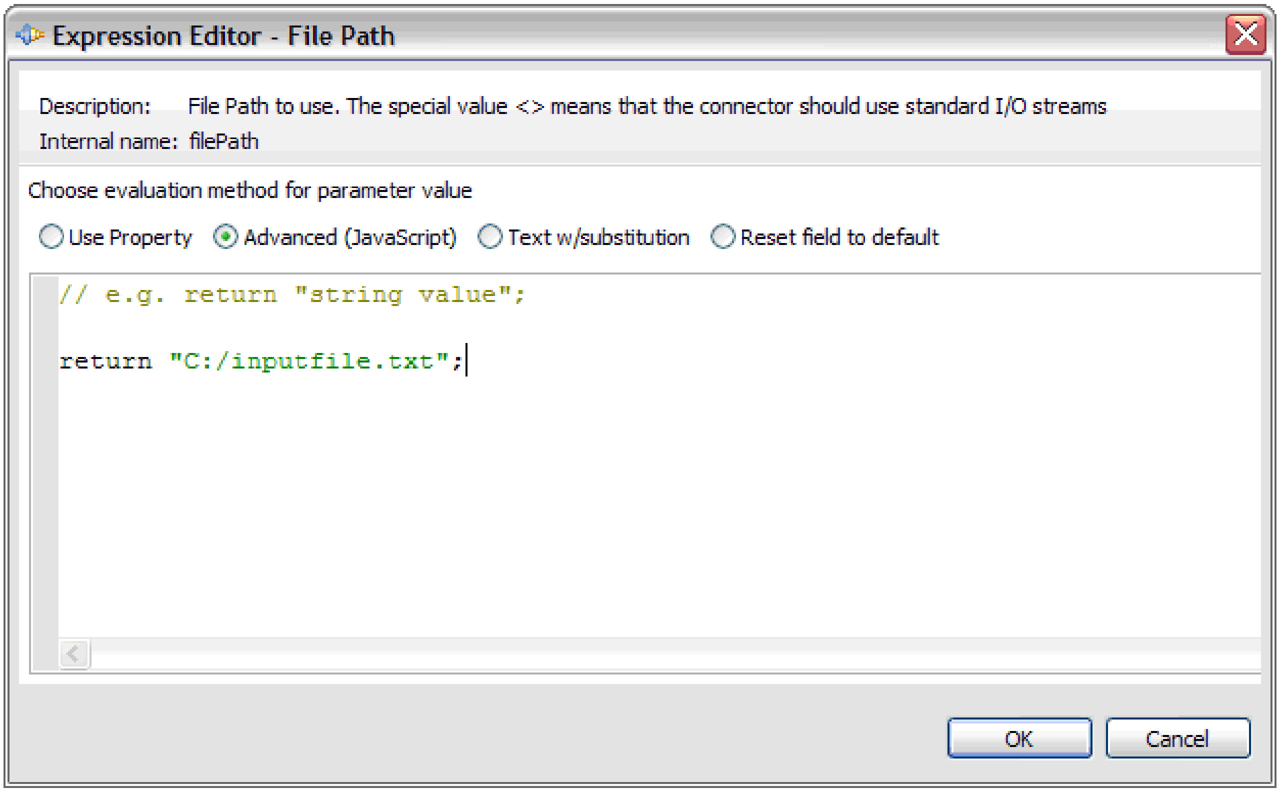
● Use Property ○ Advanced (JavaScript) ○ Text w/substitution ○ Reset field to default

Name	Value
Example	
LDAPHost (new)	ldap://localhost:900
Username (new)	username
Java-Properties	

Name: Example:Username

Figure 44. Expression Editor window, simple mode

In the expression editor you can conveniently choose a property from one of your property files or use advanced mode where you return a value based on JavaScript code. Check the **Advanced (JavaScript)** checkbox to toggle between property selection and JavaScript code. You can only use one or the other.



Description: File Path to use. The special value <> means that the connector should use standard I/O streams
Internal name: filePath

Choose evaluation method for parameter value

○ Use Property ● Advanced (JavaScript) ○ Text w/substitution ○ Reset field to default

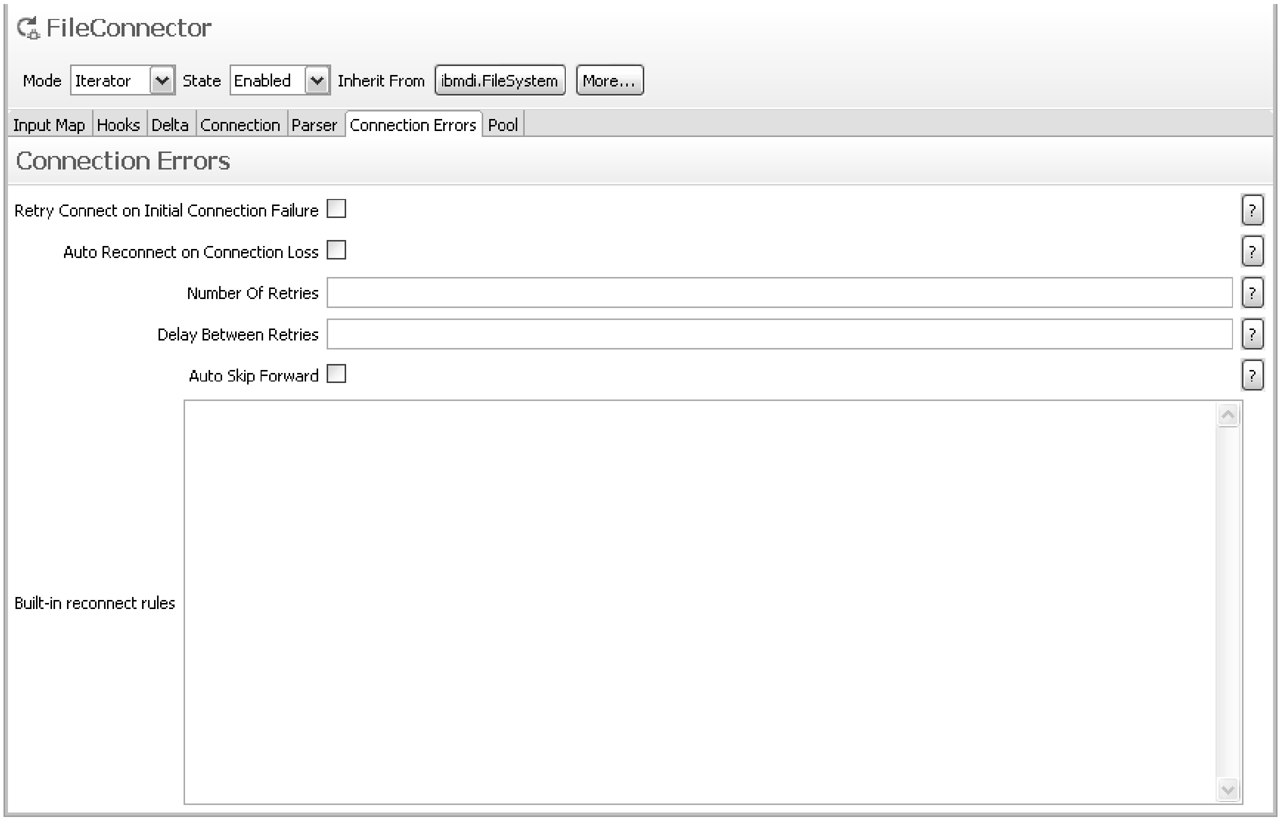
```
// e.g. return "string value";

return "C:/inputfile.txt";
```

Figure 45. Expression Editor window, Advanced (JavaScript) mode

Connection Errors

The Connection Errors tab is where you configure connection resilience, that is automated behavior to take when the connection that your component has made, fails.

Optionally, when you select **Retry Connect on Initial Connection Failure**, you can configure whether your connection attempt will throw an exception when it fails during startup, or whether it will retry. If this flag is set, and a connection cannot be established when the connector is being initialized, a "reconnect" attempt will be made; it is not really a reconnect, since a connection was not established in the first place, but generally the same mechanism as for the situations that can occur when an established connection is lost.



Mode: Iterator
State: Enabled
Inherit From: ibmdi.FileSystem

Retry Connect on Initial Connection Failure □
Auto Reconnect on Connection Loss □
Number Of Retries
Delay Between Retries
Auto Skip Forward □
Built-in reconnect rules

For established connections, the remaining parameters have the following significance:

Auto Reconnect on Connection Loss
If this flag is set, and the connection is lost after the connector is initialized, a reconnect attempt will be made.

Number Of Retries
The number of times a reconnect attempt will be made when a problem occurs, before giving up. If a new problem occurs later on, the same number of attempts will be made.

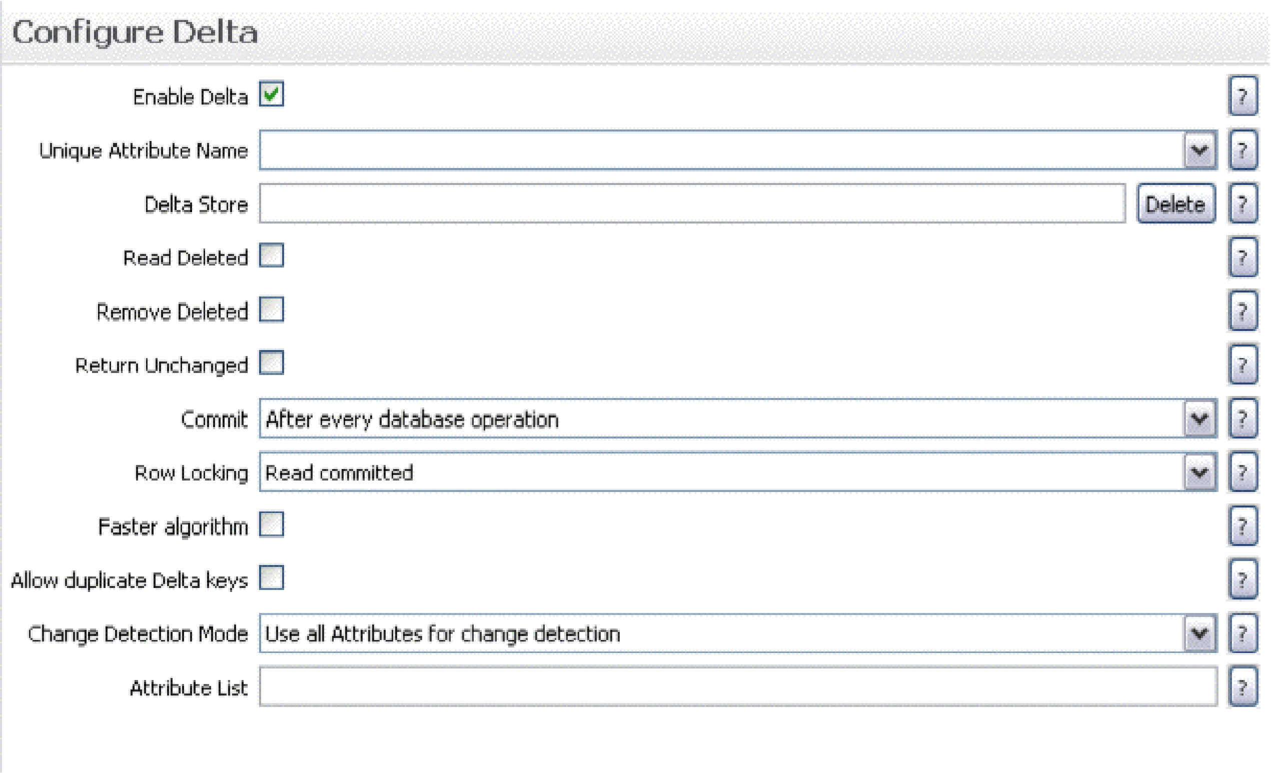
Delay Between Retries
The number of seconds to wait between each reconnect attempt, and before the first reconnect attempt.

Auto Skip Forward
After a reconnect, automatically skip forward as many times as the number of successful reads.

Built-in reconnect rules
These tie in with the Reconnect Rule Engine; see the corresponding chapter in the *IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide*

for more information.

Delta



Configure Delta

Enable Delta ☑

Unique Attribute Name

Delta Store

Read Deleted □

Remove Deleted □

Return Unchanged □

Commit: After every database operation

Row Locking: Read committed

Faster algorithm □

Allow duplicate Delta keys □

Change Detection Mode: Use all Attributes for change detection

Attribute List

This tab is only available in Iterator mode.

The parameters in this tab have the following significance:

Enable Delta
This is the master switch for the Delta engine for this Connector. If not selected, the parameters below are not enabled.

Unique Attribute Name
This is the name of an attribute, or a choice of multiple input attributes separated by "+", that holds a unique value in a given data source. Data sources with duplicate keys cannot be subjected to the delta function, except when **Allow duplicate Delta keys** is enabled. See “Delta Detection” on page 13 for more information.

Delta Store
The table in the System Store that holds the Delta information from previous runs for this Connector, so as to be able to detect differences on subsequent runs.

Read Deleted
If checked, the AssemblyLine will inject deleted entries into the AssemblyLine run when the Iterator has completed iterating, that is, finished input. The operation code will indicate that this entry was deleted in the input source. Note that delete-tagged Entries are not removed from the Delta Store unless you also enable the Remove Deleted flag.

Remove Deleted
If checked, the deleted entries from the input source are deleted from the Delta Store, such that they will not be detected again in subsequent runs.

Return Unchanged
If checked, any unchanged entries in this run are injected into the AssemblyLine.

Commit

Selects when to commit changes to the Delta Store as a result of iterating through the input. Choices are:

- After every database operation
- On end of AL Cycle
- On Connector close
- No autocommit

The default is **After every database operation**.

Row Locking

Selects the transaction isolation level for the connection to the Delta Store. This parameter addresses the need for row locking in a Delta Store table when multiple DB Clients access the same data, by setting a *Transaction Isolation Level*. Setting a higher isolation level reduces the transaction anomalies known as 'dirty reads', 'non-repeatable reads' and 'phantom reads' by using row and table locks

Choices are:

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

The default is READ_COMMITTED.For more information, see section “Row Locking” on page 218.

Attribute List

This is a list of comma separated attributes whose changes will be detected or ignored during the compute changes process. The changes in listed attributes will be affected by the **Change Detection Mode** parameter, which specifies whether to ignore or detect them. For more information about this parameter and the next, see section “Detect or ignore changes only in specific attributes” on page 219.

Change Detection Mode

Specify whether to detect or ignore changes in Attributes listed in the **Attribute List** parameter. Possible values are:

- IGNORE_ATTRIBUTES
- DETECT_ATTRIBUTES
- DETECT_ALL

Faster algorithm

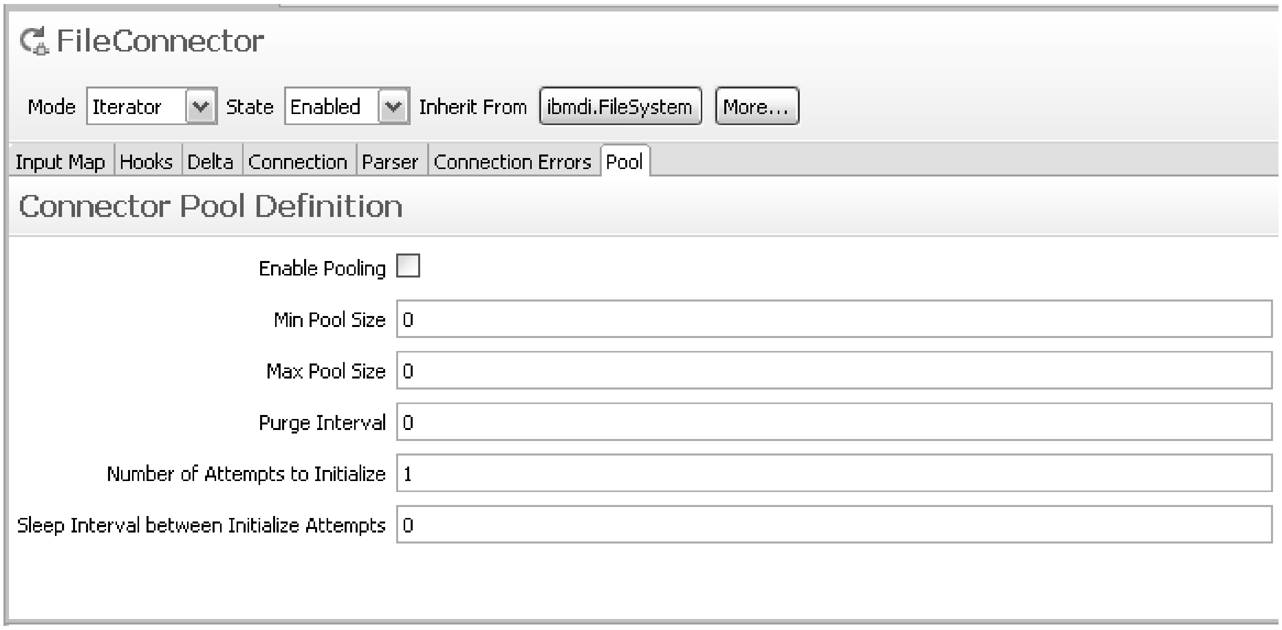
When checked, instructs the AssemblyLine to use a faster algorithm to compute changes, at the expense of more memory use.

Allow duplicate Delta keys

When the Delta feature is enabled for Changelog/Change Detection Connector in long running AssemblyLines, it is possible that an Entry can be modified more than once. These modifications will result in receiving the Entry second time and this will cause a Duplicate delta key exception to be thrown. Checking this parameter allows Entries with duplicate key attributes (specified in the **Unique Attribute Name** parameter) to be processed by Iterator Connectors with enabled Delta.

Pool

The pool definition for a connector is only visible when the connector resides in the connector library (that is, a file in Project/Resources/Connectors). Connectors that are opened from the AssemblyLine will not have this tab.



FileConnector

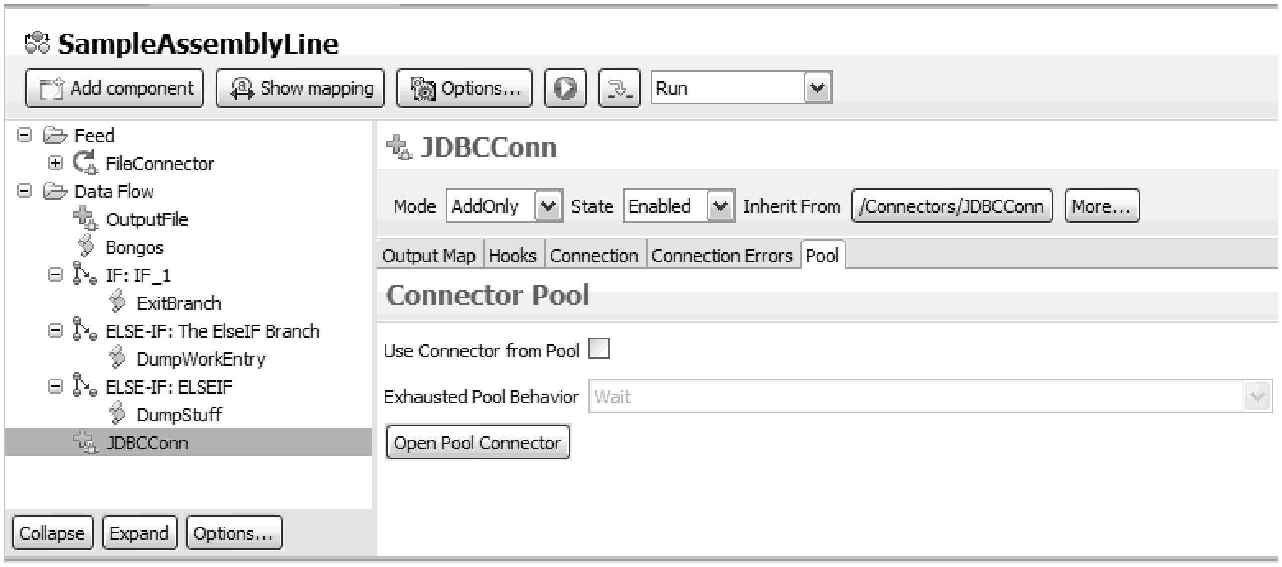
Mode: Iterator
State: Enabled
Inherit From: ibmdi.FileSystem

Connector Pool Definition

Enable Pooling □
Min Pool Size: 0
Max Pool Size: 0
Purge Interval: 0
Number of Attempts to Initialize: 1
Sleep Interval between Initialize Attempts: 0

Figure 46. Pool tab: Connector Pool definition

When a pooled connector is used in an AssemblyLine it will show the following tab instead:



SampleAssemblyLine

JDBCConn

Mode: AddOnly
State: Enabled
Inherit From: /Connectors/JDBCConn

Connector Pool

Use Connector from Pool □
Exhausted Pool Behavior: Wait

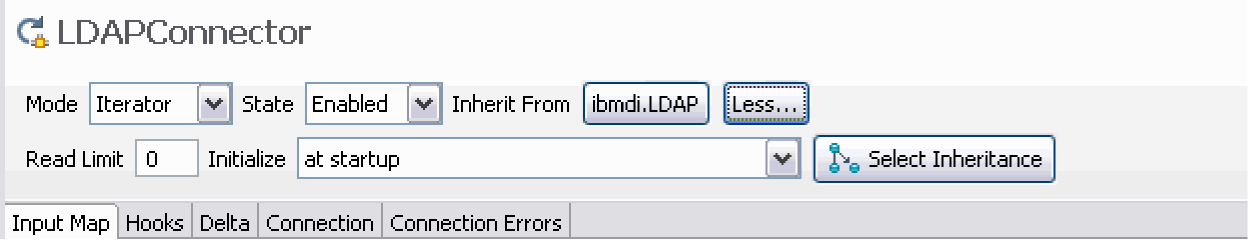
Figure 47. Pool tab: Connector in AssemblyLine

Use the **Open Pool Connector** button to open the pooled connector.

Connector Inheritance

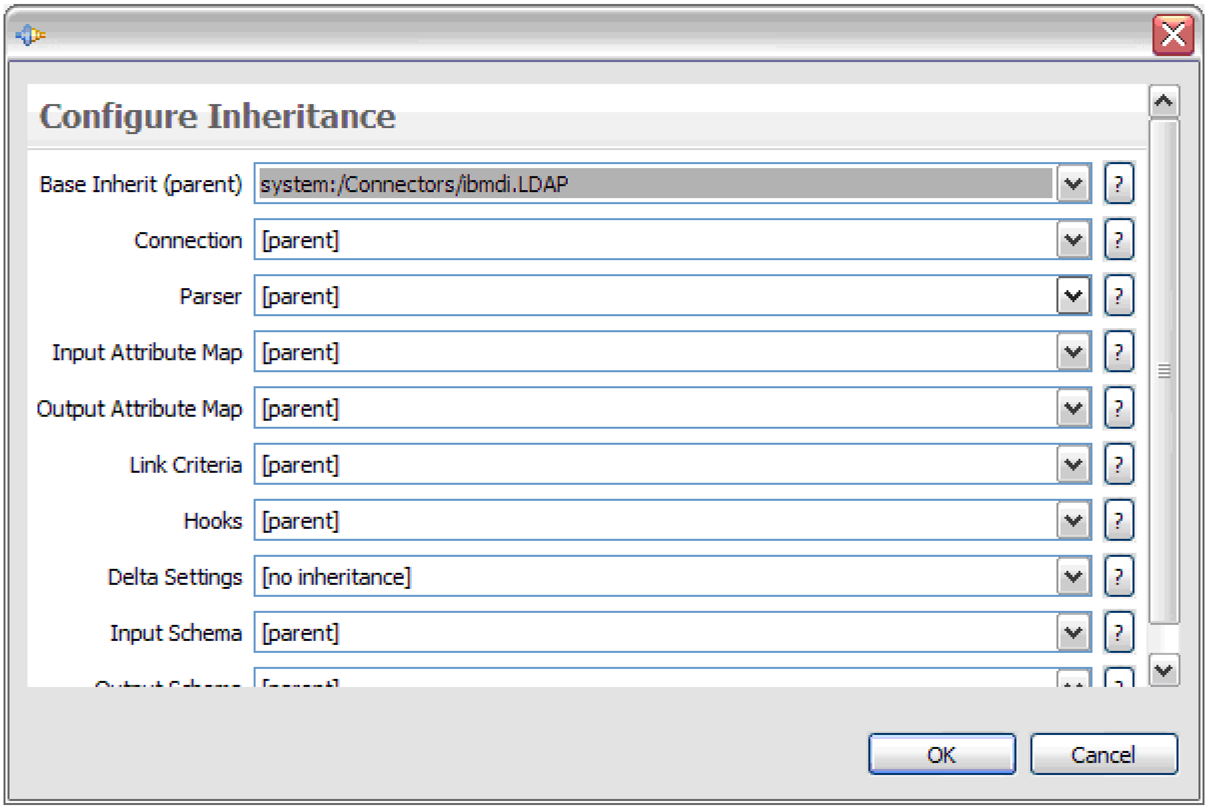
While you can change inheritance at various places in the editor you can also use the connector inheritance button for a complete list of inheritance settings.

Use the **More...** button to expand the connector editor's header and click the **Inheritance** button.


LDAPConnector

Mode: Iterator
State: Enabled
Inherit From: ibmdi.LDAP
Read Limit: 0
Initialize: at startup

The **Inheritance** button will bring up a dialog that shows all inheritance settings for the connector:


Configure Inheritance

Base Inherit (parent): system:/Connectors/ibmdi.LDAP
Connection: [parent]
Parser: [parent]
Input Attribute Map: [parent]
Output Attribute Map: [parent]
Link Criteria: [parent]
Hooks: [parent]
Delta Settings: [no inheritance]
Input Schema: [parent]
Output Schema: [parent]

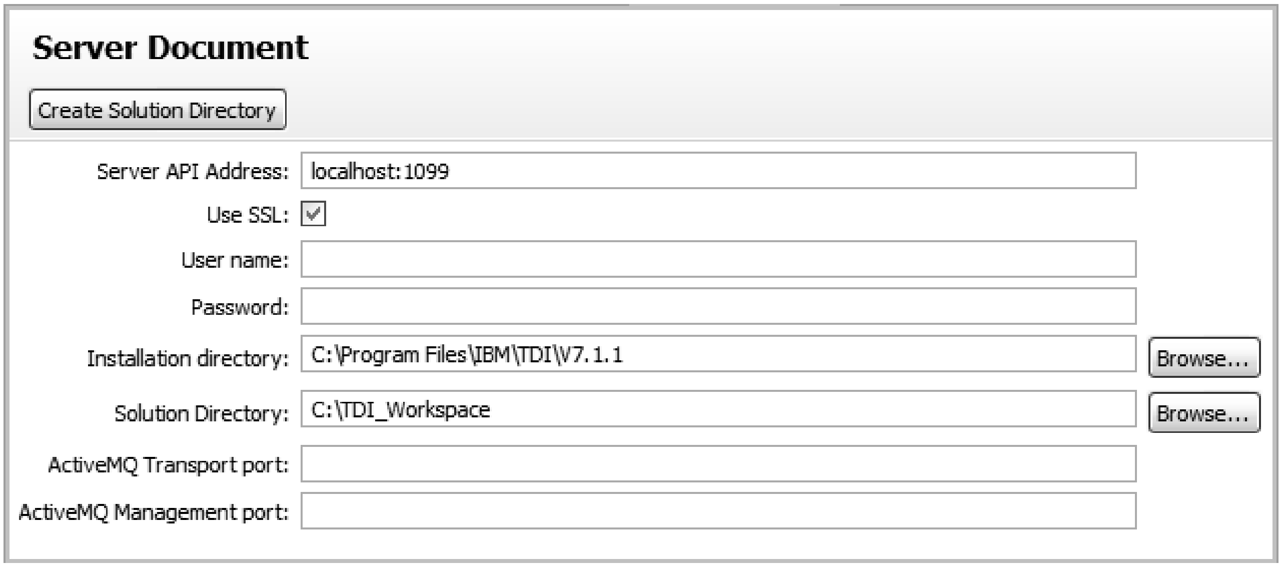
Figure 48. Connector Editor: Configure Inheritance

The notation [parent] means that the item is inherited from the component listed in the **Base Inherit (parent)** field.

Server Editor

The server editor is where you define how to reach a Tivoli Directory Integrator server.

One server is always defined by the CE and is named "Default". In the Tivoli Directory Integrator servers view you can add new ones to your project.



Server Document

Server API Address: localhost:1099

Use SSL: ☑

User name:

Password:

Installation directory: C:\Program Files\IBM\TDI\V7.1.1

Solution Directory: C:\TDI_Workspace

ActiveMQ Transport port:

ActiveMQ Management port:

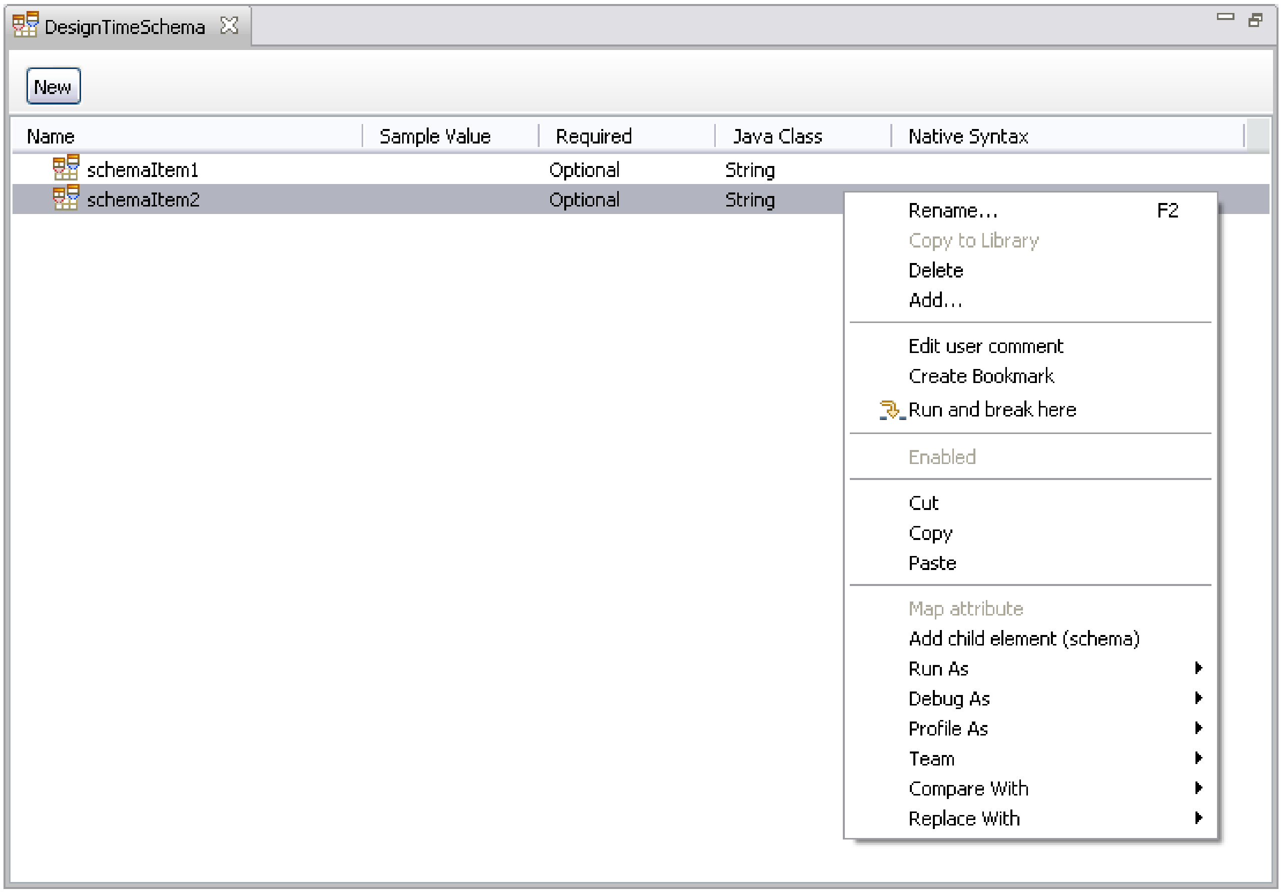
Figure 49. Server Document editor

The server API address is the `host:port` address of the Tivoli Directory Integrator server. If you specify the installation directory, the CE can start the server. Before starting a new Tivoli Directory Integrator server, configure all parameters and use the **Create Solution Directory** button to create the necessary files for the server. You can also specify unique ports for Apache ActiveMQ transport and management.

Schema Editor

The schema editor manages design schema files.

These files can be used by input and output maps in other editors. The schema is design time only (that is, in the CE only) and is typically used when you have huge schemas that you don't want to appear in the runtime configuration file.



Name	Sample Value	Required	Java Class	Native Syntax
schemaItem1		Optional	String	
schemaItem2		Optional	String	

Figure 50. Schema Editor

The editor provides a **New** button to add a new top level schema item and a context menu to operate on existing schema items.

Data Browser

The Data Browser provides an in depth look at a target system. Currently there are only the LDAP and JDBC connectors that provide extra details for a connector. The data browser is opened by right clicking a Connector in either the library or in an AssemblyLine.

In the navigator you can right click and choose **Browse Data** to open a new editor window where you can browse data within the current connection setup by the connector.

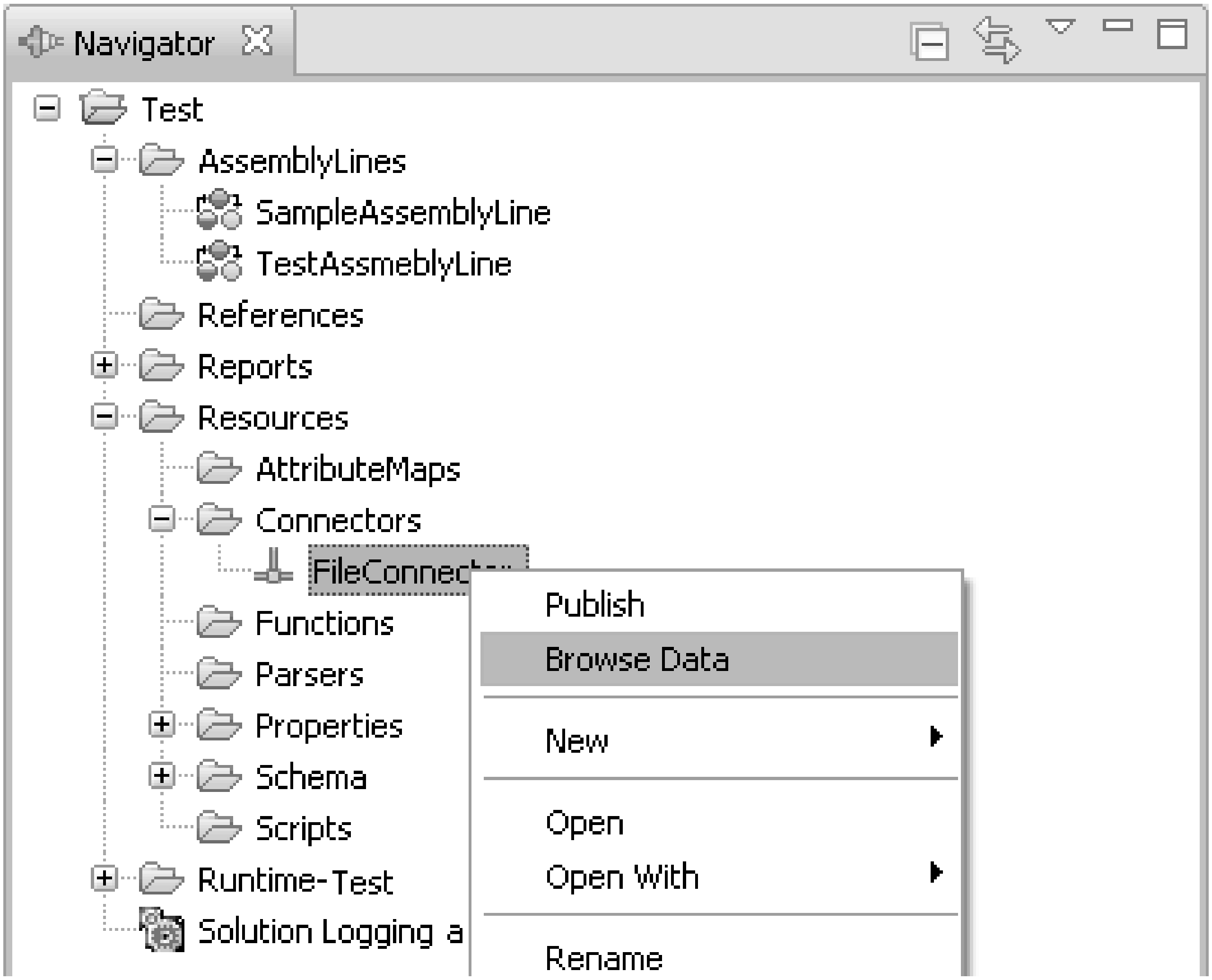


Figure 51. Data Browser

In the AssemblyLine you can do the same and have a new window opened for the data browser:

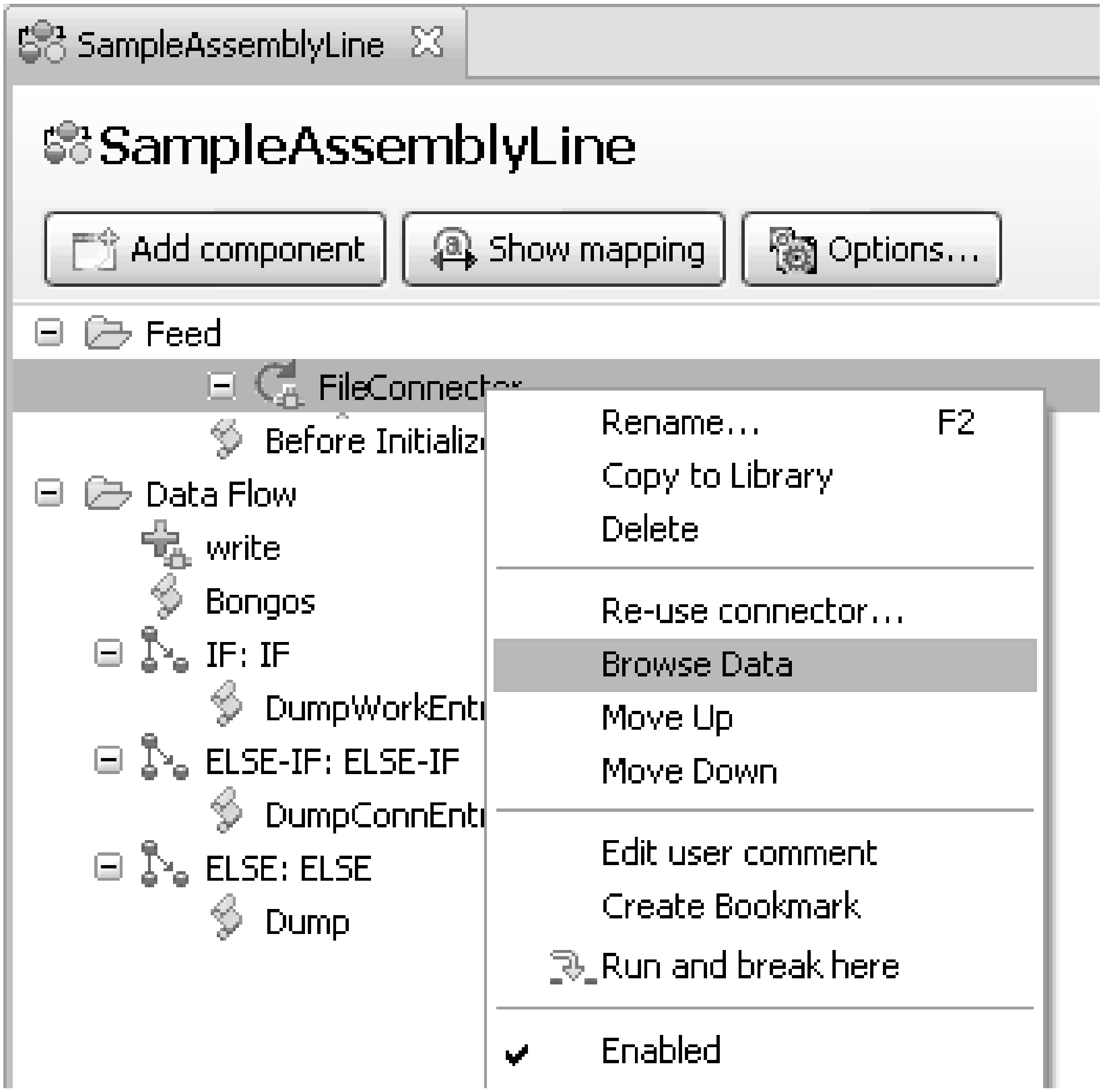


Figure 52. Data Browser

Generic Data Browser

This is the data browser used for those connectors that the Configuration Editor has no explicit knowledge of. It provides a simple way to browse a result set from the data source.

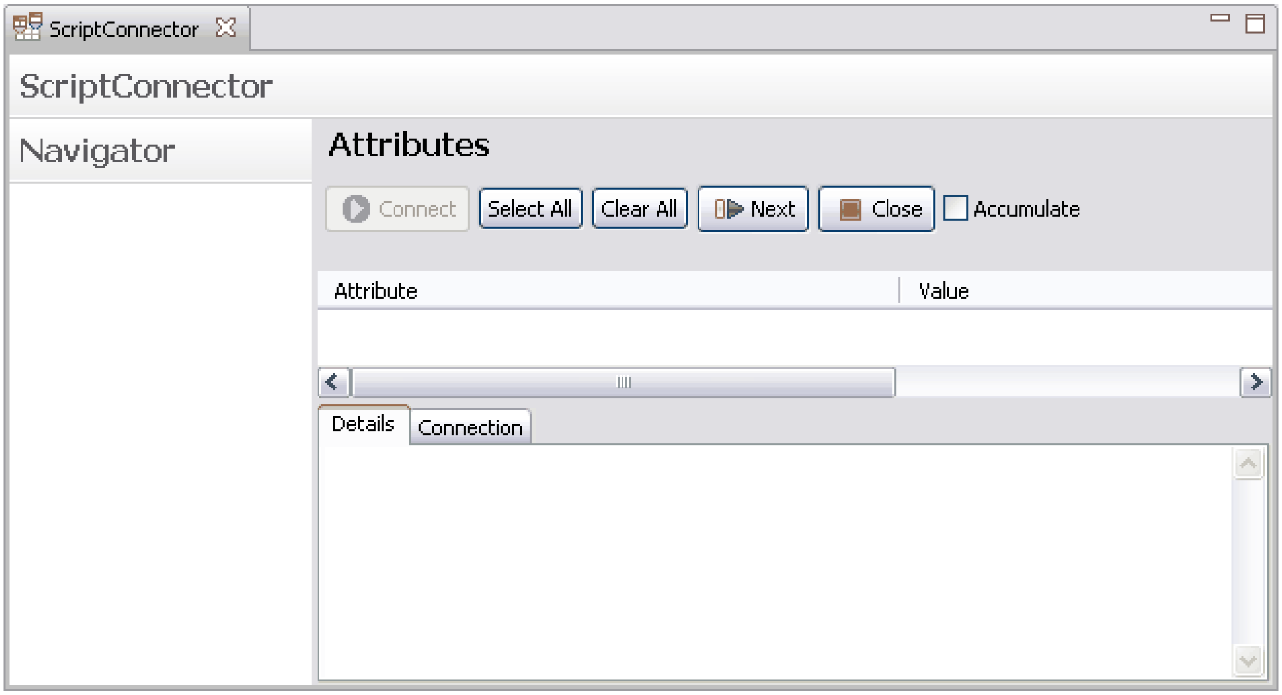


Figure 53. Generic Data Browser

The upper part shows a list of attributes read from the connector and a toolbar. Attributes in the list have checkboxes; when you check an attribute an attribute mapping is created for that attribute using a simple `conn.attributename -> work.attributename` expression. When you uncheck an attribute the attribute mapping for that attribute is removed.

The toolbar has the following functions:

Table 8. Data Browser toolbar

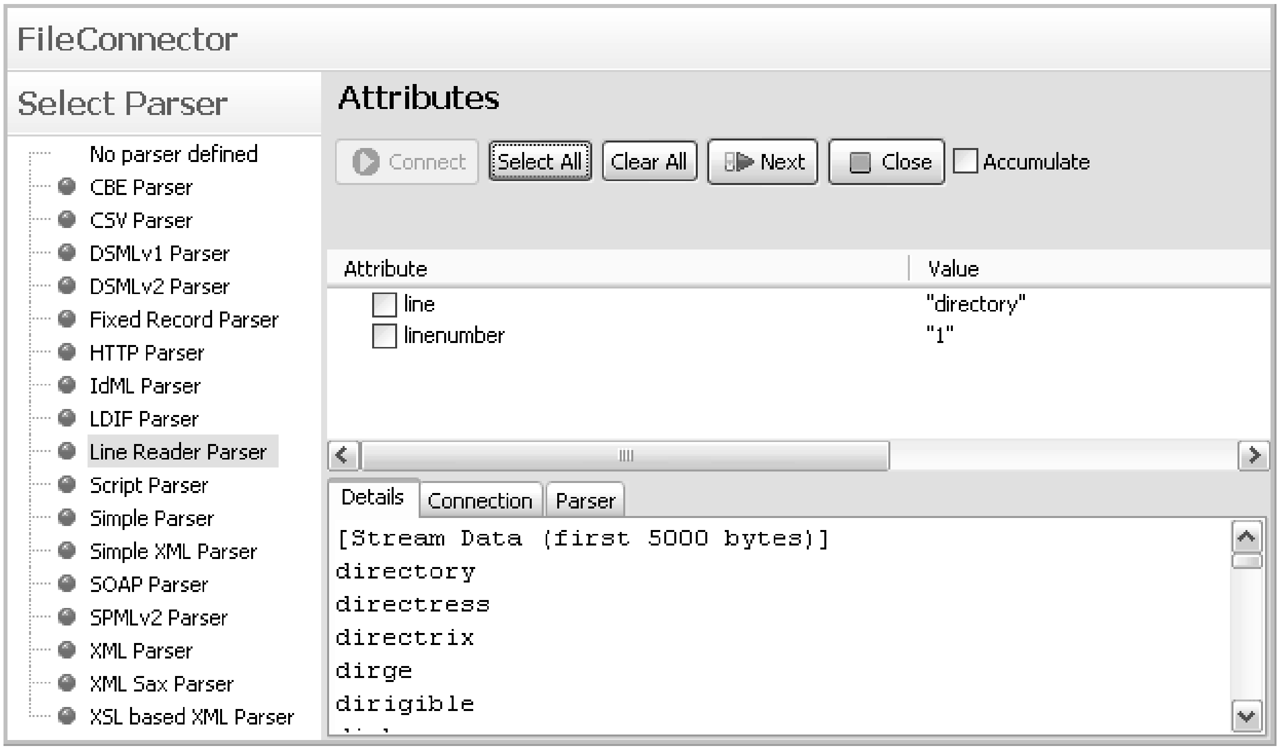
Toggle All	This command will toggle the checkboxes for every attribute in the list. This will cause a modification to the attribute map for all attributes listed.
Accumulate	This command toggles whether the list of discovered attributes are accumulated or not. When you accumulate attributes, every record read from the connector is merged with the existing list of attributes. When you don't accumulate, all attributes are removed before the next record is shown in the list.
🟥	Click this button to close the connection. When you close the editor window for the browser, the connection is automatically closed. You typically want to close the connection before you read the next record if you have modified the connection settings in this editor.
▶	Click this button to read the next record from the connector. When there no more records returned from the connector, a message is shown to the left of the toolbar to indicate there are no more entries from the connector. Pressing this button again after this condition causes the connector to start reading from the beginning of its result set.

The lower part of the screen shows two tabs. The first tab is the **Details** tab that contains details about the current selection. For the generic data browser this tab will always be empty.

The second tab is the **Connection** tab. This tab shows the connection configuration for the connector. You can modify the connection parameters and save it just as you do when you open the Connector editor.

Stream Data Browser

The stream based data browser is used when a connector uses a parser. The stream based data browser will first initialize the connector and try to obtain the input stream and show the first 20K of input data in the details tab.



Attribute	Value
□ line	"directory"
□ linenumber	"1"

```
[Stream Data (first 5000 bytes)]
directory
directress
directrix
dirge
dirigible
```

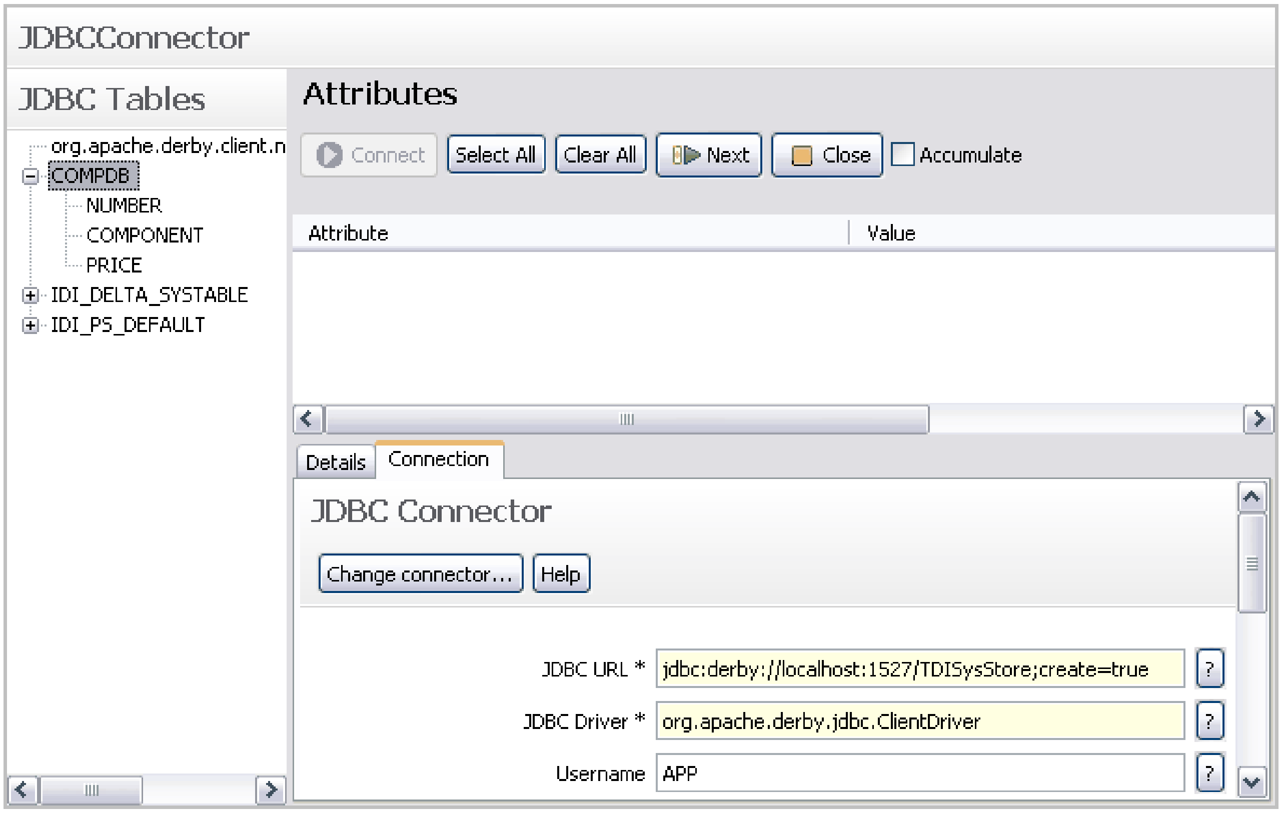
Figure 54. Stream Data Browser

The left side now contains a **Select Parser** list where you can select a parser, in order to try to read the contents of the input stream to see if it matches your expectations. When you select a parser, the **Parser** tab will be updated with the configuration form for that parser. Whenever you change the parser, the connector will be closed so you can easily see if a parser is able to interpret the input by continuously selecting a parser followed by a read-next.

Note: When you select a parser in the table, you also modify the connector configuration to use that parser. When you close and save (or use the **File** > **Save** function) you effectively update the configuration with those parameters you have currently configured.

JDBC Data Browser

The JDBC data browser shows all tables and views in the left hand side. The details tab shows information for the selected item in that list.



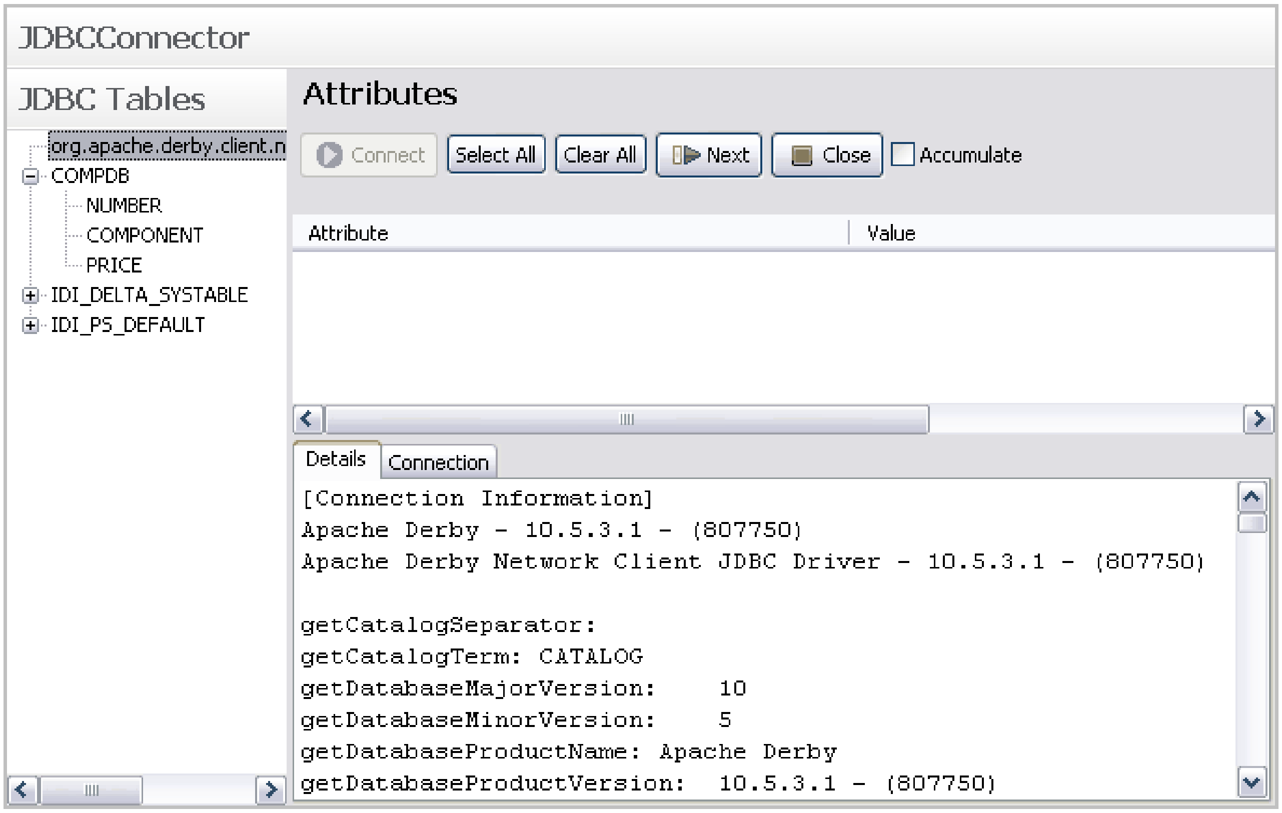
JDBC URL * jdbc:derby://localhost:1527/TDISysStore;create=true

JDBC Driver * org.apache.derby.jdbc.ClientDriver

Username APP

Figure 55. JDBC Data Browser

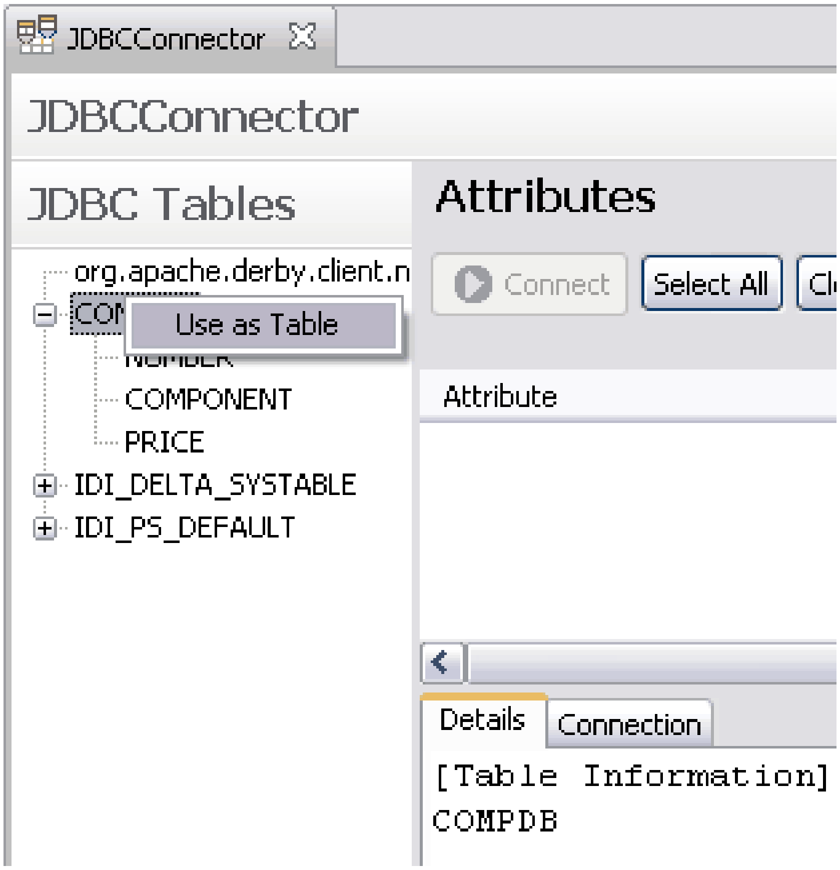
The **Details** tab shows the system information obtained from the JDBC connection object. The tree view to the left also shows all tables and views with their columns as child entries. When you select a table or view the details tab will be populated with the syntax for the entire table. For example, if you select the "IDI_PS_DEFAULT" table, you should see something like this:



```
[Connection Information]
Apache Derby - 10.5.3.1 - (807750)
Apache Derby Network Client JDBC Driver - 10.5.3.1 - (807750)

getCatalogSeparator:
getCatalogTerm: CATALOG
getDatabaseMajorVersion:    10
getDatabaseMinorVersion:    5
getDatabaseProductName: Apache Derby
getDatabaseProductVersion:  10.5.3.1 - (807750)
```

Figure 56. JDBC Table details

When you select a column in a table or view you will see the details for that column only.



```
[Table Information]
COMPDB
```

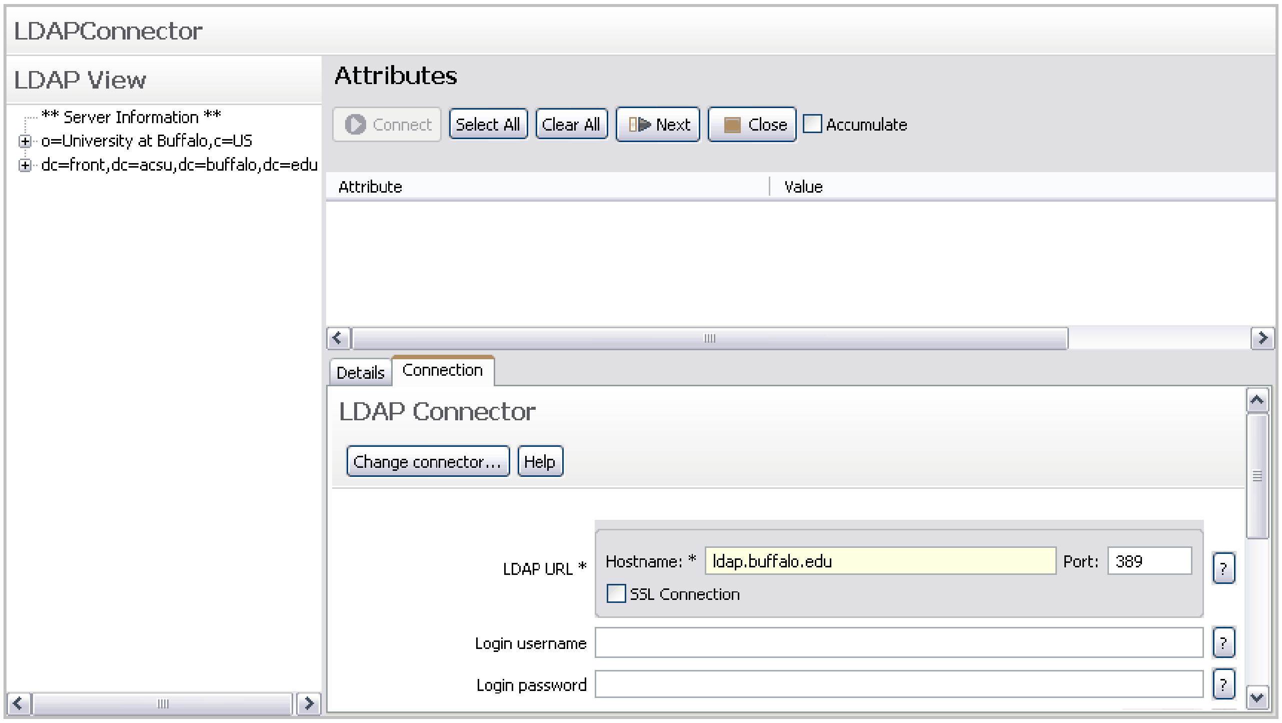
Figure 57. "Use as table" option

You can also right-click a table name and use the **Use as table** function to update the **`Table Name`** parameter of the JDBC connector configuration.

The ▶ button in the toolbar of the **JDBC Tables** header does a rediscovery of the connection. You only need to use this if the initial discovery failed, or if you have changed the JDBC URL in the connection tab.

LDAP Data Browser

The LDAP data browser shows the schema and context prefixes (search bases) that the LDAP server provides.



LDAP URL *

Hostname: * ldap.buffalo.edu Port: 389

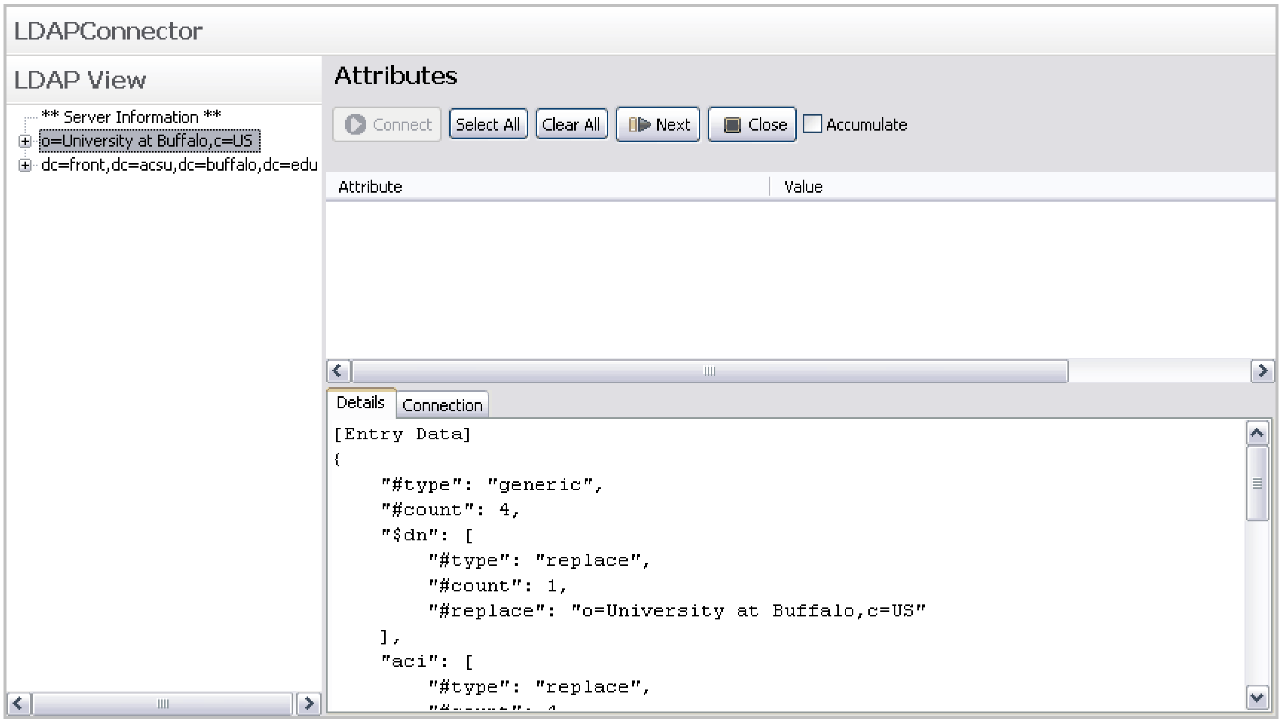
□ SSL Connection

Login username

Login password

Figure 58. LDAP Data Browser

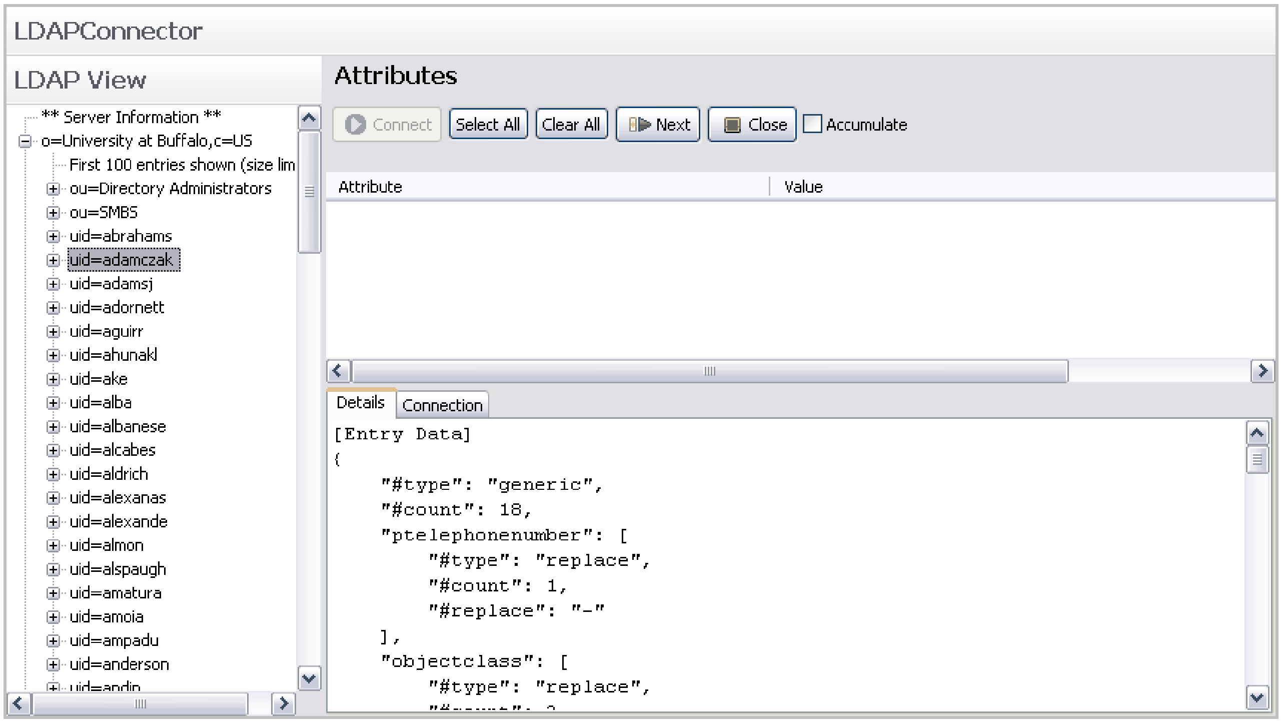
The LDAP View contains both server information and search bases that the LDAP server provides. Based on your selection in this tree you will see different results. If you select one of the non-schema nodes, you should see a detailed dump for that specific entry in the details tab as shown below:



```
[Entry Data]
{
    "#type": "generic",
    "#count": 4,
    "$dn": [
        "#type": "replace",
        "#count": 1,
        "#replace": "o=University at Buffalo,c=US"
    ],
    "aci": [
        "#type": "replace",
```

Figure 59. LDAP Data Browser entry

When you select a schema item you see the details in the details tab as well as having your attribute list updated with information from the schema item. This is very useful if you are going to read or write a specific schema:

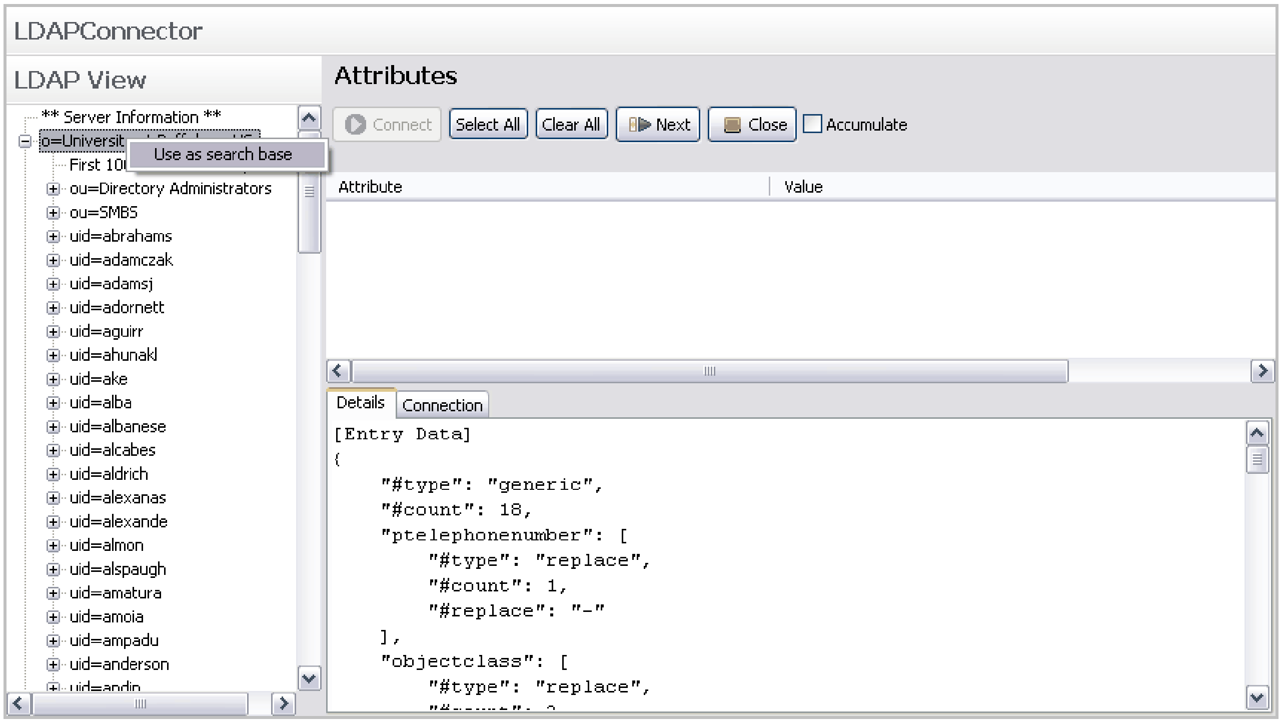


```
[Entry Data]
{
    "#type": "generic",
    "#count": 18,
    "ptelephonenumber": [
        "#type": "replace",
        "#count": 1,
        "#replace": "-"
    ],
    "objectclass": [
        "#type": "replace",
```

Figure 60. LDAP Data Browser schema item

Choosing an object class in the schema node lets you quickly create an attribute mapping for that class. The value column now contains information about the attribute. The value "MAY" means it is optional, whereas "MUST" means it is required (when adding entries). The value in parenthesis shows the object class that defines the attribute; LDAP object classes are hierarchical.

You can also quickly update the **`Search Base`** parameter of the LDAP connector by choosing **Use as search base** from the context menu.



```
[Entry Data]
{
    "#type": "generic",
    "#count": 18,
    "ptelephonenumber": [
        "#type": "replace",
        "#count": 1,
        "#replace": "-"
    ],
    "objectclass": [
        "#type": "replace",
```

Figure 61. "Use as search base" context menu choice

Forms Editor

The forms editor is used to customize the connection parameters form for a component. This can only be applied to components in the Resources folder (except properties files).

To customize a form, you must open the component with the *Forms Editor*.

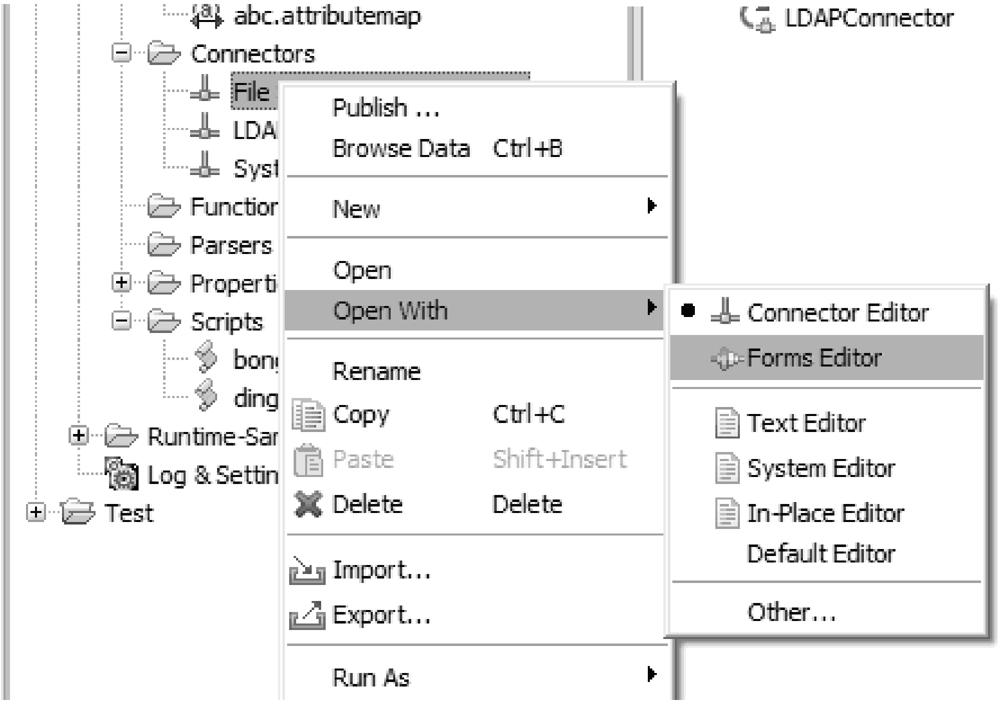
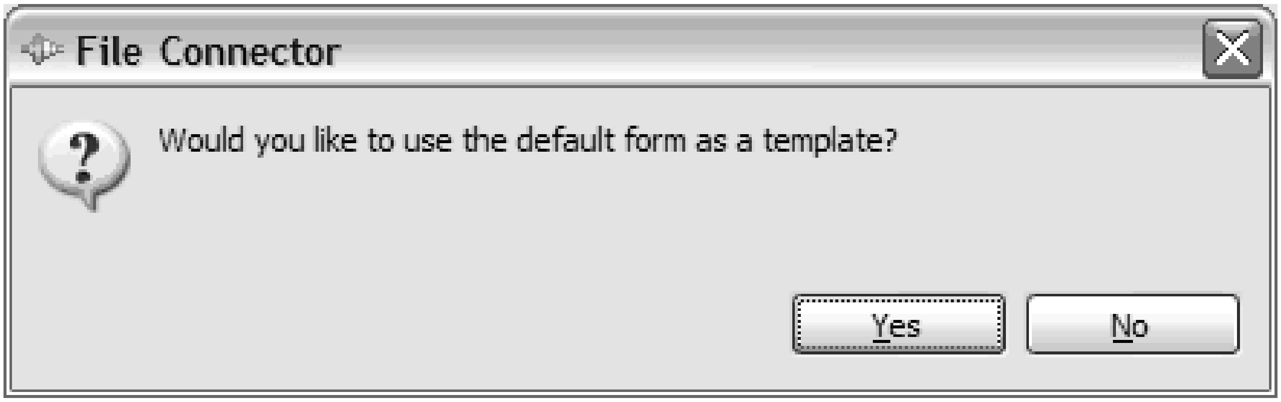


Figure 62. Context menu - Forms Editor choice

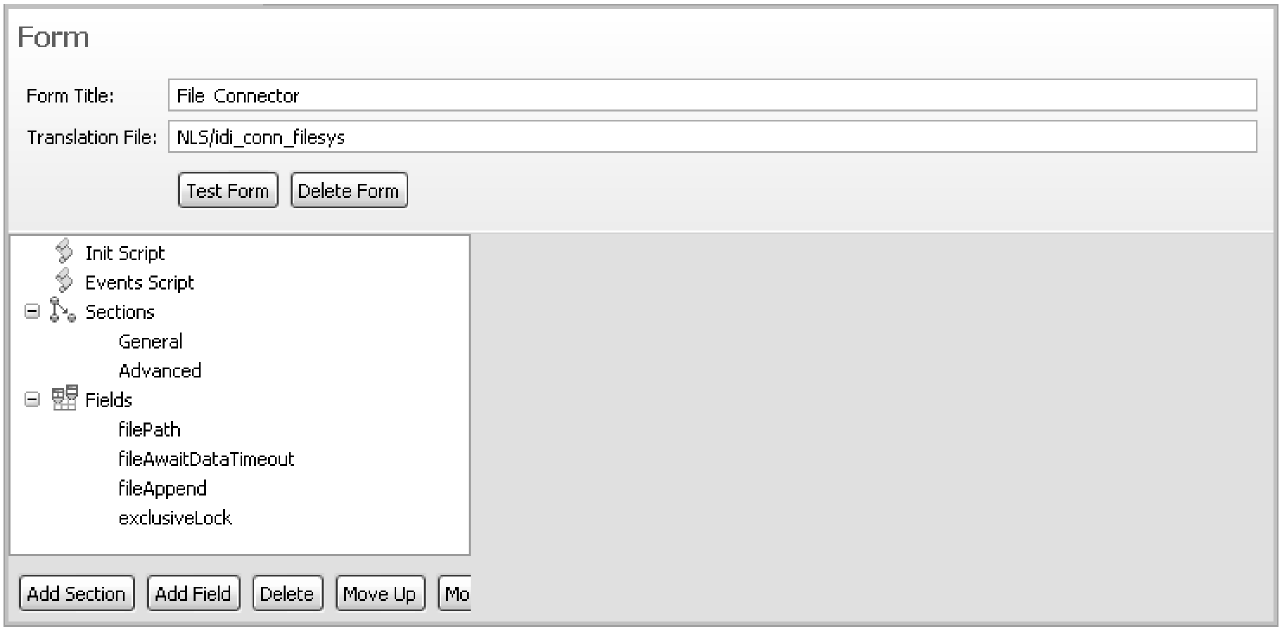
Note that when you choose a different editor for a file than the default, the CE will remember your choice, so that the next time you double click on the file it will open the file with the editor you used last. To open the component with the default editor, simply choose the appropriate editor in this menu (typically the top most editor).

If the component you open has no custom form, you will be prompted to populate the form with the default form:


File Connector

Would you like to use the default form as a template?

Choosing **Yes** will create an initial form definition based on the default form for the component. In this case the FileSystem connector is uesed in the example, which results in the following screen:



Form

Form Title: File Connector

Translation File: NLS/idi_conn_filesys

Figure 63. Default Forms Editor screen for FileSystem Connector

The various elements you see in this form have the following function:

Form Title
This is the main title of the component form (leave blank for no title). This is what the user sees at the top of the custom form.

Translation file
This is the file used to translate the labels and tooltips in the form. All labels and tooltips are attempted translated if you define a translation file. If you create a label with "file_name" as the text, the translation will try to retrieve a string from the translation file using "file_name" as the key. The translation files themselves are simple property files with "key=value" on each line.

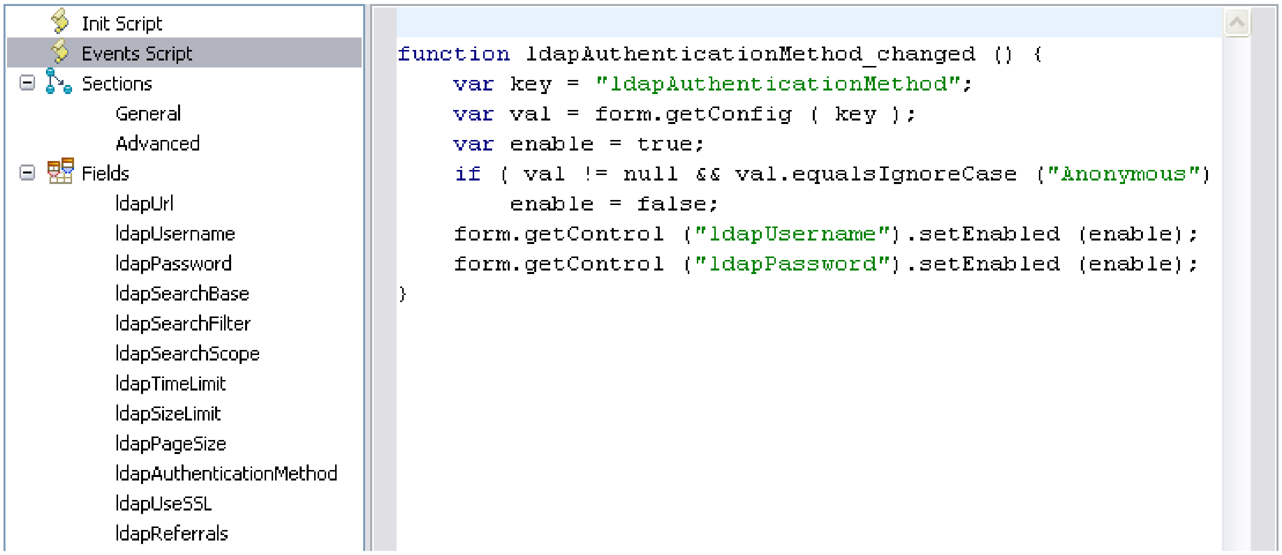
To localize a form you must create files with the locale identifier. So, for a French translation you would create a file called *base-name*`_fr.properties`.

Test Form
This button will show a dialog window with the current form definition.

Delete Form
This button will delete the form from the component. You must close and choose **save** for this to have permanent effect.

Init Script
The init script is executed when the form is loaded. This is where you place code to initialize the state of the form and any global script variables for the form.

Events Script
When a field value changes, the Form will execute an event handler defined in this script. Every field has an internal name the component uses. For example, the FileConnector uses "filePath" as its internal name for the **File path** parameter. You can see all the component parameter names when you choose to populate your form with the default form in the Fields section. To react to changes in the form, you write event handlers in the events script editor using the internal name suffixed by "_changed" as the script function name. Below you see an example from the LDAP connector that disables two fields based on the authentication method:

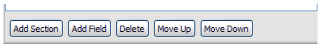


```
function ldapAuthenticationMethod_changed () {
    var key = "ldapAuthenticationMethod";
    var val = form.getConfig ( key );
    var enable = true;
    if ( val != null && val.equalsIgnoreCase ("Anonymous")
        enable = false;
    form.getControl ("ldapUsername").setEnabled (enable);
    form.getControl ("ldapPassword").setEnabled (enable);
}
```

Figure 64. Forms Editor, Events Script in LDAP Connector

Sections and Fields

Sections and fields are what make up the form. You manage the sections and fields by adding, removing and rearranging the order of them using the toolbar below the tree.

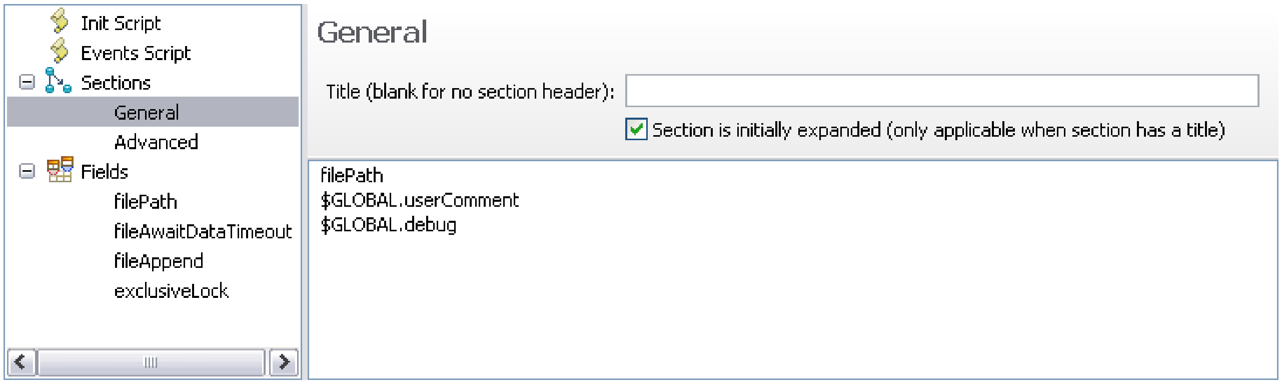


- **Add Section –** adds a new and empty section to the form
- **Add Field –** adds a new field to the form. Field names must be unique.
- **Delete –** removes a section or field from the form.
- **Move Up/Down –** rearranges the order of sections and forms. When you have sections defined, the order of fields is defined by the section and not the list of fields. When no sections are defined, the order in this tree determines the field order in the form.

Sections

This part is optional. If you don't specify sections then the form will have all fields displayed in its form, all at once.

Sections are used to arrange fields in the form. Sections are like folders the user can expand and collapse to display/hide its contents. When you define a section you specify whether the section is initially expanded and which fields are to be displayed in it. In the FileSystem connector example, there are two sections.



General

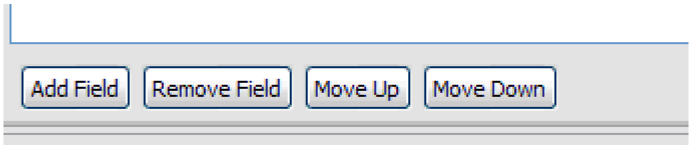
Title (blank for no section header):

☑ Section is initially expanded (only applicable when section has a title)

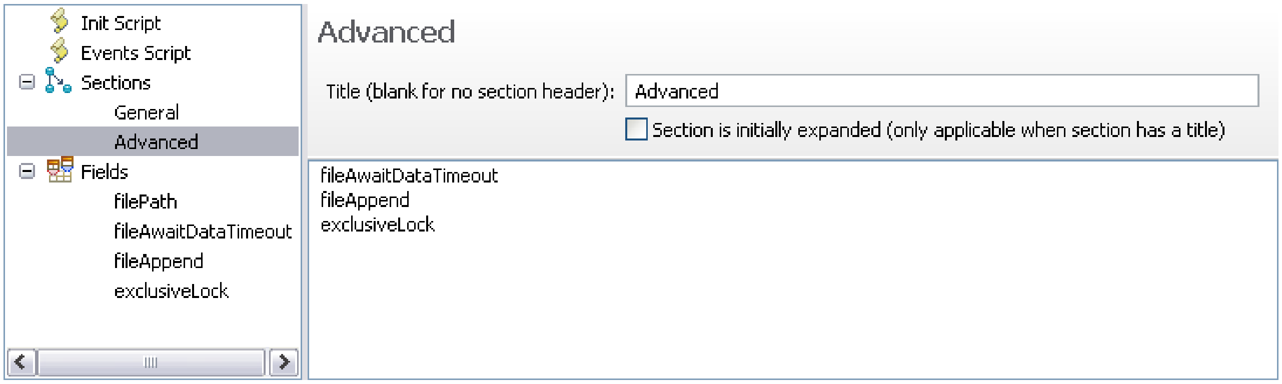
filePath
$GLOBAL.userComment
$GLOBAL.debug

Figure 65. Forms Editor - General section

The first section is the *General* section. This section has no title, which means that there will be no section header the user can click to expand or collapse contents. This makes it a static section since the section cannot be collapsed or expanded. You can add, remove and reorder the fields in the list of fields using the toolbar at the bottom of the panel:



The second section is the *Advanced* section:



Advanced

Title (blank for no section header): Advanced

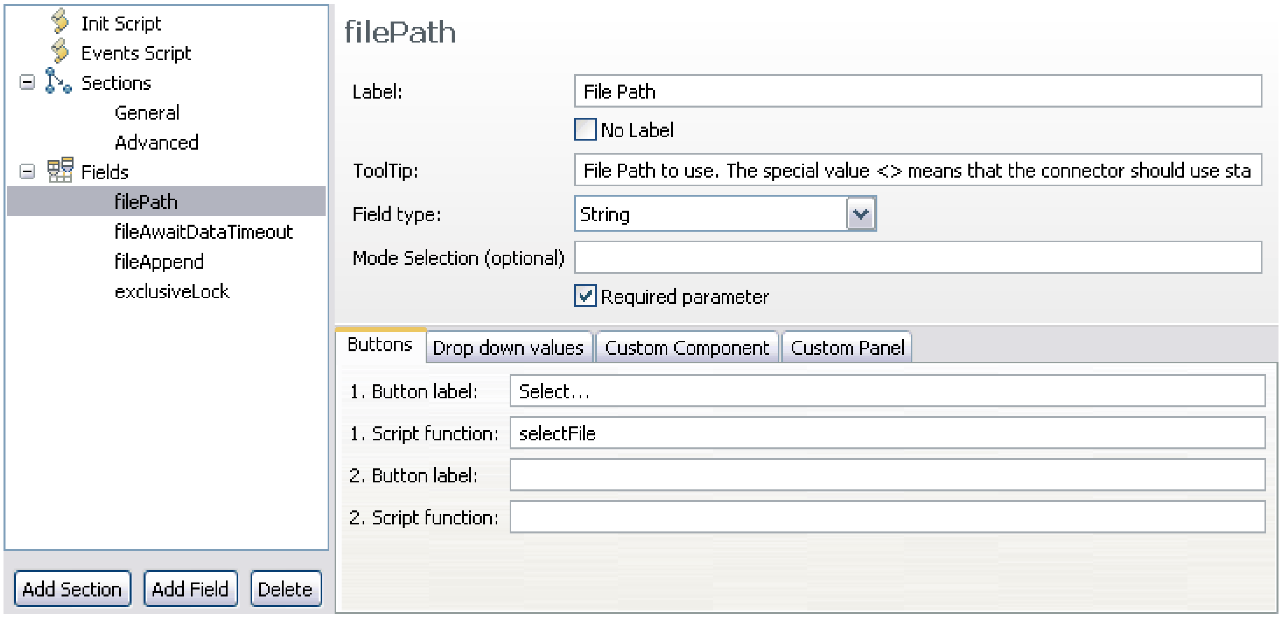
☐ Section is initially expanded (only applicable when section has a title)

fileAwaitDataTimeout
fileAppend
exclusiveLock

Figure 66. Forms Editor - Advanced section

This section does have a title, but it is not initially expanded. This will cause the form to display the form with this section title collapsed and the fields within are hidden until the user expands the section.

Fields Fields are the parameters for the component. If you reuse a component like the FileSystem connector you should either provide the required parameters in the form, or set the parameter in the "Init Script" section so that the component can function properly. Every field has its definition shown when you select it.



filePath

Label: File Path

□ No Label

ToolTip: File Path to use. The special value <> means that the connector should use sta

Field type: String

Mode Selection (optional):

☑ Required parameter

1. Button label: Select...

1. Script function: selectFile

2. Button label:

2. Script function:

Figure 67. Forms Editor - field definitions

In this panel you see the definition for the connector parameter “filePath”. In the order of appearance:

Table 9. Forms Editor - parameter definition

Field	Description
Label	The label that the form will show
ToolTip	The tooltip displayed when the user mouses over the input field
Field Type	The type of input field: • String for single line text input • Drop down (editable) for an editable drop down input field • Drop down (non-editable) for a drop down with a fixed set of selections • Boolean for a checkbox • Text Area for a multi line text input field • Static Text for simple text display (that is, no input) • Password for a single line password protected input field • Script Editor for editing scripts • Custom Component for a user defined SWT/JFace control
Mode Selection	This optional field can specify component modes where the field is excluded or included. Specify modes separated by a comma with a minus sign to exclude. “Iterator” – only show in Iterator mode “-Iterator” – Do not show in Iterator mode “Iterator,Lookup” – only show in Iterator and Lookup mode

The three tabs at the bottom let you specify buttons, drop down values for Drop-down lists and the Java class name for the custom component.

Wizards

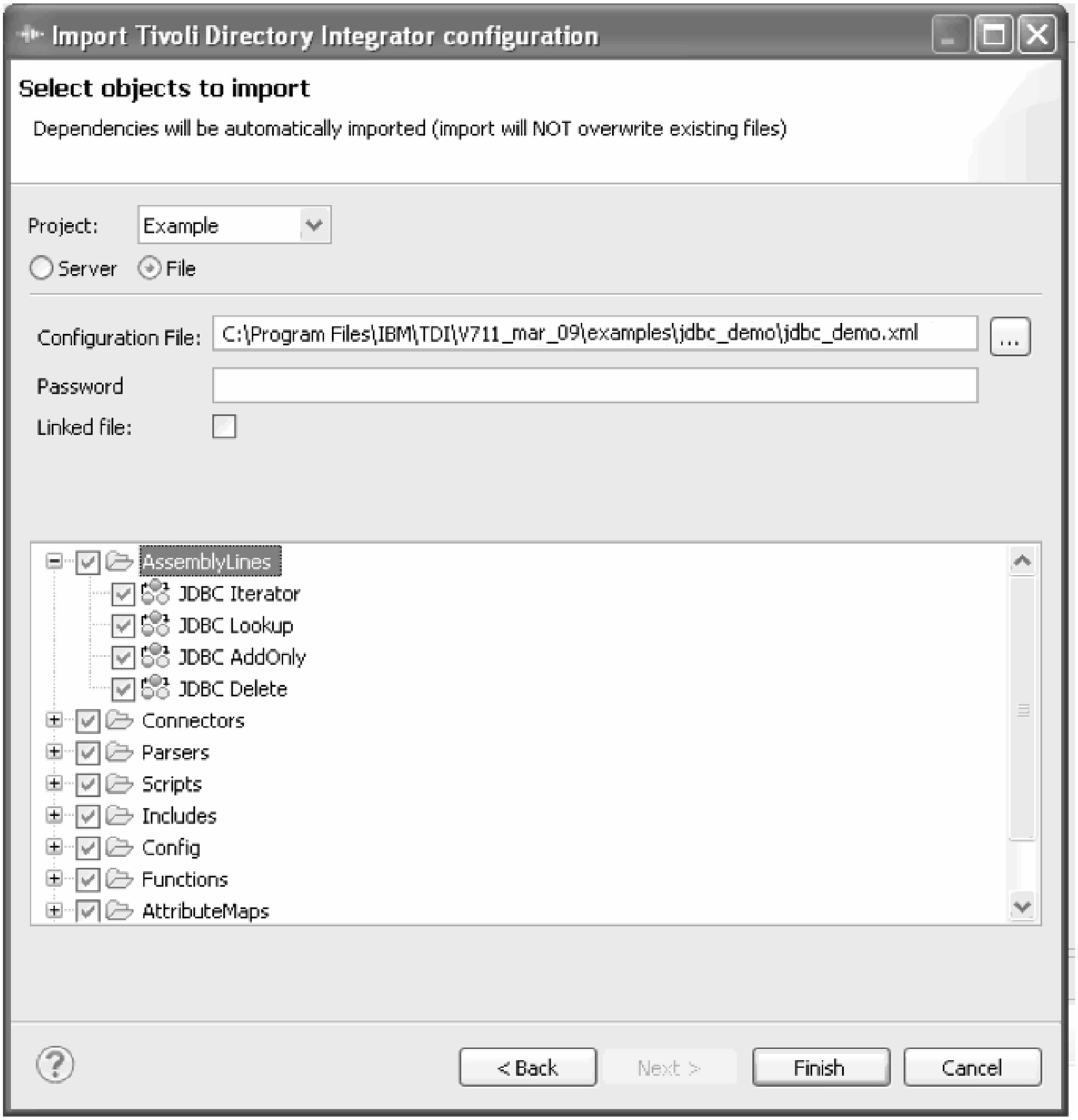
There are a number of Wizards (graphically-assisted stepped procedures) in the TDI Config Editor. These are:

1. “Import Configuration Wizard”
2. “New Component Wizard” on page 145
3. “Connector Configuration form characteristics” on page 150

Import Configuration Wizard

You can import configuration files of previous versions using the Import Configuration wizard.

In this wizard you choose the target project and which components you want to import.



Import Tivoli Directory Integrator configuration

Select objects to import

Dependencies will be automatically imported (import will NOT overwrite existing files)

Project: Example

○ Server ● File

Configuration File: C:\Program Files\IBM\TDI\V711_mar_09\examples\jdbc_demo\jdbc_demo.xml

Password

Linked file: □

- ☑ AssemblyLines
  - ☑ JDBC Iterator
  - ☑ JDBC Lookup
  - ☑ JDBC AddOnly
  - ☑ JDBC Delete
- ☑ Connectors
- ☑ Parsers
- ☑ Scripts
- ☑ Includes
- ☑ Config
- ☑ Functions
- ☑ AttributeMaps

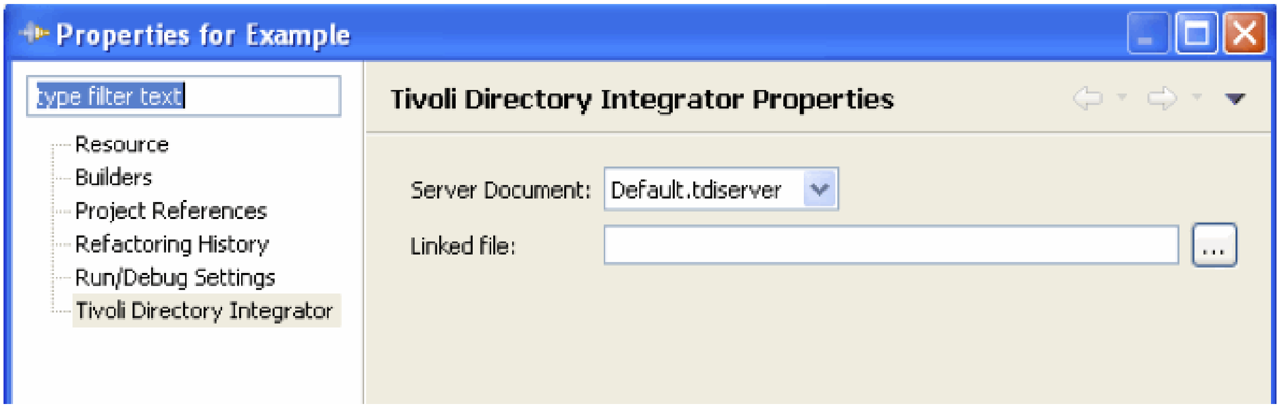
Figure 68. Import Configuration wizard

By default all components are selected for import. You can however check only those you are interested in. If you check a single AssemblyLine and that AssemblyLine uses connectors in the configuration file, those connectors will automatically be imported as well.

The **Project** input field is the target project into which the configuration is imported. Select the blank option to create a new project.

The **Configuration File** input field is the configuration file you are going to import. If the configuration is password protected you enter the password in the **Password** input field.

When the **Linked file** field is checked, any changes made to the imported project are written back to the file the project was imported from. You can change this setting in the project properties by clearing or changing the file name for the **Linked file** field, as illustrated below:

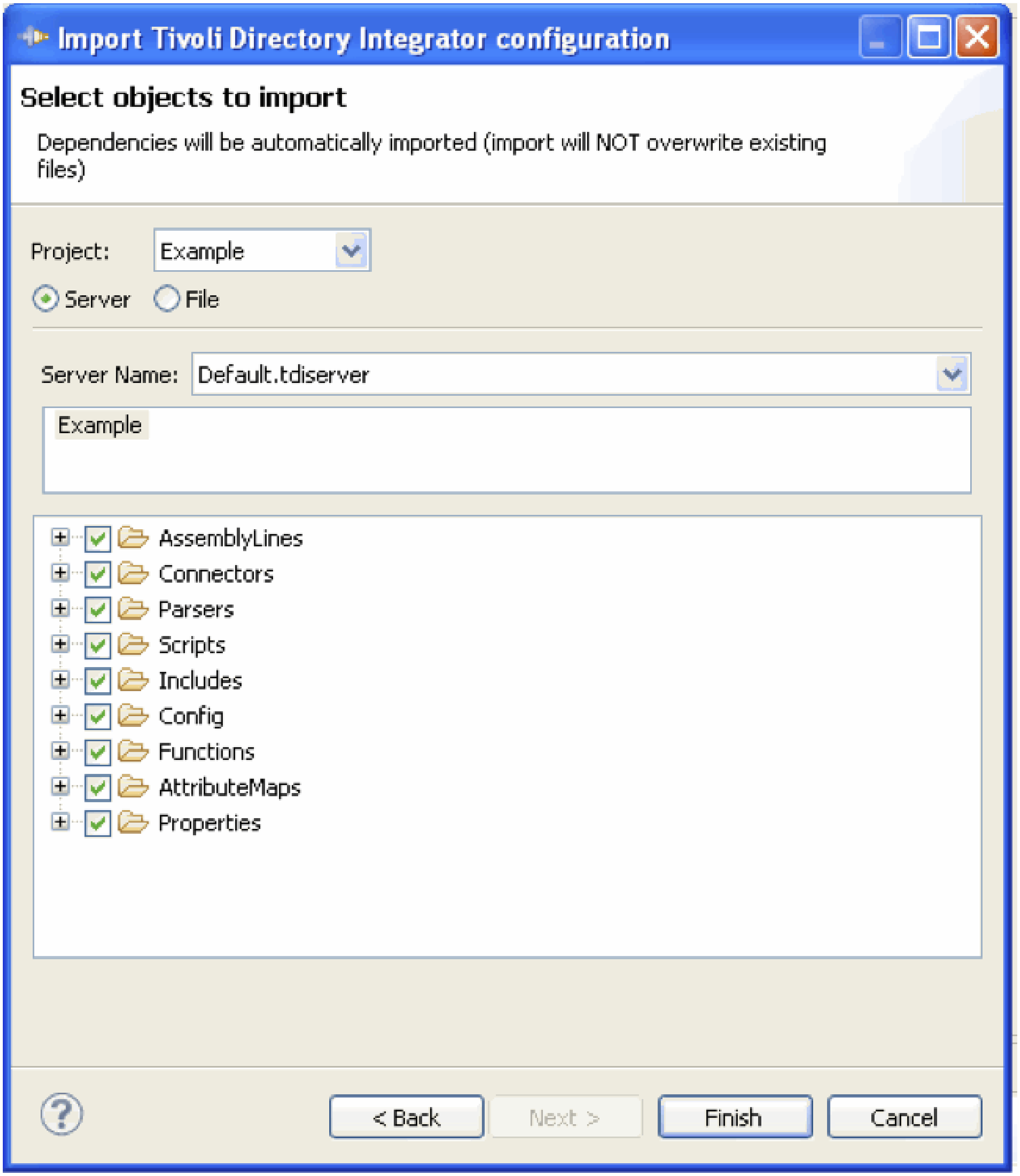


Server Document: Default.tdiserver

Linked file:

Figure 69. Linked file field

You can also import configurations from servers. Switch to the server view by checking the **Server** radio button.



Import Tivoli Directory Integrator configuration

Select objects to import

Dependencies will be automatically imported (import will NOT overwrite existing files)

Project: Example

● Server ○ File

Server Name: Default.tdiserver

Example

- ☑ AssemblyLines
- ☑ Connectors
- ☑ Parsers
- ☑ Scripts
- ☑ Includes
- ☑ Config
- ☑ Functions
- ☑ AttributeMaps
- ☑ Properties

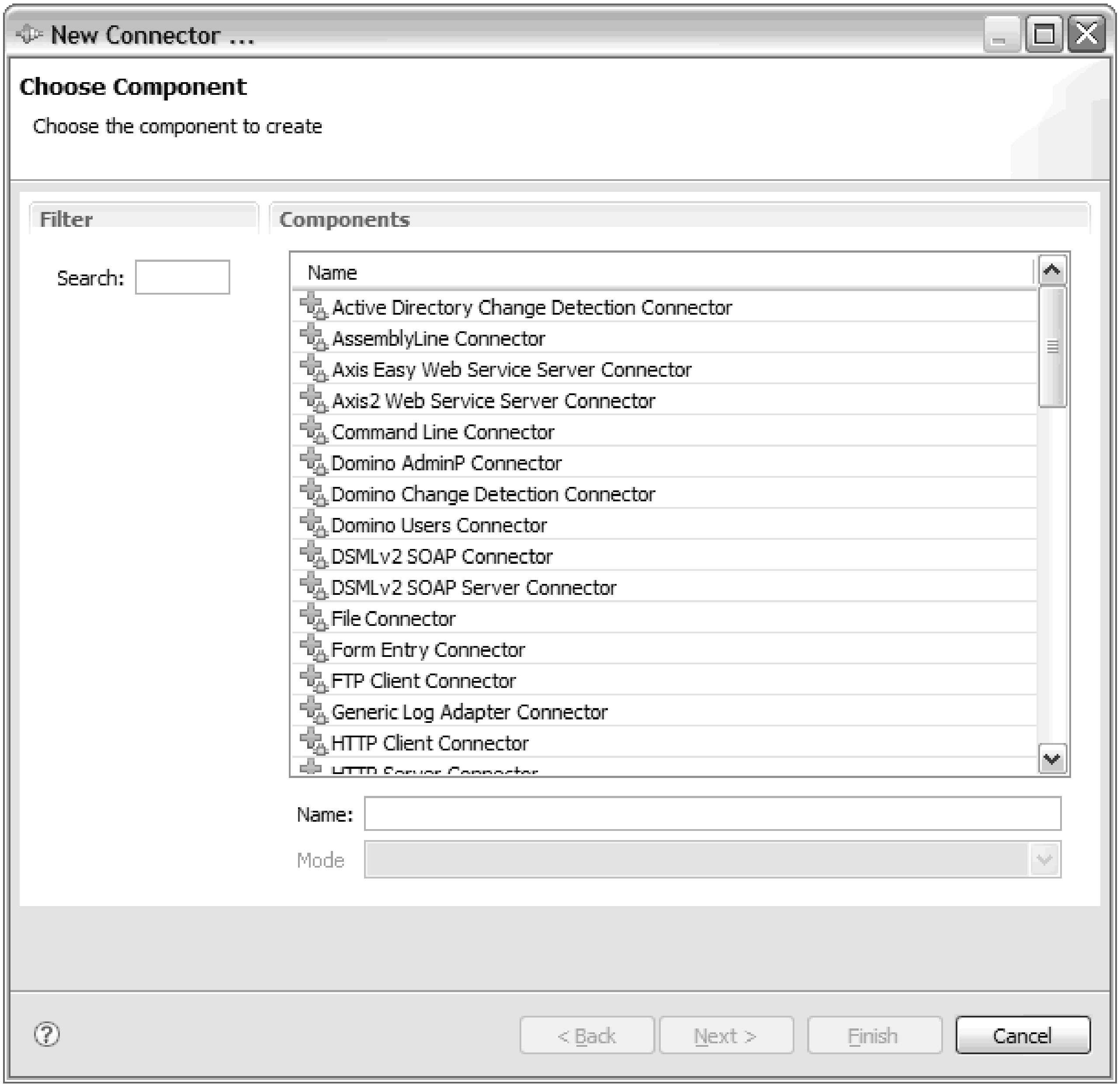
Figure 70. "Import from server" wizard

You start this type of import by selecting the server from the list of servers in the **Server name** field. Once you have selected a server, the list of configurations on that server is shown in the list below the server name. Since this is a network operation you may see some of the controls disabled while the list is being refreshed. From the list of configurations you select the configuration you want to import. Selecting a configuration downloads that configuration from the server and populates the tree below with its contents so you can choose which components to include in the import.

New Component Wizard

This wizard is invoked when you use **Add Component** in an AssemblyLine or when you use **File** > **New...** from the main menu bar.

When you create a new component in the `Resources` folder you will have a slightly different wizard layout. For a new connector, for example, the wizard looks like this:



Choose Component

Choose the component to create

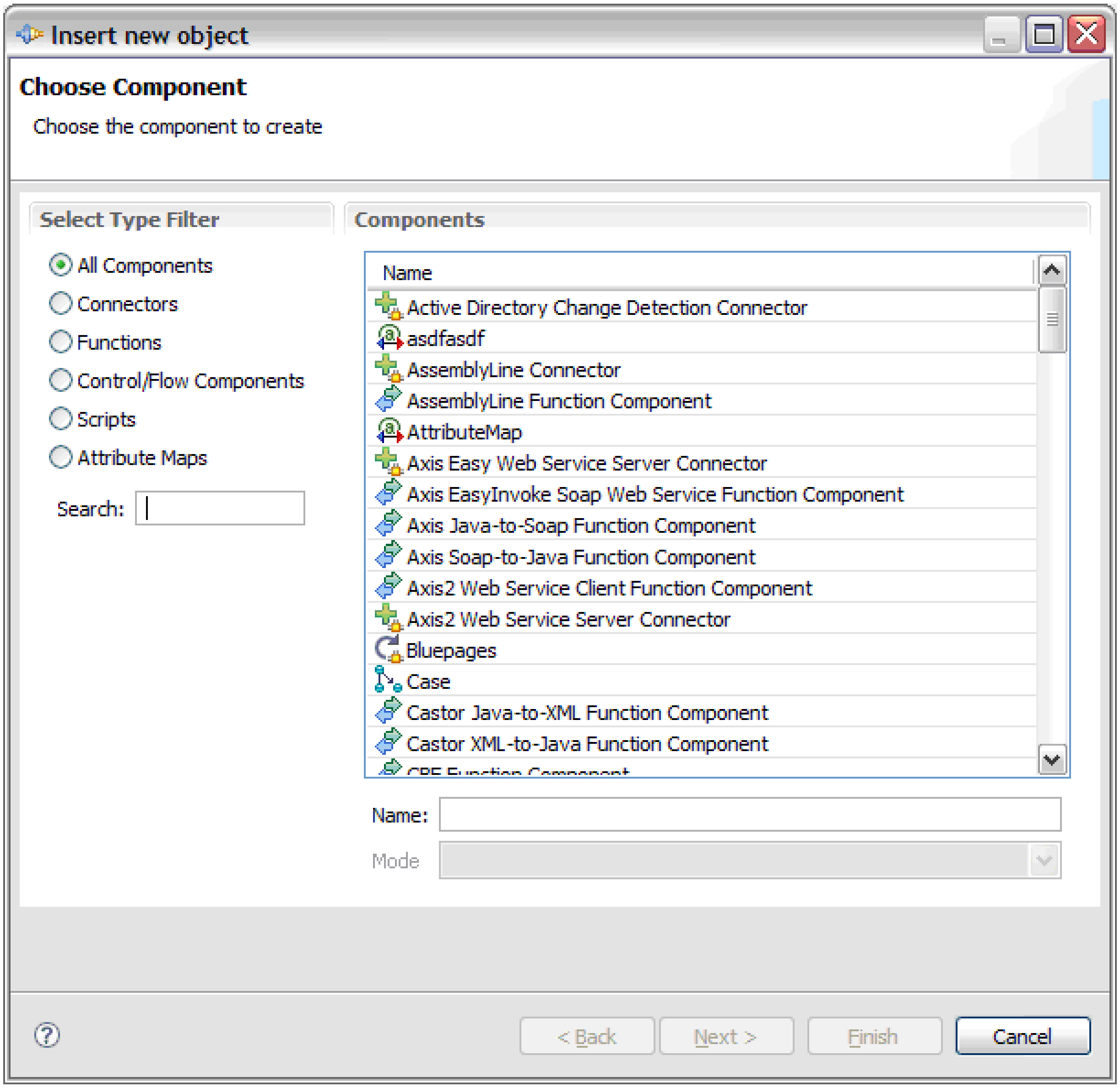
Name
Active Directory Change Detection Connector
AssemblyLine Connector
Axis Easy Web Service Server Connector
Axis2 Web Service Server Connector
Command Line Connector
Domino AdminP Connector
Domino Change Detection Connector
Domino Users Connector
DSMLv2 SOAP Connector
DSMLv2 SOAP Server Connector
File Connector
Form Entry Connector
FTP Client Connector
Generic Log Adapter Connector
HTTP Client Connector
HTTP Server Connector

Figure 71. New Connector in Resources folder wizard

The component name will be used as the suggested file name in the folder; it is advisable to change that name into something meaningful.

When you create a new component in an AssemblyLine, you will have many more options in this wizard.

The first page of the wizard is the component type selection page. In this page you choose the component type you want to add or create. The left side contains a list of filters that will select relevant components based on the label in the list.



Insert new object

Choose Component

Choose the component to create

Select Type Filter

● All Components
○ Connectors
○ Functions
○ Control/Flow Components
○ Scripts
○ Attribute Maps

Search:

Components

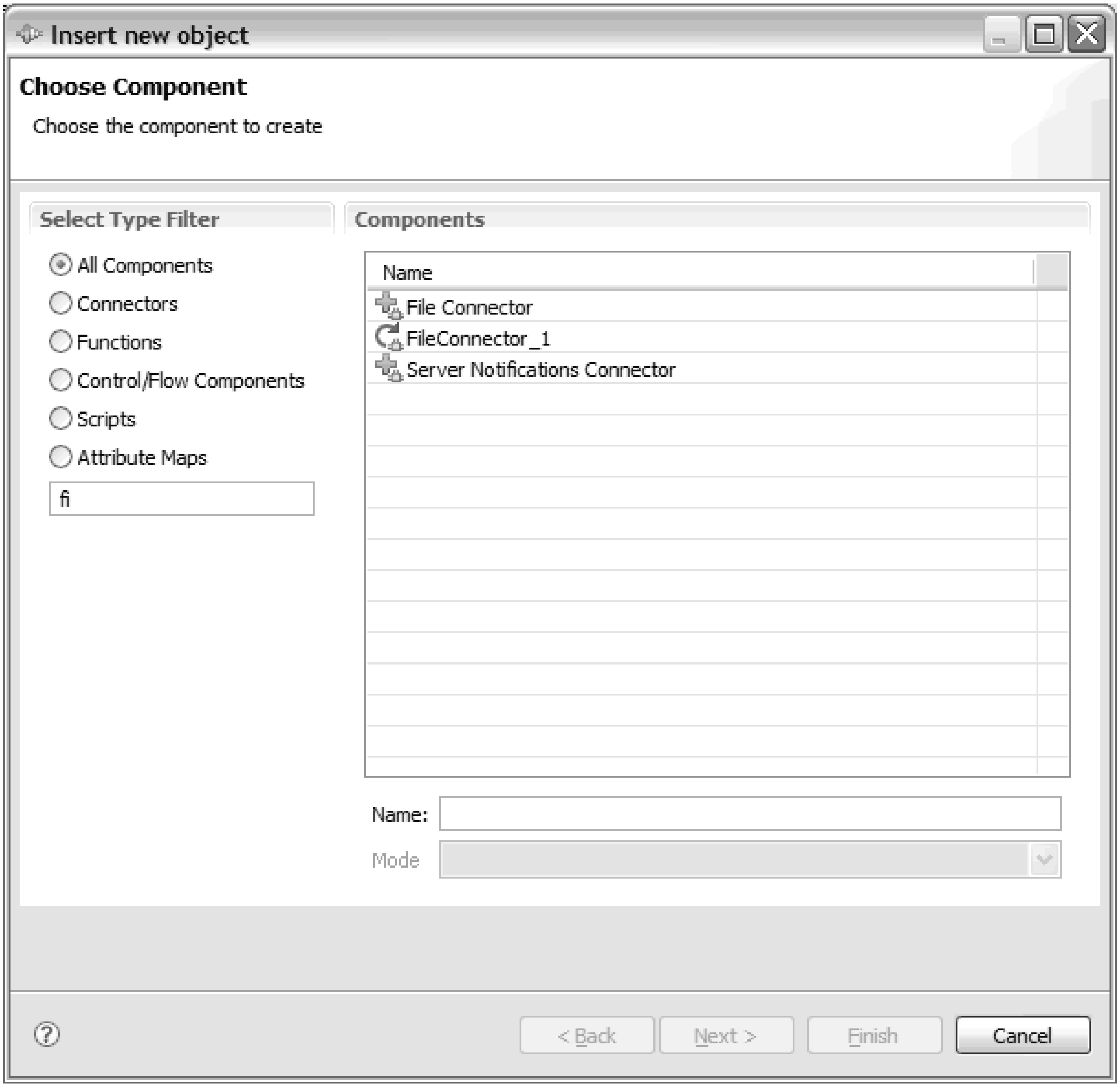
Name
Active Directory Change Detection Connector
asdfasdf
AssemblyLine Connector
AssemblyLine Function Component
AttributeMap
Axis Easy Web Service Server Connector
Axis EasyInvoke Soap Web Service Function Component
Axis Java-to-Soap Function Component
Axis Soap-to-Java Function Component
Axis2 Web Service Client Function Component
Axis2 Web Service Server Connector
Bluepages
Case
Castor Java-to-XML Function Component
Castor XML-to-Java Function Component
CBE Function Component

Name:

Mode

Figure 72. New Component wizard

You can filter the contents by typing in the Search field. As you type, the list will be matched against what you have typed in the search field (case insensitive contains match).



Choose Component

Choose the component to create

Select Type Filter

● All Components
○ Connectors
○ Functions
○ Control/Flow Components
○ Scripts
○ Attribute Maps

fi

Components

Name
File Connector
FileConnector_1
Server Notifications Connector

Name:

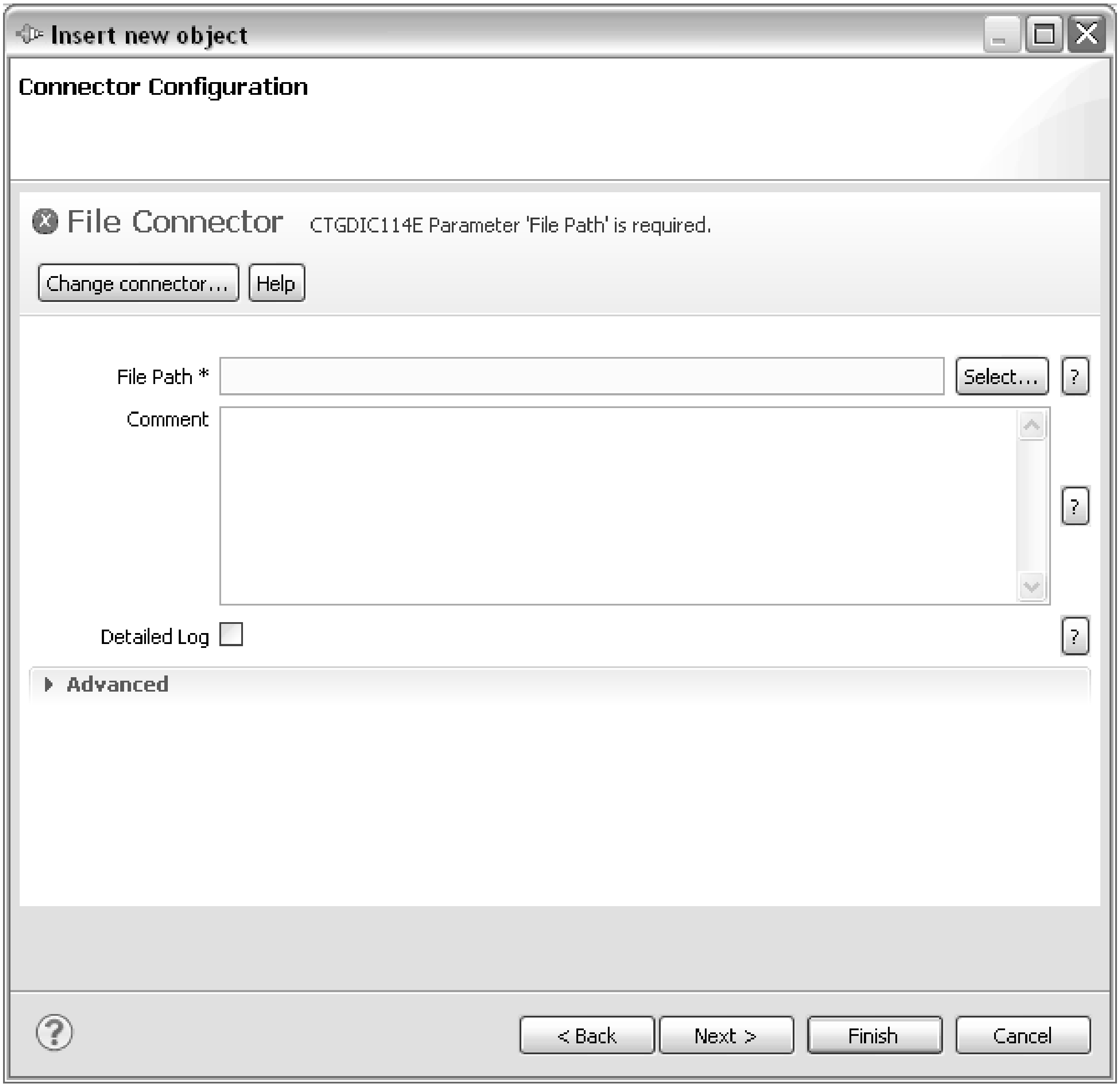
Mode

Figure 73. New Component wizard, with filtering

At this point you can choose to finish the wizard and the component is inserted into your AssemblyLine.

When you choose a connector you will be presented with a series of forms to properly define the connector. For all other types you can only finish the wizard and configure the component in the AssemblyLine.

After selecting the type you are presented with the Connector configuration panel.



Insert new object

Connector Configuration

File Connector CTGDIC114E Parameter 'File Path' is required.

File Path *

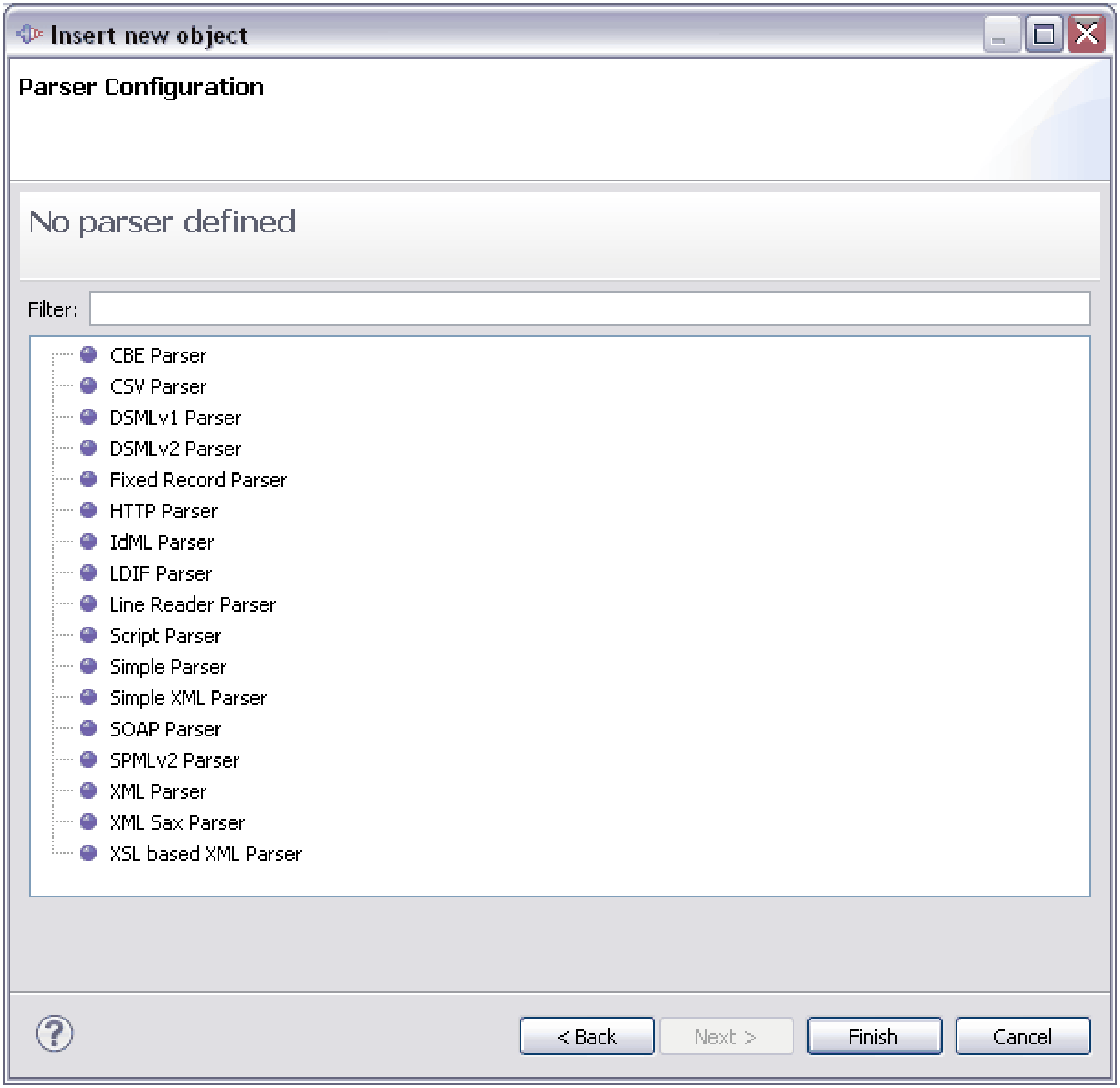
Comment

Detailed Log □

Figure 74. Connector Configuration panel

See also “Connector Configuration form characteristics” on page 150 for some specifics in completing this type of form.

The next step will show the Parser configuration if the connector can use one.



Insert new object

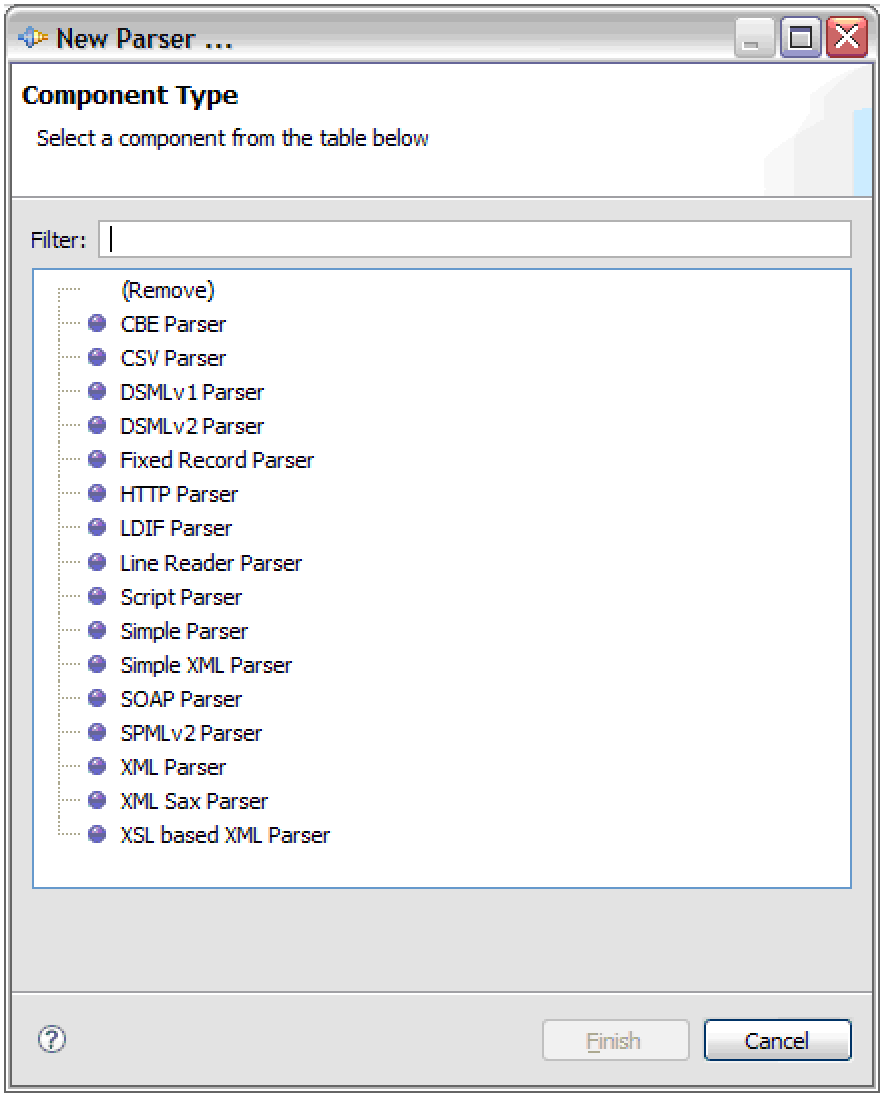
Parser Configuration

No parser defined

Filter:

Figure 75. Parser Configuration panel

Use the **Select Parser** button to choose the parser for the component:



New Parser ...

Component Type

Select a component from the table below

Filter:

- (Remove)
- CBE Parser
- CSV Parser
- DSMLv1 Parser
- DSMLv2 Parser
- Fixed Record Parser
- HTTP Parser
- LDIF Parser
- Line Reader Parser
- Script Parser
- Simple Parser
- Simple XML Parser
- SOAP Parser
- SPMLv2 Parser
- XML Parser
- XML Sax Parser
- XSL based XML Parser

Figure 76. Parser selection dialog

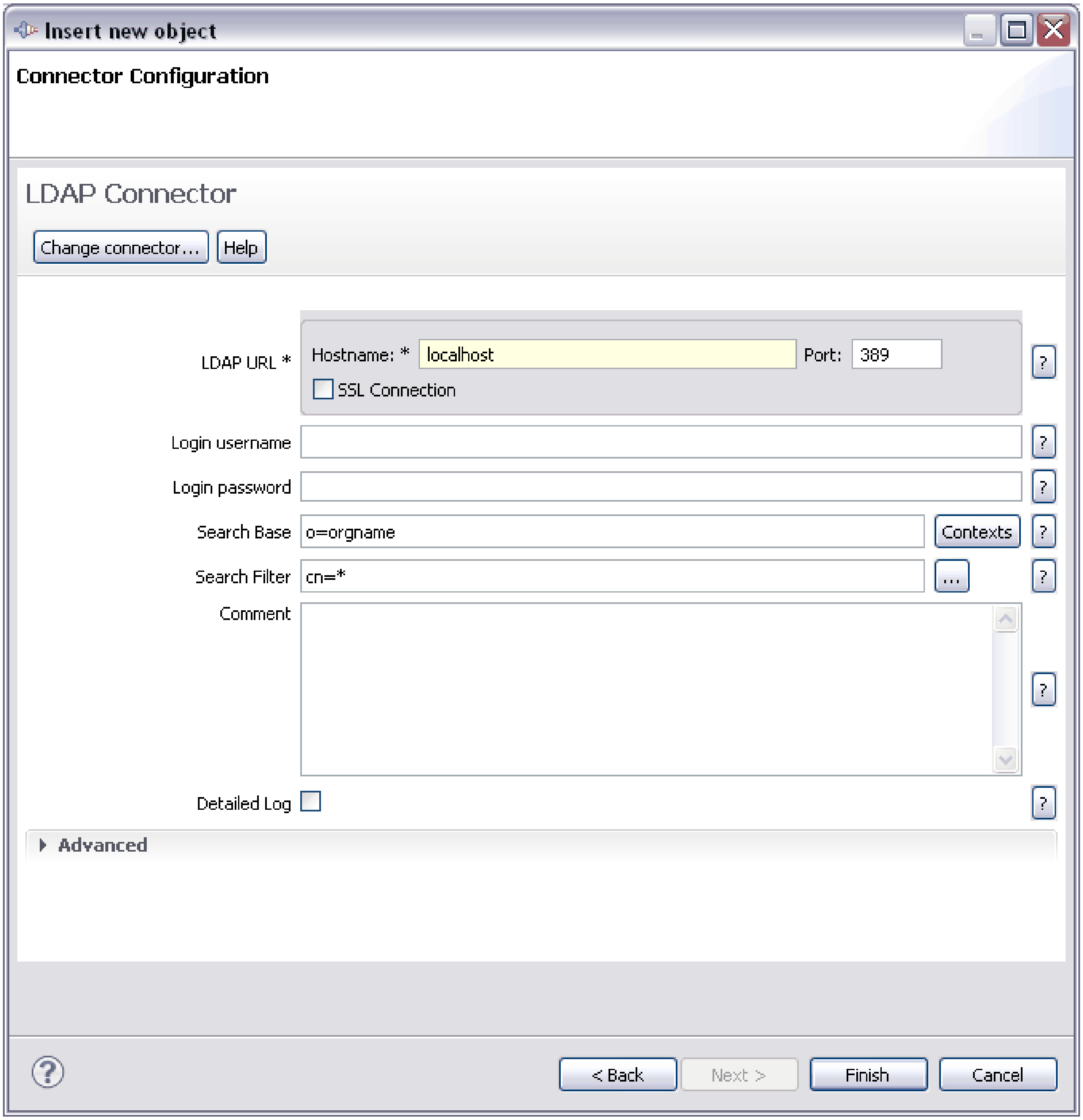
You can remove the current parser from the component by selecting the "(Remove)" option.

Connector Configuration form characteristics

Fields in a form can be grouped into optionally titled sections, and fields have a property that specifies whether the field is required.

In the Connector Configuration window, required fields are indicated by a * following the field name.

Values that are inherited will have a blue label.



Insert new object

Connector Configuration

LDAP Connector

LDAP URL * Hostname: * localhost Port: 389

□ SSL Connection

Login username

Login password

Search Base o=orgname

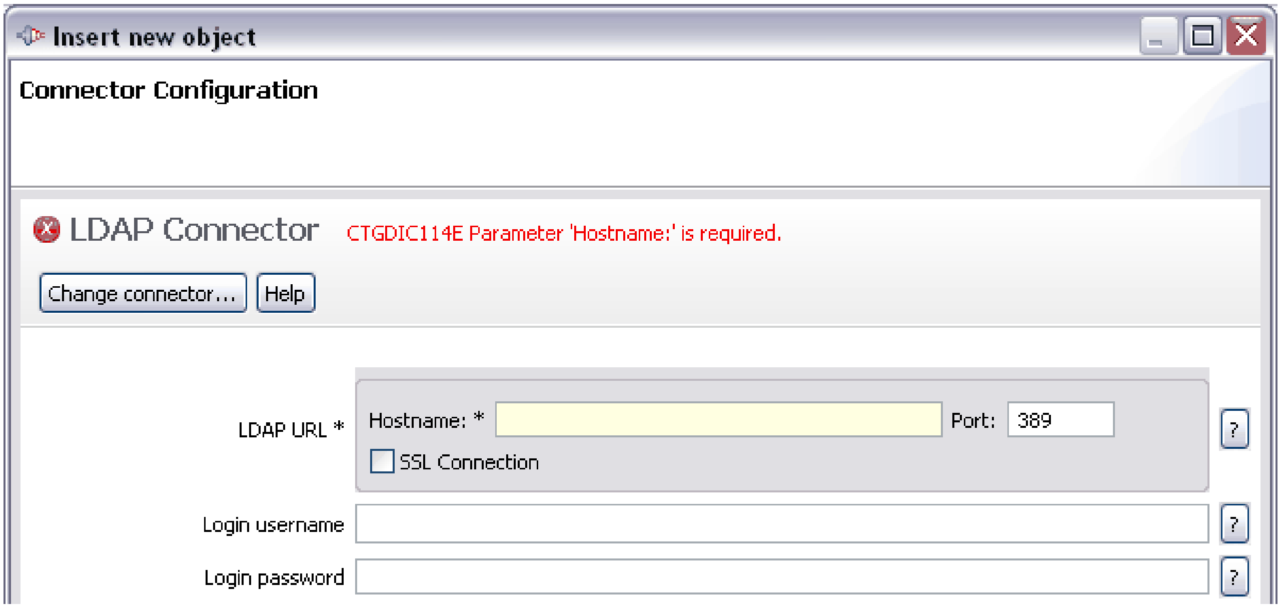
Search Filter cn=*

Comment

Detailed Log □

Advanced

In the sample above, the LDAP connector configuration shows two sections with a required field. When a required field has no value, the form will mark this in the title.



Insert new object

Connector Configuration

LDAP Connector CTGDIC114E Parameter 'Hostname:' is required.

LDAP URL * Hostname: * Port: 389

When you mouse over the red icon or text, you will see any error messages and which fields the errors apply to. This is useful when there is more than one error in the form.

Running and Debugging AssemblyLines

The Config Editor is equipped with a number of mechanisms that aid you in developing AssemblyLines, including facilities to test and debug the logic of them.

AssemblyLine Reports

AssemblyLine reports can be run from the context menu in the navigator.

Right-click an AssemblyLine and choose the **Create AssemblyLine Report** submenu. This menu contains all the report templates that are in the *TDI_Install_dir*/`XSLT/ConfigReport` directory.

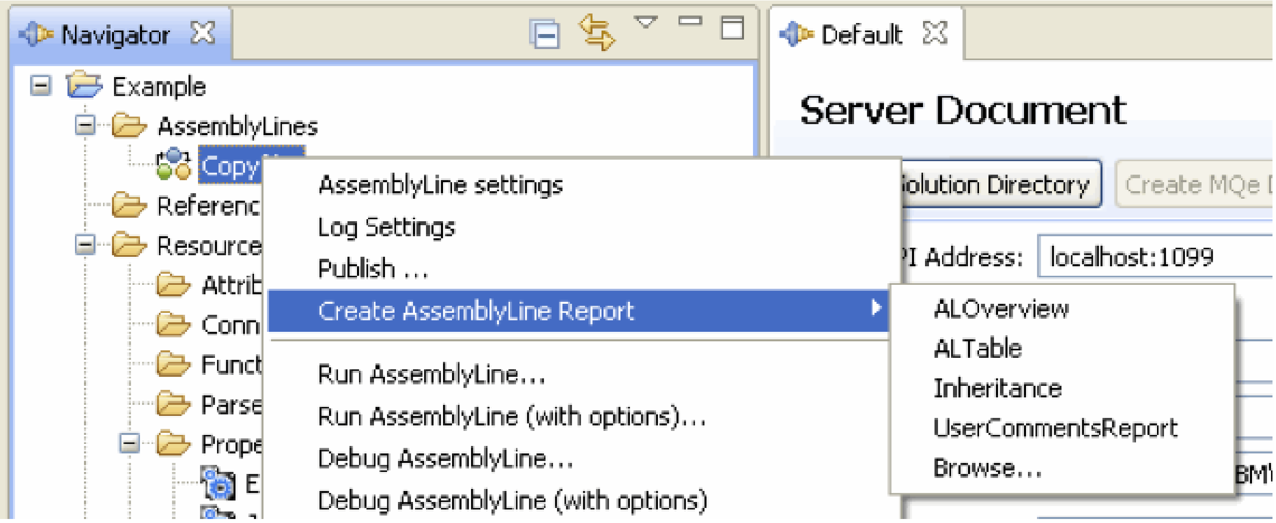


Figure 77. "Create AssemblyLine Report" command

Choose the **Browse...** option to browse the local file system for a report template.

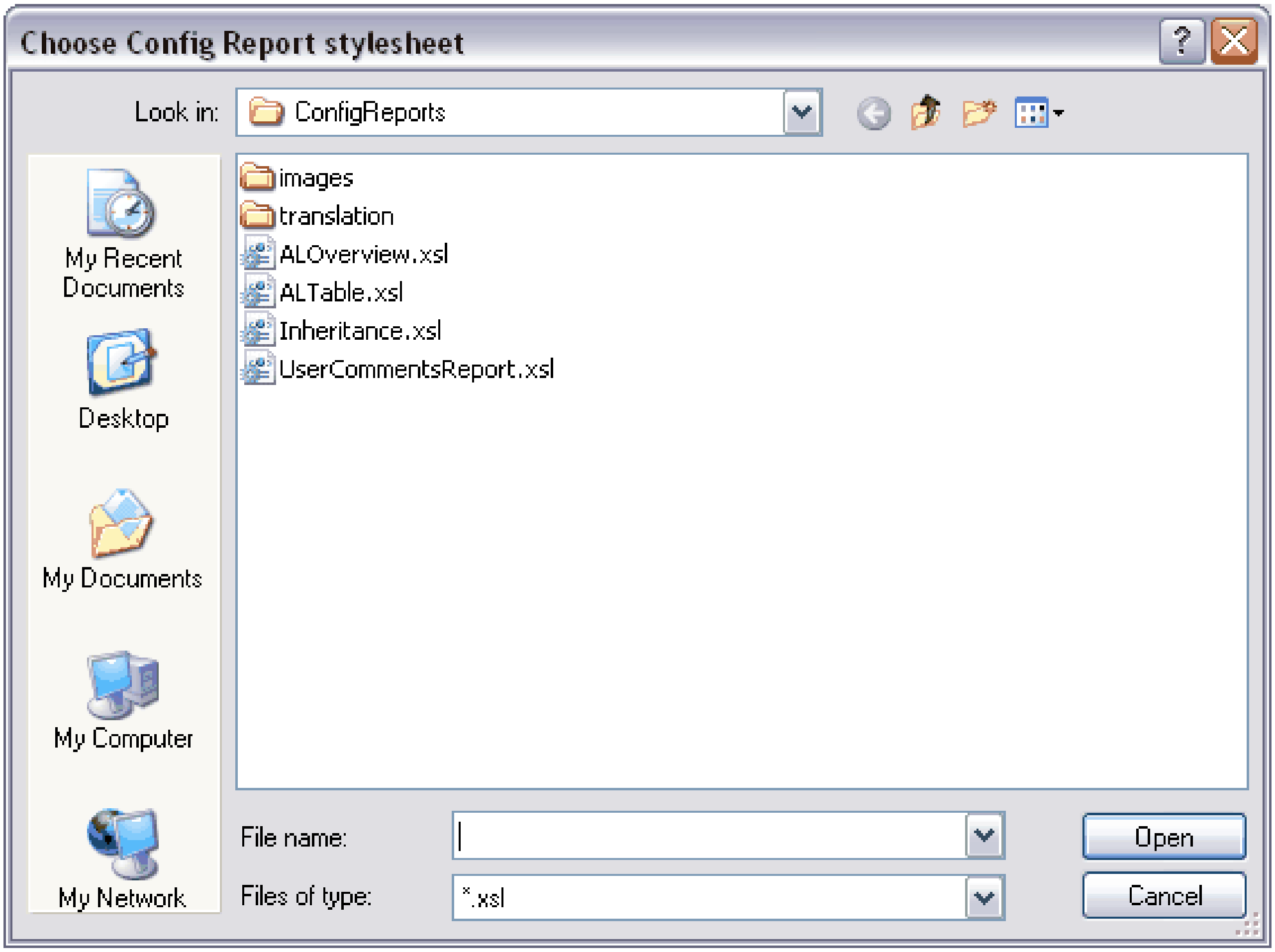


Figure 78. Choose Config Report stylesheet dialog

When you select one and click **Open**, the report is generated and placed in the `Reports` directory of your project as seen in the next picture. The editor associated with the *.html* file extension is opened to view the report, typically your default internet browser.

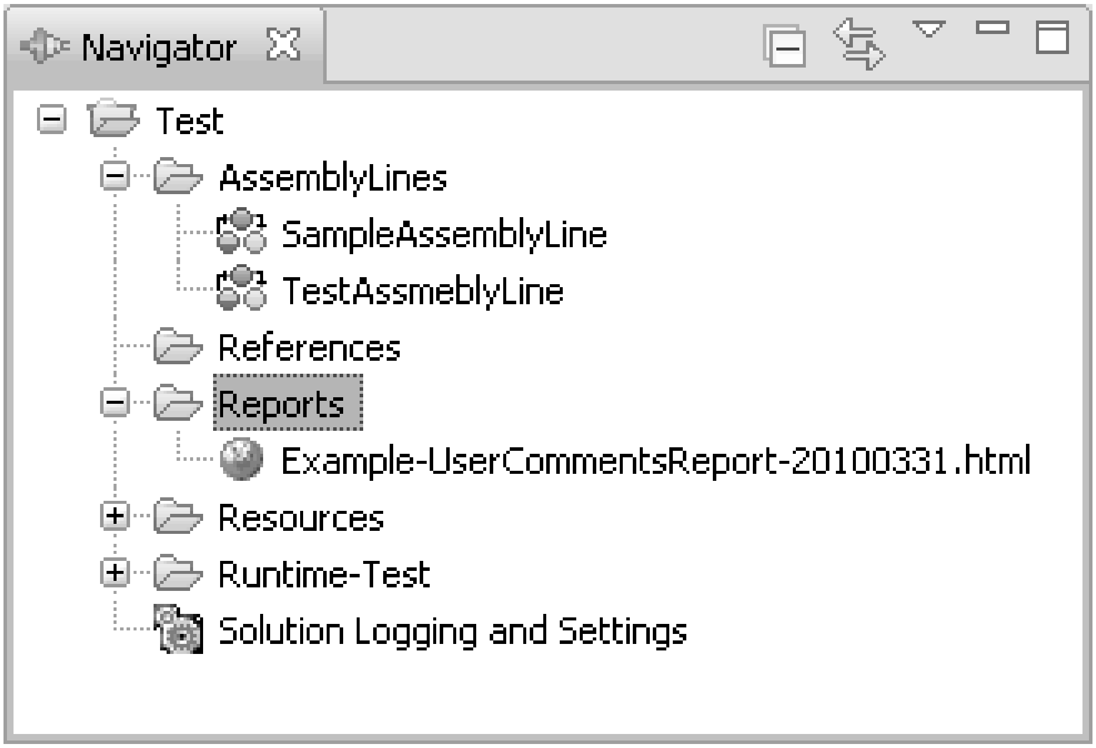
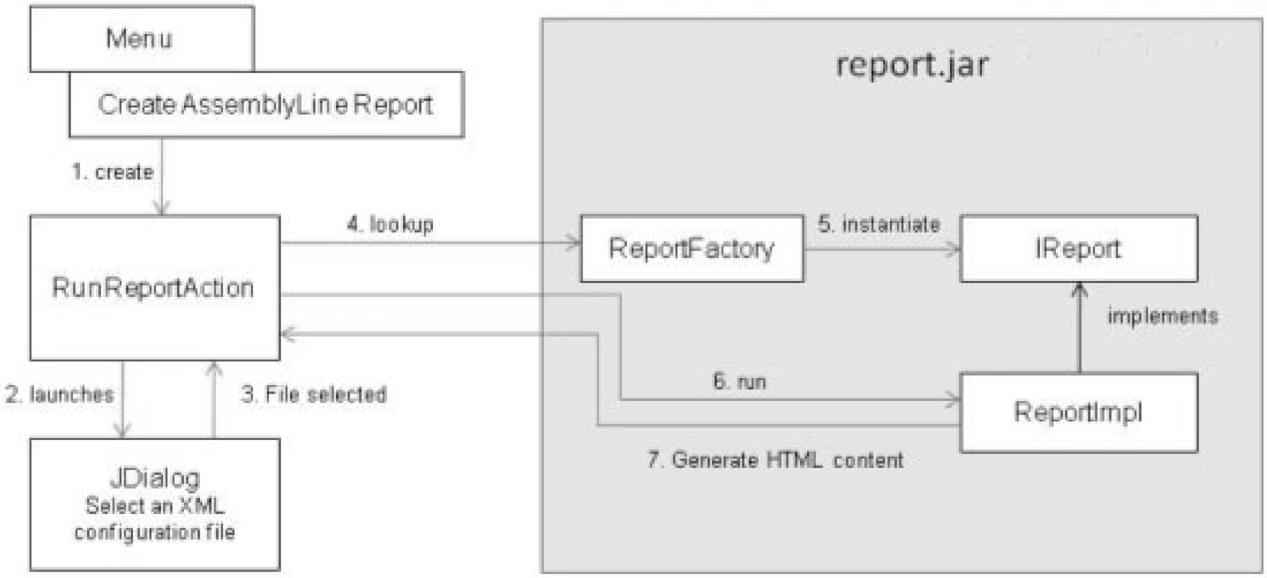


Figure 79. Reports folder in the Project hierarchy

The AssemblyLine report generates report file names based on the AssemblyLine with the current date inserted.

Overview of XML based AssemblyLine reports

You can generate a report for the selected configuration element based on a report style sheet or for the specified report configuration XML file. The following diagram depicts the architecture of XML based AssemblyLine reports.



Report configuration XML file format

The AssemblyLine report configuration XML file has the following format:

```
<tdiReport>
<reportClass)com.ibm.di.report.aloverview.AssemblyLineOverview</reportClass>
    <reportConfig>
        <report specific configuration>
    </reportConfig>
</ tdiReport>
```

The configuration XML file has the following elements:

- **reportClass** - specifies the Java class name for the report.
- **ReportFactory** - instantiates an instance of the report.
- **reportConfig** - contains report specific parameters.
-

Running the AssemblyLine

As you develop the AssemblyLine you can test it by either running to completion or by stepping through the components one by one.

There are two buttons to run the AssemblyLine. The first button (play icon, ▶) starts the AssemblyLine and shows the output in a console view. The second button (**Debugger**) runs the AssemblyLine with the debugger.

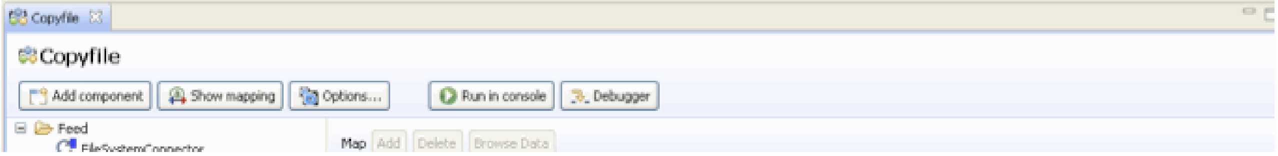
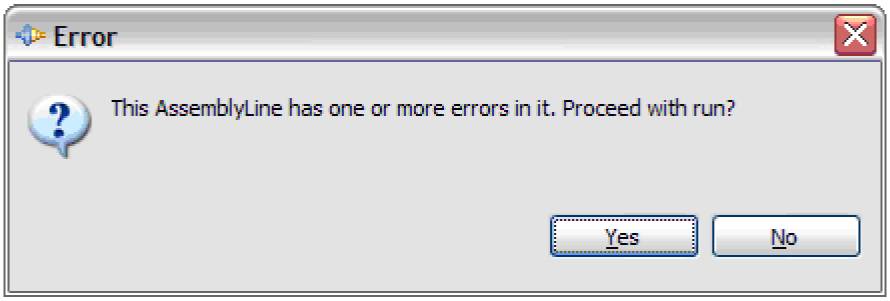


Figure 80. Three options to start an AssemblyLine

The process of starting an AssemblyLine goes through three steps.

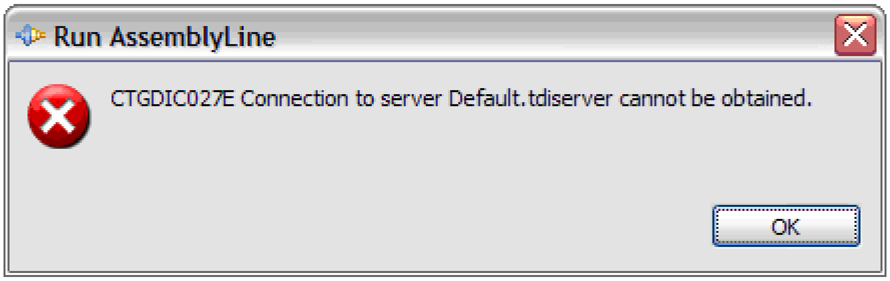
If the AssemblyLine contains errors (like missing output maps and so forth) you will be prompted to confirm running the AssemblyLine:



This AssemblyLine has one or more errors in it. Proceed with run?

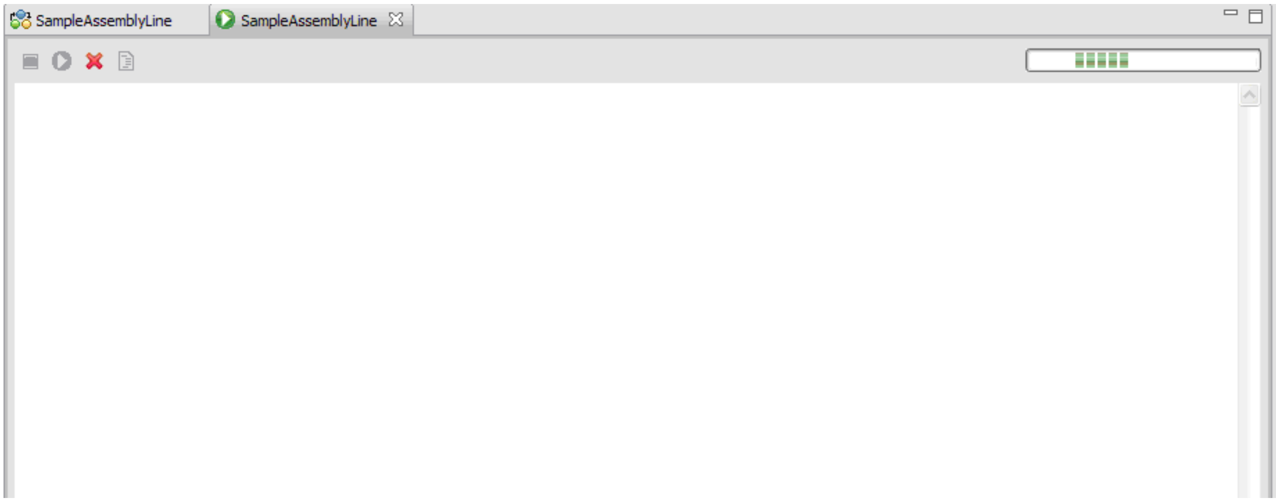
If you get this dialog you should check the Problems view to see which errors are potentially going to break the AssemblyLine. Often though, during development you are aware of these and still want to run the AssemblyLine, in that case hit Enter or click the **Yes** button to run the AssemblyLine.

The next check is whether the TDI server is available. If the server is unreachable you will see this message:



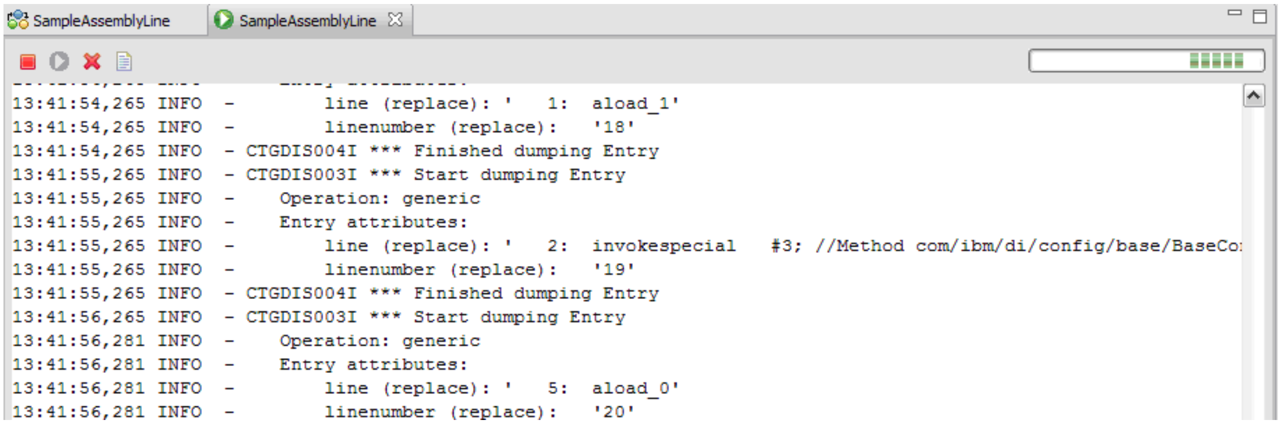
CTGDIC027E Connection to server Default.tdiserver cannot be obtained.

When running an AssemblyLine from the CE, the first step is when the CE transfers the runtime configuration to the server and waits for the AssemblyLine to be started. In this step you will see a progress bar in the upper right part of the window. The toolbar button to stop the AssemblyLine is also grayed out as it hasn't started yet.



The second step is when the AssemblyLine is running. The progress bar will be spinning and you should start seeing messages in the log window. You can now stop the AssemblyLine by hitting the stop button (far left) in the toolbar.

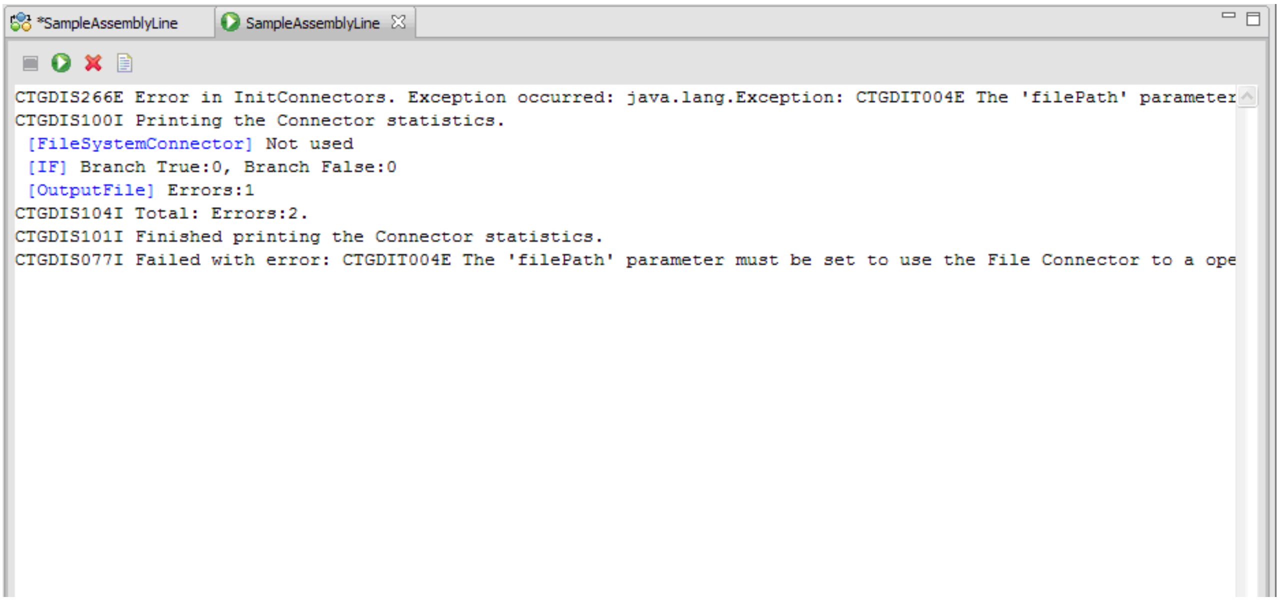
Note: The stop button will only work if the server gains control in the execution of the thread. If the thread is executing something outside of TDI, clicking the stop button may not have much effect.



```
13:41:54,265 INFO  -       line (replace): '   1:  aload_1'
13:41:54,265 INFO  -       linenumber (replace):   '18'
13:41:54,265 INFO  - CTGDIS004I *** Finished dumping Entry
13:41:55,265 INFO  - CTGDIS003I *** Start dumping Entry
13:41:55,265 INFO  -    Operation: generic
13:41:55,265 INFO  -    Entry attributes:
13:41:55,265 INFO  -       line (replace): '   2:  invokespecial   #3; //Method com/ibm/di/config/base/BaseCo
13:41:55,265 INFO  -       linenumber (replace):   '19'
13:41:55,265 INFO  - CTGDIS004I *** Finished dumping Entry
13:41:56,265 INFO  - CTGDIS003I *** Start dumping Entry
13:41:56,281 INFO  -    Operation: generic
13:41:56,281 INFO  -    Entry attributes:
13:41:56,281 INFO  -       line (replace): '   5:  aload_0'
13:41:56,281 INFO  -       linenumber (replace):   '20'
```

If you have multiple open AssemblyLine windows, you can tell which of the corresponding AssemblyLines are running because their names will be prefixed by a '*' (asterisk).

When the AssemblyLine has stopped (either normally or you hitting the stop button) the progress bar disappears and the toolbar item to rerun the AssemblyLine is enabled. The stop button is now disabled since the AssemblyLine is no longer running.



```
CTGDIS266E Error in InitConnectors. Exception occurred: java.lang.Exception: CTGDIT004E The 'filePath' parameter
CTGDIS100I Printing the Connector statistics.
 [FileSystemConnector] Not used
 [IF] Branch True:0, Branch False:0
 [OutputFile] Errors:1
CTGDIS104I Total: Errors:2.
CTGDIS101I Finished printing the Connector statistics.
CTGDIS077I Failed with error: CTGDIT004E The 'filePath' parameter must be set to use the File Connector to a ope
```

Figure 81. Console log window

At any one time you can clear the log window. The log window only shows the last few hundred lines of the AssemblyLine log, but every log message is written to a temporary log file so you can open the log file in a separate editor window using the View log button in the toolbar (far right).

Note: You can change the size of the log buffer underlying the log window from the default 300 lines to another value by going into **Window** > **Preferences** > **Tivoli Directory Integrator Preferences** > **Maximum number of lines for Run AssemblyLine window**.

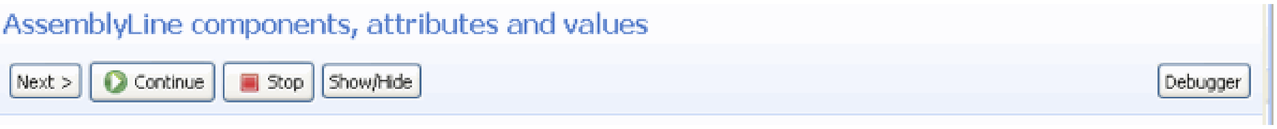
Once the AssemblyLine has terminated you can rerun the AssemblyLine with the rerun button.

Notice the blue text in the log window. When you see this, you can hold CTRL down while clicking the word to take you to that part of the AssemblyLine.

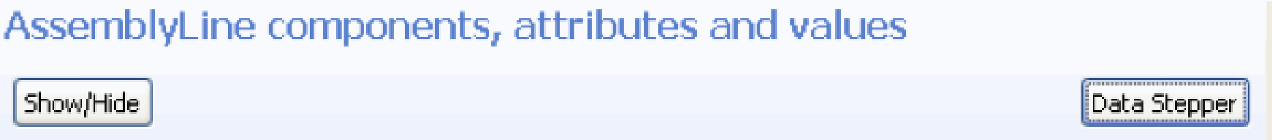
The stepper and debugger

The stepper and debugger are tools built in to the Config Editor that can help you develop your AssemblyLines interactively.

You can run the stepper in two modes. One is the normal advanced debugger (activated by the **Debugger** button, see “The Debugger” on page 160) where you have access to all parts of the AssemblyLine and the other is the stepper (activated by the **Data Stepper** button, see “The Data Stepper” on page 158) that provides a simpler view of the components and flow. You can toggle between the two by clicking the button in the column view:

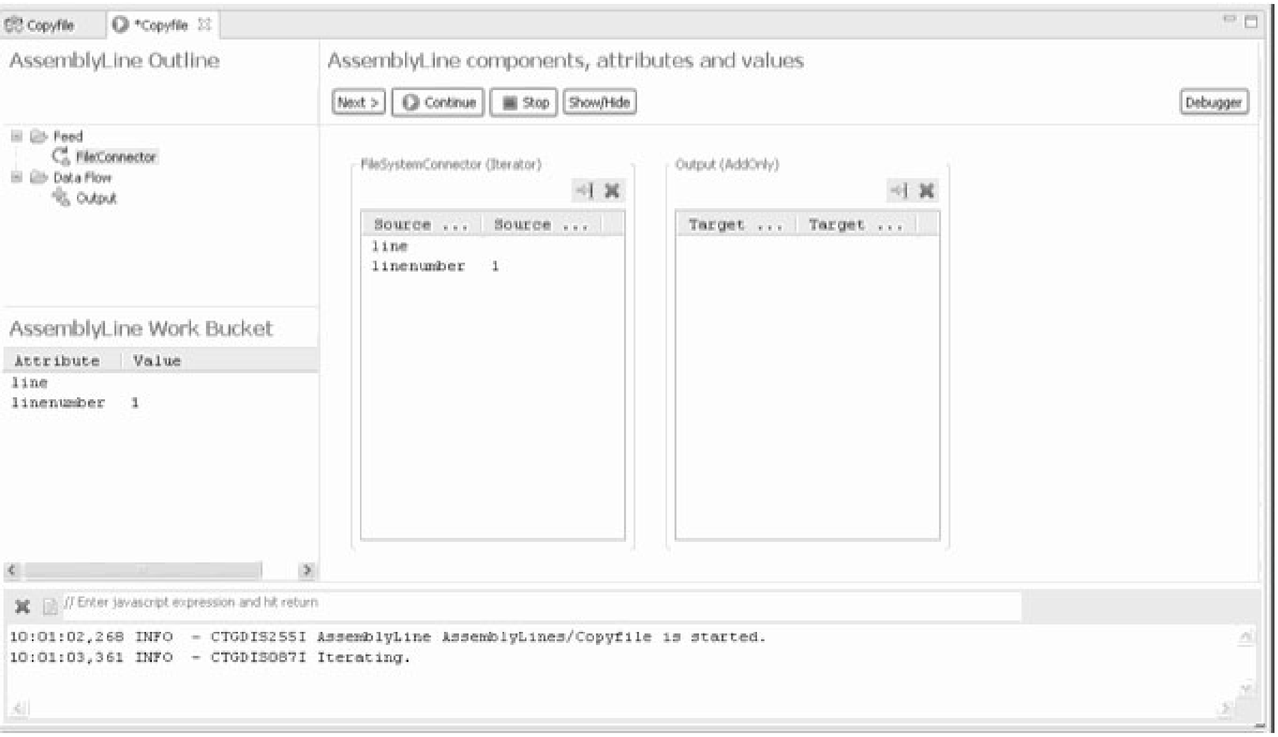


In the stepper view you can switch to debugger view by clicking the **Debugger** button. Conversely, switching from debugger to the stepper you click the **Data Stepper** button.



The Data Stepper

The data stepper provides a column view of all connectors in the AssemblyLine. Stepping through the AssemblyLine will show the data read or written for each component. All data shown in these tables are always after a connector has completed its operation.



Source ...	Source ...
line	
linenumber	1

Target ...	Target ...

Attribute	Value
line	
linenumber	1

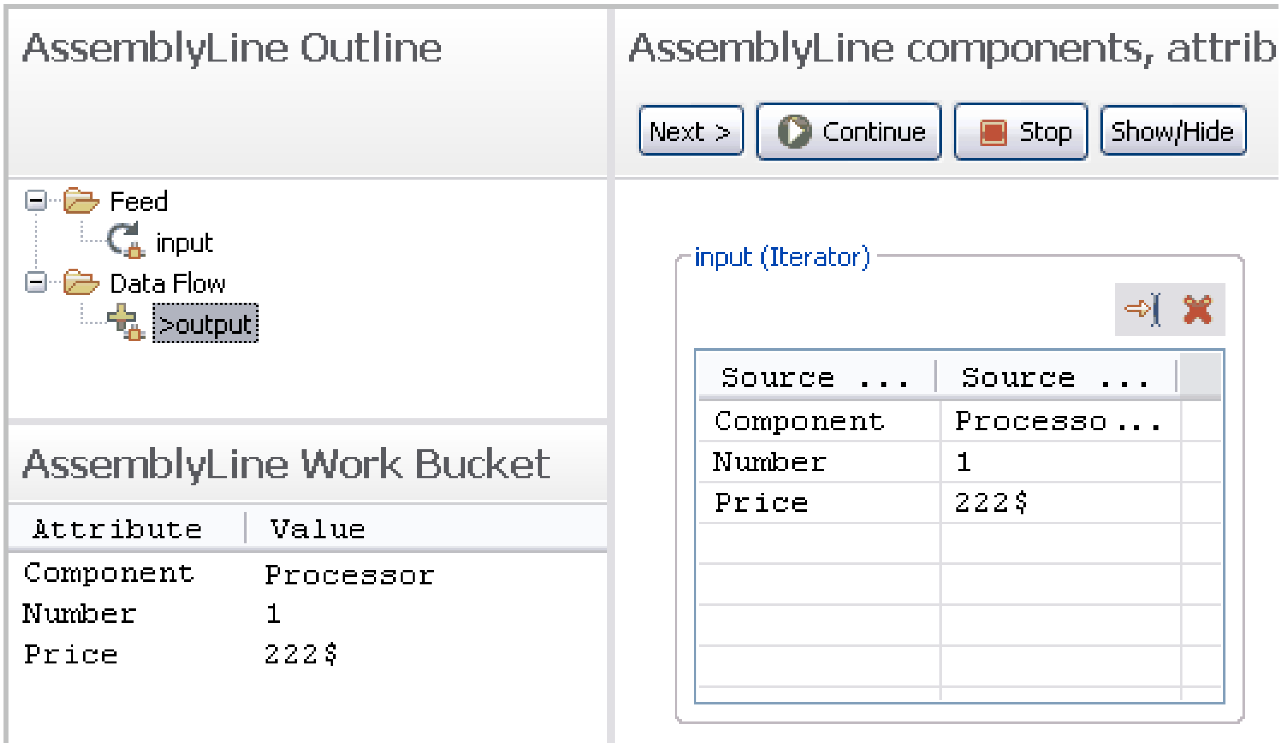
```
10:01:02,268 INFO  - CTGDIS255I AssemblyLine AssemblyLines/Copyfile is started.
10:01:03,361 INFO  - CTGDIS087I Iterating.
```

Figure 82. Data Stepper main window

In the right part, the data stepper shows components with an attribute map vertically. Each component can be removed from the view by clicking the close button or deselecting it from the **Show/Hide** dialog.

The button to the left of the close button (⇥) in each component is a shortcut for “Run to here”.

The **Next** and **Run** buttons are used to step one component at a time or to run the AssemblyLine to completion. The **Stop** button is used to pause a running AssemblyLine, or terminate a paused AssemblyLine. The left part of the data stepper shows the outline for the AssemblyLine and the work entry below. In the outline view you can choose to re-launch the AssemblyLine from the context menu. This will start the debugger in a new session and run until the selected component. This is a quick way of getting into the debugger from the data stepper.



Attribute	Value
Component	Processor
Number	1
Price	222$

Source ...	Source ...
Component	Processo...
Number	1
Price	222$

Figure 83. ***Show/Hide*** *button in the Data Stepper*

The **Show/Hide** Components dialog lets you choose which components to show in the view.

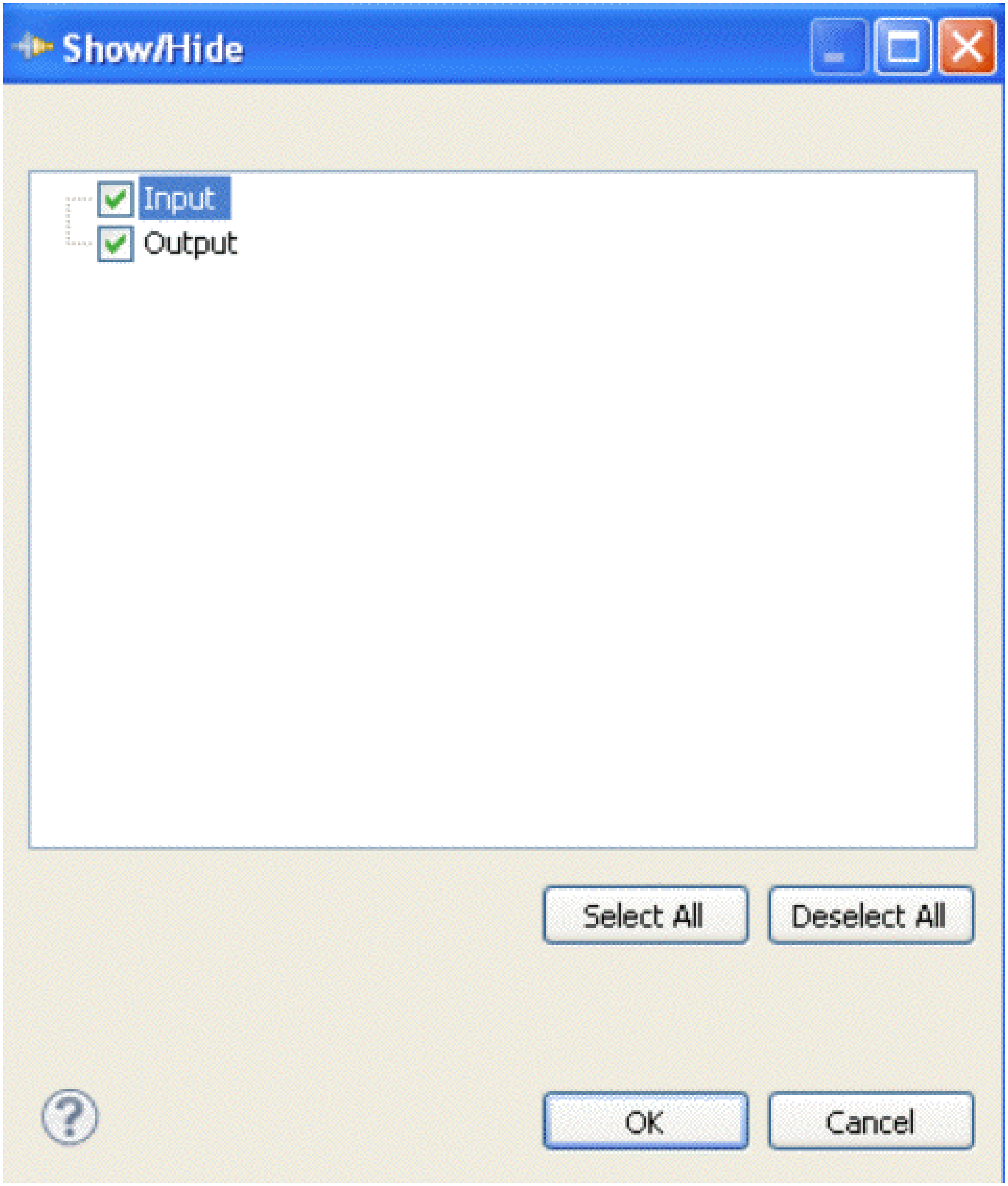
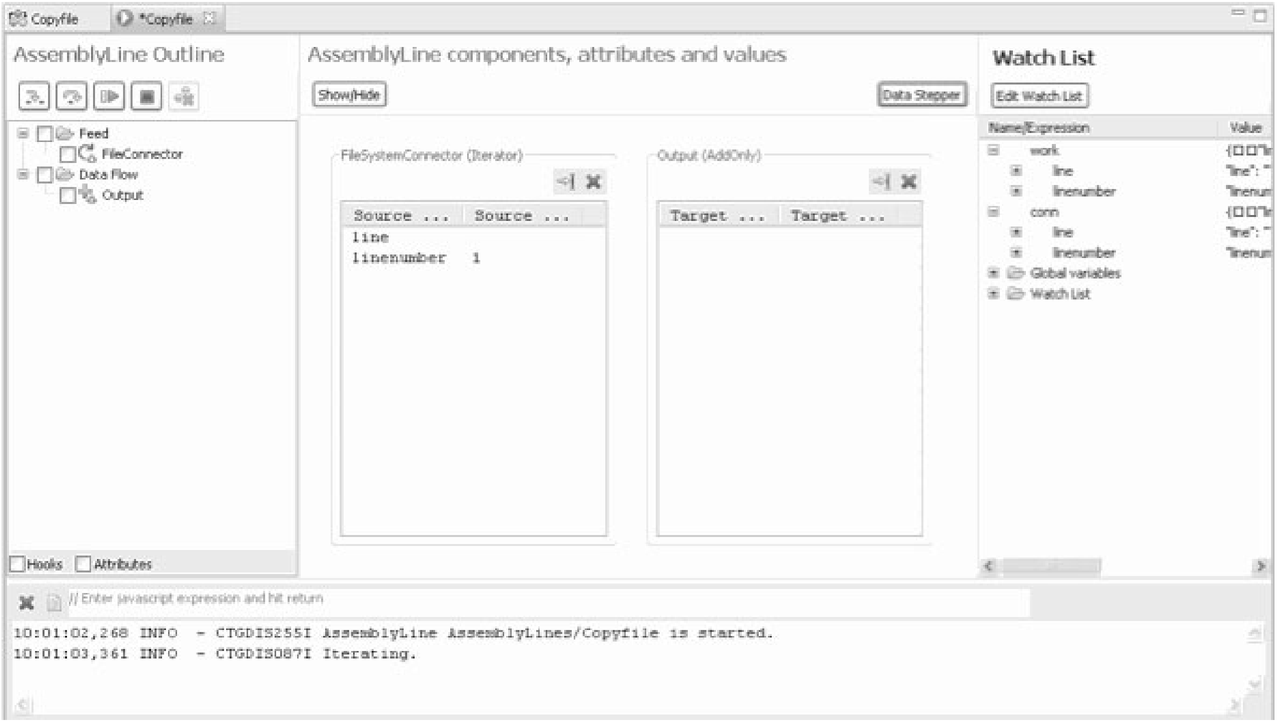


Figure 84. Show/Hide components dialog

The Debugger

When you choose the **Debugger** view you will be presented with a layout very similar to the stepper view, but in the debugger view you see much more of the AssemblyLine components and you also have the watch window where you can have custom expressions. The AssemblyLine components tree has a check box for each item you can check or uncheck to set or remove a breakpoint. Also, there are more command buttons that enable stepping into components and hooks.



Source ...	Source ...
line	
linenumber	1

Target ...	Target ...

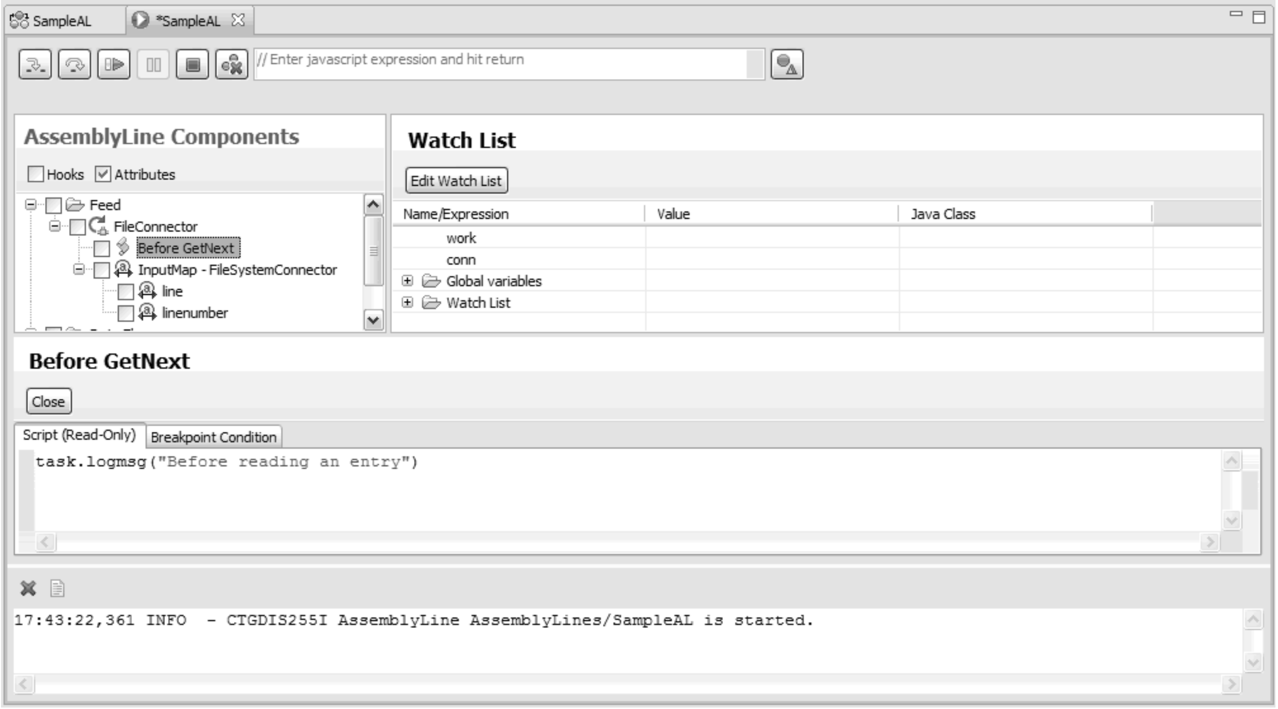
```
10:01:02,268 INFO  - CTGDIS255I AssemblyLine AssemblyLines/Copyfile is started.
10:01:03,361 INFO  - CTGDIS087I Iterating.
```

Figure 85. Debugger window

(In the debugger window it can be useful to maximize the editor with Ctrl-M to get a better overview.)

The tree to the left shows the AssemblyLine components and hooks. You can toggle the check box for each item to set a break point there. Double-click the item to view the script, or to enter the script for a conditional break.

The picture below shows the conditional break tab after double-clicking the **Before GetNext** hook. Also note that you can hide all hooks that are inactive. Showing all hooks lets you set a breakpoint regardless of whether the hook is active or not. The **Attributes** check box lets you hide attribute maps from the tree view.



☐ Hooks ☑ Attributes

Name/Expression	Value	Java Class
work		
conn		
Global variables		
Watch List		

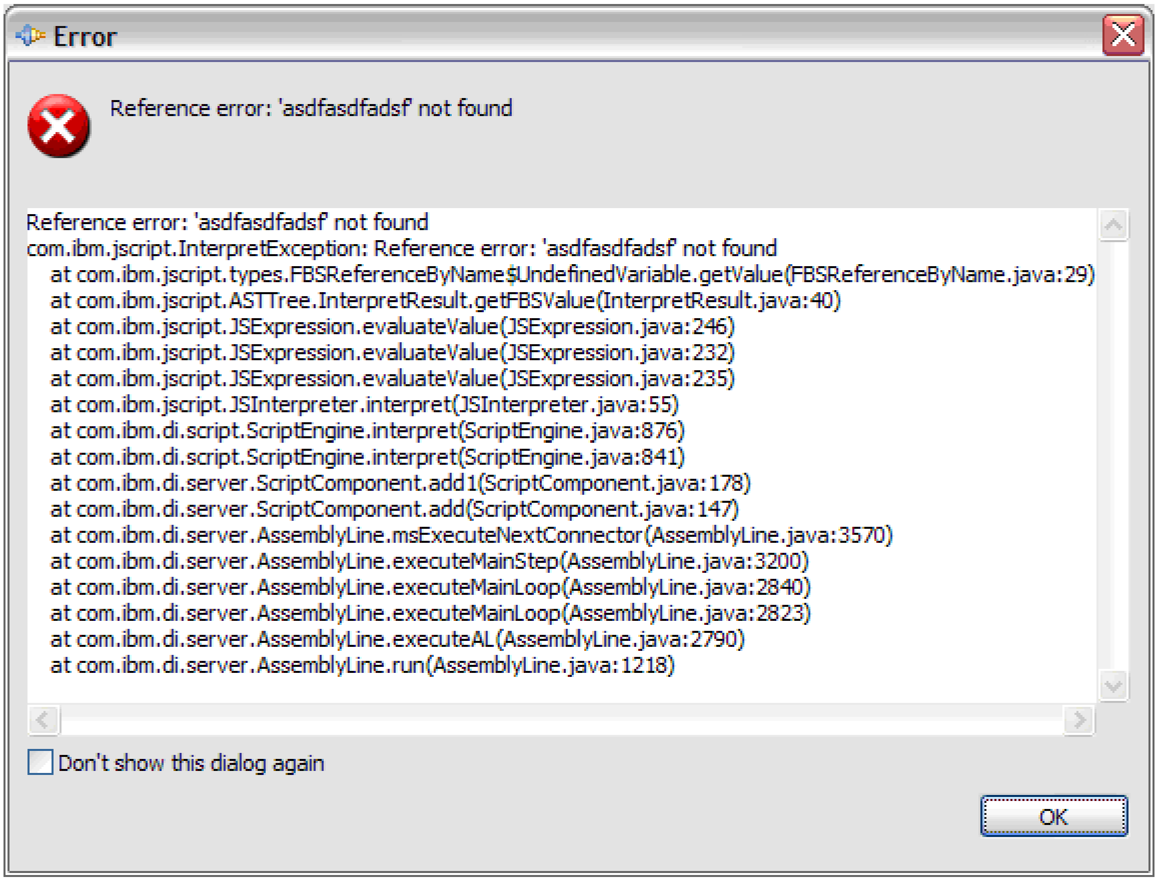
```
task.logmsg("Before reading an entry")
```

```
17:43:22,361 INFO  - CTGDIS255I AssemblyLine AssemblyLines/SampleAL is started.
```

Figure 86. Debugger at Before GetNext

The attribute map panel shows the components and their assignments and also the value assigned to the work attribute. The value is a snapshot from the last break in the AssemblyLine and the **Previous Value** is the snapshot value before that. When you step through the AssemblyLine, the values will reflect maps and scripts that affect the work entry.

When an error occurs the stepper will show the exception message and the stack trace in a separate dialog. The log file (on screen) does not include this stack trace. You can choose to not show this dialog by selecting the check box or in the Configuration Editor's preferences page.



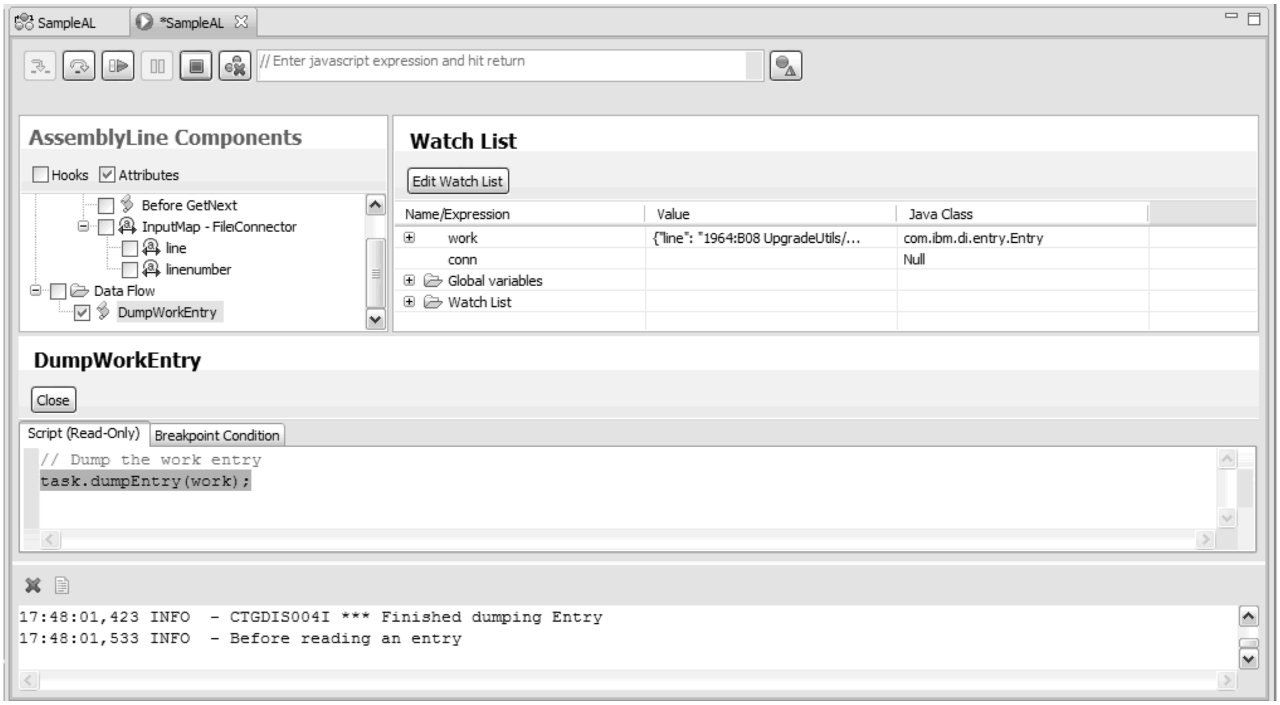
Reference error: 'asdfasdfadsf' not found

```
Reference error: 'asdfasdfadsf' not found
com.ibm.jscript.InterpretException: Reference error: 'asdfasdfadsf' not found
   at com.ibm.jscript.types.FBSReferenceByName$UndefinedVariable.getValue(FBSReferenceByName.java:29)
   at com.ibm.jscript.ASTTree.InterpretResult.getFBSValue(InterpretResult.java:40)
   at com.ibm.jscript.JSExpression.evaluateValue(JSExpression.java:246)
   at com.ibm.jscript.JSExpression.evaluateValue(JSExpression.java:232)
   at com.ibm.jscript.JSExpression.evaluateValue(JSExpression.java:235)
   at com.ibm.jscript.JSInterpreter.interpret(JSInterpreter.java:55)
   at com.ibm.di.script.ScriptEngine.interpret(ScriptEngine.java:876)
   at com.ibm.di.script.ScriptEngine.interpret(ScriptEngine.java:841)
   at com.ibm.di.server.ScriptComponent.add1(ScriptComponent.java:178)
   at com.ibm.di.server.ScriptComponent.add(ScriptComponent.java:147)
   at com.ibm.di.server.AssemblyLine.msExecuteNextConnector(AssemblyLine.java:3570)
   at com.ibm.di.server.AssemblyLine.executeMainStep(AssemblyLine.java:3200)
   at com.ibm.di.server.AssemblyLine.executeMainLoop(AssemblyLine.java:2840)
   at com.ibm.di.server.AssemblyLine.executeMainLoop(AssemblyLine.java:2823)
   at com.ibm.di.server.AssemblyLine.executeAL(AssemblyLine.java:2790)
   at com.ibm.di.server.AssemblyLine.run(AssemblyLine.java:1218)
```

Figure 87. Error dialog: Stack Trace

Stepping through scripts

When you reach a breakpoint that contains JavaScript you can step into the script and have it executed line by line. The script tab will show the script about to be executed; you can follow the flow by using the **Step Into** command (the leftmost of the two Stepper buttons).



Name/Expression	Value	Java Class
work	{"line": "1964:B08 UpgradeUtils/...	com.ibm.di.entry.Entry
conn		Null
Global variables		
Watch List		

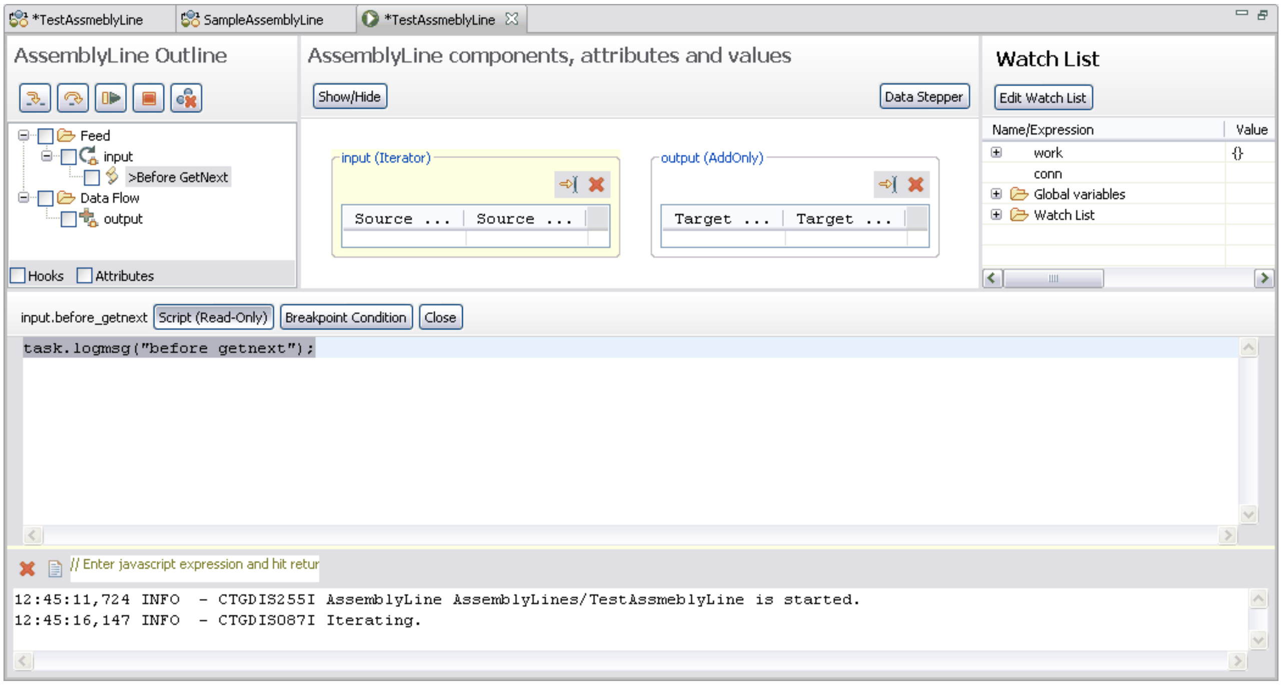
```
// Dump the work entry
task.dumpEntry(work);
```

```
17:48:01,423 INFO  - CTGDIS004I *** Finished dumping Entry
17:48:01,533 INFO  - Before reading an entry
```

Figure 88. Debugger window: stepping through a script line-by-line.

As you step through the script, each line will be highlighted before it is executed. You can also use the **Evaluate** function to display script engine variables while you step through a script.

When a script function is about to be executed you can use the **StepInto** button to step into the JavaScript function. If the function is defined outside the current context (for example, hook, attribute map script) the window is replaced.



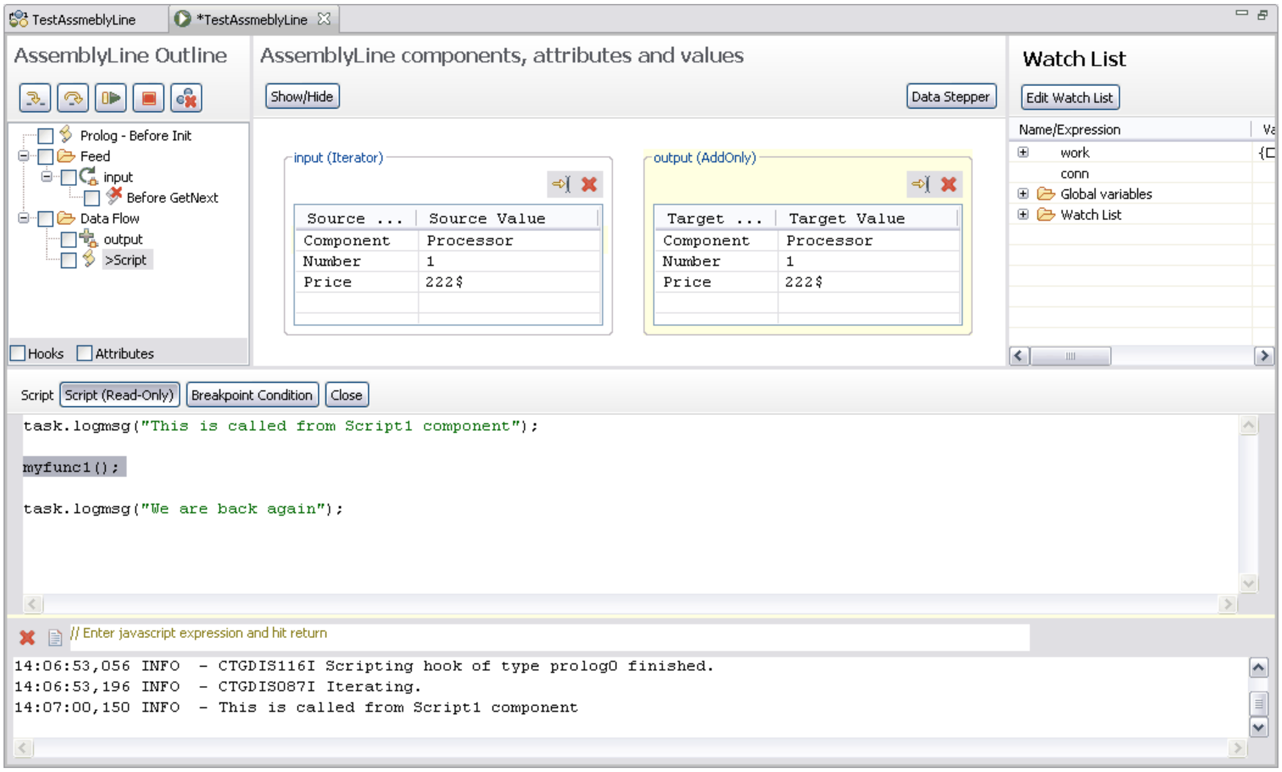
Name/Expression	Value
work	{}
conn	
Global variables	
Watch List	

```
task.logmsg("before getnext");
```

```
12:45:11,724 INFO  - CTGDIS255I AssemblyLine AssemblyLines/TestAssmeblyLine is started.
12:45:16,147 INFO  - CTGDIS087I Iterating.
```

Figure 89. StepInto function

In this AssemblyLine we have defined a function called `myfunc1()` in the before init hook. We will call it from the Script1 component to show how it appears to you:



Source ...	Source Value
Component	Processor
Number	1
Price	222$

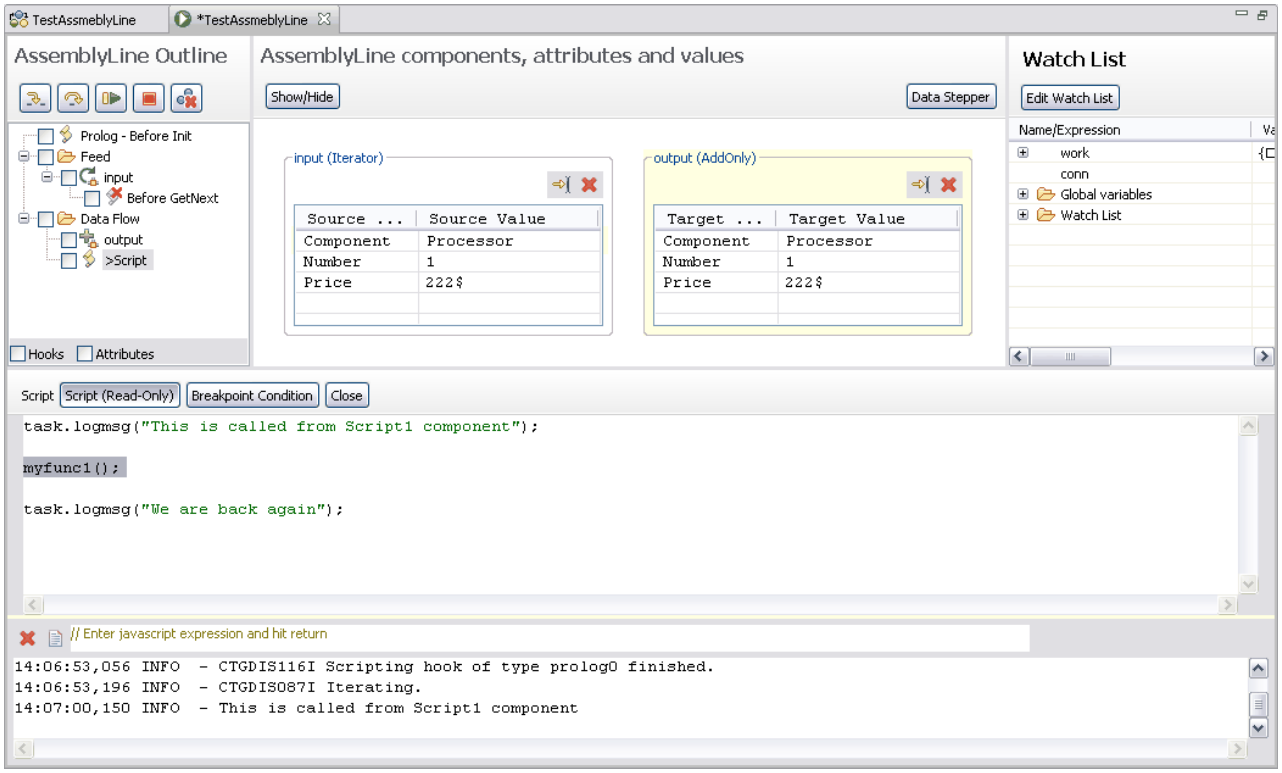
Target ...	Target Value
Component	Processor
Number	1
Price	222$

```
task.logmsg("This is called from Script1 component");

myfunc1();

task.logmsg("We are back again");
```

```
14:06:53,056 INFO  - CTGDIS116I Scripting hook of type prolog0 finished.
14:06:53,196 INFO  - CTGDIS087I Iterating.
14:07:00,150 INFO  - This is called from Script1 component
```

Figure 90. Stepped-into function

At this point we can press **StepOver** to continue with the next statement or press **StepInto** to follow the JavaScript function call:



Source ...	Source Value
Component	Processor
Number	1
Price	222$

Target ...	Target Value
Component	Processor
Number	1
Price	222$

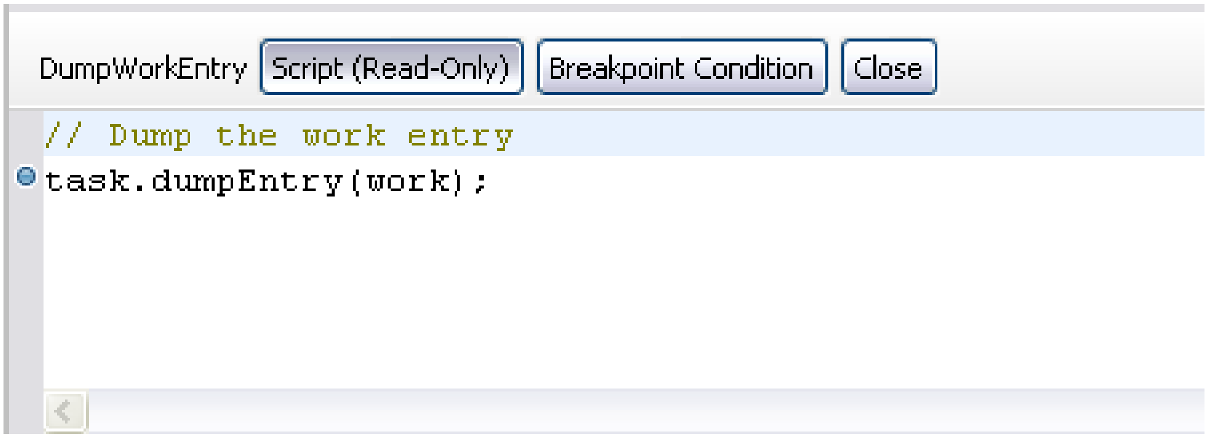
```
task.logmsg("This is called from Script1 component");

myfunc1();

task.logmsg("We are back again");
```

```
14:06:53,056 INFO  - CTGDIS116I Scripting hook of type prolog0 finished.
14:06:53,196 INFO  - CTGDIS087I Iterating.
14:07:00,150 INFO  - This is called from Script1 component
```

Figure 91. Follow the function call

Step into causes the script editor to change to the script from the *before init* hook. Notice the changed label that shows where the script is defined.

You can also set breakpoints inside scripts. Open the editor for a script or attribute map and double-click the left margin. A blue bullet will appear to denote that there is a breakpoint set for that location. Double-click again to remove it. You can also right-click the left margin or in the text field to toggle break points.



```
// Dump the work entry
task.dumpEntry(work);
```

Server Debugging

Debugging a Tivoli Directory Integrator server means that every AssemblyLine started on a Tivoli Directory Integrator server will automatically establish a debug session with the Configuration Editor as if it were started in step mode.

Server debugging is activated by selecting **Debug Server** from the drop-down menu on a server in the Servers view. When you choose this option, the CE will connect to the server and set a Java property pointing to the CE for debugging.

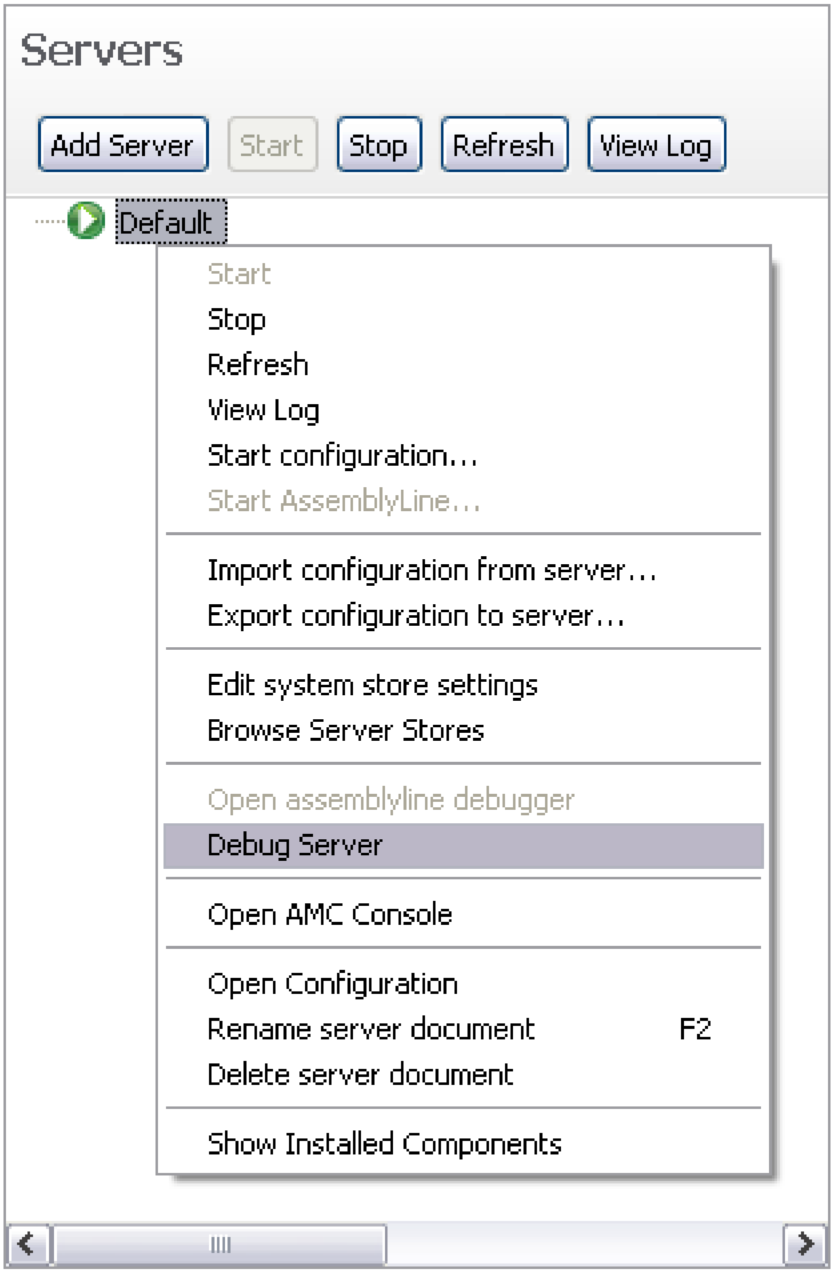
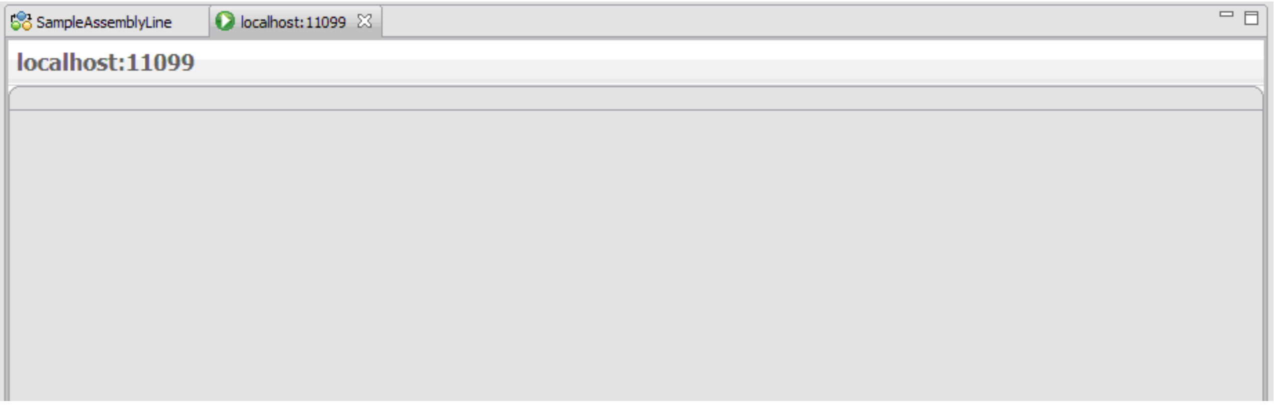


Figure 92. Debug Server option

Selecting **Debug Server** starts a new window where the AssemblyLines appear as they are started on the server. This is different from actively starting an AssemblyLine in the CE. When you start a debug session from the CE, it will have its own window and will not appear in this server debug window.

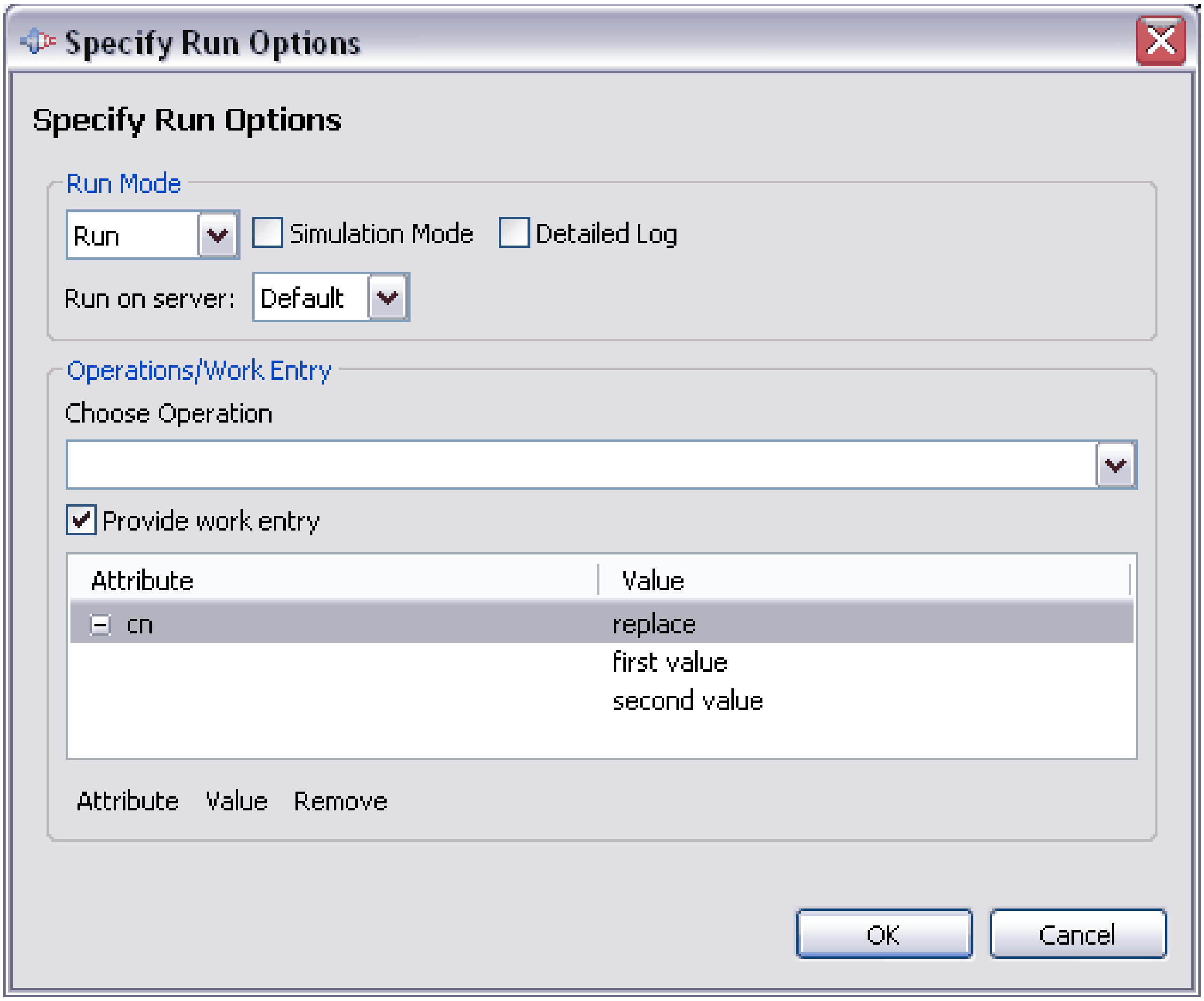


Each AssemblyLine started by another CE or component on the targeted server will have its own stepper panel inside this window. See section "The stepper and debugger" on page 157 for a description of the stepper panel.

Run Options

You can specify additional options when you run an AssemblyLine. These options are saved so every time you run the AssemblyLine it will use these options.

You can select the run mode, the AssemblyLine operation and provide an initial work entry (IWE) to the AssemblyLine.



Specify Run Options

Run Mode

Run □ Simulation Mode □ Detailed Log

Run on server: Default

Operations/Work Entry

Choose Operation

☑ Provide work entry

Attribute	Value
⊟ cn	replace
	first value
	second value

Figure 93. Run with Options window

Use the **Attribute** and **Value** buttons to add attributes and values to the initial work entry.

Choosing the Server

When you run an AssemblyLine it will execute on the local development server. This server is started by the CE and its solution directory is located in the *TDI Servers* project. When you create a new Tivoli Directory Integrator server, you can change the preferred server for a given project through the **Properties** dialog for the project:

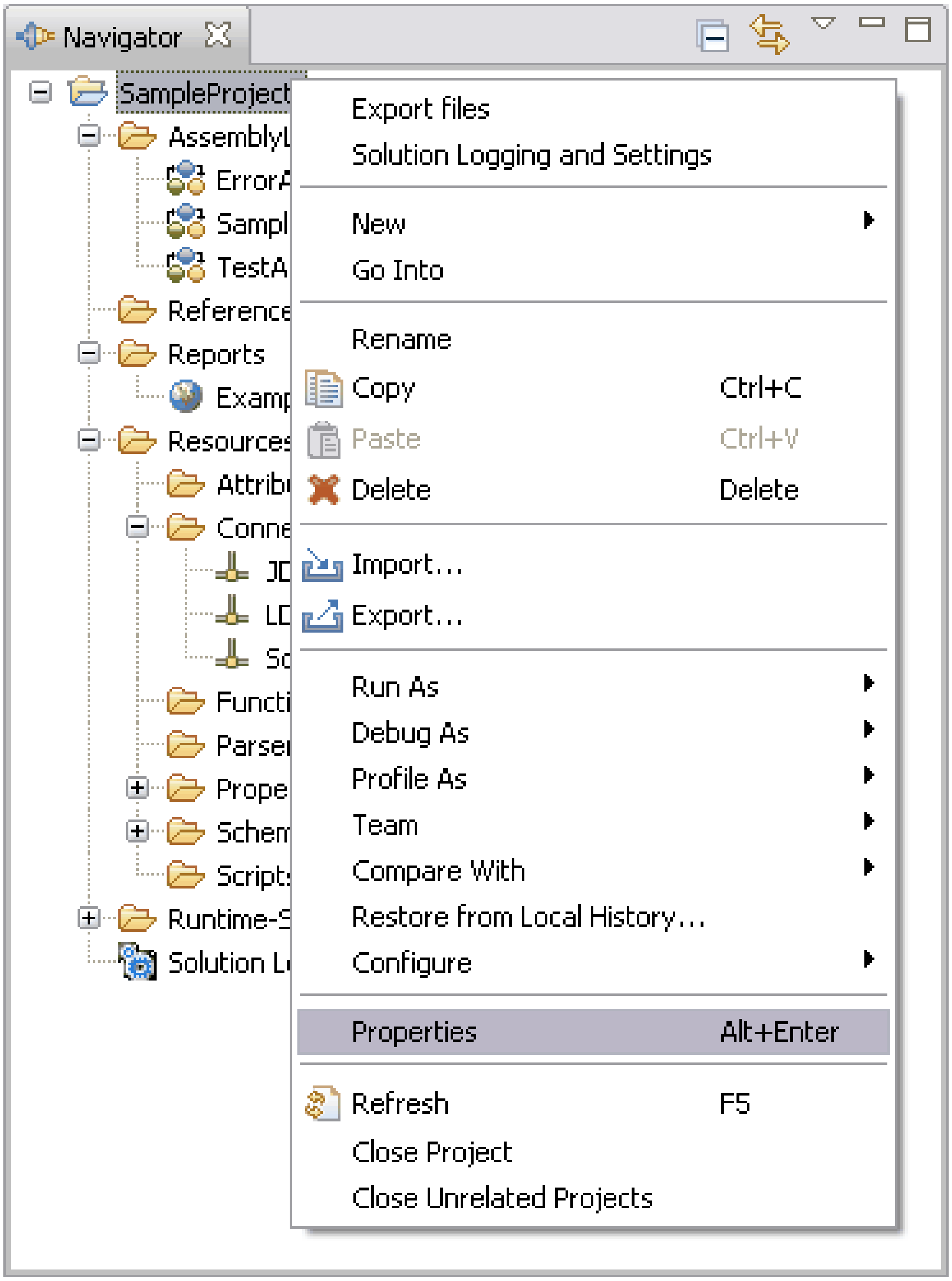
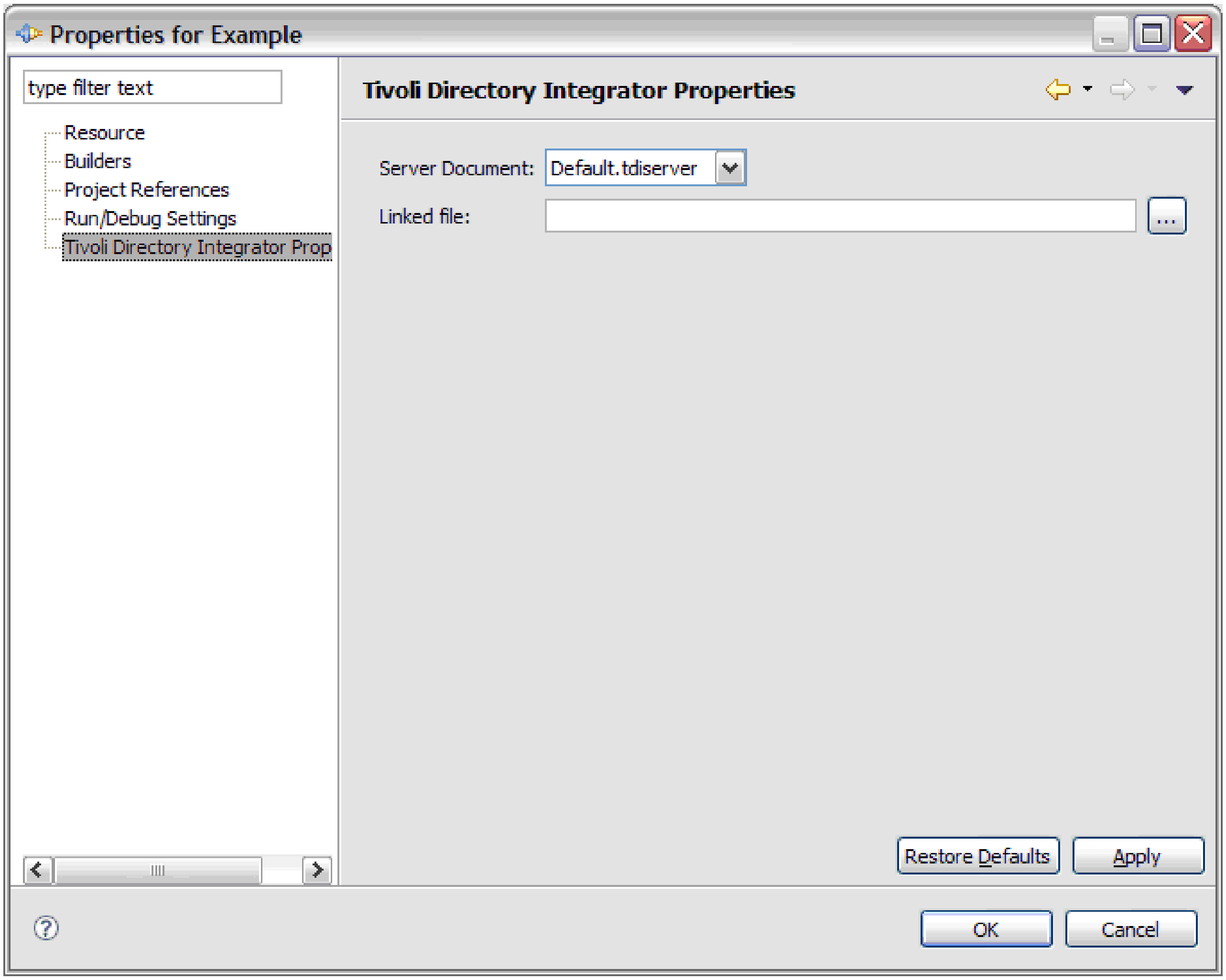


Figure 94. Project Properties menu choice

Selecting this item will bring up the properties dialog for the project. Select **Directory Integrator Properties** to change the default server for the project.



Server Document: Default.tdiserver

Linked file:

Figure 95. Project Properties

Team Support

The Tivoli Directory Integrator Configuration Editor includes the Eclipse plug-ins that enable sharing of projects between users using a source code control repository.

The V7.1.1 release of Tivoli Directory Integrator includes CVS libraries that support pserver, pserverssh2, ext and extssh type connections. For more in depth information about the eclipse CVS plugins consult the Eclipse CVS site at http://www.eclipse.org/eclipse/platform-cvs.

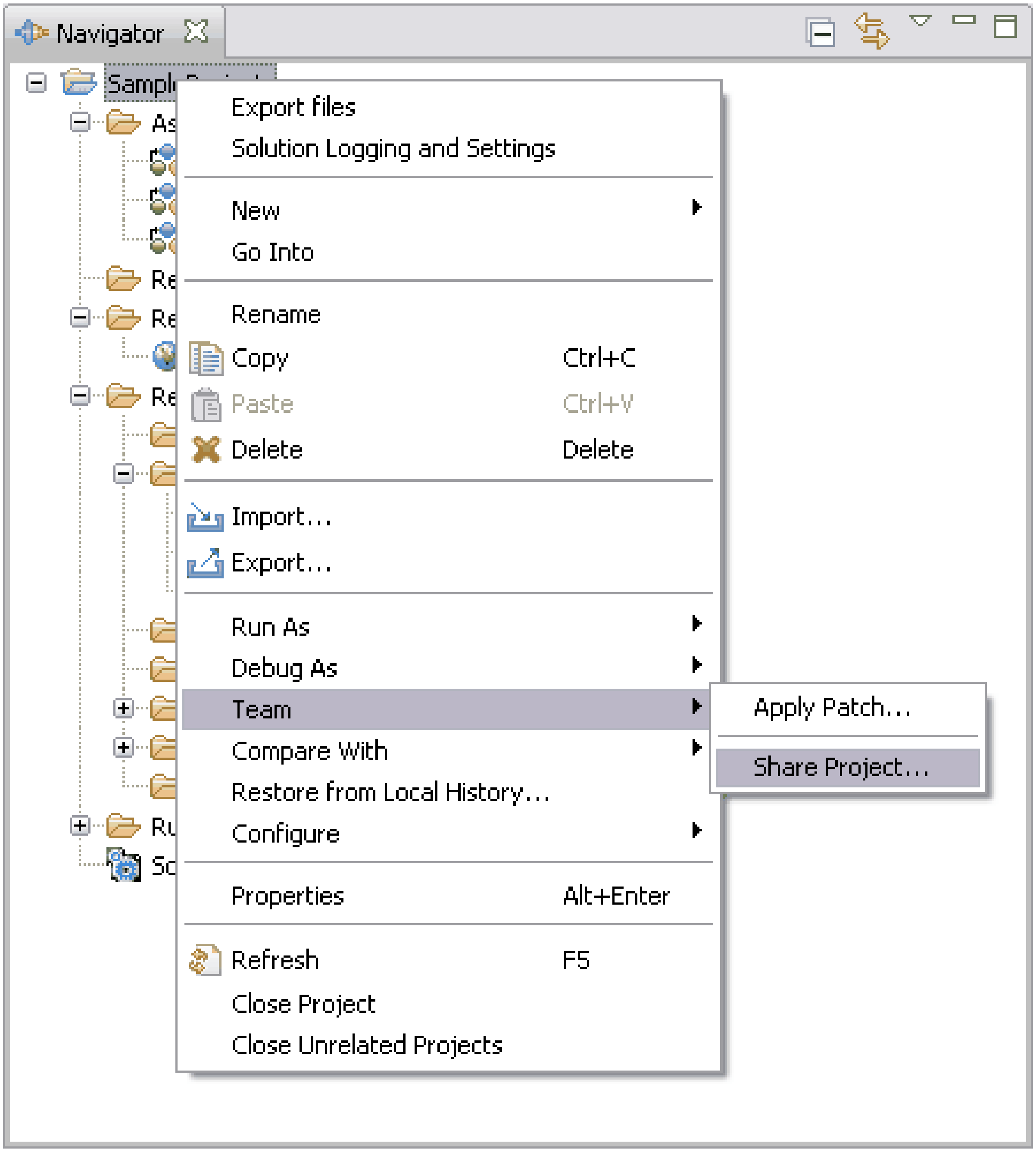
In order to use the team sharing facilities you need access to a CVS repository. CVS repositories can be hosted by most operating systems and binary packages are commonly available on the net. The Eclipse CVS site has an FAQ that will help you with many common problems you may run into.

For help with installing and configuring a CVS server consult the CVS wiki site at http://ximbiot.com/cvs. Searching the web for "how to install a cvs server" will also bring up a host of web sites that will describe in detail how to install and configure a CVS repository on a variety of operating systems and platforms.

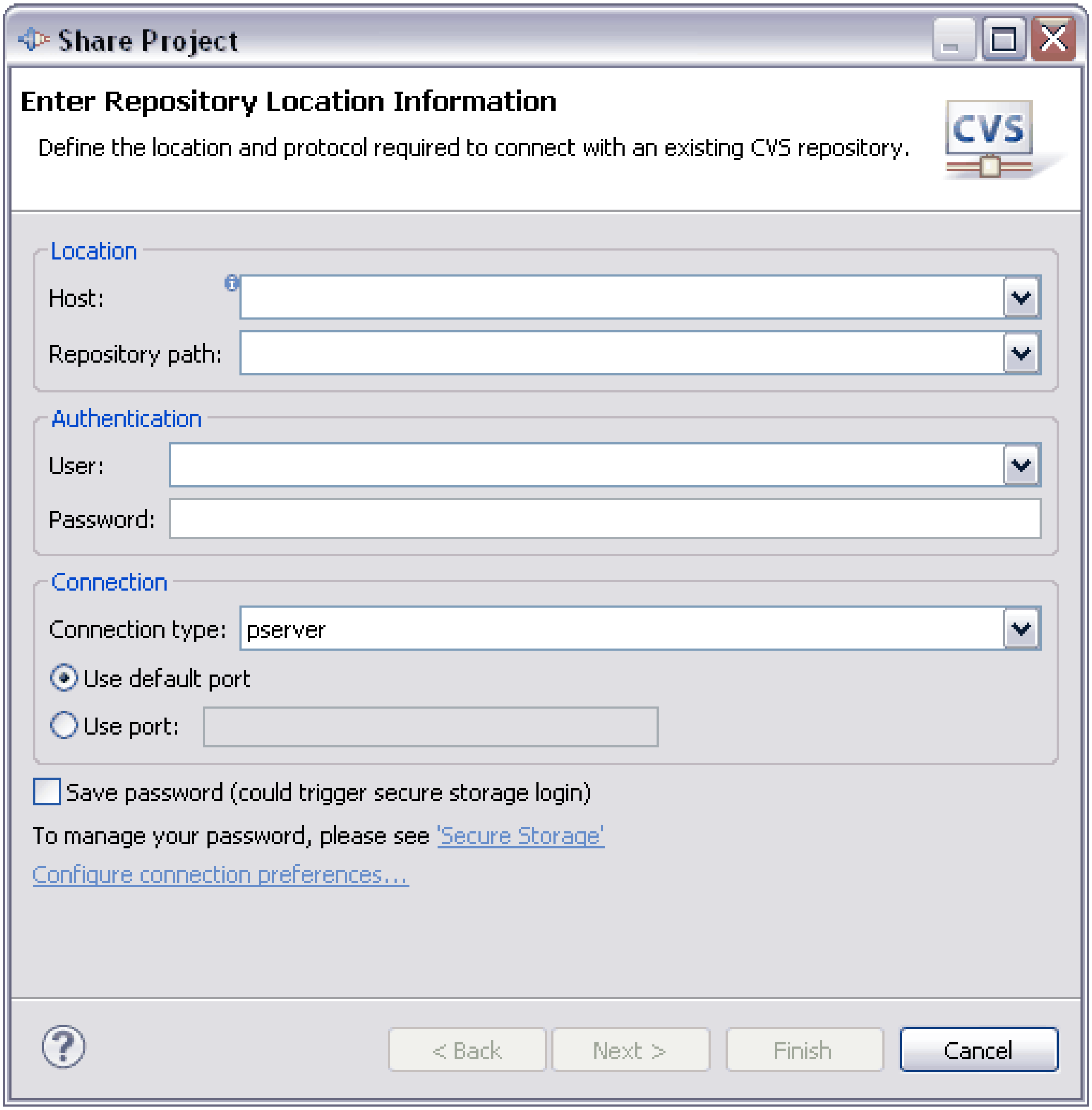
Sharing a project

You can share a project with other users using CVS.

To share a project you choose the **Team** > **Share Project...** option in the drop-down menu on a project.



This opens another screen, where you can specify parameters for the source control repository.



Share Project

Enter Repository Location Information

Define the location and protocol required to connect with an existing CVS repository.

Location

Host:

Repository path:

Authentication

User:

Password:

Connection

Connection type: pserver

● Use default port

○ Use port:

□ Save password (could trigger secure storage login)

To manage your password, please see 'Secure Storage'

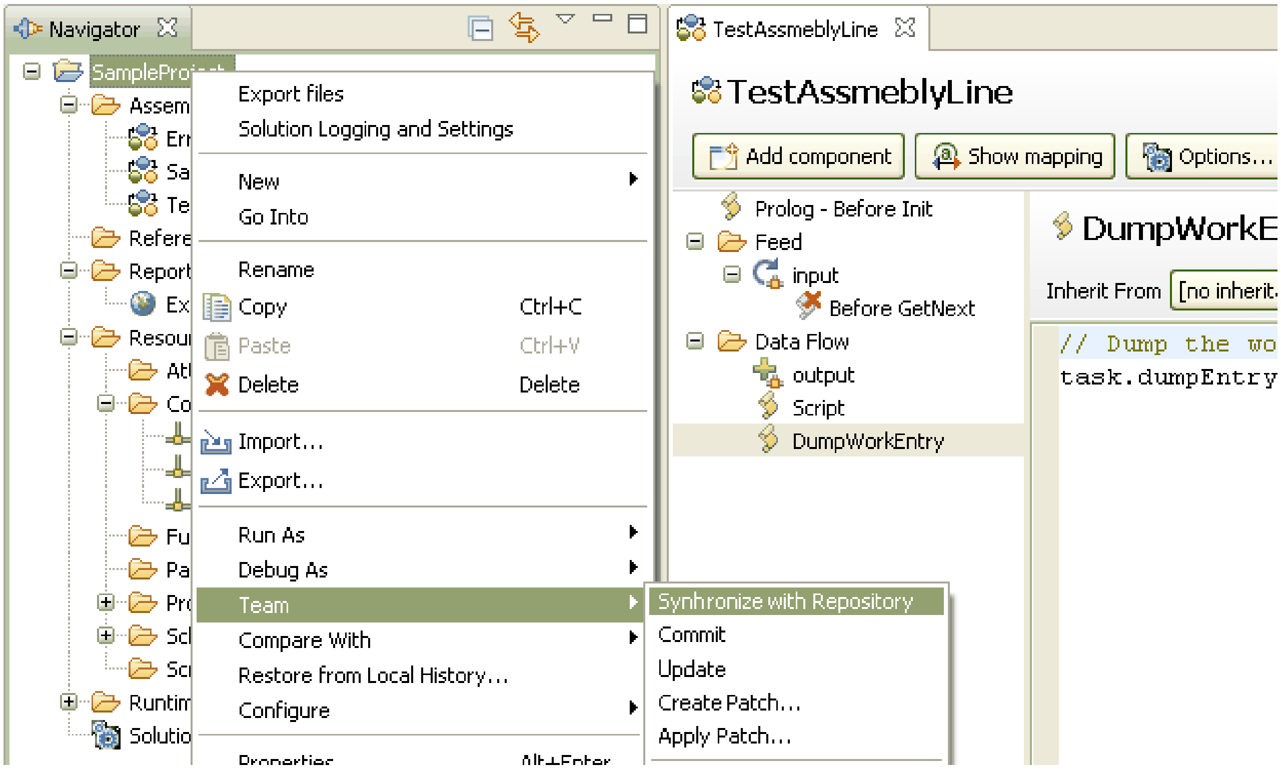
Configure connection preferences...

Figure 96. CVS Share Project window

Complete the wizard to share the project to the CVS server.

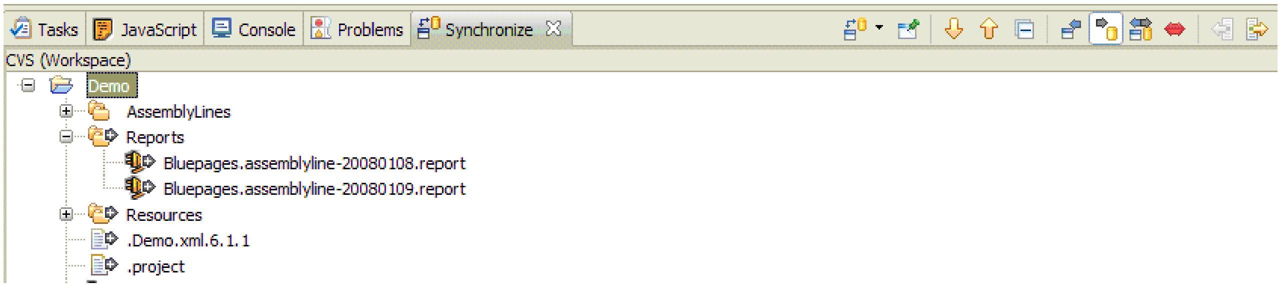
Note: Only actual files in the project, and the directory structure containing them, can be properly controlled by the repository. Empty directories are not taken into account.

When the project has been shared, you can synchronize with the repository to commit your own changes as well as receiving changes made by other people.



```
// Dump the wo
task.dumpEntry
```

Select **Team** > **Synchronize with Repository** to open the synchronize view:

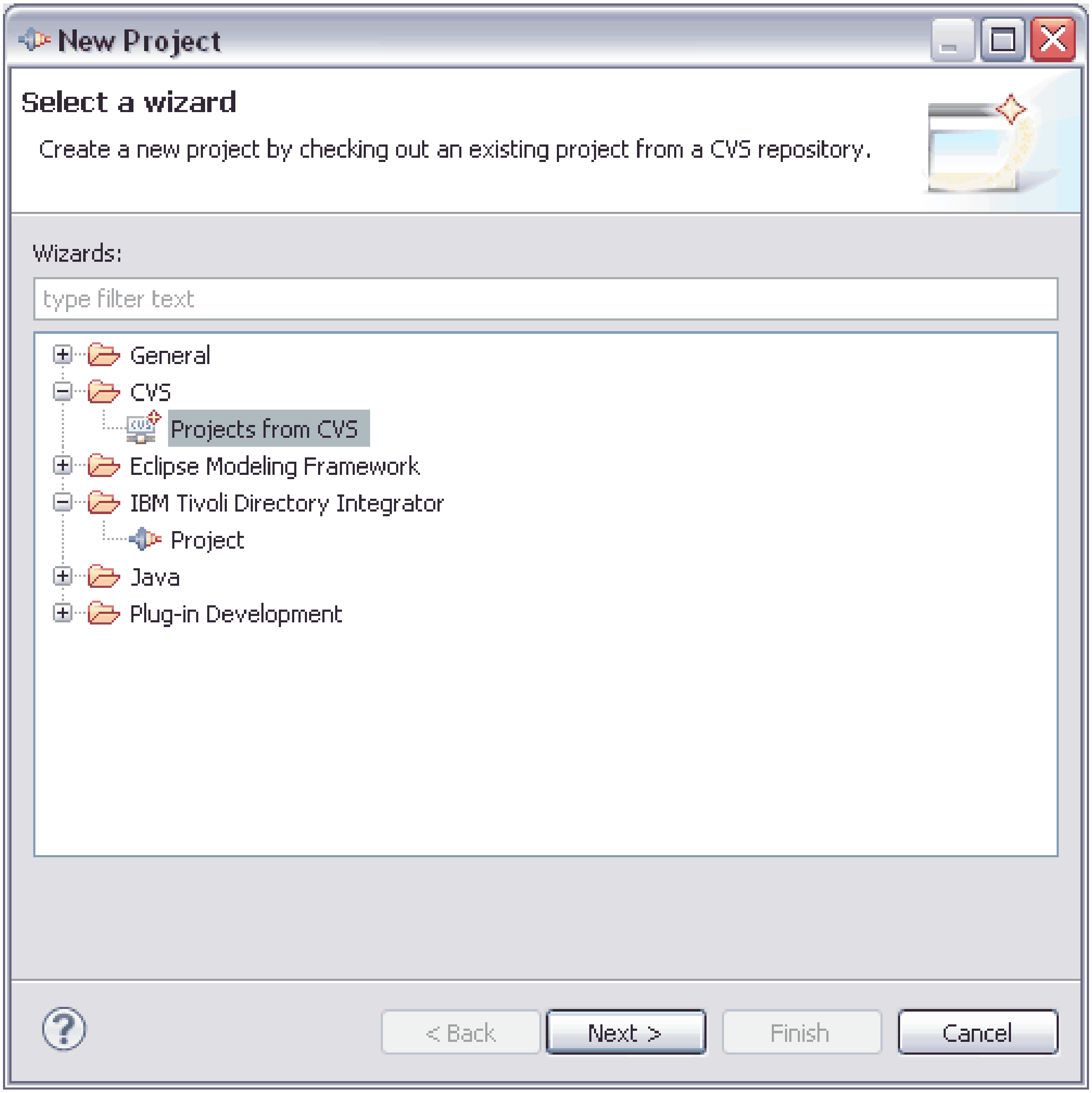


This view shows which files need to be updated as well as which files there are updates to. After committing the changes the navigator will show additional info for the files.

Using a shared project

If you want to use a project someone has shared, you use the CVS project wizard.

Select **File** > **New** > **New project...** from the main menu and choose the CVS project wizard.



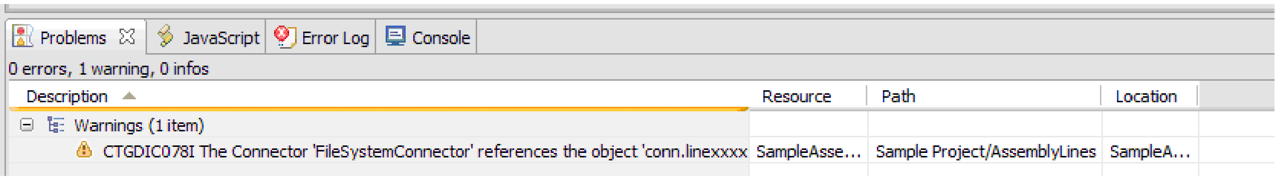
Create a new project by checking out an existing project from a CVS repository.

Complete the wizard to retrieve the project into your workspace.

The Problems View

When you save a component, the Tivoli Directory Integrator project builder will update its runtime configuration file to reflect the changes you made.

The project builder will also perform a validation check on the component and report warnings and errors in the problems view. In the problems view you will see warnings like these:



0 errors, 1 warning, 0 infos

Description	Resource	Path	Location
Warnings (1 item)			
CTGDIC078I The Connector 'FileSystemConnector' references the object 'conn.linexxxx	SampleAsse…	Sample Project/AssemblyLines	SampleA…

Figure 97. Problems View window

When you double-click a line in this view it will open the location where the problem was located.

The problems you can expect to see in the problems view are the following items:

- Reference to an undefined schema item

  If you use constructs like “conn.abc” and “abc” is not defined in the schema you get this warning.
- Reference to an undefined work attribute

  If you use constructs like “work.abc” and “abc” is not known to exist you get this warning.
- Syntactical errors in scripts
- AssemblyLines with more than one server mode connector

JavaScript Enhancements

Everywhere you edit scripts in the Configuration Editor, you will get a text editor, enhanced specifically for editing JavaScript code.

This editor provides code completion, syntax coloring as well as marking syntactical errors.

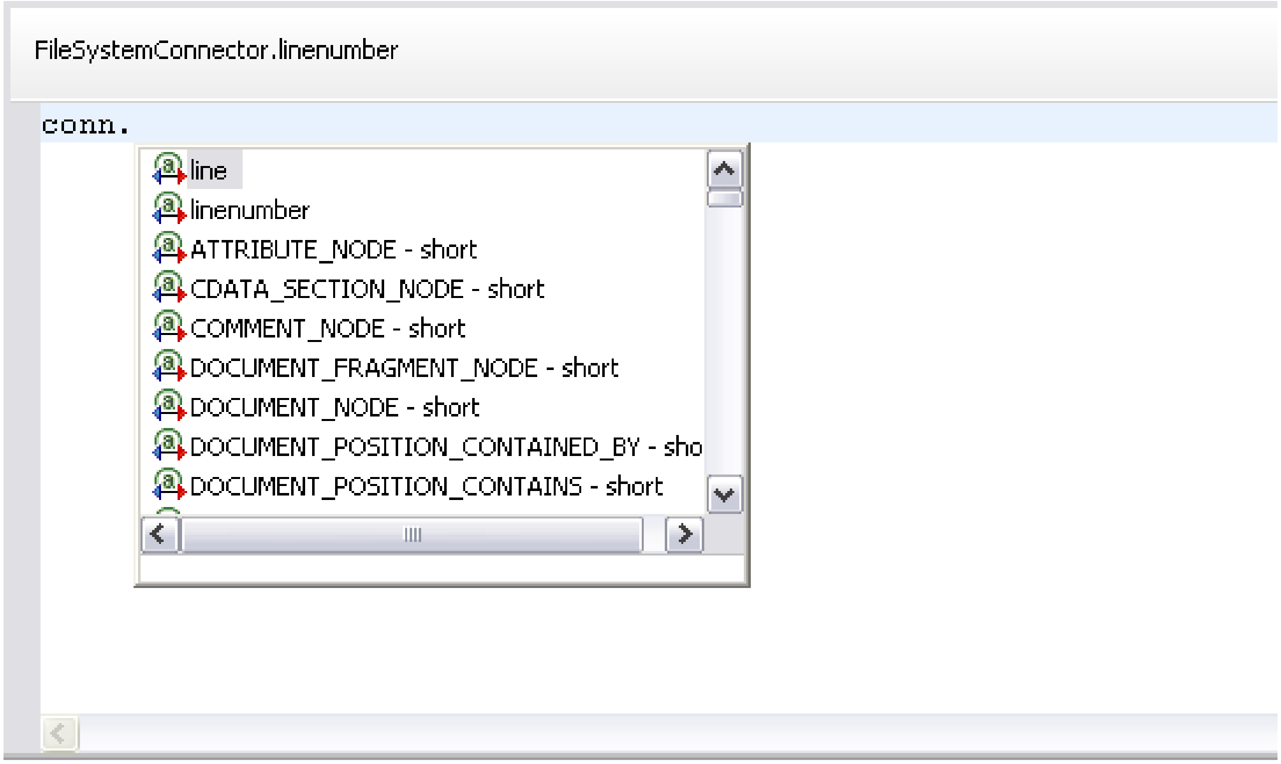
Some of the enhancements are:

- “Code Completion”
- “Syntax Coloring” on page 179
- “Syntax Checking” on page 179
- “Local Evaluation” on page 179
- “External Editors” on page 180

Code Completion

The JavaScript editor supports code completion.

As you type, the editor will react to certain keystrokes. The dot (.) activates code completion that brings up a popup menu showing all relevant methods, fields and features specific to IBM Tivoli Directory Integrator. Code completion can also be manually activated by the `Ctrl-<Space>` key combination. Code completion covers standard JavaScript script types (that is, string, number and so forth) as well as custom completion for TDI-specific objects. Objects like *conn* and *work* provide a list of available attributes as well as fields and methods of the object.

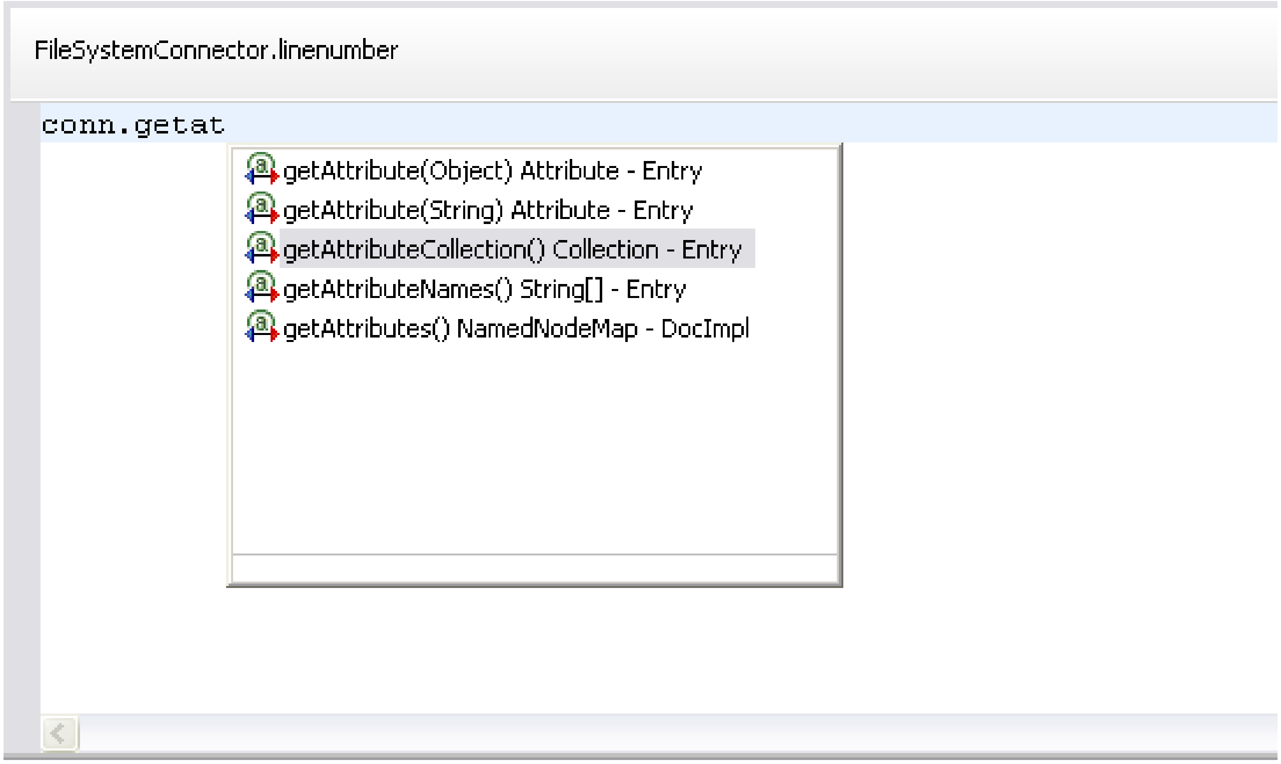


FileSystemConnector.linenumber

```
conn.
```

line
linenumber
ATTRIBUTE_NODE - short
CDATA_SECTION_NODE - short
COMMENT_NODE - short
DOCUMENT_FRAGMENT_NODE - short
DOCUMENT_NODE - short
DOCUMENT_POSITION_CONTAINED_BY - sho
DOCUMENT_POSITION_CONTAINS - short

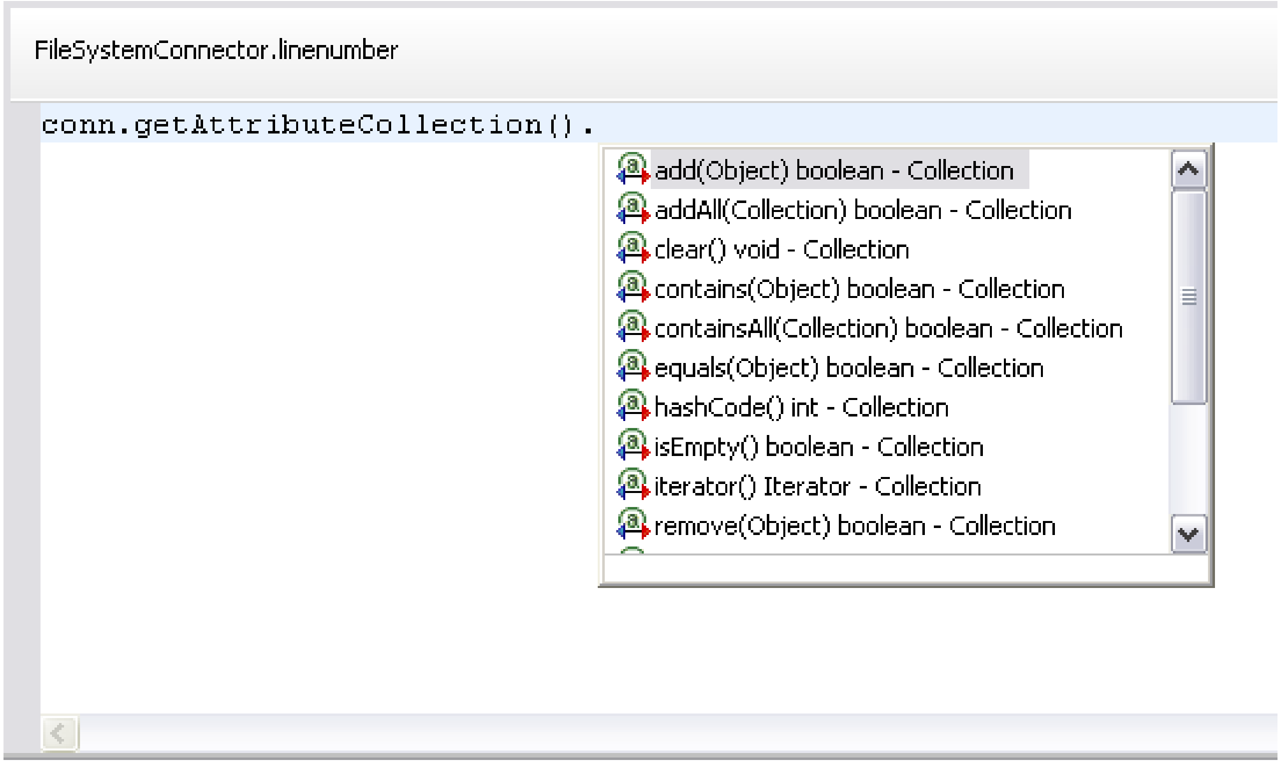
Code completion works as long as the editor can derive the Java class name in the expression:



FileSystemConnector.linenumber

```
conn.getat
```

getAttribute(Object) Attribute - Entry
getAttribute(String) Attribute - Entry
getAttributeCollection() Collection - Entry
getAttributeNames() String[] - Entry
getAttributes() NamedNodeMap - DocImpl



FileSystemConnector.linenumber

```
conn.getAttributeCollection().
```

add(Object) boolean - Collection
addAll(Collection) boolean - Collection
clear() void - Collection
contains(Object) boolean - Collection
containsAll(Collection) boolean - Collection
equals(Object) boolean - Collection
hashCode() int - Collection
isEmpty() boolean - Collection
iterator() Iterator - Collection
remove(Object) boolean - Collection

The editor will also react to single or double quotation marks and curly ("{}") braces. When you type a single or double quotation mark, the editor automatically inserts another quotation mark and positions the caret in between the two. This is done to make it easier to enter string constants.

Typing curly braces will auto indent to accommodate indentation of blocks. When you type an opening curly brace and hit enter, tabs will be inserted and the caret positioned on the next line with proper indentation. Conversely, typing the closing brace will un-indent the curly brace.

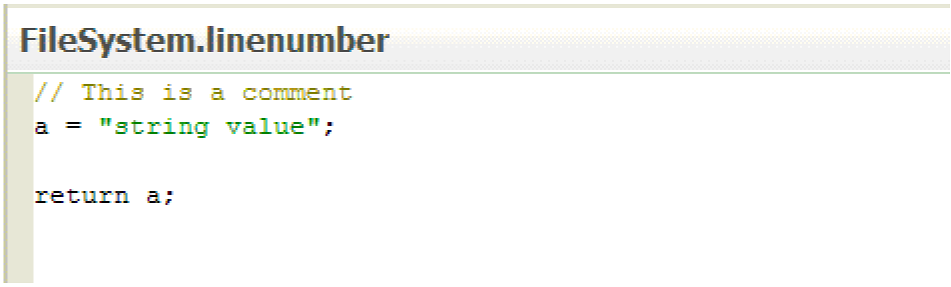
Mapping and Code Completion

When you add attributes in the CE we will generate simple expressions based on the attribute name. Any name that contains dots or other characters that are not valid script object identifiers, will be enclosed in ["attribute name"] (for example, `conn["http.body"]`). Whether the Entry is hierarchical or not is irrelevant when we use this notation since it works in both cases. Attributes that are valid javascript identifiers are used as is (for example, conn.cn). If you have a hierarchical schema we will still generate a simple expression for the map. If for example you have a hierarchy of a->b->c and you map the "c" part we generate ["a.b.c"] to reference the attribute.

In the script editor you will see that code completion works in a similar fashion. The code completion will show completions in plain text (for example, http.body) but when you hit enter to complete the expression the same enclosure is used for attribute names that are not valid script object names.

Syntax Coloring

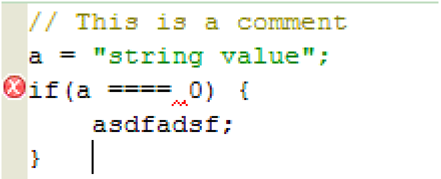
Syntax coloring in the editor is basic. The editor decorates comments and strings.


FileSystem.linenumber

```
// This is a comment
a = "string value";

return a;
```

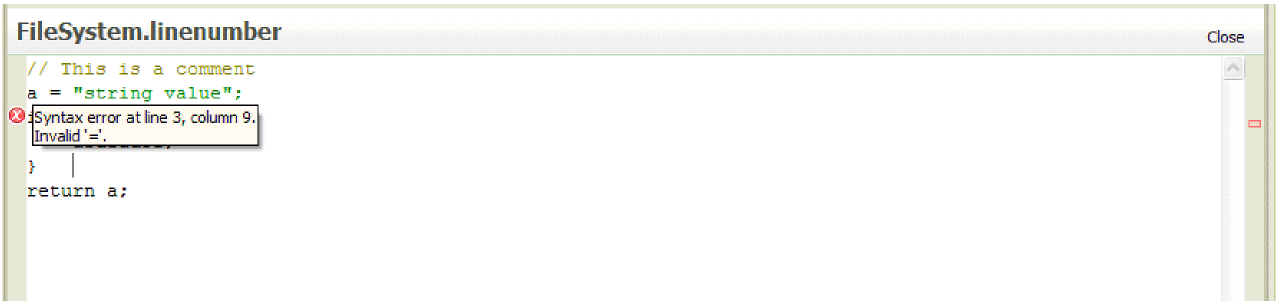
Syntax Checking

As you type, the editor performs syntax checking on the script in the background. When you have errors in the script, it shows the error in the margins and the text will get a red squiggle under it to mark where the JavaScript interpreter found an error.



```
// This is a comment
a = "string value";
if(a ==== 0) {
    asdfadsf;
}
```

When you mouse over the error mark in the *ruler* (the left margin) you see the error message in a popup window.



FileSystem.linenumber

```
// This is a comment
a = "string value";
}
return a;
```

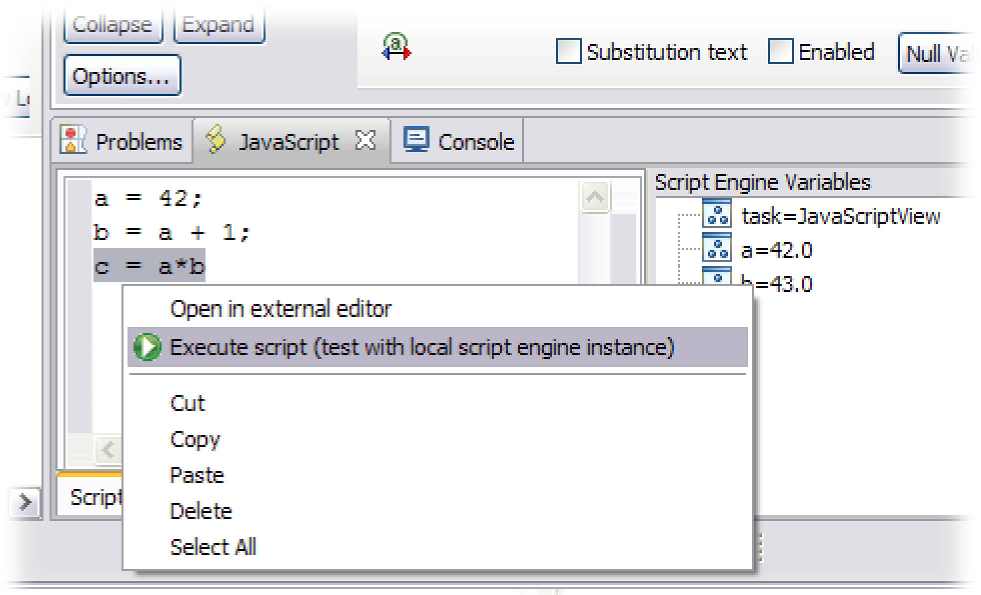
Syntax error at line 3, column 9.
Invalid '='.

The ruler on the left is synchronized with the text window, and a mark is directly related to the line in the text window. If there are no errors in the text you see in the window, there will be no marks in this ruler either.

However, the overview ruler on the right shows all errors in the entire script regardless of where the text editor is positioned. When you mouse over the mark in the ruler you get the same error message popup as you get in the left ruler. Clicking on the marker repositions the editor to the line where the problem was identified.

Local Evaluation

When you edit a script, you can right-click and choose **Execute Script** from the dropdown menu to do a quick evaluation of the selected text. A more detailed test environment for scripts is the JavaScript view where you can execute scripts and look at the script engine variables in a separate window:



```
a = 42;
b = a + 1;
c = a*b
```

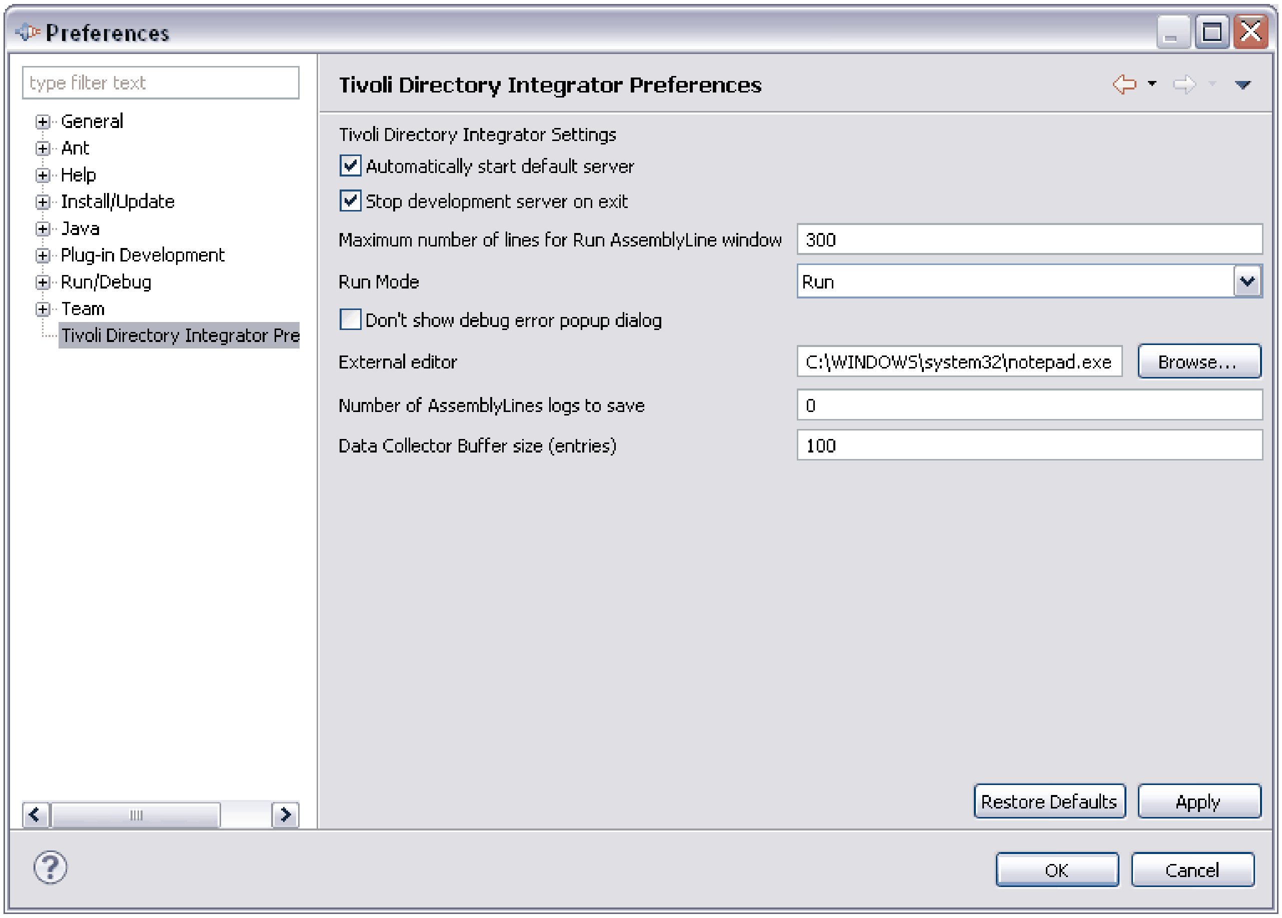
Script Engine Variables

```
task=JavaScriptView
a=42.0
b=43.0
```

External Editors

You can edit the script in your favorite JavaScript editor using the context menu.

Right-click in the text field and choose **Open in external editor**. The editor is configured in the **Window** > **Preferences** > **Tivoli Directory Integrator** preferences tab:



Tivoli Directory Integrator Preferences

Tivoli Directory Integrator Settings

☑ Automatically start default server

☑ Stop development server on exit

Maximum number of lines for Run AssemblyLine window: 300

Run Mode: Run

□ Don't show debug error popup dialog

External editor: C:\WINDOWS\system32\notepad.exe [Browse...]

Number of AssemblyLines logs to save: 0

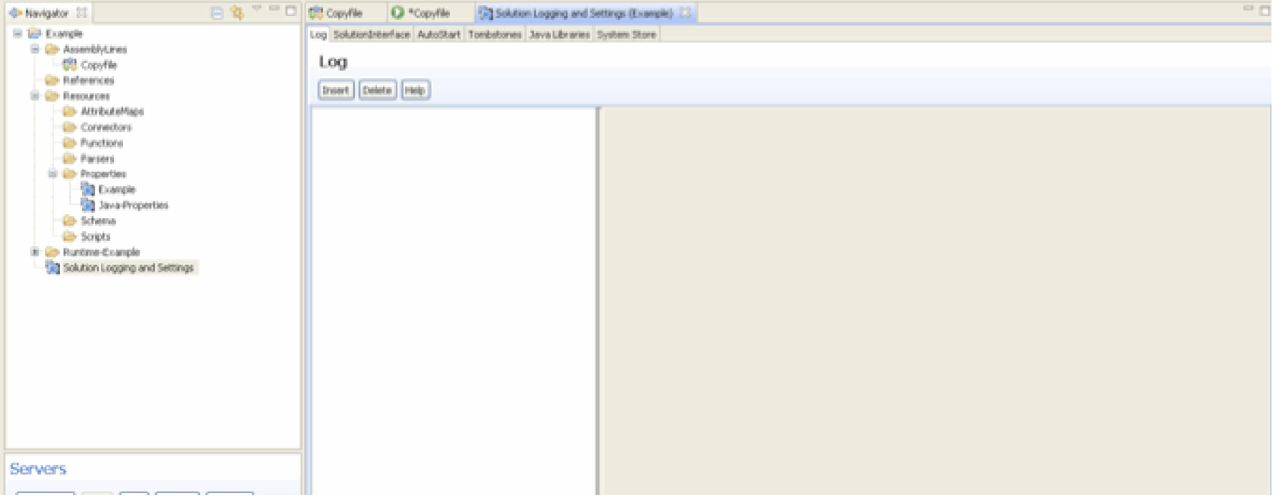
Data Collector Buffer size (entries): 100

Figure 98. Configuration Editor Preferences window

Solution Logging and Settings

The Solution logging and Settings window lets you edit the solution specific areas of the project.

You can open the window by selecting a project, or a project file, and double-click **Solution Logging and Settings** in the project tree.



Under Solution Logging and Settings you can modify the following items:

- "System Store settings"
- "Logging" on page 184
- "Tombstones" on page 185
- "Java Libraries" on page 185
- "AutoStart" on page 185
- "Solution Interface settings" on page 186

System Store settings

The system store settings for a project can override the default system store defined by the Tivoli Directory Integrator server. When this is enabled, the configured system store will be used by the configuration instead of the system store of the Server.

System store settings can occur in two places:

1. At the Tivoli Directory Integrator Project level in your workspace; and
2. at the Tivoli Directory Integrator Server level.

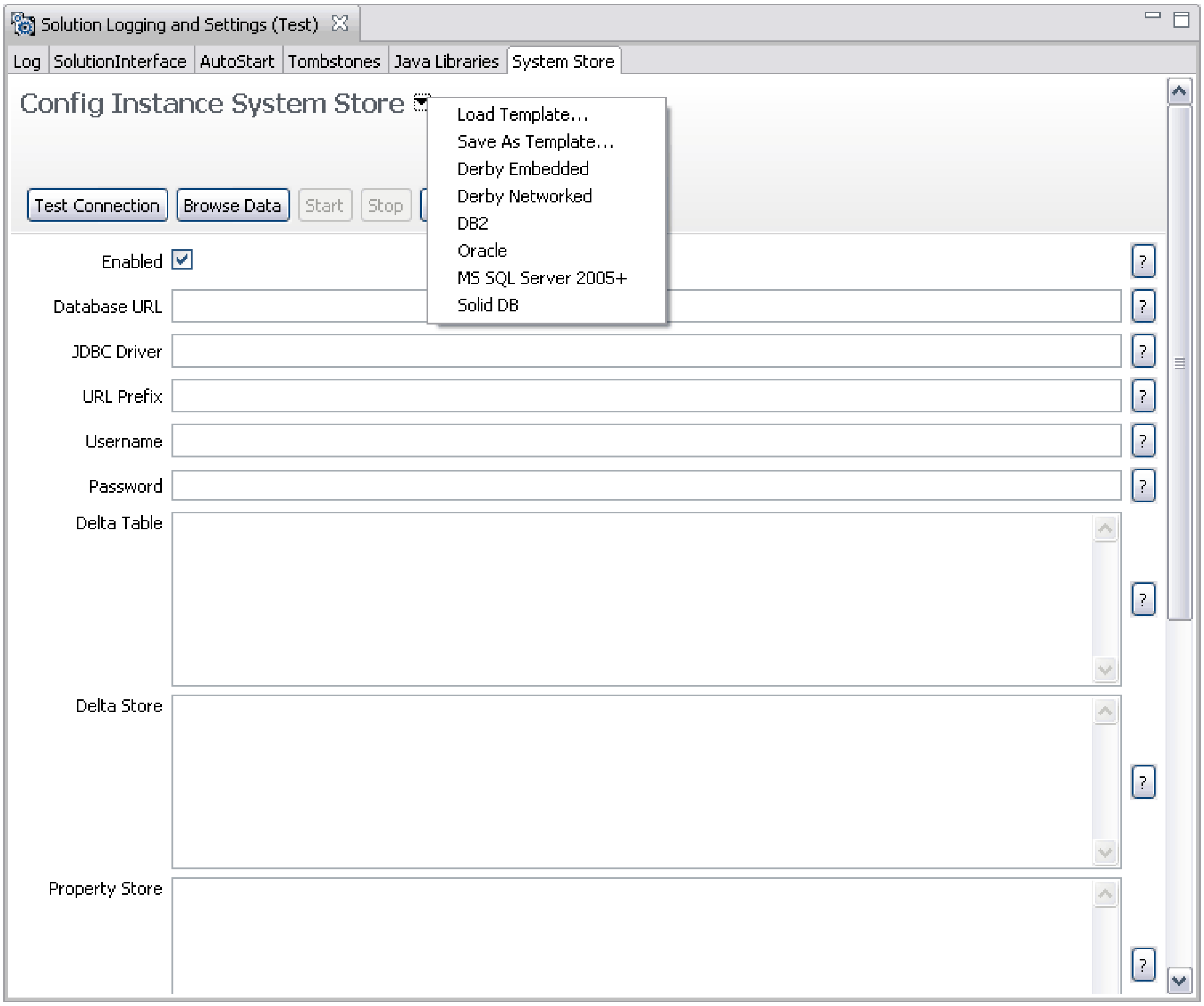
The settings at the Project level take precedence over the settings at the Server level, that is, if you have defined a system store specifically for your project, then that system store will be used while running your components in the Config Editor; if you have not defined any system store settings in the project, the system store of the Server you use to run your components will be used.

Settings for Project level

In the drop-down menu of the title you can choose predefined templates as well as loading and saving system store settings to files in the workspace. Note that all tables (delta, properties and so forth) will be stored in this database.

The menu items listed as **Derby Embedded** and so forth are pre-defined templates that you can load into the configuration panel, after which you can modify them to your needs, and subsequently update them in the configuration. You can also **Load Template...** from your local filesystem.

Changes made to this panel are saved in the Configuration file (when the project is exported) and are used by all its AssemblyLines, despite the settings of the server on which they are executed.



Enabled ☑

Figure 99. Configuration system store settings

Settings for Server level

The same screen appears when you select **Edit system store settings** from the drop-down menu on a server in the Servers view. This will update and save the system store settings variables to the solution properties file. A server restart is required to activate the new settings.

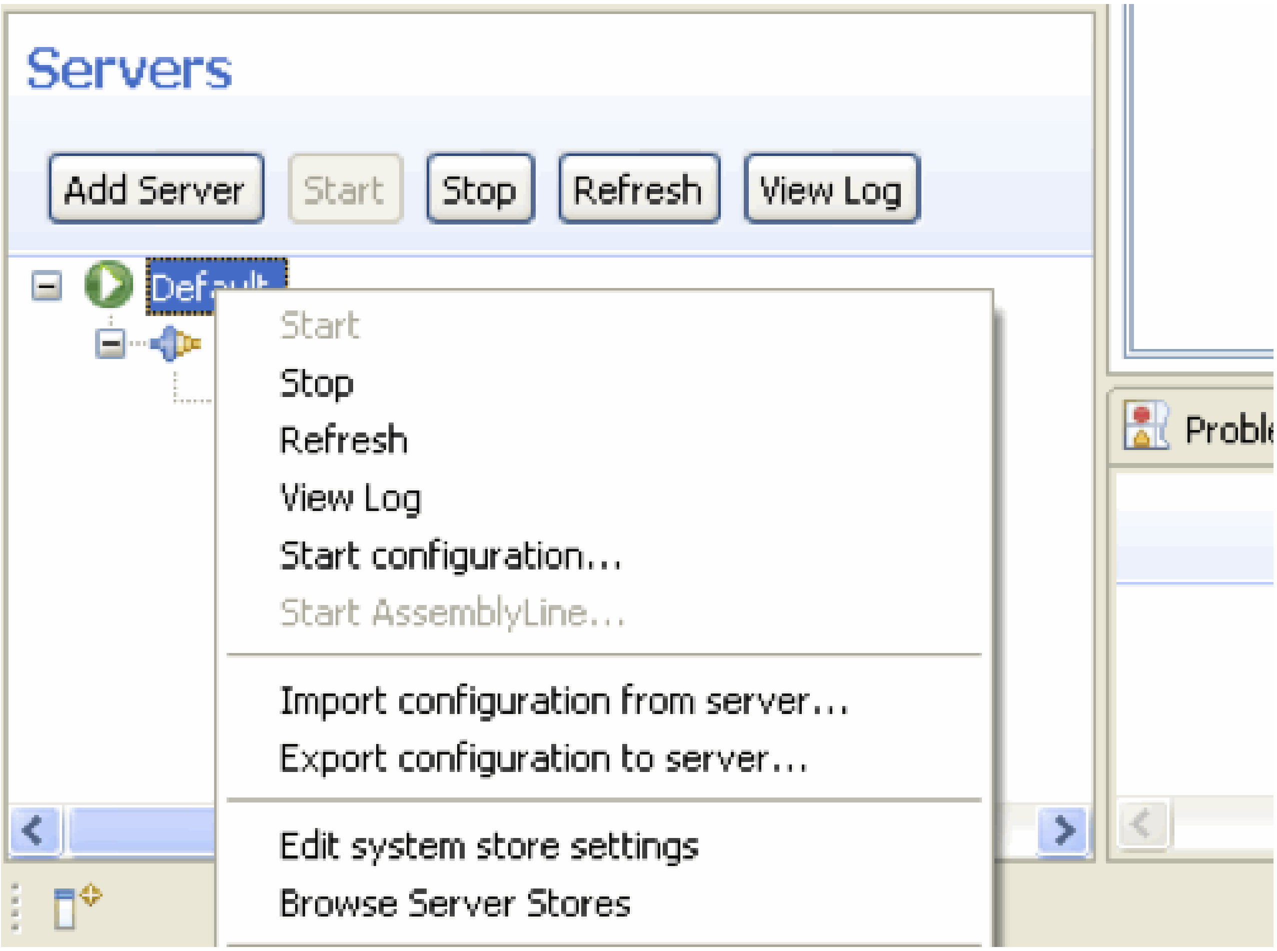
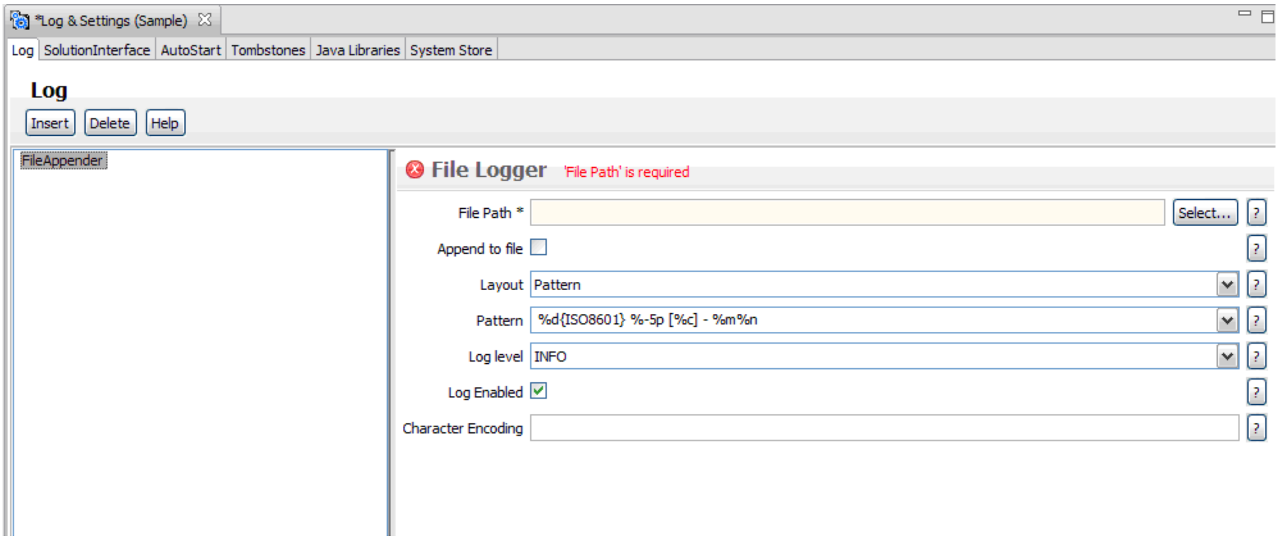


Figure 100. Server document context meu

Logging

The **Logging** view shows the loggers for the solution.

You can add and remove loggers by clicking **Insert** and **Delete** in the title bar.



File Logger 'File Path' is required

File Path *

Append to file ☐

Layout Pattern

Pattern %d{ISO8601} %-5p [%c] - %m%n

Log level INFO

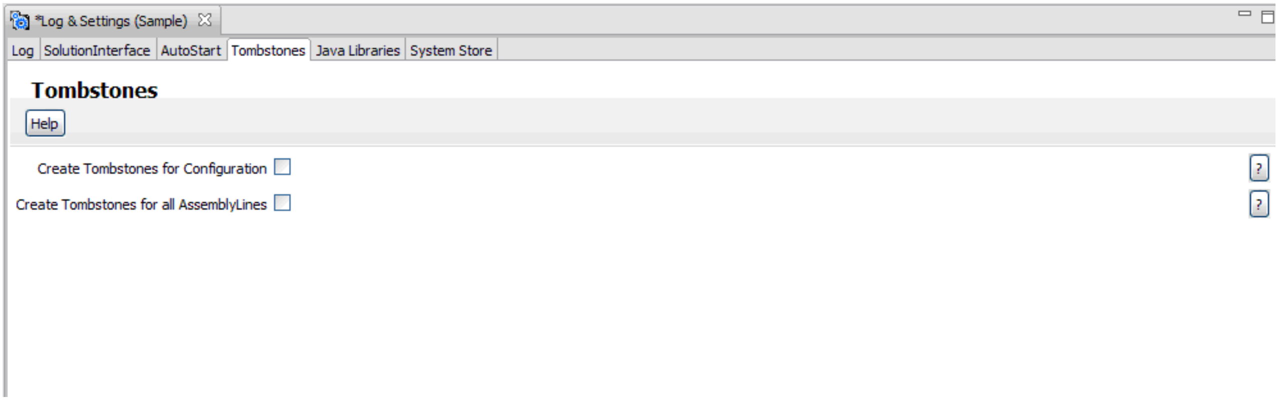
Log Enabled ☑

Character Encoding

See chapter "Logging and Debugging" in *IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide* for more information.

Tombstones

The Tombstones configuration for a project is found in the **Tombstones** tab.



Create Tombstones for Configuration □

Create Tombstones for all AssemblyLines □

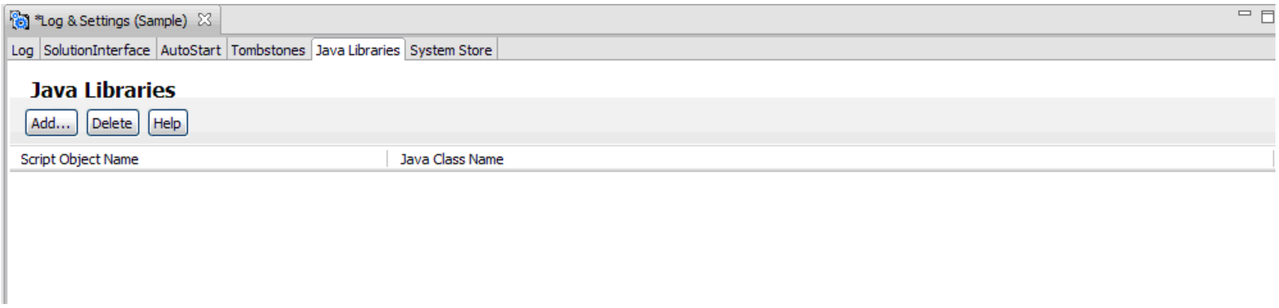
A Tombstone is a record of a run of an AssemblyLine, containing some statistics like number of entries read by Iterators, number of records skipped, number of records updated, and so forth.

Selecting **Create Tombstones for Configuration** causes a Tombstone to be generated every time the configuration file derived from this project, loaded on a Tivoli Directory Integrator Server, terminates.

Selecting **Create Tombstones for all AssemblyLines** causes a Tombstone to be generated every time an AssemblyLine in this project, run on a Server, terminates.

Java Libraries

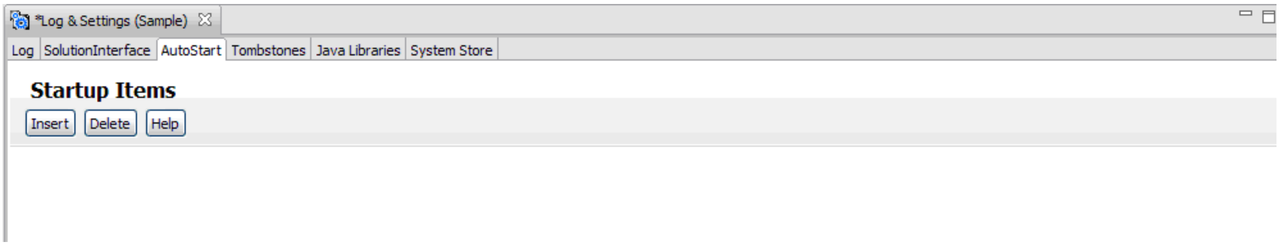
The **Java Libraries** tab shows the Java classes that are automatically loaded and defined in each instance of the script engine.



Script Object Name	Java Class Name

AutoStart

The **AutoStart** tab shows a list where you can enter the name of AssemblyLines that are automatically started when the configuration instance is started.



Startup Items

Figure 101. AutoStart settings

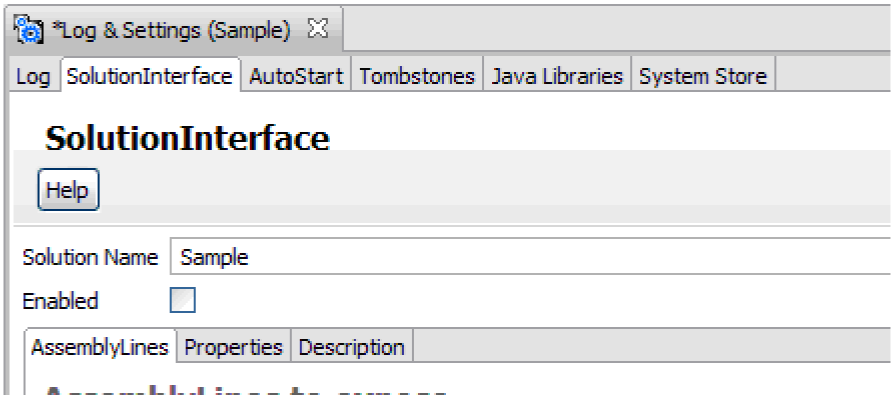
You can add items to the list by clicking **Insert**; this lets you choose from existing AssemblyLines in your workspace.

You can remove AssemblyLines from the Startup Items list by selecting them, and clicking **Delete**.

Solution Interface settings

The solution interface settings can be enabled to provide additional information about the configuration. This information is typically used by the Webadmin tool (AMC) but can be used by other clients that have access to the solution interface configuration.

The first two fields in this panel are the solution name and the enabled status. The default value for the solution name is the project name itself; if the solution name is left blank then the exported configuration file will not contain any Solution ID. The enabled checkbox determines whether the configuration is in use or not.



SolutionInterface

Solution Name Sample

Enabled □

Figure 102. Solution Interface settings

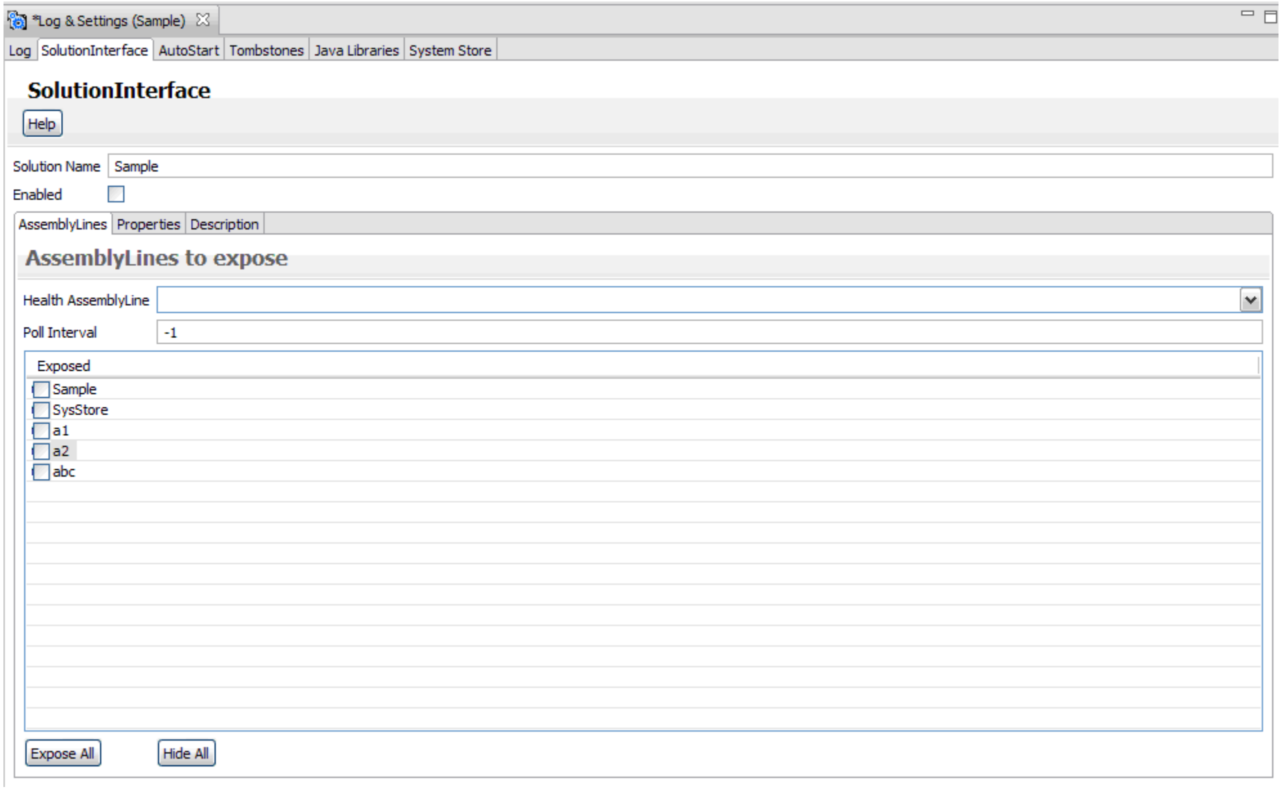
Solution Interface settings are divided into three sections, each with its corresponding tab:

- "AssemblyLines"
- "Properties" on page 187
- "Description" on page 188

AssemblyLines

This tab shows all AssemblyLines in the project and you can check those that should be visible to the user for starting/stopping.

The health AssemblyLine is a special AssemblyLine that is used to report the health of the running configuration. This AssemblyLine is simply one that can provide custom feedback to the user as to the state of the configuration. The health AssemblyLine is called and should return two fields in its work entry to report status (`healthAL.result` and `healthAL.status`). See the Action Manager (AM) documentation (online in AMC, or in *IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide*) for more information on how these fields are used in that context. The poll interval specifies how often the client (for example, AMC or AM) will call the health AssemblyLine.



Solution Name: Sample

Enabled □

Health AssemblyLine:

Poll Interval: -1

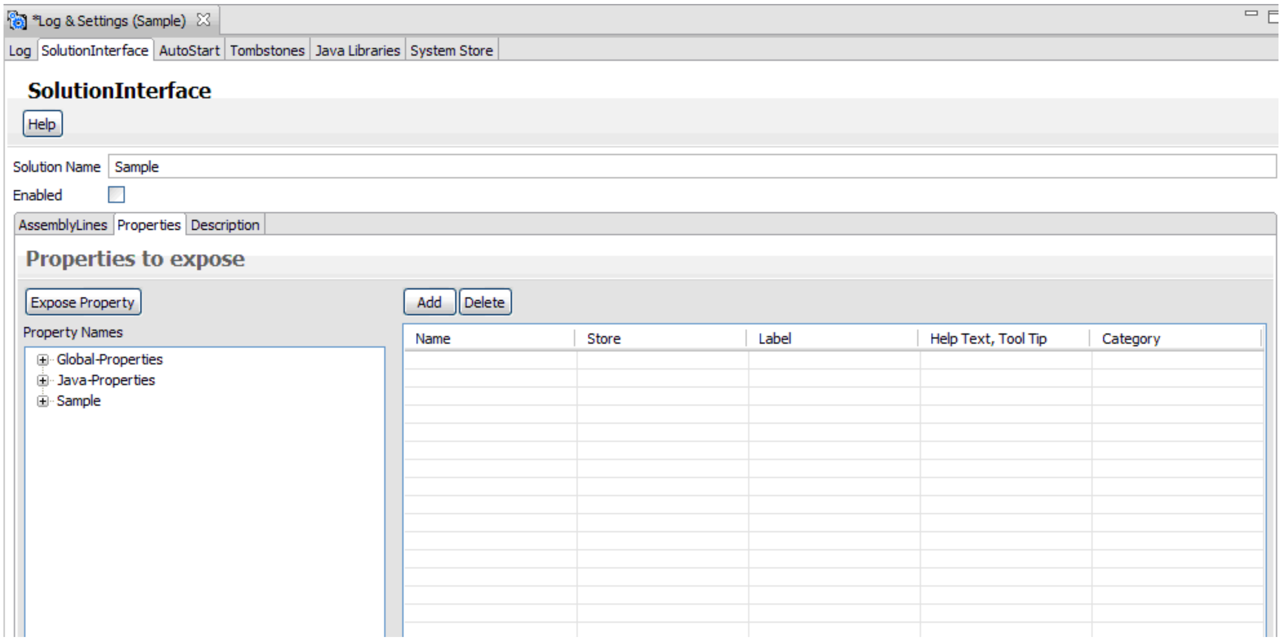
Exposed

- □ Sample
- □ SysStore
- □ a1
- □ a2
- □ abc

Figure 103. Solution Interface settings: AssemblyLines

Properties

This tab lets you define which properties the users see, and how it is presented to the user.



Solution Name Sample

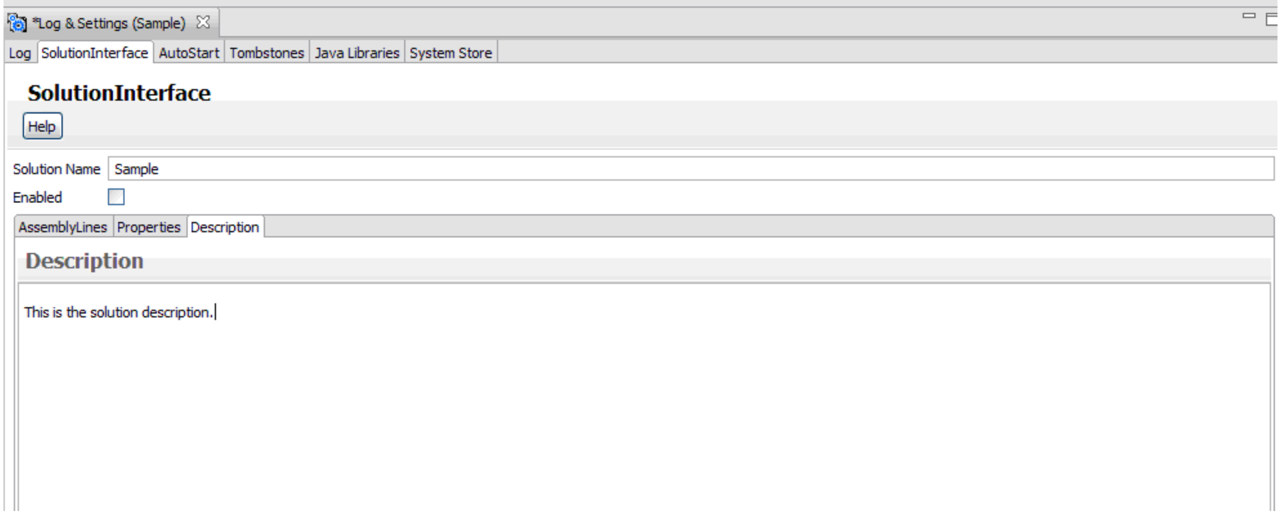
Enabled □

Name	Store	Label	Help Text, Tool Tip	Category

Figure 104. Solution Interface settings: Properties

Description

This tab lets you enter text that describes the solution. It is for documentation purposes only; it is not used in any other way by Tivoli Directory Integrator.



Solution Name Sample

Enabled □

This is the solution description.

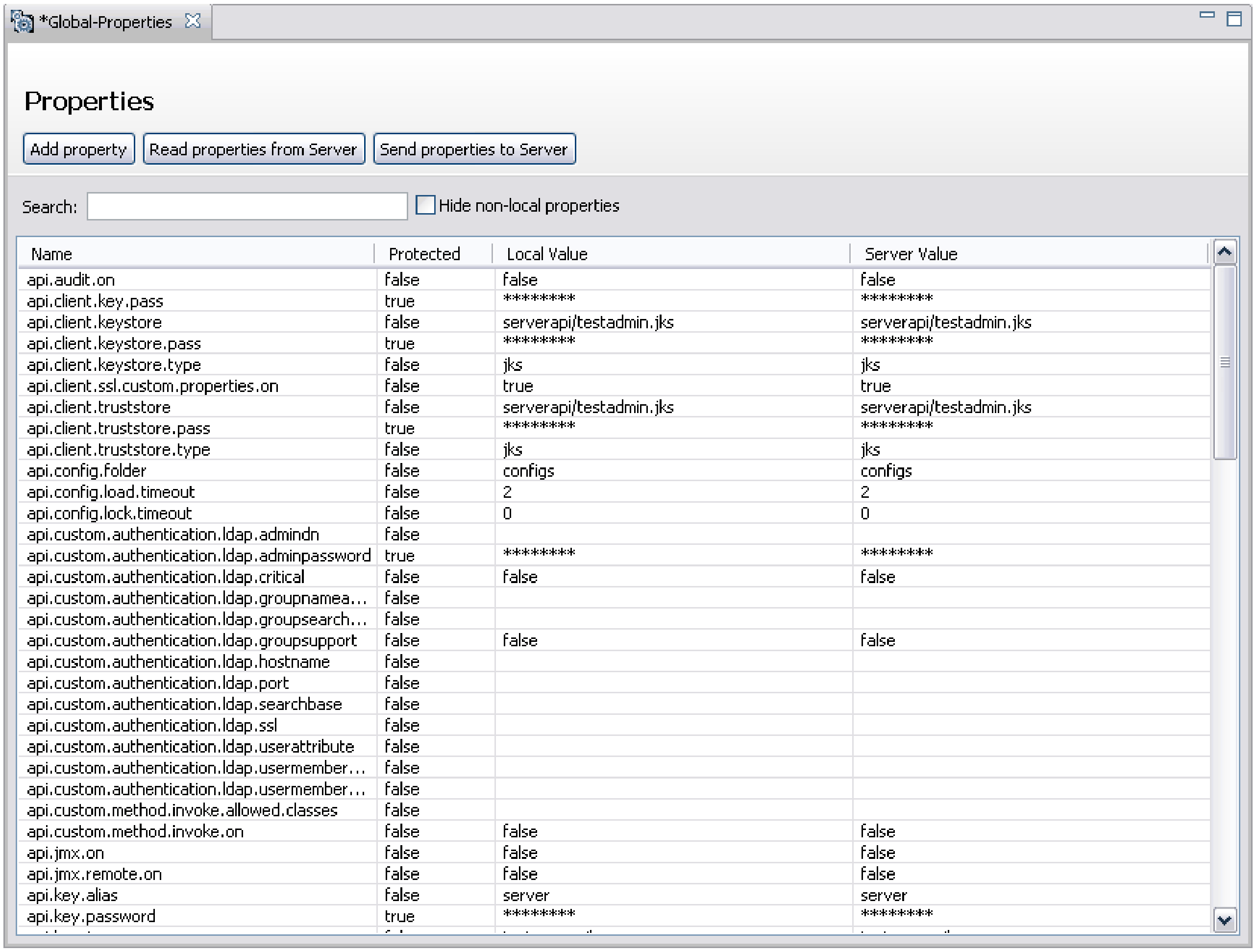
Figure 105. Solution Interface settings: Description

Server Properties

You can edit properties for a project using the Properties editor.

Create a new properties file using the **File** > **New** > **Properties** wizard and type the name of a property store.

In the properties editor you can exchange the contents of the property store using the **Download** and **Upload** commands:



Properties

Search:

□ Hide non-local properties

Name	Protected	Local Value	Server Value
api.audit.on	false	false	false
api.client.key.pass	true	********	********
api.client.keystore	false	serverapi/testadmin.jks	serverapi/testadmin.jks
api.client.keystore.pass	true	********	********
api.client.keystore.type	false	jks	jks
api.client.ssl.custom.properties.on	false	true	true
api.client.truststore	false	serverapi/testadmin.jks	serverapi/testadmin.jks
api.client.truststore.pass	true	********	********
api.client.truststore.type	false	jks	jks
api.config.folder	false	configs	configs
api.config.load.timeout	false	2	2
api.config.lock.timeout	false	0	0
api.custom.authentication.ldap.admindn	false		
api.custom.authentication.ldap.adminpassword	true	********	********
api.custom.authentication.ldap.critical	false	false	false
api.custom.authentication.ldap.groupnamea...	false		
api.custom.authentication.ldap.groupsearch...	false		
api.custom.authentication.ldap.groupsupport	false	false	false
api.custom.authentication.ldap.hostname	false		
api.custom.authentication.ldap.port	false		
api.custom.authentication.ldap.searchbase	false		
api.custom.authentication.ldap.ssl	false		
api.custom.authentication.ldap.userattribute	false		
api.custom.authentication.ldap.usermember...	false		
api.custom.authentication.ldap.usermember...	false		
api.custom.method.invoke.allowed.classes	false		
api.custom.method.invoke.on	false	false	false
api.jmx.on	false	false	false
api.jmx.remote.on	false	false	false
api.key.alias	false	server	server
api.key.password	true	********	********

Figure 106. Properties editor window

Use **Download** to retrieve all properties from the property store (for example, the properties file assigned to this store). Note that these values are read and passed to the CE from the Tivoli Directory Integrator Server currently selected for the project. Conversely, the **Upload** button updates the property store (via the current server) with the values in the editor. Only those properties that have a local value are updated.

Use the **Search** text field to show properties matching the text in this field. Checking **Hide non-local properties** causes the editor to only show those properties that have a local value.

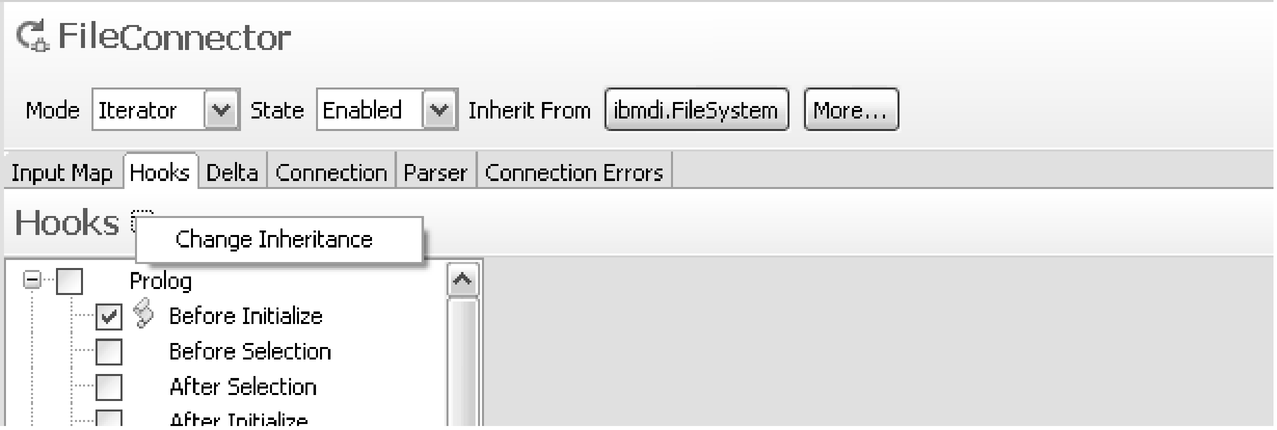
The project builder will include the property store configuration in the runnable configuration file. However, the property values in this document are only transferred as needed.

Note: The order in which Properties stores are initialized and accessed in the Tivoli Directory Integrator Server is undefined. Therefore, it is not possible to reliably store in a Property store any properties that define access parameters (for example, filenames) of other Property stores.

Inheritance

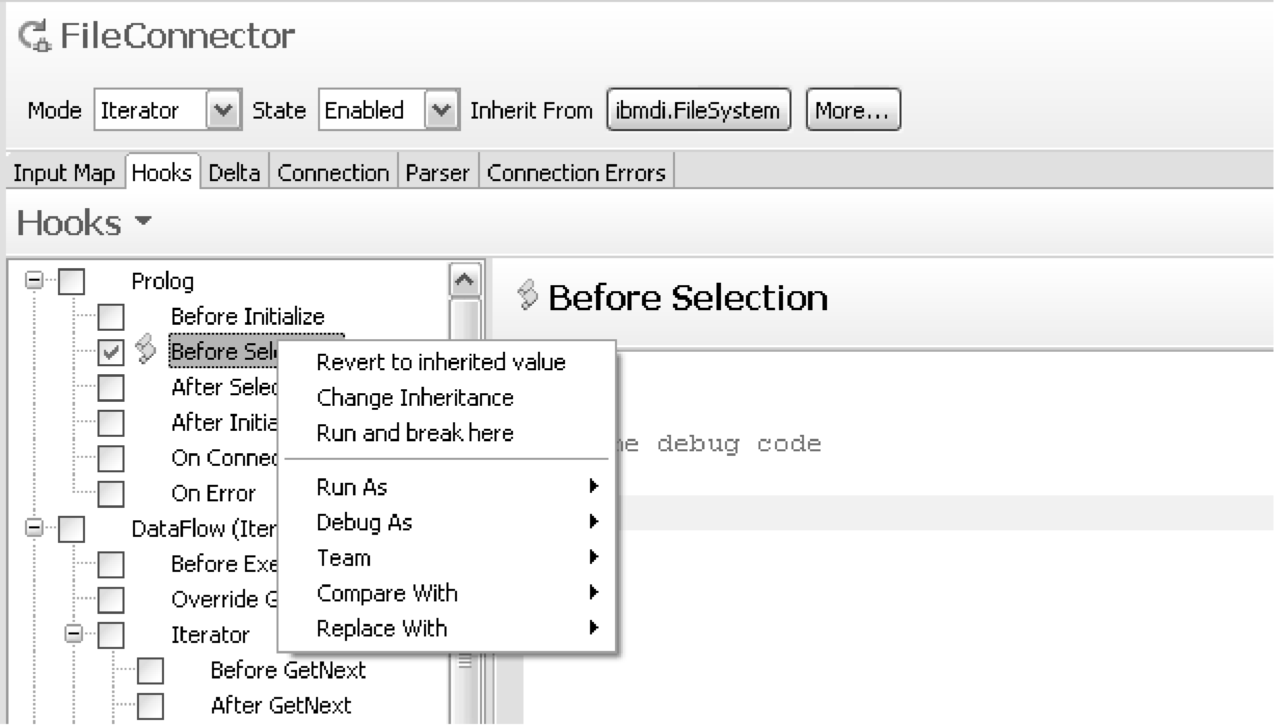
Components can inherit elements from other components by dragging and dropping an object onto the title bar of a component or subsection of the component.

For subsections there is also a menu choice (**Change inheritance**) available on the title bar.



Mode Iterator State Enabled Inherit From ibmdi.FileSystem

For hooks and attribute maps there is a separate inheritance selection on individual items where you can choose to inherit from a script or function component in the workspace.



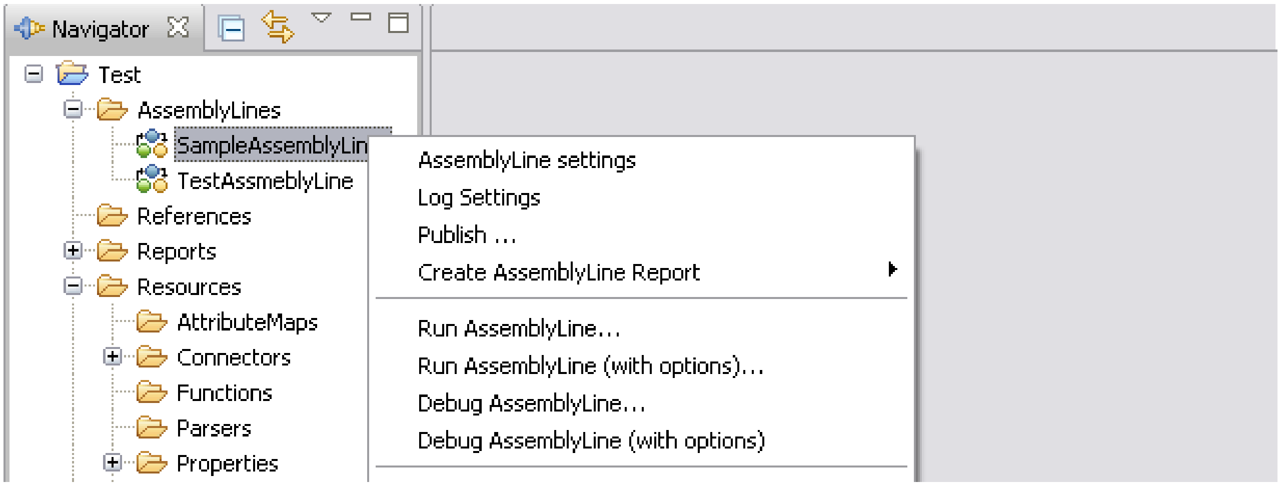
Mode Iterator State Enabled Inherit From ibmdi.FileSystem

When a configuration item overrides an inherited item the user interface offers a way to revert to the inherited value through a dropdown menu action item: **Revert to inherited value**.

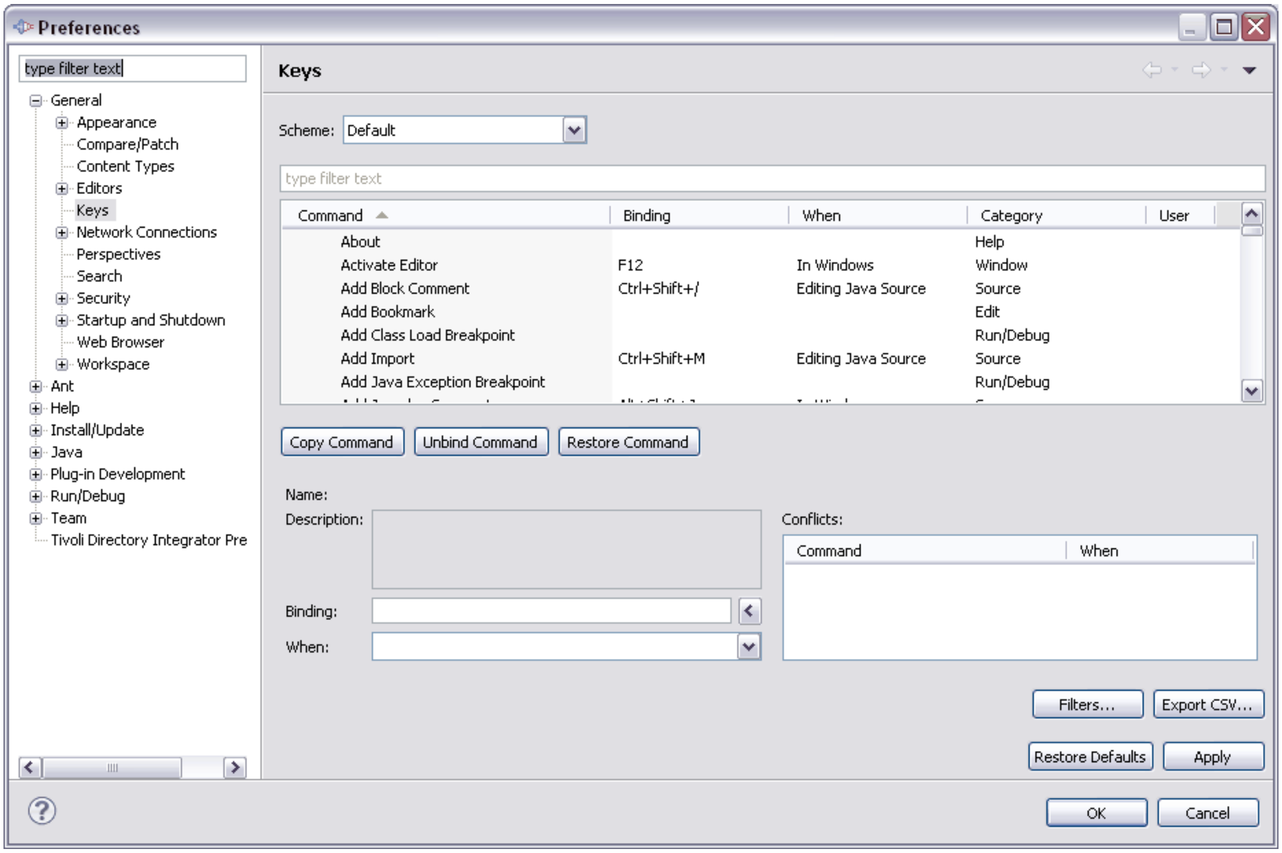
Actions and Key Bindings

The CE contributes a number of actions to itself as well as to objects in the workbench. These actions perform specific operations on specific objects.

For example, **Run AssemblyLine Report** is an action that is contributed to all files with an extension of `.assemblyline`. When you right-click an AssemblyLine in the navigator, the drop-down menu will include this command as well as all other contributions to this type of object.



These actions also have an associated command definition. A command definition lets the user define the keyboard shortcut for a command. This is done in the **Window** > **Preferences** > **Keys** panel:



Scheme: Default

Command	Binding	When	Category	User
About			Help	
Activate Editor	F12	In Windows	Window	
Add Block Comment	Ctrl+Shift+/	Editing Java Source	Source	
Add Bookmark			Edit	
Add Class Load Breakpoint			Run/Debug	
Add Import	Ctrl+Shift+M	Editing Java Source	Source	
Add Java Exception Breakpoint			Run/Debug	

Name:

Description:

Binding:

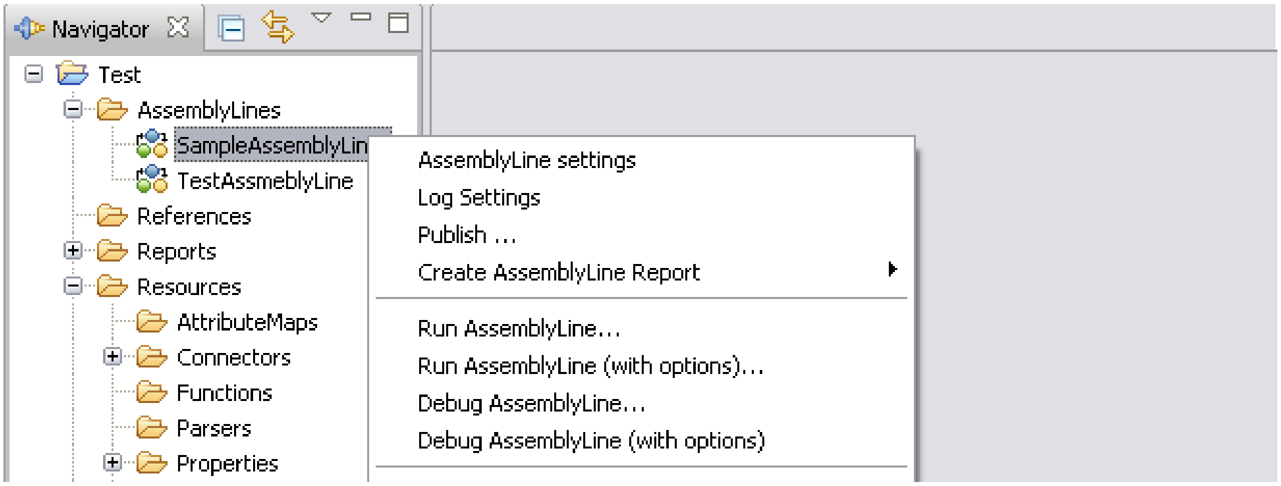
When:

Conflicts:

Command	When

Figure 107. Key Assignments window

The picture above shows how keyboard assignments are done in the user interface. In this example, the Run report command has been assigned `Alt+Shift+I` as its shortcut. When you open the menu again on the AssemblyLine you will see this reflected in the menu.



You can obtain a list of all specific Tivoli Directory Integrator commands by entering the text `tivoli directory integrator` in the search field, which can be found underneath the **Scheme** selector.

Chapter 4. Debugging features in TDI

Once you have created your IBM Tivoli Directory Integrator solution, usually using the Config Editor (CE), there are a number of ways you can put it through its paces and test it. Some of the debugging features TDI offers are available from within the CE, and some are based on scripts that are available in the TDI installation directory.

Debugging features in IBM Tivoli Directory Integrator are the Sandbox, AssemblyLine Simulation Mode and the stepper and debugger.

The stepper and debugger are part of the Configuration Editor; see "The stepper and debugger" on page 157.

Sandbox

IBM Tivoli Directory Integrator includes a Sandbox feature that enables you to record the operation of one or more Connectors in an AssemblyLine for later replay without the necessary data sources being available.

The Sandbox feature uses the System Store. See Chapter 6, "System Store," on page 209 for more information.

Recording a component means that the AssemblyLine will intercept every call to the connector instance used by the component. Recording a connector component for example, will record all calls to its connector instance methods like `selectEntries`, `getNextEntry` and so forth. The result of each of those calls is recorded in the configured sandbox database. The information recorded is the return value or the exception thrown by the connector.

To run a test session, create an AssemblyLine and add a file system connector reading data from a file. Add a script component that dumps the work entry. Then check the file system connector in this panel and select **Run/Record** on the Run button menu. After you have done that you can move the file it just read, and run the AssemblyLine in Playback mode. You should see the same log output even though the file connector would normally abort since the file is no longer accessible.

This feature can be very useful when providing support materials. Often, the time to reproduce the environment for an AssemblyLine and the state of data sources to reproduce a condition can be quite comprehensive. With a sandbox database with a recorded session, a support person can run the AssemblyLine without having access to all data stores the AssemblyLine requires. In addition, the AssemblyLine configuration can be modified to print out more information if that is necessary. The only change that cannot be done to the AssemblyLine configuration is to make additional calls or reorder the calls to recorded components. This would cause an error during playback as calls to the connector would not match the next expected call to the connector.

Before you can record or replay an AssemblyLine, you must first tell Tivoli Directory Integrator where to store the AssemblyLine recording data. This is done in the Sandbox window, which you can open by selecting **AssemblyLine Settings...** > **Sandbox Settings** in the AssemblyLine Editor. At the top of this window is a field labeled **Database** where you can enter the directory path for the system to use.

The Sandbox facility is not supported in AssemblyLines containing a Connector in Server mode, or an Iterator Connector with Delta enabled. The server will abort the running of the AssemblyLine if this is discovered.

Recording AssemblyLine input

Recording a component means that the AssemblyLine will intercept every call to the connector instance used by the component. Recording a connector component for example, will record all calls to its connector instance methods like `selectEntries`, `getNextEntry` and so forth. The result of each of those call are recorded in the configured Sandbox database. The information recorded is the return value or the exception thrown by the connector.

To run a test session, create an AssemblyLine and add a file system connector reading data from a file. Add a script component that dumps the work entry. Then check the file system connector in this panel and select **Run/Record** on the **Run** button menu. After you have done that you can move the file it just read and run the AssemblyLine in Playback mode. You should see the same log output even though the file connector would normally abort since the file is no longer accessible.

Sandbox playback of AssemblyLine recordings

When an AssemblyLine is in Sandbox mode, all the Connectors set for playback are said to be in *virtual mode*. This means that their Connector Interface operations (for example, `getNext()`, `findEntry()`) are not actually called. Instead, these operations are simulated during playback.

In order to run an AssemblyLine in Sandbox Playback Mode, you must select the Connectors to be run in virtual mode by the corresponding **Playback Enabled** check box in the AssemblyLine **Sandbox** settings window.

Note: Not all recorded Connectors need to be enabled for playback. You can enable them to access live data sources, although this might affect the results of the playback operation.

To run an AssemblyLine from the command line, start the server with the **-q2** switch. An AssemblyLine in Sandbox mode runs with input (including its Initial Work Entry) coming from a recorded set of data. For example, if you have a Java Messaging Service (JMS) Connector in your AssemblyLine in Sandbox mode, the JMS Connector retrieves input from the previously recorded data and is never actually initialized.

When recording an AssemblyLine, the server creates a Derby database in the specified **Database** directory using the AssemblyLine name as the database name. This database contains tables for each Connector in the AssemblyLine. An AssemblyLine in Sandbox mode can have one or more of its virtual Connectors replaced by renaming the recorded Connector and then adding a new one with its original name.

AssemblyLine Simulation Mode

AssemblyLines can be debugged without actually exchanging data with connected systems using the AssemblyLine Simulation Mode. When an AssemblyLine has been started in this mode all AssemblyLine Components which potentially could cause changes in the target systems will be skipped.

This implies that the AL executes normally, but Connectors in Update, Delta, Delete and AddOnly Mode do not perform the actual operation.

Note: An AssemblyLine run this way can only approximate what would happen in normal mode; in many cases the very fact that a connected system does not receive any updates in Simulation Mode may cause your business logic to behave differently, negating the usefulness of the simulation.

The simulation state should not to be confused with the state of a component. The component's state has higher priority than the component's simulation state.

The component's state has two values – Enabled or Disabled. If the component is in Enabled state then it will initialize and its simulation state will be checked when the appropriate operation is called. When the state is set to Disabled then no check for the simulation state will be made since no operation will be called, and the component's initialization will not be called.

Connectors and FCs have another state called Passive. When their state is set to Passive then they only will be initialized. Their operations can be executed only via a user script. When their specific operation is called then the simulation state is checked.

There are a number of ways in which an AssemblyLine can be started in Simulation Mode:

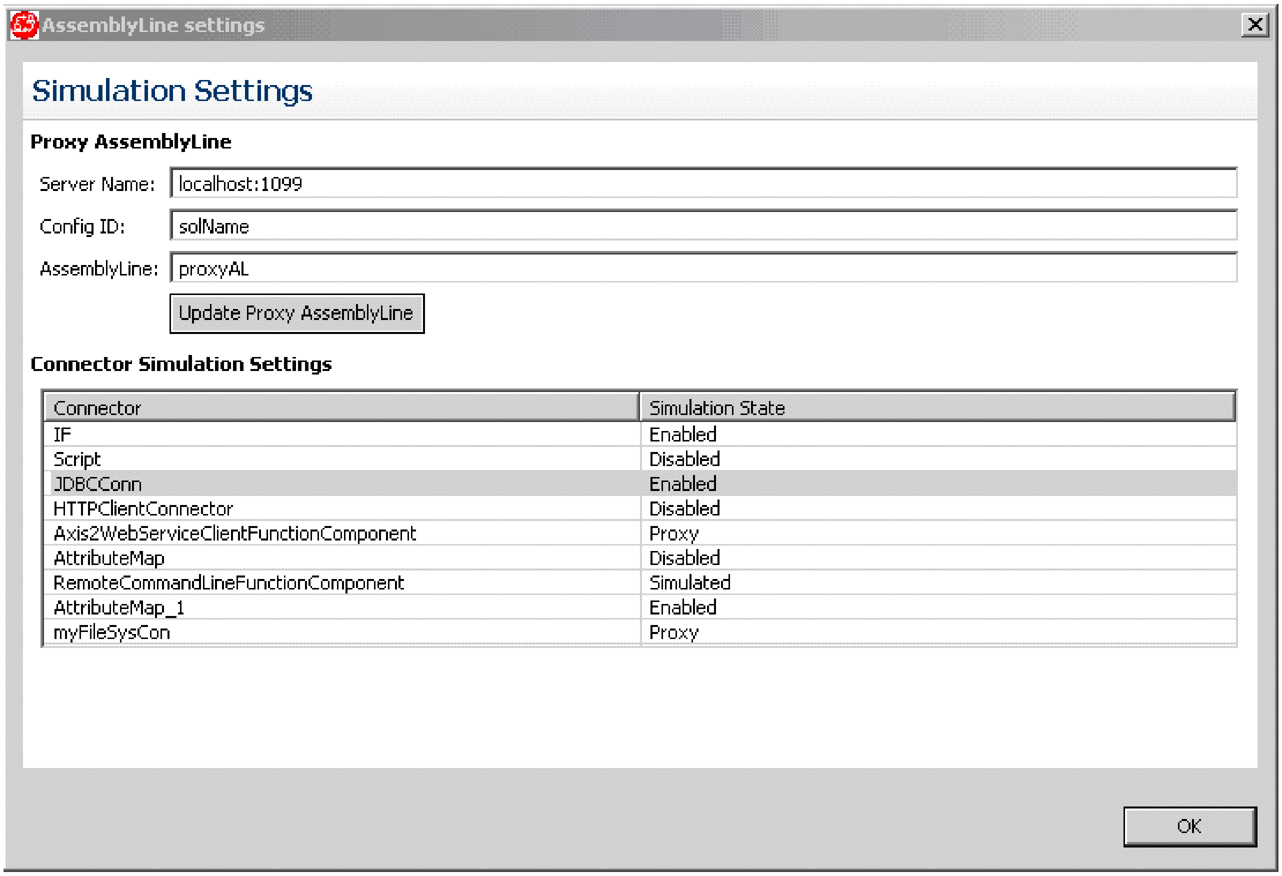
From the Configuration Editor:
A checkbox in the Run-mode drop-down menu is available to enable and disable simulation for the AL. You must choose the desired run-mode Run with Options from the drop-down menu and check the checkbox **Simulation mode** in the pop-up Options dialog box in order to make the AL simulate while running. The default state for this checkbox is unchecked.

Using the Server startup command `ibmdisrv`:
This command recognizes the the switch `–M`, and will start the AL with simulation turned on if this switch is provided.

Using the API:
In order to make an AL run in Simulation Mode, you must set the property `AssemblyLine.TCB_SIMULATE_MODE` to true in the TCB object. This object then should be provided to the `startAssemblyLine(String, TaskCallBlock)` method. If no property is set then by default its value is considered to be false.

In order to run an AssemblyLine in Simulate Mode, a new configuration is created. It is a child of the AssemblyLine Config. This configuration object contains parameters that configure the connection to an AL that will be used as a Proxy AssemblyLine (ProxyAL). The SimulationConfig object has a method that creates or updates a template of a ProxyAL based on the components' states of the AssemblyLine that is being simulated. The SimulateConfig object also contains all the hooks defined for Components that are in Scripted simulation state. The name of each hook is the same as the name of the component in Scripted simulation state. It also contains the simulation state for each component.

AssemblyLines to be run in Simulation Mode can be configured in more detail; relevant settings can be configured by selecting **AssemblyLine Settings** > **Simulation Settings** in the AssemblyLine Editor window.



Simulation Settings

Proxy AssemblyLine

Server Name: localhost:1099

Config ID: solName

AssemblyLine: proxyAL

Connector Simulation Settings

Connector	Simulation State
IF	Enabled
Script	Disabled
JDBCConn	Enabled
HTTPClientConnector	Disabled
Axis2WebServiceClientFunctionComponent	Proxy
AttributeMap	Disabled
RemoteCommandLineFunctionComponent	Simulated
AttributeMap_1	Enabled
myFileSysCon	Proxy

Figure 108. Simulation Settings window

The configuration of the simulation states for each component is done using the Simulation Settings dialog box. This dialog box configures the ProxyAL to be used by the components which simulation state is set to *ProxyAL*. When you click **Update Proxy AssemblyLine** the Config Editor will either create a new ProxyAL in the current project or will update an existing Proxy AssemblyLine. The created or updated ProxyAL is provided as a template and its structure is based on the configuration you have done in the Simulation Settings dialog box. Note that in this process only the name of the Proxy AssemblyLine is taken into account. The Server Name and Config ID are taken into account during the execution of the Simulated AssemblyLine. If there is already an existing AssemblyLine with the name specified in the dialog box, then only new branches will be added and no old branches will be modified or removed. This is because some of them can contain a user-specific configuration. Individual components in the AssemblyLine can be set to one of the following states:

Enabled

This is the equivalent of running this component in the AL in normal mode (that is, the component is executed as it normally would).

Disabled

This is the equivalent of disabling the component (that is, no operations, hooks, or anything else except the initialization is executed for the component).

Simulated

Generally, the statistics for all of the components described below will contain information for the operation that would have completed if the AL was not simulating. Since no critical operation is done during simulation, there is no possible way to predict what would be the result from the execution of a critical operation (success or error), thus the statistics will state that the operation has completed successfully.

- Connectors in AddOnly mode: executed as normal, only the potentially unsafe call to the `connector.putEntry()` method and the call to the `override_add` hook are skipped.
- Connectors in Update mode: executed as normal, only the potentially unsafe call to the `connector.modEntry()` method and the call to the `override_modify` hook are skipped.
- Connectors in Delete mode: executed as normal, only the potentially unsafe call to the `connector.deleteEntry()` method and the call to the `override_delete` hook are skipped.
- Connectors in Delta mode: executed as normal, but because of the fact that this connector relies on calls to the methods described above, it will be simulated when the above methods and hooks are skipped.
- Connectors in Iterator mode with Delta tagging: executed as normally; for these components the change is similar to the Connectors described above, that is, the calls to the potentially unsafe methods of the BTree class (`putEntry`, `modEntry`, `deleteEntry`) are skipped. For the CDDeltaTaskComponent the commit state is overridden to disable commit ("No autocommit") but committing will still be possible for users that explicitly call the `CSDeltaTaskCoponent#commitDeltaState()` method.
- Function Components (FCs): execute as normal, but the invocation of "`before_functioncall`", "`after_functioncall`" and "`no_reply`" hooks as well as the call to the `function.perform()` method are disabled. The invocation of these hooks is disabled because they are associated with an object that is returned from the perform method which is also disabled. The only thing that will be done is a change to the statistics that will indicate that the FC has successfully executed this operation.

  Also the Input and Output Maps will be executed but the actual entry before the InputMap will be an empty one (this could lead to an exception being thrown, it is better to disable each simple attribute mapping from the InputMap that relies on an attribute returned from the Function Component or to add a check for validity of the retrieved attribute in the advanced attribute Map; alternatively, you can use the NullBehaviour mechanism to override potential errors).
- Connectors in any other mode run as usual.

Proxy Connectors in this simulation state will start another AL (specified in the Simulation tab) that will override the potentially unsafe execution of the operation. This external AL is shared between all the Components in this simulation mode. It will be executed with different operation which name matches the name of the Component being simulated. See "Proxy AssemblyLine workflow" on page 198.

Scripted

A user-defined script will override the potentially unsafe operation. Each component in this simulation mode has its own hook, unlike the Proxy state where an AL is shared between the components. See "Simulation script workflow" on page 199.

You have the ability to check whether the AL is simulating and also to switch simulation on and off using the following methods:

- `boolean AssemblyLine#isSimulating()` Used as follows from a hook: task.isSimulating();
- `void AssemblyLine#setSimulating(boolean)` Used as follows from a hook: task.setSimulating(true);

You have the ability to check the state of each component, and to set it dynamically by using the following methods:

- `String AssemblyLineComponent.getSimulatingState()` Used as follows from a hook: ConnectorName.getSimulatingState();
- `void AssemblyLineComponent.getSimulatingState(String)` Used as follows from a hook: ConnectorName.setSimulatingState("Proxy");

Note: Only Connectors and FCs have the full set of simulation states as described above, the remaining components (and Connectors in Server mode) have only Enabled and Disabled states.

Tivoli Directory Integrator considers some FCs safe and their default simulation state will be Enabled, that is by default they will not simulate and will execute as usual; these are all FCs that are unable to change a target system since they simply do not connect to any. The list of safe FCs is as follows:

- CBE FC
- JavaToXML FC
- XMLToJava FC
- SDOToXML FC
- XMLToSDO FC
- Parser FC
- MemQueue FC
- JavaToSOAP FC
- SOAPToJava FC
- WrapSOAP FC

Attention: You still can cause changes to underlying datasystems by explicit coding; this is out of scope of the Simulation Mode.

Any remaining FCs not listed here are considered potentially unsafe and their default simulate state will be set to Simulated.

Proxy AssemblyLine workflow

In the context of Simulation Mode, the call to a proxy AL works similar to how you would use the AssemblyLine Connector to drive an AL of your choice, however some significant differences are observed.

The call to the override hook of the specific operation is disabled when the component is in Proxy simulation state.

The table below shows the methods that are called when the Connector is in one of those Modes or when the component is a Function component.

Table 10. Method invocation according to mode

Mode	Method
Connector: AddOnly	putEntry
Connector: Update	findEntry, modEntry, putEntry
Connector: Delete	findEntry, deleteEntry
Connector: Delta	findEntry, modEntry, putEntry, deleteEntry
Connector: Iterator	selectEntries, getNextEntry
Connector: CallReply	queryReply
Connector: Lookup	findEntry
Function component	perform

When the time for the calling of a specific method arrives and the component is in Proxy simulation state then the call to that method will be delegated to the proxy AL. When the proxy AL is started an op-entry is passed to it with the following attributes:

- $operation – this attribute contains the name of the operation to be executed. When the proxy AL is called instead of a component's specific method then the value of this attribute will be the same as the name of the component.

- $method – this attribute contains the name of the method that would be normally called, but since the component is in Proxy simulation state and Components in some modes (for example, Update) execute several methods before actually do the modification (that is, findEntry) then the proxy AL must recognize which method to implement. This $method attribute just tells the proxy AL which method it is executed instead so the proxy AL can handle the operation properly.
- search – this attribute is available when the $method attribute is findEntry. Its value is an object of type SearchCriteria and represents the Search Criteria defined by the user. For example if the component is in Proxy simulation state and its mode is Update, then a Search Criteria must be defined in order to be able to update the right entry on the target system. Since the simulation state is Proxy, the actual lookup operation that takes place before the modify operation delegates its execution to the proxy AL. Then the proxy AL uses this Search Criteria for proper lookup simulation.
- current – this attribute is only available when the $method attribute is modEntry. Its value is an entry object that represents the entry found in the target system before the actual modification occurs. This is the entry that the findEntry method, executed before modEntry method, returns.

An Initial Work Entry (IWE) is passed to the proxy AL when it is called. When the $method is findEntry, selectEntry or getNextEntry the IWE is a copy of the work entry from the calling AL. In any other case the IWE is the entry retrieved from the OutputMap procedure (a.k.a. the *conn* entry). In particular, for the deleteEntry operation the IWE is the entry retrieved from the preceding findEntry operation.

After the proxy AL execution is done and the $method is findEntry, the result entry is checked for the attribute *conn*. If it is available then it is assumed that this attribute contains all the entries found from the findEntry operation and according to its value, the appropriate hooks will be called, that is, no_match and multiple_match. If no attribute with the name conn is found then the result entry of the proxy AL execution is treated as the entry found by the findEntry $method. The entry retrieved from the proxy AL that overrides the selectEntries $method is automatically merged with the work entry of the calling AL. The entry retrieved from the proxy AL that simulates those methods that expect an entry (that is, findEntry, getNextEntry, queryReply and perform) is sent to the defined InputMap. For all other methods that are not expected to return a result, the entry from the proxy AL is ignored.

Simulation script workflow

Each component which simulation state is set to Scripted has a Simulation Script (SS) defined in the Simulate tab.

Here is list of objects exposed to you for direct use from a SS:

- *work* – the work entry.
- *conn* – the entry retrieved from the lookup operation or directly from the OutputMap or null.
- *resEntry* – the entry that is used as a result if the operation being simulated requires a result to be returned; otherwise if the result will not be used, it is null.
- *current* – the entry found in the target system that will be modified or null.
- *search* – the SearchCriteria object defined by the user or null.
- *method* – a String object containing the name of the method that is being override by the SS.

The table below explains the methods being simulated, it shows when the SS will be invoked.

Table 11. Method invocation according to mode

Mode	Method
Connector: AddOnly	putEntry
Connector: Update	modEntry, or putEntry if the entry couldn't be found
Connector: Delete	deleteEntry
Connector: Delta	modEntry, putEntry or deleteEntry (depends on inner logic)

Table 11. Method invocation according to mode (continued)

Mode	Method
Connector: Iterator	getNextEntry
Connector: CallReply	queryReply
Connector: Lookup	findEntry
Function component	perform

If a Connector is in Server mode with a Scripted simulation state it would still require to receive a request from a client. The SS will be called when the response is to be sent.

The internal lookup operation that is executed for Connectors in Update, Delete and Delta mode will be done internally and it won't allow overriding from a SS like it does for the proxy AL.

Connectors in this simulation state will not execute the operation's override hook if any.

Chapter 5. Easy ETL

The EasyETL perspective in the Configuration Editor is a highly specialized way of looking at your IBM Tivoli Directory Integrator configurations, dedicated to get you up and running fast with projects that revolve around simple tasks to Extract, Transform and Load data (ETL) into some form of database.

The EasyETL perspective shows EasyETL projects and lets you run, open and create new ETL projects. The ETL perspective can be shown by choosing **Window** > **Open Perspective** and then selecting the Easy ETL perspective.

To start the CE with this perspective add `-perspective com.ibm.tdi.rcp.perspective.etl` to the CE command line (`ibmditk`).

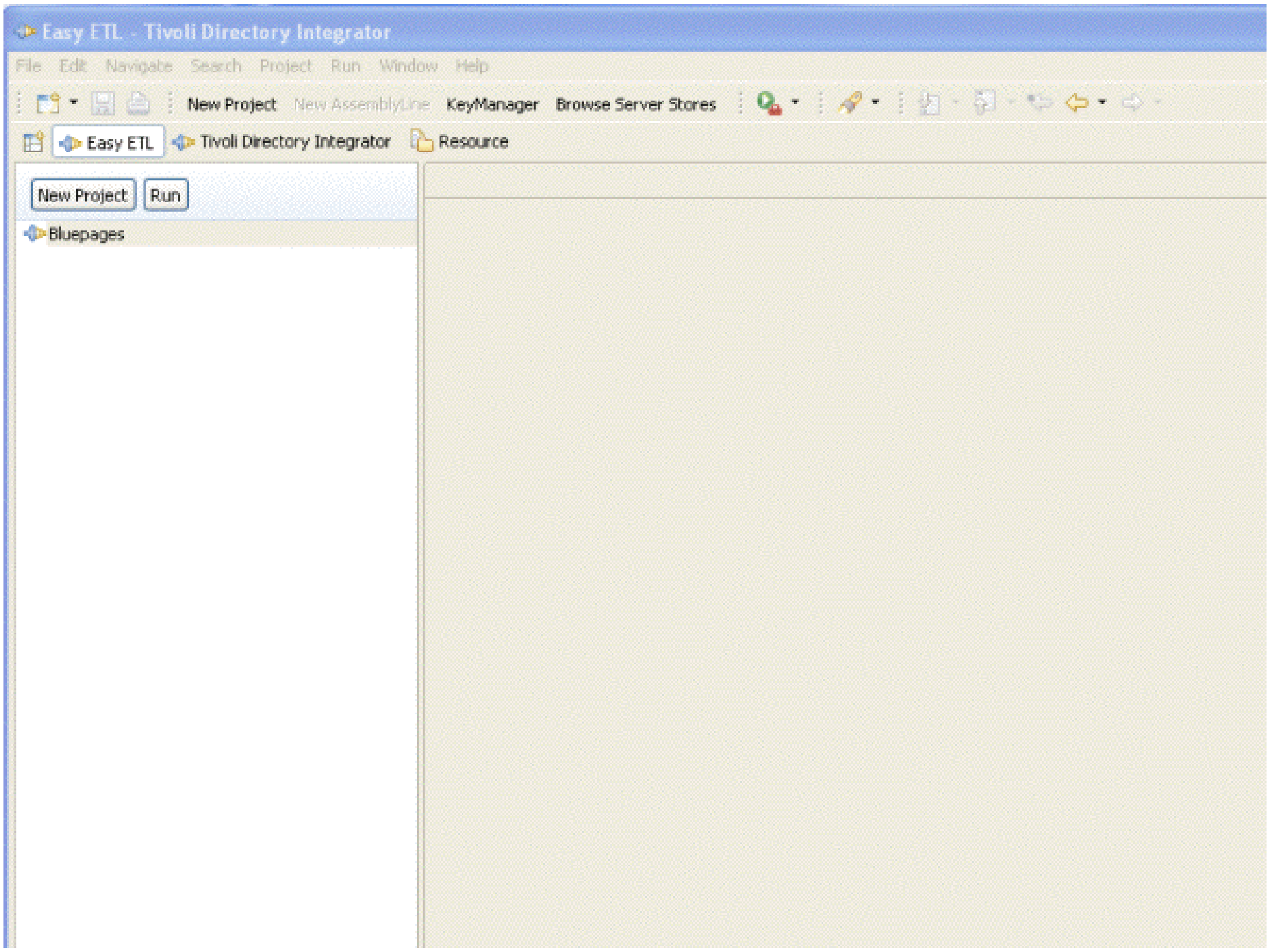

Easy ETL - Tivoli Directory Integrator
File Edit Navigate Search Project Run Window Help
New Project
New AssemblyLine
KeyManager
Browse Server Stores
Easy ETL
Tivoli Directory Integrator
Resource
New Project
Run
Bluepages

Figure 109. EasyETL main window

Use the **New Project** button to create a new EasyETL project. An EasyETL project is a normal TDI project with a single AssemblyLine with two connectors. Double click or select the project and press the Enter key to open the ETL editor.

The context menu for an ETL project has the following items:

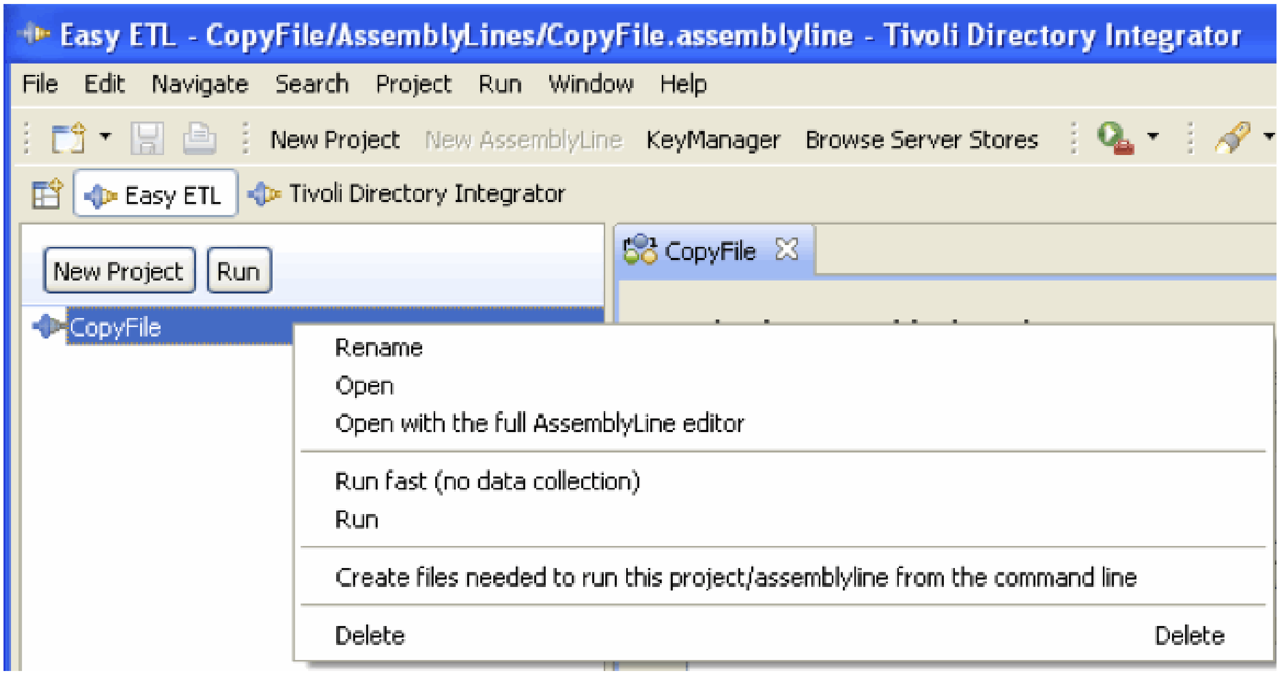
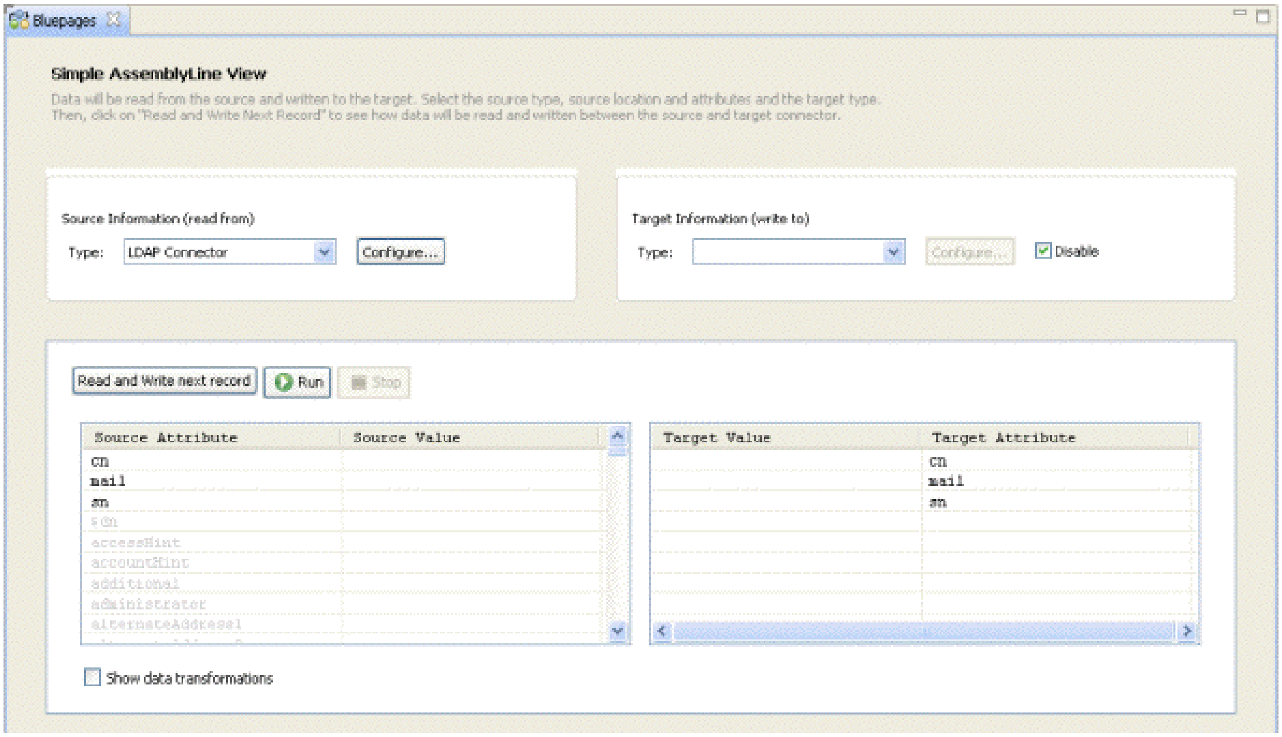

Easy ETL - CopyFile/AssemblyLines/CopyFile.assemblyline - Tivoli Directory Integrator
File
Edit
Navigate
Search
Project
Run
Window
Help
New Project
New AssemblyLine
KeyManager
Browse Server Stores
Easy ETL
Tivoli Directory Integrator
New Project
Run
CopyFile
CopyFile
Rename
Open
Open with the full AssemblyLine editor
Run fast (no data collection)
Run
Create files needed to run this project/assemblyline from the command line
Delete
Delete

Figure 110. EasyETL project context menu

- **Open** – Open the project in the editor
- **Open with full AssemblyLine Editor** – Open the project in the advanced AssemblyLine editor
- **Run fast** – Run the project without collecting data from the AssemblyLine
- **Run**– Run the project and display collected data in the Data-Collector view
- **Create files ...** – Generate files needed to run the project from the command line
- **Rename** – Rename the project
- **Delete** – Delete the project

The EasyETL editor shows the two connectors with a table that shows the mapping between the two connectors. The initial screen shows an empty selection for both connectors as shown in this picture:



Simple AssemblyLine View

Data will be read from the source and written to the target. Select the source type, source location and attributes and the target type.
Then, click on "Read and Write Next Record" to see how data will be read and written between the source and target connector.

Source Information (read from)

Type: LDAP Connector

Target Information (write to)

Type:

☑ Disable

Source Attribute	Source Value
cn	
mail	
sn	
$dn	
accessHint	
accountHint	
additional	
administrator	
alternateAddress1	

Target Value	Target Attribute
	cn
	mail
	sn

□ Show data transformations

Figure 111. Initial EasyETL project window

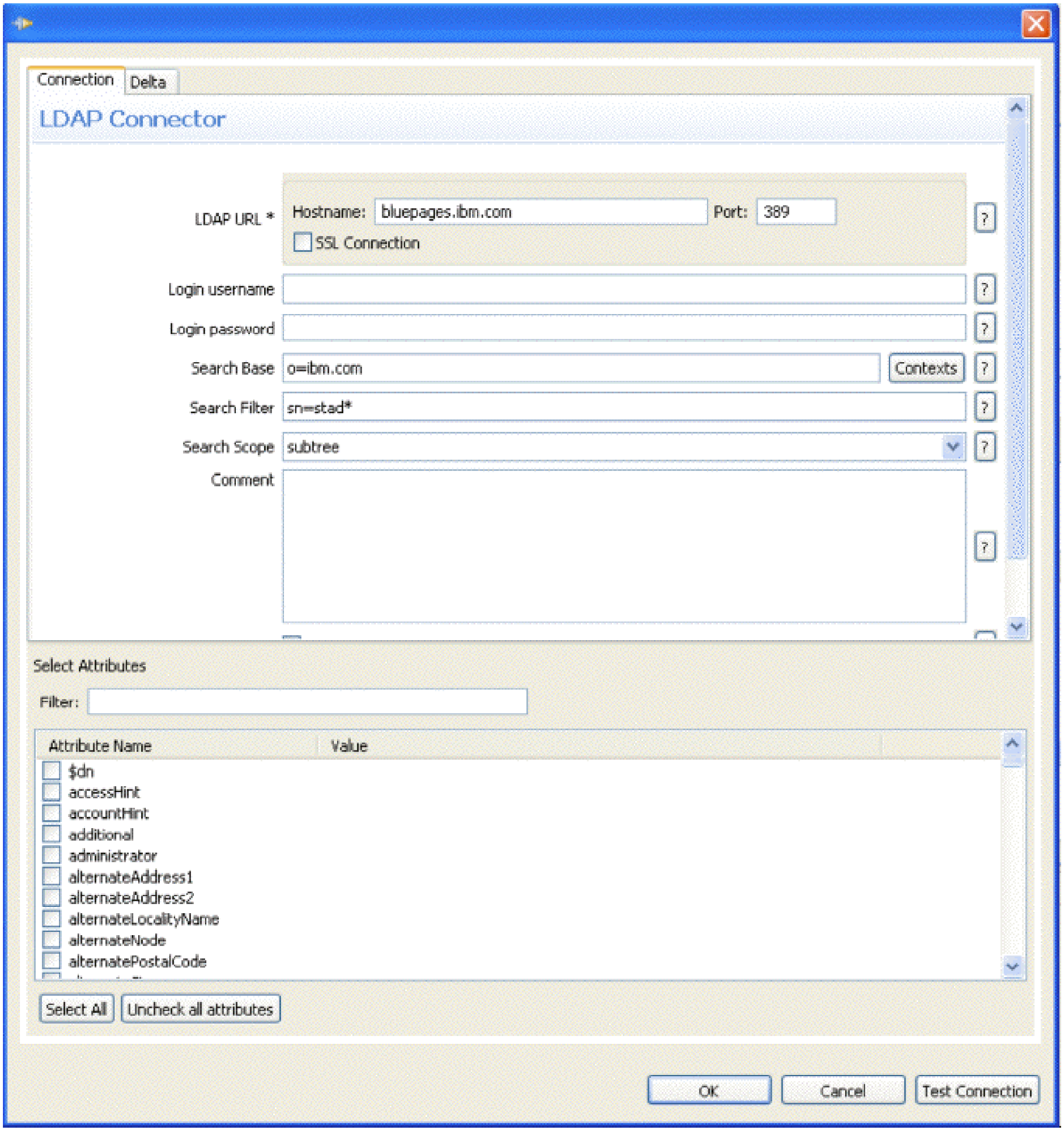
You then typically start by choosing the source connector. There are four options in the type dropdown:

- File System Connector
- LDAP Connector
- Database Connector (JDBC)
- Select Connector...

The top three are quick selections since they are commonly used. The last option, **Select Connector...** brings up the standard connector selection dialog where you can choose any connector you want. However, the list of connector is limited to those that implement the mode for the source (Iterator) and target (AddOnly) connector.

Once the connector is chosen you can configure the connector. The configuration dialog contains the forms to configure the connector and parser if it is required. In addition, the *Delta* screen is available for the source connector.

If you select the LDAP Connector you will see the following window:



LDAP URL *
Hostname: bluepages.ibm.com Port: 389
□ SSL Connection
Login username
Login password
Search Base o=ibm.com
Search Filter sn=stad*
Search Scope subtree
Comment

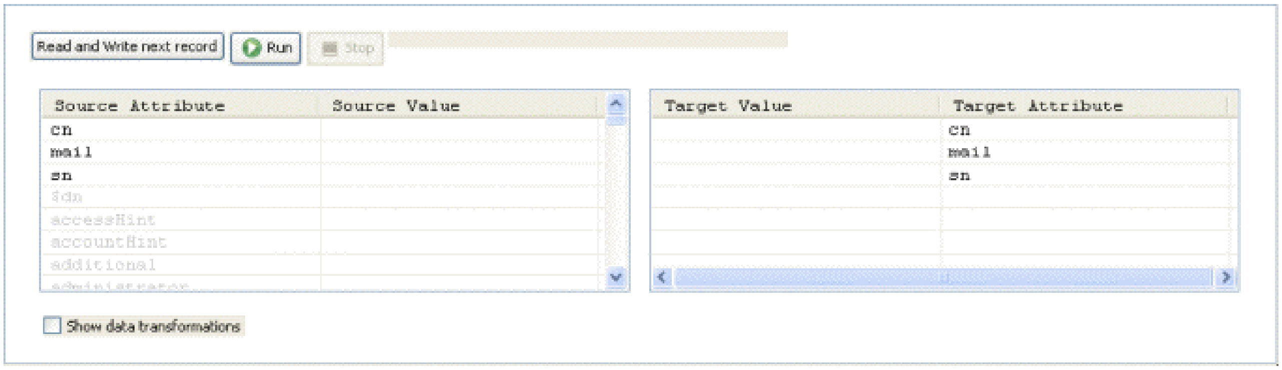
Select Attributes
Filter:

Attribute Name	Value
□ $dn	
□ accessHint	
□ accountHint	
□ additional	
□ administrator	
□ alternateAddress1	
□ alternateAddress2	
□ alternateLocalityName	
□ alternateNode	
□ alternatePostalCode	

Figure 112. LDAP Connector in Easy ETL

After discovering the attributes in the connector you can check those attributes you want to read in from the connector. For the target connector, only the list of schema items is present since the mapping is based on the attributes of the input connector.

Once the schema and attributes are available they will be shown in the source attribute column.



Source Attribute	Source Value
cn	
mail	
sn	
$dn	
accessHint	
accountHint	
additional	
administrator	

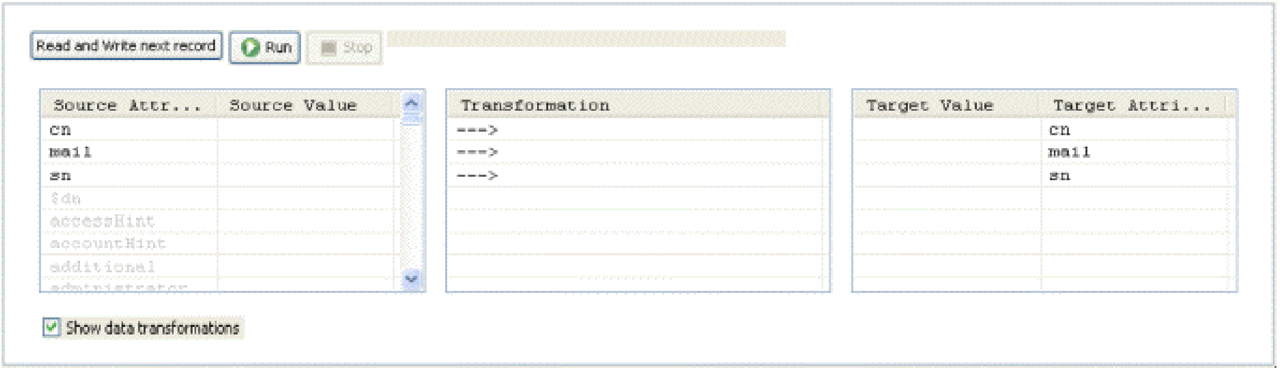
Target Value	Target Attribute
	cn
	mail
	sn

□ Show data transformations

Figure 113. Input/Output mapping

Items that are grayed out are attributes that are not mapped. Right click and choose **Map Attribute** to map the attribute. You can also double click or press the Enter key to map the current selection. In the target attribute column you can click and choose a different output attribute name. Conversely, you can do the same on a mapped attribute to return it to the list of unmapped attributes.

To customize the mapping between the two attribute check the **Show Transformations** checkbox. This will add a new table between the source and target tables.



Source Attr...	Source Value
cn	
mail	
sn	
$dn	
accessHint	
accountHint	
additional	
administrator	

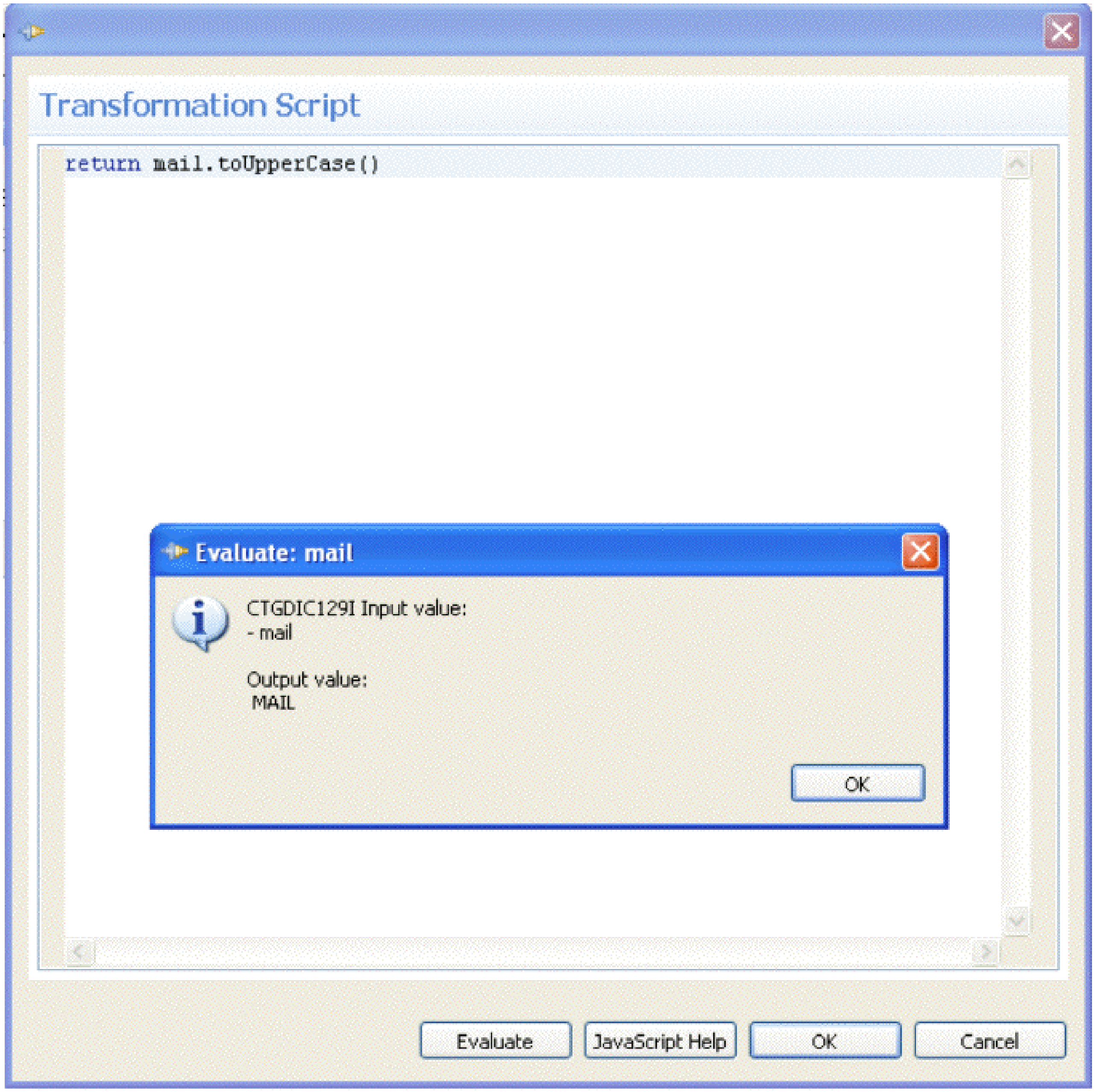
Transformation
--->
--->
--->

Target Value	Target Attri...
	cn
	mail
	sn

☑ Show data transformations

Figure 114. Input/Output mapping, with Transformations

The transformation table shown has arrows that denote verbatim copies between two attributes. Double click a transformation item to bring up the JavaScript editor for that map.



```
return mail.toUpperCase()
```

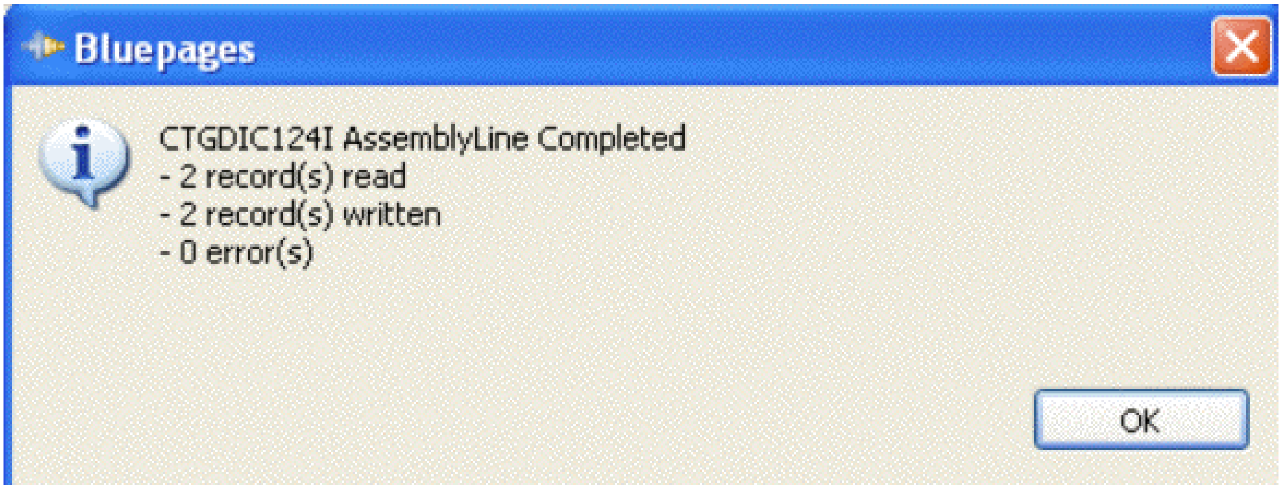
```
CTGDIC129I Input value:
- mail

Output value:
 MAIL
```

Figure 115. Transformation script

Enter the script that will perform the custom transformation of the value. Note that all mapped attributes in the source connector are available as top-level beans. This means that you can refer directly to `cn` instead of using the `work.cn` notation. The editor is also aware of the Java class based on what has been read by the *Read and Write Next Record* action. The last entry read is also used when you test the script with the **Evaluate** button so the evaluation of the script can be tested against real live data as shown in the picture above. The message shows the input value and the result of the transformation script (output). The **JavaScript Help** button is a quick way to access the help page for JavaScript.

The target connector has a checkbox labeled **Disable**. This checkbox, when checked, will disable the output connector and instead dump the entry (after custom transformations) to the console log.

After configuring the connectors you can now run the AssemblyLine either to completion or by stepping one record at a time through the AssemblyLine. When stepping through the AssemblyLine the table will reflect the last entry read and written to the target connector. When you click the **Run** button the AssemblyLine will execute continuously until it completes or you press the **Stop** button. When the stop button is pressed during execution, the AssemblyLine breaks and gives control back to you. When the stop button is pressed while you have control, the AssemblyLine terminates. When the AssemblyLine has terminated it shows a completion dialog with some statistics about the run:



CTGDIC124I AssemblyLine Completed
- 2 record(s) read
- 2 record(s) written
- 0 error(s)

Figure 116. Completion dialog

Chapter 6. System Store

The System Store addresses the various needs of IBM Tivoli Directory Integrator for persistent storage and by default uses the Apache Derby RDBMS (previously known as Cloudscape) as its underlying storage technology.

Other relational databases, like IBM DB2, can be used to hold the System Store. The System Store can be shared by multiple instances of Tivoli Directory Integrator servers if the Derby database runs in networked mode, or if a multiuser relational database system is used. If Derby runs embedded in a Directory Integrator server, it cannot be shared simultaneously with other servers.

The System Store implements three types of persistent stores for Directory Integrator components:
- "User Property Store"
- "Delta Store" on page 210
- Sandbox tables

Each store offers its own set of features and built-in behavior, as well as a callable interface that users can access from their scripts, for example, to persist their own data and state information.

You can configure the "System Store settings" on page 182 for a project by selecting the project in the Tivoli Directory Integrator Navigator, and select **Solution Logging and Settings**. Then select the **System Store** tab.

You can set up a JDBC Connector to directly access any of the tables in the System Store, although changing data in these tables must be avoided, as this can cause your solution to malfunction.

Attention: If you are running Derby embedded in TDI as opposed to running it in networked mode as a server, then be sure to **Close** the database again before trying to test or run your Config. Because the Config Editor starts up a separate instance of the server, running in its own JVM, the System Store is not available to this server. Closing the System Store Details window also closes your connection to the database.

Note: Although the Sandbox feature also uses the System Store technology, you specify a new database directory for each AssemblyLine.

User Property Store

The User Property Store is a System Store table used for maintaining serialized Java objects associated with a key value. This is where persistent component parameters and properties, such as the **Iterator State Store** are maintained, as well as data you store.

For example, when you set the **`Iterator State Store`** parameter for the Active Directory Change Detection Connector, you are specifying the key value that the Connector uses to save and restore the Iterator state. If you want the Iteration to start with the first (or last) change entry, simply delete the Iterator State Store entry in the User Property Store; that is, click **Delete** next to the parameter.

You can persist your own objects with the following `system` calls:

`system.setPersistentObject(`*`keyValue`*`,obj)`
Saves the object **obj** in the User Property Store using the specified *keyValue*. The object is returned if it was saved successfully, otherwise the function returns **null**.

`system.getPersistentObject(`*`keyValue`*`)`
Returns the object with the specified *keyValue* from the User Property Store. If the *keyValue* is not found, then the function returns NULL.

`system.deletePersistentObject(`*`keyValue`*`)`
Deletes the object with the specified *keyValue* in the User Property Store. This function returns the object that was deleted, or NULL if the *keyValue* was not found.

These methods access the default User Property Store.

However, you can create and use your own stores using the *Store Factory*.

If you view the User Property Store from the System Store window, note that it has the following table definition:

Key The unique key (512 chars)

Entry The object associated with the key

Note: Any object to be persisted in the User Property Store must be serializable.

Delta Store

The Delta Store is found under the **Delta Tables** folder in the System Store browser. Each table represents one **Delta Store** parameter setting (in the **Delta** tab of an Iterator). There are a number of classes and methods for working directly with the Delta Store, although this is not recommended. For more information on the Delta feature, see the section entitled Chapter 7, “Deltas,” on page 213.

Store Factory methods

The following examples are of methods that can be used with the Store Factory:

`public static PropertyStore getDefaultPropertyStore () throws Exception;`
Returns the default Property Store.

`public static PropertyStore getPropertyStore ( String table ) throws Exception;`
Returns the Property Store identified by name. Only one instance of a given name is present at one time.

@param name
The Property Store name.

@return
The Property Store object associated with name.

`public static String getSystemDatabaseURL ();`
Returns the System Store JDBC URL.

`public static Connection getConnection () throws Exception;`
Returns a connection object to the default database.

`public static Connection getConnection ( String database ) throws Exception;`
Returns a connection object to the named database with AutoCommit set to TRUE.

@param database
The database name.

`public static Connection getConnection ( String database, boolean autoCommit ) throws Exception;`
Returns a connection object to the named database.

@param database
The database name.

@param autocommit
The AutoCommit flag.

@return
A connection object to the named database.

`public static boolean dropTable ( Connection connection, String table );`
Drops a table in the database associated with connection.

@param connection
The connection object obtained by `getConnection()`.

@param table
The table to drop.

`public static void verifyTable ( Connection connection, String table, Vector sql ) throws Exception;`
Verifies that a table is accessible in the database.

@param connection
The connection object obtained by `getConnection()`. If null, a connection to the default table is obtained.

@param table
The table name to verify.

@param sql
A vector of SQL statements to create the table if it does not exist.

`public static Exception dropTable ( String tableName );`
Drops a table in the default database.

@param tableName
The name of the table to drop.

`public static byte[] serializeObject ( Object obj ) throws Exception;`
Serializes an object to a byte array.

@param obj
The object to serialize.

@return
The byte array containing the serialized object.

`public static Object deserializeObject ( byte[] array ) throws Exception;`
Deserializes a byte array into a Java object.

@param array
The byte array with the serialized Java object.

@return
The resurrected Java object.

Property Store methods

The following examples are methods that you can use with the Property Store:

`public Object setProperty ( String key, Object obj ) throws Exception;`
Adds or updates a value in the Property Store. If an update is performed the old value is returned.

@param key
The unique identifier.

@param obj
The value.

@return
The old value in case of an update.

`public Object getProperty ( String key ) throws Exception;`
Returns a value in the Property Store.

@param key
The unique identifier.

@return
Value in the store or NULL if not found.

`public Object removeProperty ( String key ) throws Exception;`
Removes a value in the Property Store.

@param key
The unique identifier to remove.

@return
The old value or **null** if key is not in the table.

UserFunctions (system object) methods

The UserFunctions class (for example, the system object) has additional methods defined to get or set objects in the System Property Store:

`public Object getPersistentObject ( String key ) throws Exception;`
This method retrieves a named object from the default system Property Store.

@param key
The unique key.

`public Object setPersistentObject ( String key, Object value ) throws Exception;`
This method stores a named object in the default system Property Store.

@param key
The unique key.

@param value
The object to store (must be Java serializable).

@return
The old object if any.

`public Object removePersistentObject ( String key ) throws Exception;`
This method removes a named object in the default System Property Store.

@param key
The unique key.

@return
The old object if any.

Chapter 7. Deltas

The Delta Engine feature is available to Connectors in Iterator mode. If enabled from the Iterator's Delta tab, the Delta engine feature uses the System Store to take a snapshot of data being iterated. Each successfully read Entry is compared with the snapshot database called the Delta Store to see what has changed. Based on the differences between the read Entry and the Entry stored in the Delta store, a new Entry called a Delta Entry is created by the Delta Engine. This Entry is tagged with special delta operation codes to indicate what has changed, and how.

Delta mode is a Connector mode which enables a connector to "understand" and use the delta operation codes. A connector in this mode uses the delta operation codes of a received Delta Entry to determine what type of change needs to be applied to the connected system. The Delta mode supports all types of modifications – add, modify and delete. This mode is used to facilitate synchronization between different systems (for example, synchronizing two LDAP servers on different machines).

Attention: The Delta Engine introduces an underlying local repository for storing snapshots of data in order to compute changes during the synchronization process. The data source that is being scanned for changes becomes the master in a master-slave relationship, and it is then vital that all changes made to the slave (for example, the Delta store) be made using the Delta mechanism, and not by directly manipulating the underlying database table. Otherwise, the Delta snapshot information that Tivoli Directory Integrator maintains becomes inconsistent, and the Delta Engine fails.

Delta Features

Delta features in IBM Tivoli Directory Integrator are the following:

Delta Entry

This is a regular Entry object that has been tagged with special delta operation codes. These codes describe the type of change (*add*, *modify*, *delete* or *unchanged*) and can be assigned on different levels – the Entry, Attribute or AttributeValue level.

Components producing Delta Entries

- Delta Engine – detects changes in a data source. This is useful when the data source itself does not provide a convenient access to changes (for example, changelog or change notification mechanism). Changes are detected by comparing the current state of the data source against a historical snapshot. The snapshot is saved in a repository called the "Delta Store". Physically the Delta Store consists of a number of Delta Tables located in the System Store.

  The next time the data is read it will be compared to the one already saved in the Delta store. After the changes information is computed a Delta Entry is created and returned by the Connector. This Delta Entry can then be used to transfer the detected changes to other connected systems by using Connectors in *Update*, *AddOnly*, *Delete* or *Delta* mode.
- Change Detection Connectors – these are the LDAP Connectors, RDBMS Change Detection Connector and Domino Change Detection Connector.
- LDIF and DSMLv2 Parser – when reading or writing the LDIF and DSMLv2 Parser support Delta tagging at Entry, Attribute and AttributeValue level.

Components consuming Delta Entries

- Delta mode – this Connector mode facilitates synchronization solutions between systems. It can be used to apply all types of changes to a connected system. The Delta mode is the only mode that requires (and uses) Delta Entries. This mode detects change types by using delta operation codes of an Entry.
- Update Connectors with Compute Changes – Connectors in Update mode with enabled "Compute Changes" parameter do not trigger the "Compute Changes" logic if the received Entry is delta tagged.

Delta Entry

A Delta Entry is an Entry that posseses all features and functions of a regular Entry. In addition to that the Delta Entry also contains delta operation codes. They indicate the type of change applied to the Entry – add, delete, modify or unchanged. The delta operation codes can be attached to Entries, Attributes and values to reflect their particular changes.

Overview

The process of assigning delta operation codes is called delta tagging and the delta codes are referred to as operation codes and delta tags. In a few words, the Delta Entry is actually a Delta tagged regular Entry. Here is an example of a regular Entry and a Delta Entry.

Regular Entry:

```
{
 "#type": "generic",
 "#count": 3,
 "UserName":
  "#type": "replace",
  "#count": 1,"tanders",
 "FullName":
  "#type": "replace",
  "#count": 1,"Teddy Anderson",
 "id":
  "#type": "replace",
  "#count": 1,"66"
}
```

Delta Entry:

```
{
 "#type": "modify",
 "#count": 3,
 "@delta.old": "{
 "UserName": "manders",
 "FullName": "Mary Anderson",
 "id": "66"
 }",
 "UserName": [
  "#type": "modify",
  "#count": 2,
  "tanders",
  "manders"
 ],
 "FullName": [
  "#type": "replace",
  "#count": 1,
  "Mary Anderson"
 ],
 "id":
  "#type": "unchanged",
  "#count": 1,"66"
}
```

The process of evaluation of delta codes goes from top to bottom or from Entry level → Attribute level → AttributeValue level. The operation at the higher level takes precedence.

If an Entry operation is *delete* all other delta tags are ignored. If it is *replace*, *modify* or *add* the evaluation continues with the Attributes delta tags.

If an Attribute is tagged as *delete*, *add* or *replace* the delta tags of its values are ignored. Only if an Attribute is tagged as *modify*, the ' delta tags of AttributeValues will be considered. These delta tags indicate that AttributeValues can have different operation codes (for example, some of them are added and others deleted).

The AttributeValue delta tags have the following meaning in the context of Delta tagging:
- *add* – the value is added to the list of values for the specified Attribute;
- *delete* – the value is removed from the list of values for the specified Attribute;
- *replace* – the value is replaced; this is the default delta tag.

Delta Entries are produced by the following components:
1. Connectors in Iterator mode with enabled Delta;
2. Change Detection Connectors;
3. LDIF and DSMLv2 Parser when reading.

They are consumed by these components:
1. Connectors in Delta mode;
2. LDIF and DSMLv2 Parser when writing;
3. Connectors in Update mode.

Getting and setting Delta operation codes using Script

Delta tagging is supported at Entry, Attribute and AttributeValue level. Here is how you can get/set the Entry operation using script:

```
var entryOper = work.getOperation();  //get Entry operations as string (e.g. 'add')
var entryOp = work.getOp();      //get Entry operations as char (e.g. 'a')

work.setOperation("modify");      //set Entry operation
work.setOp ('m');
```

If you want to set/get the Delta flags for an Attribute you can do that with the following code:

```
var attr = work.getAttribute("sn");   // get Attribute object

var attrOper = attr.getOperation();   // get delta operation as string (e.g. 'replace')
var attrOp = attr.getOper();      // get delta operation as char (e.g. 'r')

attr.setOperation("replace");     // set Attribute delta operation
attr.setOper('r');
```

Delta tags at the AttributeValue level can be set/get using this script:

```
var attr = work.getAttribute("sn");   // get Attribute object

var valOper = attr.getValueOperation(0);// get value delta operation for first value
var valOp = attr.getValueOper(0);

attr.setValueOperation(1, "add");    // set value delta operation for second value
attr.setValueOper(1, 'a');
```

Producing Delta Entries

Delta Entries can be generated either using the Delta feature or suitable parser of Connectors in Iterator mode, or by Tivoli Directory Integrator's Change Detection Connectors.

In particular, Delta tagged Entries are returned by the following components:

- Active Directory Change Detection Connector
- Domino Change Detection Connector
- IBM Tivoli Directory Server Changelog Connector
- RDBMS Change Detection Connector
- Sun Directory Change Detection Connector
- z/OS LDAP Changelog Connector
- DSMLv2 Parser
- LDIF Parser

Delta feature for Iterator mode

Connectors in Iterator mode can generate Delta entries. This feature uses the Delta Engine and the Delta Store to detect changes.

Delta Engine

The Delta Engine allows you to read through a data source, and detect changes from the previous time you did this. This way you can detect new entries, changed entries and even deleted entries. For certain data sources (such as LDIF files and LDAP servers), Tivoli Directory Integrator can even detect if attributes and values within entries have been changed. You can configure Delta settings on Connectors in Iterator mode only.

The Delta Engine knows whether Entries or Attributes have been added, changed or deleted by keeping a local copy of each Entry in a persistent store, which is part of the System Store. This local repository is called a *Delta Store* and consists of *Delta tables*. Each time the AssemblyLine is run, the Delta Engine compares the data being iterated with its copy in the Delta Table. When a change is detected the Connector returns a *Delta Entry*.

Note: Do not manually modify Delta Store tables. Otherwise, the Delta snapshot information will become inconsistent, and the Delta Engine will fail.

Note: In versions earlier than IBM Tivoli Directory Integrator V6.1, snapshots written to the Delta 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. This limitation is addressed through the Commit parameter on the Connector Delta tab. Setting this parameter controls when the Delta engine commits snapshots taken of incoming data to the System Store.

Unique Attribute name

In order for the Delta mechanism to be able to uniquely identify each Entry, you must specify a unique Attribute to use as a Delta key. The values of this attribute must be unique in the used data source. You can specify the Delta key in the Delta tab of the Connector, by entering or selecting an attribute name in the Unique Attribute Name parameter. This attribute must be found in the Input Map of your Iterator, and can either be an attribute read from the connected system or a computed attribute (using script in the Attribute Mapping).

You can also specify multiple attributes by separating them with a plus sign (+):

```
LastName+FirstName+BirthDate
```

At least one of the attributes specified in the Unique Attribute Name parameter must contain a value. When several attributes are specified, their string values are concatenated into one string, which then becomes the unique Delta identifier. Attributes with no values (for example, blank or NULL) are skipped when the Delta key is built for an Entry.

Delta Store

The Delta Store is physically located in the System Store. It consist of one Delta Systable (DS) and one or more Delta Tables. Each Delta Table is used for the Delta Store of a different Iterator Connector with enabled Delta.

Although Delta Store tables can be accessed with both the JDBC Connector and the System Store Connector, it is unadvisable to change them without a deep understanding of how these tables are structured and handled by the Delta Engine.

Delta Table structure

Every Delta Table (DT) contains information about each Entry processed by the Delta Engine for a particular Connector. A Delta Systable (DS) maintains a list of all Delta Tables currently in use by the Delta Store.

- Delta Systable – The Delta Systable (DS) contains information about each Delta Table (DT) in the System Store. The purpose of the DS is to maintain the sequence counter for each DT. The structure for the DS is as follows:

Table 12. Delta Systable structure

Column	Type	Description
ID	Varchar	The DT identifier (name)
SequenceID	Int	The sequence ID from the last run
Version	Int	The DS version (1)

- Delta Table – Each Connector that requests a Delta store needs to specify a unique Delta identifier to be associated with the Connector. This identifier is also used as the name of the Delta Table in the System Store. The Delta Table structure is as follows:

Table 13. Delta Table structure

Column	Type	Description
ID	Varchar	The unique value identifying an Entry
SequenceID	Int	The sequence number for the Entry
Entry	Long Varbinary	The serialized Entry object

Delta process

Given the above Delta Store structure, the sequence number is used to determine which entries are no longer part of the source data set. Every time an AssemblyLine is run the sequence number for the Delta Table used in particular by the Connector is read from the Delta Systable. Then it is incremented, and this incremented value will be used for marking the updated entries during the entire AssemblyLine execution.

The Delta Engine process works in two passes.

1. Read → Look up → Compare → Update → Set current SequenceID
   a. The Iterator reads entries from the input data source.

b. The Delta process looks for corresponding Entry in the Delta Table using the unique attribute's value.

c. If a match is found the Delta process compares each Attribute (and its values) to determine if there have been modifications to the Entry. Based on the result from the comparison, the Delta Engine returns Delta Entry tagged with the relevant operation codes: *modify* or *unchanged*:

   - Modify Entry – the Entry that was read and the corresponding Entry from the Delta Table are considered different; the Entry is updated in the Delta Table
   - Unchanged Entry – the Entry that was read and the corresponding Entry from the Delta Table are considered equal.

d. If a match is not found in the Delta Table the Entry is treated as new:

   - Add Entry – the Entry is added to the Delta Table.

e. In both case c. and d. the sequence number value in the Delta table is updated with the sequence number used for the current AssemblyLine execution.

2. Check for data with (SequenceID < current SequenceID) → Mark as Deleted

   Once End of Data is reached by the Iterator, the Delta Engine makes a second pass through the Delta Table looking for those entries not accessed during the first pass. These Entries are easily recognized because their sequence number is not updated with the current sequence number. Therefore any Entries in the Delta Table with a sequence number lower than the current sequence number are considered to be deleted entries and are returned as deleted.

   Note: This pass happens only when the iteration trough the input data completes successfully. If for some reason an error occurs during that iteration, no Entries will be tagged as deleted and returned by the AssemblyLine or removed from the Delta Table. This will not affect the original data source and the next time the AssemblyLine is executed successfully the deleted Entries will be processed correctly.

Row Locking

This parameter is available in the Delta tab for Iterator connectors and the Delta Function Component configuration. It allows you to set the transaction isolation level used by the connection established to the Delta Store database. Setting a higher isolation level reduces the transaction anomalies known as 'dirty reads', 'non-repeatable reads' and 'phantom reads' by using row and table locks. This parameter has the following values:

READ_UNCOMMITTED
Corresponds to `java.sql.Connection.TRANSACTION_READ_UNCOMMITTED`; indicates that dirty reads, non-repeatable reads and phantom reads can occur. This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

READ_COMMITTED
Corresponds to `java.sql.Connection.TRANSACTION_READ_COMMITTED`; indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.

REPEATABLE_READ
Corresponds to `java.sql.Connection.TRANSACTION_REPEATABLE_READ`; indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur. This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

SERIALIZABLE
Corresponds to `java.sql.Connection.TRANSACTION_SERIALIZABLE`; indicates that dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in

TRANSACTION_REPEATABLE_READ and further prohibits the situation where one transaction reads all rows that satisfy a WHERE condition, a second transaction inserts a row that satisfies that WHERE condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read. This is generally the slowest but safest option, and the default value for the **Row Locking** parameter.

For more information about transaction isolation levels, see the online documentation of the java.sql.Connection interface: http://docs.oracle.com/javase/1.6.0/docs/api/java/sql/Connection.html.

Each database server sets a default transaction isolation level; the default value for Derby, Oracle and MS SQL Server is TRANSACTION_READ_COMMITTED. However, the default value of the **Row Locking** parameter of SERIALIZABLE will override this when using a Delta component (that is, the Delta functionality in Iterator Connectors or the Delta Function Component).

Some database servers may not support all transaction isolation levels, therefore please refer to the specific database documentation for accurate information about supported transaction isolation levels.

Note: Transaction isolation levels are maintained by the database server itself for every connection established to the database. Therefore when a Delta component (with **Transaction isolation level** set to REPEATABLE_READ or SERIALIZABLE and the **Commit** parameter set to On Connector Close starts its transaction, all other queries trying to modify the same data will be blocked. This means that other components which need to modify the same data will have to wait until the first component commits its transaction on termination. This waiting may cause the issued SQL queries to timeout and leave the data unmodified.

Also when a component has the **Commit** parameter set to No autocommit you should manually commit the transactions in such manner that other components will not wait forever to perform a modification.

Detect or ignore changes only in specific attributes

The parameters **Attribute List** and **Change Detection Mode** configure the ability of the Delta Engine to detect changes only in specific attributes instead of in all received attributes.

The **Attribute List** parameter is a list of comma separated attributes which will be affected by **Change Detection Mode**. This **Change Detection Mode** parameter specifies how changes in these attributes will be handled. It has three values:

IGNORE_ATTRIBUTES
(“Ignore changes for the following Attributes”) – Changes in every attribute specified in the **Attribute List** parameter will be ignored during the compute changes process.

DETECT_ATTRIBUTES
(“Detect changes for the following Attributes”) – This option has the opposite effect – the only detected changes will be in the attributes listed in the **Attribute List** parameter.

DETECT_ALL
(“Use all Attributes for change detection”) – This instructs the Delta Engine to detect changes in all attributes. When this option is selected the **Attribute List** parameter is disabled since no list of affected attributes is needed.

Example use case

When using the Delta Engine, sometimes the received entries contain attributes that you consider as not important and wish to ignore. In such cases, these attribute must not affect the result of the Delta computation, as when several Entries differentiate only by these attribute it leads to unnecessary updates of the Delta Store table.

The solution for this case is using the **Attribute List** and **Change Detection Mode** parameters

Here is an example scenario where two AssemblyLines are receiving changelog entries from two replicas of a LDAP server and these changes are applied to one Delta Store. To illustrate this we will use the following example changelog entries:

Entry1:

```
Entry attributes:
 targetdn (replace): 'cn=Niki,o=IBM,c=us'
 changetime (replace): '20071015094646'
 $dn (replace): 'changenumber=78955,cn=changelog'
 ibm-changeInitiatorsName (replace): 'CN=ROOT'
 changenumber (replace): '78955'
 objectclass (replace): 'top' 'changelogentry' 'ibm-changelog'
 changetype (replace): 'modify'
 cn (replace): 'Niki' 'Niky'
 changes (replace): 'replace: cn
       cn: Niki
       cn: Niky
       -
       '
```

Entry2:

```
Entry attributes:
 targetdn (replace): 'cn=Niki,o=IBM,c=us'
 changetime (replace): '20071015094817'
 $dn (replace): 'changenumber=10076,cn=changelog'
 ibm-changeInitiatorsName (replace): 'CN=ROOT'
 changenumber (replace): '10076'
 objectclass (replace): 'top' 'changelogentry' 'ibm-changelog'
 changetype (replace): 'modify'
 cn (replace): 'Niki' 'Nikolai'
 changes (replace): 'replace: cn
       cn: Niki
       cn: Nikolai
       -
      '
```

Entry3:

```
Entry attributes:
 targetdn (replace): 'cn=Niki,o=IBM,c=us'
 changetime (replace): '20071037454817'
 $dn (replace): 'changenumber=112,cn=changelog'
 ibm-changeInitiatorsName (replace): 'CN=ADMIN'
 changenumber (replace): '112'
 objectclass (replace): 'top' 'changelogentry' 'ibm-changelog'
 changetype (replace): 'modify'
 cn (replace): 'Niki' 'Nikolai'
 changes (replace): 'replace: cn
       cn: Niki
       cn: Nikolai
       -
          '
```

Modified attributes are marked in **bold** and attributes that can be ignored are marked in *italics*. The ignored attributes (such as changenumber, changetime, and so forth) will not be considered when comparing the received Entry with the stored Entry. Therefore these attributes have to be listed in the **Attribute List** parameter. In order to specify that we want to ignore them the **Change Detection Mode** parameter needs to be set to `Ignore changes for the following Attributes`.

This is the workflow when the AssemblyLines receive the entries:

1. When AL1 receives Entry1, it will be returned as *modify* and saved in the Delta Store table.

2. When AL2 receives Entry2 , its changetime, $dn, bm-changeInitiatorsName, changenumber attributes are modified but will be ignored. However the cn and changes attributes are also modified and therefore the resulted Delta Entry will be tagged as *modify* and saved in the Delta Store table.
3. When AL2 receives Entry3, its changetime, $dn, bm-changeInitiatorsName, changenumber attributes are modified but will be ignored. The rest of the attributes are equal so the resulted Delta Entry will be tagged as *unchanged* and will be returned to the AssemblyLine (only if the **`Return unchanged`** parameter is checked) or skipped. The returned Delta Entry will be identical to the received Entry3. In this case the Delta Store is not updated. If the Attribute List and Change Detection Mode parameter were not used, the last Entry3 would have been tagged as *modify* and saved in the Delta Store.

Change Detection Connectors

Change Detection Connectors leverage information in the connected system to detect changes, and are either used in Iterator or Server Mode, depending on the Connector. For example, Iterator mode is used for many of the Change Detection Connectors, like those for LDAP, RDBMS, Active Directory and Notes/Domino. They are designed to behave in a common way, as well as provide the same parameter labels for common settings.

Change Detection Connectors in Tivoli Directory Integrator are:

- IBM Tivoli Directory Server (TDS) Changelog
- AD Change Detection (Active Directory)
- Domino Change Detection
- Sun Directory Change Detection (openLDAP, SunOne, iPlanet, and so forth)
- RDBMS Change Detection (DB2, Oracle, SQL server, and so forth)
- z/OS LDAP Changelog

The Delta engine feature reports specific changes all the way down to the individual values of attributes. Delta tagging at the AttributeValue level is also available when parsing with either LDIF or DSMLv2 Parsers. The LDIF Parser is used internally by the IBM Tivoli Directory Server Changelog Connector, Sun Directory Change Detection Connector and z/OS LDAP Changelog Connector, therefore these connectors support Delta tagging at AttributeValue level as well. The rest of the change detection connectors are limited to simply reporting if an entire Entry is added, modified or deleted. For more information about the Delta tagging support of a particular component refer to the specific description of that component in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

In some cases, long running AssemblyLines may need to process the same entries more than once. These entries will have a duplicate delta key and will cause the AssemblyLine to throw an exception. If you want to allow duplicate delta keys, you can select the **Allow Duplicate Delta Keys** check box in the Iterator's Delta tab. This means that duplicate entries can be handled by AssemblyLines having both Iterator Connectors with Delta enabled or Change Detection Connectors and Delta mode Connectors.

Note: It is possible to have, for example, an AssemblyLine with a number of Changelog and Delta mode Connectors. In this case, if the Delta mode Connector is pointing to the same underlying system as the Changelog Connector, the Delta operation could trigger the Changelog again. As there is no way to differentiate between newly-received changes and those triggered by the Delta engine, you should carefully consider your scenario in order not to enter into an endless loop.

The delta information computed by the Delta engine is stored in the Work Entry object, and depending on the Change Detection component or feature used can be stored as an *Entry-Level*, *Attribute-Level* or *Attribute Value-Level* operation code.

Consuming Delta Entries

Delta Entries in the AssemblyLine can be acted upon ("consumed") by Connectors in Delta Mode, and Connectors in Update Mode inside the Compute Changes logic.

Delta Mode is available in the following Connectors:

- JDBC Connector
- DSMLv2 SOAP Connector
- JNDI Connector
- LDAP Connector

Delta Mode Connectors

The Delta mode is designed to simplify the application of changes to data by providing incremental modifications to the connected system, based on delta operation codes. Incremental modifications means to write only the specific values that have been changed.

Firstly, Delta mode handles all types of deltas: adds, modifies and deletes. The Delta mode requires receiving a Delta Entry to operate. Therefore when using a Connector in Delta mode, it must be combined with components that produce Delta Entries; these are Iterator Connectors with enabled Delta, Change Detection Connectors or Connector using an LDIF or DSMLv2 Parser. For example, you can synchronize two systems with the use of only two Connectors: one Change Detection Connector in the Feeds section to pick up the changes, and a second Connector in Delta mode to apply these changes to a target system.

Furthermore, Delta mode will apply the delta information at the lowest level supported by the target system itself. This is done by first checking the Connector Interface to see what level of incremental modification is supported by the data source. If you are working with an LDAP directory, then Delta mode will perform AttributeValue additions and deletes. In the context of a traditional RDBMS (JDBC), deleting and then adding a column value does not make sense, so this is handled as a value replacement for that Attribute.

Also, incremental modifications are automatically dealt by the Delta mode for those data sources that support this functionality. If the data source offers optimized calls to handle incremental modifications, and these are supported by the Connector Interface, then Delta mode will use these. On the other hand, if the connected system does not offer "intelligent" delta update mechanisms, Delta mode will simulate these as much as possible, performing pre-update lookups (like Update mode), change computations and subsequent application of the detected changes.

Note: The only Connectors that support incremental modifications are the LDAP Connectors, since LDAP directories provide this functionality.

The Delta features in Tivoli Directory Integrator are designed to facilitate synchronization solutions using not only data sources providing change detection mechanism but also, for example, flat files. The following diagram shows such a synchronization solution using Connectors in Iterator and Delta mode. The Iterator Connector reads entries from a data source. The read entries are compared to the ones saved in the Delta Store on the previous iteration. The result of the comparison is a Delta Entry assigned with delta operation codes defining the made change – add/delete/modify. Then the Delta Entry is used by the Connector in Delta mode to apply the detected changes to a destination system.

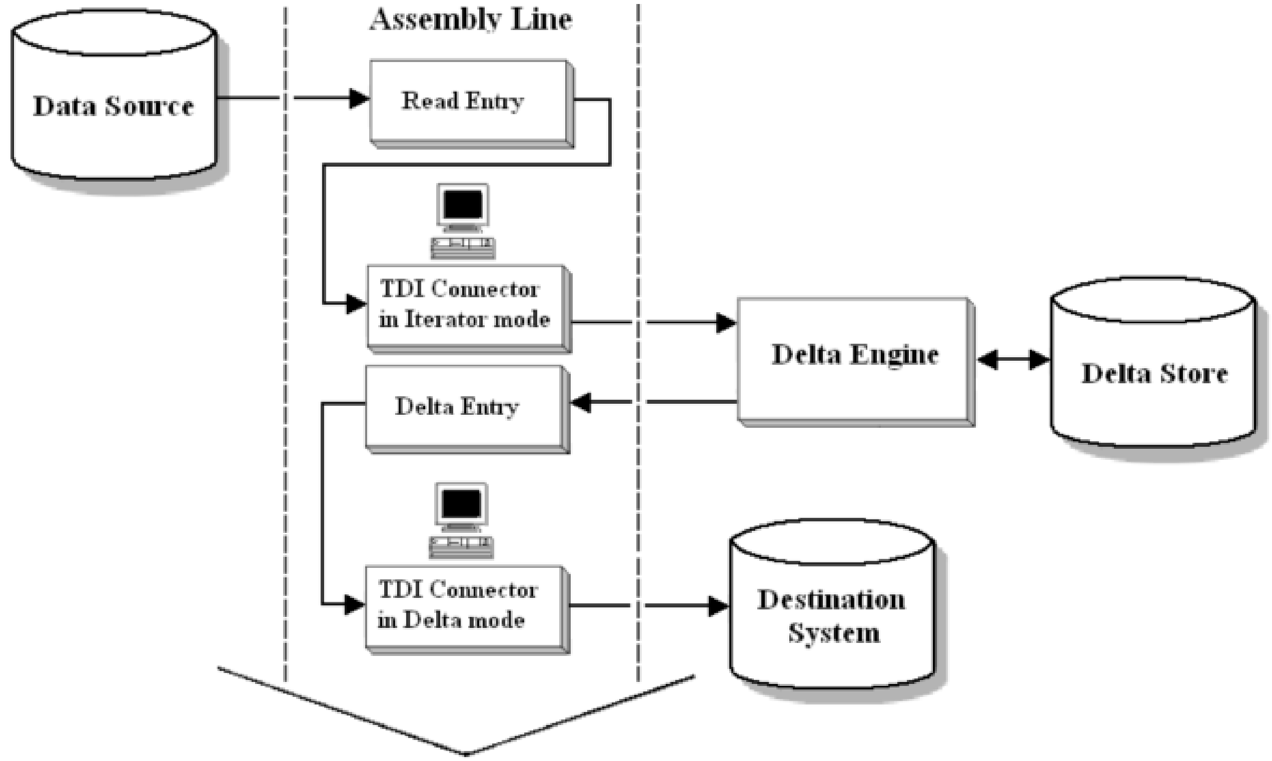


Figure 117. Synchronization AssemblyLine using Delta functionality

The result of the AssemblyLine execution is that the data contained in the data source is synchronized with the data in the destination system

Update Mode and Delta Entries

Connectors in Update mode have an additional change-control feature called “Compute Changes”. This feature can be made to work with Delta Entries.

The “Compute Changes” feature can be enabled in an Update mode Connector by selecting a check box of that name in the Connector Detail pane. When turned on the Connector will compare its Entry with the actual Entry in the connected system. If the two Entries are equal no updating is performed by the Connector. So the “Compute Changes” option allows the Connector in Update mode to skips unnecessary updates. This is useful for data sources with relatively heavy update operations.

However when a Connector in Update mode with enabled **Compute Changes** parameter receives a Delta Entry, the “Compute Changes” logic will not be triggered; it will use the delta operation codes assigned to the Delta Entry to apply the necessary updates to the connected system.

Examples

Simple Delta example

The instructions below will show you how to utilize some of the Delta features discussed above.

Set up a File System Connector with the Delta engine feature enabled. Have it iterate over a simple XML document that you can easily modify in a text editor. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3333</ValueTag>
        </Telephone>
        <Birthdate>1958-12-24</Birthdate>
        <Title>Full-Time TDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
    </Entry>
</DocRoot>
```

Be sure to use the special map-all attribute map character, the asterisk (*). This is the only Attribute you need in your map to ensure that all Attributes returned are mapped in to the Work Entry object.

Now add a Script Component with the following code:

```
// Get the names of all Attributes in work as a String array
var attName = work.getAttributeNames();

// Print the Entry-level delta op code
task.logmsg(" Entry (" +
work.getString( "FullName" ) + ") : " +
work.getOperation() );

// Loop through all the Attributes in work
for (i = 0; i < attName.length; i++) {

 // Grab an Attribute and print the Attribute-level op code
 att = work.getAttribute( attName[ i ] );
 task.logmsg("    Att ( " + attName[i] + ") : " + att.getOperation() );

 // Now loop through all the Attribute's values and print their op codes
 for (j = 0; j < att.size(); j++) {
  task.logmsg( "      Val (" +
                 att.getValue( j ) + ") : " +
                 att.getValueOperation( j ) );
 }
}
```

The first time you run this AL, your Script Component code will create this log output:

```
12:46:31   Entry (John Doe) : add
12:46:31      Att ( Telephone) : replace
12:46:31        Val (111-1111) :
12:46:31        Val (222-2222) :
12:46:31        Val (333-3333) :
12:46:31      Att ( Birthdate) : replace
12:46:31        Val (1958-12-24) :
12:46:31      Att ( Title) : replace
12:46:31        Val (Full-Time TDI Specialist) :
12:46:31      Att ( uid) : replace
12:46:31        Val (jdoe) :
12:46:31      Att ( FullName) : replace
12:46:31        Val (John Doe) :
```

Since this Entry was not found in the previously empty Delta Store, it is tagged at the Entry-level as new. Furthermore, each of its Attributes has a replace code, meaning that all values have changed (which makes sense because the Delta is telling us that this is new data).

Make the following changes to your XML file:

1. Change the last Telephone number value to 333-4444.
2. Delete Birthdate.
3. Add a new Address Attribute.

Your resulting Config should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-4444</ValueTag>
        </Telephone>
        <Title>Full-Time TDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
        <Address>123 Willowby Lane</Address>
    </Entry>
</DocRoot>
```

Run your AL again. This time your log output should look like this:

```
13:53:22   Entry (John Doe) : modify
13:53:22      Att ( Telephone) : modify
13:53:22        Val (111-1111) : unchanged
13:53:22        Val (222-2222) : unchanged
13:53:22        Val (333-4444) : add
13:53:22        Val (333-3333) : delete
13:53:22      Att ( Birthdate) : delete
13:53:22        Val (1958-12-24) : delete
13:53:22      Att ( uid) : unchanged
13:53:22        Val (jdoe) : unchanged
13:53:22      Att ( Title) : unchanged
13:53:22        Val (Full-Time TDI Specialist) : unchanged
13:53:22      Att ( Address) : add
13:53:22        Val (123 Willowby Lane) : add
13:53:22      Att ( FullName) : unchanged
13:53:22        Val (John Doe) : unchanged
```

Now the Entry is tagged as *modify* and the Attributes reflect what the modifications for each of them. As you can see, the Birthdate Attribute is marked as *delete* and Address as *add*. That's the reason you used the special map-all character for our Input Map. If you had mapped only the Attributes that existed in the first version of this XML document, we would not have retrieved Address when it appeared in the input.

Pay special attention to the last two value entries under the Telephone Attribute, marked as *modify*. The change to one of the values of this Attribute resulted in two Delta items: a value *delete* and then *add*.

To build a data synchronization AssemblyLine in earlier versions of Tivoli Directory Integrator, you had to script in order to handle flow control. Although you may receive *adds*, *modifies* and *deletes* from your change component or feature, a Connector could only be set to one of the two required output modes: Update or Delete.

So either you need two Connectors pointing to the same target system and put script in the Before Execute Hook of each to ignore the Entry if its operation code did not match the mode of this component; or you could have a single Connector (either Update or Delete mode) in *Passive* state, and then control its execution from script code where you checked the operation code. This still meant that even though you knew what had changed in the case of a modified Entry, your Update Mode Connector would still read in the original data before writing the changes back to the data source. This can lead to unwanted network or data source traffic when you are only changing a single value in a multi-valued group-related Attribute containing thousands of values.

Further examples

Go to the `root_directory/examples/` directory of your IBM Tivoli Directory Integrator installation.

- In subdirectory `deltas` you will find a simple IBM Tivoli Directory Integrator configuration that demonstrates the Delta functionality. This demo runs on MS Windows systems (Windows 2000) only as it uses MS Access database and the JDBC:ODBC bridge. If you use another platform, you must create your own database and configure the JDBC settings of the Connectors for this database.
- In subdirectory `delta_tagging` you will find an example that demonstrates how to convert an Entry tagged at value level to a regular Entry and a regular Entry to a Delta Entry.

Chapter 8. Tivoli Directory Integrator Dashboard

Use the Tivoli Directory Integrator Dashboard to install, configure, deploy, and monitor data integration solutions.

You can also use Dashboard to create and configure EasyETL solutions (ETL stands for Extract, Transform, and Load) for simple data integrations. An EasyETL solution contains a single AssemblyLine with a source and a target connector.

Introducing Tivoli Directory Integrator Dashboard

The Dashboard, a web application, is developed by using the Open Service Gateway Initiative (OSGI) framework. It is used to configure and manage data integration solutions on a server through RESTful (Representational State Transfer) interface. From the Dashboard, you can:

- Upload existing data integration solutions and install it as a normal solution or save as a template
- Configure data integration solutions to reflect changes that are specific to your local environment
- Add schedules to run AssemblyLines at specified times
- Build simple AssemblyLines with a single source connector and a target connector
- Browse contents of the selected connector
- Deploy, monitor, and audit the processes of data integration solutions
- View server details, server log, and system store data
- Filter log file data by setting the filter criteria to extract only the required information
- View AssemblyLine execution history in the form of a graph
- Create reports that contain data on AssemblyLine execution status, sent automatically to the user through an email

Advantages of Tivoli Directory Integrator Dashboard

The advantages are:

- Facilitates simpler and faster deployment of data integration solutions
- Creates and configures simple integrations without using Configuration Editor
- Generates scheduled email report of the AssemblyLine history, sent by using the Dashboard RunReport feature. You can use the report to address high availability/failover requirements of the deployment
- Facilitates integration with Eclipse development environment
- Ensures effective and efficient management of AssebmlyLine executions

Accessing Dashboard application

You can access the Dashboard web application from either your browser or from the Configuration Editor.

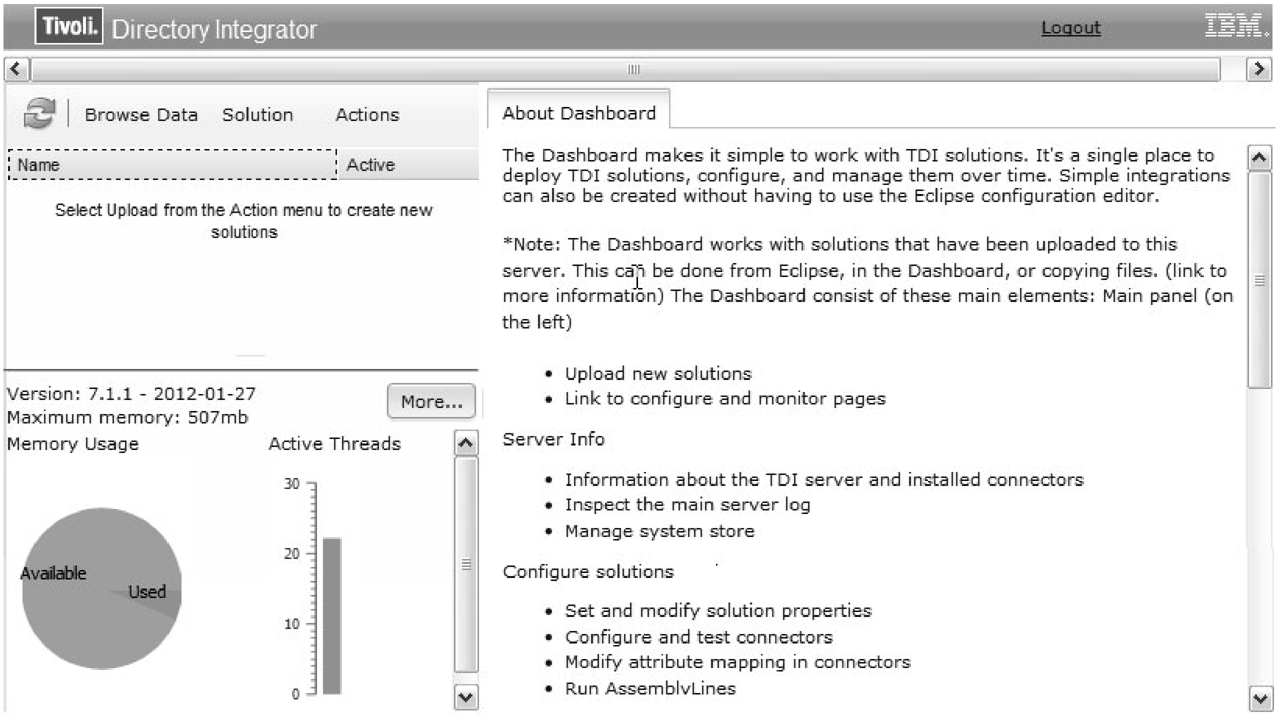
Before you begin

Start Tivoli Directory Integrator Server before accessing the Dashboard.

Opening from a browser

Procedure

From a browser, go to `http://<host name>:<rest-api-port>/dashboard`. The Dashboard home page is displayed.



Select Upload from the Action menu to create new solutions

Version: 7.1.1 - 2012-01-27
Maximum memory: 507mb

The Dashboard makes it simple to work with TDI solutions. It's a single place to deploy TDI solutions, configure, and manage them over time. Simple integrations can also be created without having to use the Eclipse configuration editor.

*Note: The Dashboard works with solutions that have been uploaded to this server. This can be done from Eclipse, in the Dashboard, or copying files. (link to more information) The Dashboard consist of these main elements: Main panel (on the left)

- Upload new solutions
- Link to configure and monitor pages

Server Info

- Information about the TDI server and installed connectors
- Inspect the main server log
- Manage system store

Configure solutions

- Set and modify solution properties
- Configure and test connectors
- Modify attribute mapping in connectors
- Run AssemblvLines

Note: By default, you can access Dashboard on the local host. To authenticate a user, set the Tivoli Directory Integrator properties file to add LDAP user name and password for both remote and local access.

Opening from the Windows Start menu

Procedure

From the Windows **Start** menu, choose **Start** > **All Programs** > **IBM Tivoli Directory Integrator v7.1.1** > **Open Dashboard**. The Tivoli Directory Integrator Dashboard home page is displayed.

Table 14. Tivoli Directory Integrator Dashboard Menu Options

Menu Option	Description
Browse Data	Use this option to configure the selected connector and browse its contents.

Table 14. Tivoli Directory Integrator Dashboard Menu Options (continued)

Menu Option		**Description**
Solution	**Monitor**	Use this option to track the progress of AssemblyLine execution and view the execution history of your solution.
	Start	Use this option to start the selected solution.
	Stop	Use this option to terminate a running solution.
	Configure	Use this option to open the selected solution for modification.
	Delete	Use this option to delete the selected solution from the server.
	Unlock Solution	Use this option to unlock a solution.
Actions	**Browse Data**	Use this option to configure the selected connector and browse it contents.
	Show System Log	Use this option to view the contents of system log (`ibmdi.log`).
	Create Solution	Use this option to choose a template and create a data integration solution.
	Upload Solution	Use this option to upload an existing solution to the Tivoli Directory Integrator Server.
	Show Server Details	Use this option to view information about Tivoli Directory Integrator Server such as version, installed components, and other server information.

Uploading a data integration solution

Use the Upload Solutions window of Dashboard to upload an existing data integration solution to install it on the Tivoli Directory Integrator Server. You can also save the uploaded solution as a template.

About this task

The Dashboard can be used to upload an existing Tivoli Directory Integrator data integration solution, which is an XML configuration file, to install it on the server. You can modify the solution with the requisite properties before deploying. You can also upload the existing solution to save it as a template. The saved template can be used to create many instances of the same integration.

Procedure

1. In the navigation pane of Dashboard window, click **Actions** > **Upload Solutions**.
2. Define the following settings on the Upload Solutions window.

Option	**Description**
Browse	Browse for the existing solution you want to upload
Upload as Template	Saves the uploaded solution as a template **Note:** The saved template in not installed on Tivoli Directory Integrator Server (config directory).
Overwrite Existing Solution	Overwrites existing installed solutions or templates
Solution Name	Specifies name for the uploaded solution **Note:** You can also view names of the currently installed solutions and template solutions in the Upload Solutions window.

3. Click **OK**.

Creating a data integration solution

Use the Create Solutions window of Dashboard to create a data integration solution, by using various options.

Procedure

1. In the navigation pane of Dashboard window, click **Actions** > **Create Solutions**.
2. Select one of the following options in the Create Solutions window.

Option	Description
Installed Solution	Solution is created based on an already installed solution on the Tivoli Directory Integrator Server.
Template	Solution is created based on an existing solution, which is uploaded as a template.
Default	EasyETL solution is created having a single AssemblyLine with two connectors.

3. Type the name of your solution in the **Solution Name** field.
4. Click **OK**.

Solution Configuration

The Dashboard consists of various editors to configure the components of your data integration solution.

In the Dashboard, you can modify the selected solution with appropriate settings that are specific to your needs before deployment. To facilitate faster configurations, you can limit the amount of information visible while configuring a solution. You can also add new configuration pages to include properties that are specific to the solution.

Only the installed data integration solutions are shown in the navigation panel of the Dashboard window. Each solution contains components such as AssemblyLines and associated connectors. These components are shown as a tree structure in the Solution Editor. From the Solution Editor you can install, create, configure, deploy, and monitor your solutions by using the following editors:

- AssemblyLine Editor – this editor is used to create and configure schedules in the AssemblyLines and to run the AssemblyLine with its output in the log viewer.
- Connector Editor – this editor is used to configure the connectors of the AssemblyLine.
- EasyETL Editor – this editor is used to configure EasyETL solution, which contains a single AssemblyLine with two connectors.
- Monitor Editor – this editor is used to monitor information of both current and past performance of your solution.

The Solution Editor has the following menu options:

Option	Description
Edit Solution Interface	Use this option to select the components of the solution, which are visible for editing.
New AssemblyLine	Use this option to create an EasyETL solution and add it the selected solution.
Rename AssemblyLine	Use this option to rename the selected AssemblyLine.
Delete AssemblyLine	Use this option to delete AssemblyLines.

Option	Description
Create RunReport	Use this option to create RunReport to send email about status of the AssemblyLine executions.

Adding solution description

Use the Solution Editor of Dashboard to add important information about your data integration solution.

Procedure

1. In the navigation pane of Dashboard window, select the solution.
2. Click **Solution** > **Configure**.
3. In the Solution Editor, select **Solution Description** and click **Edit Description**.
4. Type the description in the text area.
5. Click **Close**.
6. Click **Save**.

Configuring an AssenblyLine schedule

Use the AssemblyLine Editor to create or configure the Dashboard scheduler to run AssemblyLines of your data integration solution at the specified time.

Creating a schedule

Procedure

1. Select the AssemblyLine from the tree structure.
2. In the AssemblyLine Editor, click the **Schedule** tab.
3. Click **Create Schedule**.
4. Define the following settings:

Option		Description
Schedule	**Automatically start when solution starts**	Use this option to run the AssemblyLine based on the execution option selected.
	Don't start if already running	
	Terminate schedule if assemblyline fails	
Month	**Every Month**	Use this option to select the schedule execution months.
	Select Months	
Day	**Every Day**	Use this option to select the schedule execution days.
	Weekdays	
	Select Days	
Hours/Minutes/ Seconds	**Hours**	Use this option to select the schedule execution timing.
	Minutes	
	Seconds	
Share Logging		Use this option to specify whether the logging between the scheduler and AssemblyLines are to be shared or not.

5. Click **Save**.

Deleting a schedule

Procedure

1. Select the AssemblyLine from the tree structure.
2. In the AssemblyLine Editor, click the **Schedule** tab.
3. Click **Delete Schedule**.
4. Click **Save**.

Running and stopping Dashboard Scheduler

Procedure

1. Select the AssemblyLine from the tree structure.
2. In the AssemblyLine Editor, click the **Run** tab.
3. To run the AssemblyLine, click **Run**. The output is shown in the log viewer.
4. To stop the AssemblyLine, click **Stop**.

Configuring a connector

Use the Connector Editor to configure connector details such as mapping information of the attributes and the connection details, to suit your configuration requirements.

Modifying connection details

Procedure

1. Select the connector from the tree structure of your solution.
2. Click the **Connector** tab.
3. To view only the important fields, click **Less**.
4. To change the connector type, click **Choose Component** and select the connector from the **Select Connector** list.
5. Depending on the requirements, modify the connection settings.
6. To test the connection to the data source, click **Test Connection**.
7. Click **Save**.

Modifying attribute mapping

Procedure

1. Select the connector from the tree structure of your solution.
2. Click the **Attribute Map** tab.
3. To add an attribute:
   a. Click **Add**.
   b. Select a work attribute from the list or type a new name in the text filed.
   c. Click **OK**.
4. To read connector data and to show it in the attribute map, click **Read Next**.
5. To close connection to the data source, click **Close Connection**.
6. To modify the selected attribute, click **Actions**.
   a. To edit the assignment for the selected attribute, click **Edit Attribute** and modify the following assignment type:
      - **Simple Assignment –** you can assign an attribute from the source attribute list.
      - **Scripted Assignment –** you can write a script to the attribute in the text area.
   b. Click **Close**.
   c. To map the selected attribute, click **Map attribute**.
   d. To remove the mapping for the selected attribute, click **Unmap attribute**.
7. Click **Save**.

Dashboard EasyETL

The Dashboard EasyETL feature can be used to create and configure simple data integration solutions.

The Dashboard EasyETL provides a user interface to build and configure simple AssemblyLines with a single source and a target connector. An EasyETL solution can be:

- Scheduled in the Dashboard Scheduler
- Saved as a template
- Included in RunReports
- Deployed and monitored in the Dashboard solution monitor

Configuring EasyETL solutions

Use the EasyETL Editor to configure and add schedules to an EasyETL solution.

Procedure

1. In the navigation pane of Dashboard window, select the EasyETL solution.
2. Click **Solution** > **Configure**.

   Alternatively,

   a. Right-click on the selected solution.

   b. From the menu, click **Configure**.
3. In the Solution Editor, select the EasyETL solution.
4. Add a description to the solution. See the "Adding solution description" on page 231 topic for more details.
5. In the EasyETL Editor, select the solution and click **Configure**.
6. From the **Connector** list, select a component for the source connector and target connectors.
7. Specify the configuration details for the selected connectors in the form.
8. Select a parser from the **Parser** list.

   Note: The **Parser** list is active only when the selected connector requires or can use a parser.

   To establish the connection and browse the data:

   a. To establish a connection and read the first 25 records of the connector, click **Connect**.

   b. To view the next 25 records, click **Next**.

   c. To toggle the data record view to show one record at a time, click **Toggle View**.

   d. To view the next records in the toggled view, click **Next**.

   e. To view only the selected attributes in the table:

      1) Click the **Configure Options** icon on the menu bar.

      2) In the Configure Options window, select the attributes.

      3) To change the order of the attributes, click the up or down arrows.

      4) Click **OK**.
9. Click **Back** to return to the EasyETL Editor.
10. To view the attributes of the configured connectors, click **Discover**.
11. To map selected source attributes to the target attributes, click **Map**. The following mapping type is created based on your source attribute selection:
    - If there is a single selected attribute in the source view and also in the target view, a one-to-one map is created between the two attributes.
    - If multiple attributes are selected in the source view and none selected in the target view, a one-to-one map is created for each attribute. If there are no attributes in the target view, they are created.

- If multiple attributes are selected in the source view and a single attribute selected in the target view, do one of the following tasks:
  - Click **Copy** to copy source attributes to its equivalent target attributes.
  - Click **Merge** to merge all values from source into one attribute in target.
  - Click **Concatenate** to concatenate the values from the source attribute as a single value to the target attribute.

12. Click **Save**.

Server Configuration

Use the Server Information window to view and modify the server configuration.

You can view the following server properties and configure the behaviors in the Server Information window:

- Specifying maximum number of log files to be created and maintained
- Starting and stopping the Tombstone Manager
- Defining LDAP Server settings to authenticate users to access the Dashboard application
- Viewing server information such as Tivoli Directory Integrator version with build date, host name, IP address, Server boot time, OS name, and server ID
- Viewing connectors and parsers installed on the Tivoli Directory Integrator Server
- Viewing contents of various system stores that are in use by the Tivoli Directory Integrator Server

At the lower left corner of Dashboard window, you can view the version information of the application. You can also monitor the following metrics to get a snapshot of how the application is performing:

- Memory usage pie chart
- Active thread count for server process

Configuring log settings

Use the Log and Tombstones tab to enable or disable the default logger and to specify the maximum number of log files to save.

Procedure

1. In the Dashboard window, click **Actions** > **Show Server Details** or click **More**.
2. Click the **Log and Tombstones** tab.
3. Define the following settings:

Option	Description
Default Logger	Enables the default logger **Note:** Use the Default Logger, which is appended to every AssemblyLine that runs, if you want individual log files for your AssemblyLines.
Maximum Number of Log Files	Specifies the maximum number of log files that the logger can save.

4. Click **Update**.

Configuring tombstones

Use the Log and Tombstones tab to start the Dashboard Tombstone Manager, which creates tombstone records.

About this task

Tombstone Manager of Dashboard creates tombstone records for each AssemblyLines as they terminate. A tombstone record contains statistics such as number of entries read by Iterators, number of records skipped, and number of records updated. The statistics can be used to graphically show workload over time.

Procedure

1. In the Dashboard window, click **Actions** > **Show Server Details** or click **More**.
2. To generate the tombstone records, click **Start Tombstone Manager**.

   Note: To stop Tombstone Manager, restart the Tivoli Directory Integrator Server.

Configuring Dashboard security settings

Use the Security and Connection tab to configure LDAP Server for authenticating users for both local and remote connections.

Procedure

1. In the Dashboard window, click the **Actions** > **Show Server Details** or click **More**.
2. Click the **Security and Connection** tab.
3. Define the following settings:

Option	Description
Local	Specifies connections from local host
Remote	Specifies connections from non-local host
LDAP Host Name	Specifies name of the LDAP Server
LDAP Search Base	Specifies the LDAP search base name to locate users
LDAP Group DN	Specifies the LDAP Group DN name to check for the membership of the users

4. Click **Update**.

Viewing installed components

Use the Installed Components tab to view the components such as connectors and parsers, which are installed on the Tivoli Directory Integrator Server.

Procedure

1. In the Dashboard window, click the **Actions** > **Show Server Details** or click **More**.
2. To view the list of components, click the **Installed Components** tab.

Viewing system store data

Use the Server Stores tab to view the contents of the various system stores that are in use by the Tivoli Directory Integrator Server.

Procedure

1. In the Dashboard window, click the **Actions** > **Show Server Details** or click **More**.
2. To view the list of server stores, click the **Server Stores** tab.

Dashboard RunReports

Use the Dashboard RunReport feature to create automated email reports of the AssemblyLine execution history.

You to create an AssemblyLine that runs based on the set schedules to generate automated email reports for the monitored AssemblyLines. The RunReport AssemblyLine uses Tivoli Directory Integrator tombstones database to determine whether the AssemblyLines are run since the last time the RunReport was executed.

The email report contains information about execution history that indicates whether the monitored AssemblyLines are successfully run or not. This report is periodically mailed to the specified recipient.

Creating RunReports

Use the Create RunReport window to create a RunReport AssemblyLine. This AssemblyLine is used to generate automated email reports. The email reports contain information about execution history for the monitored AssemblyLines and are periodically sent to the specified recipients.

Creating and scheduling a RunReport

Procedure

1. From the Solution Editor, click **Actions** > **Create RunReport** .
2. In the Create RunReport window, type a name for RunReport AssemblyLine in the **Name** field.
3. Click **OK**.
4. In the AssemblyLine Editor, define the following settings:

Option		**Description**
Schedule	**Automatically start when solution starts**	Runs the AssemblyLine based on the execution option selected
	Don't start if already running	
	Terminate schedule if assemblyline fails	

Month	**Every Month**	Specifies the schedule execution months
	Selected Month(s)	

Day	**Every Day**	Specifies the schedule execution days
	Weekdays	
	Selected day(s)	
Hours/Minutes/ Seconds	**Hours**	Specifies the schedule execution timing
	Minutes	
	Seconds	
Subject (success)		Specifies the email subject line for successful execution of the monitored AssemblyLines **Note:** The generated report uses the subject line based on whether the monitored AssemblyLines are run successfully or not since the last generated report.
Subject (failure)		Specifies the email subject line to indicate failed execution of the monitored AssemblyLines
Sender Address		Specifies the email address of the sender

Recipient Address	Specifies the email address of the recipient **Note:** Specify a comma delimited or semicolon delimited list for multiple email addresses.
SMTP Host	SMTP host parameter that accepts SMTP connections and transfers mail to the recipient
Monitor AssemblyLines	A comma-separated list of AssemblyLine names to be monitored
Share Logging	Specifies whether the logging between the scheduler and AssemblyLines are to be shared or not

5. Click **Save**.

Deleting a schedule

Procedure

1. Select the RunReport AssemblyLine from the tree structure.
2. In the AssemblyLine Editor, click the **Schedule** tab.
3. Click **Delete Schedule**.
4. Click **Save**.

Running and stopping RunReport Scheduler

Procedure

1. Select the RunReport AssemblyLine from the tree structure.
2. In the AssemblyLine Editor, click the **Run** tab.
3. To run the AssemblyLine, click **Run**. The output is shown in the log viewer.
4. To stop the AssemblyLine, click **Stop**.

Configuring and browsing connector data

Use Browse Data page in the Solution Editor to configure a connector, browse data sources of the connector, or to create a data integration solution.

Procedure

1. In the navigation pane of Dashboard window, click **Browse Data**.
2. In the Browse Data window, from the **Connector** list, select a connector to configure and browse its contents.
3. Select a parser from the **Parser** list.

   Note: The **Parser** list is active only when the selected connector requires or can use a parser.
4. Configure connector details in the form on the right side of the window.
5. To create a solution for the selected connector, click **Create Integration**.
6. To establish a connection and read the first 25 records of the connector, click **Connect**.
7. To view the next 25 records, click **Next**.
8. To toggle the data record view to show one record at a time, click **Toggle View**.
9. To view the next records in the toggled view, click **Next**.
10. To view only the selected attributes in the table:
    a. Click the **Configure Options** icon on the Browse Data window.
    b. In the Configure Options window, select the attributes.
    c. To change the order of the attributes, click the up or down arrows.
    d. Click **OK**.

Solution Monitor

Use the Dashboard Monitor Editor to track progress of AssemblyLine execution and view execution history of your solution.

You can view AssemblyLines of the selected solution in the Monitor Editor, which presents information of both current and past performance of the AssemblyLines. You can monitor the following activities:

- Real-time status monitoring of the AssemblyLines, which includes:
  - Start time of the AssemblyLine execution
  - End time of the AssemblyLine execution
  - Scheduled time for the next run of the AssemblyLine
  - Number of data objects that are processed in the AssemblyLines
- Execution history of the AssemblyLines, which is graphically shown as workload over time
- Log files to record results and issues for each AssemblyLine execution
- Tombstones records for all AssemblyLines in the solution
- Server log file (`ibmdi.log`)

Starting and stopping the AssemblyLines

Use the Dashboard Monitor Editor to start and stop the AssemblyLines.

Procedure

1. In the Dashboard, select the installed solution.
2. Click **Solution** > **Monitor**.

   Alternatively,

   a. Right-click on the selected solution.

   b. From the menu, click **Monitor**.
3. In the Monitor Editor, select the AssemblyLine to perform the following actions:
   - To start an AssemblyLine, click **Start**.
   - To stop an AssemblyLine, click **Stop**.

   The execution details are shown in the editor. The green icon with AssemblyLine name indicates that the AssemblyLine is active.

Viewing AssemblyLine execution history

Use the Dashboard Monitor Editor to view the AssemblyLine execution history in the form of a graph.

Procedure

1. In the Monitor Editor, select the AssemblyLine.
2. Click **History** or double-click the selected AssemblyLine. The AssemblyLine execution details are shown in a graphical view as workload over time.
3. Select any of the following counters to graph:
   - Get
   - Errors
   - Add
   - Delete
   - Lookup
   - Modify
4. Click the node on the graph to view the log file for a specific run. You can also view the execution statistics for that run as a tooltip.

Viewing tombstone records

Use the Tombstones tab in the Dashboard Monitor Editor to view the tombstone records that are created when the execution of AssemblyLines is completed.

Procedure

1. Go to Monitor Editor of Dashboard.
2. To view the recorded tombstones, click the **Tombstones** tab.

   Note: You can view the tombstone records only when the Tombstone Manager is running. For more information, see the "Configuring tombstones" on page 234 topic.

Viewing log files

Use the Log Files tab in the Dashboard Monitor Editor to view the log files created for each AssemblyLine execution. The log files are used to quickly and easily analyze problems and debug any issues.

Procedure

1. In the Monitor Editor, click the **Log Files** tab. The log files for all the AssemblyLine execution are shown as a tree structure.
2. To view the contents of the log file, select the log file from the tree structure and double-click.
3. To search for specific information in the log contents, type the text you want to search in the **Search** field.
4. Select the **Include source** check box to message source or the component that logged the message.
5. Click **Options** to set the following details:

Option	Description
Page Size	
Message Types	Select any of the following message types to be included in the log file: • INFO • WARN • ERROR • DEBUG
Display Options	Select options to show date, time, message source, or line numbers in the log file: • Show Date • Show Time • Include Source • Show Line Numbers

Appendix A. What's new for IBM Tivoli Directory Integrator version 7.1.1

Use this section to see an overview of significant new features and enhancements for version 7.1.1 of IBM Tivoli Directory Integrator.

This chapter is an outline of significant changes and new features of this release. This outline of new and changed features gives you a quick overview on how changes in this version have been designed to improve IBM Tivoli Directory Integrator version 7.1.1.

- New Features, including, but not limited to:
  - Configuration Editor enhancements
  - New Parsers, Connectors, and Function Components
  - Web based Deployment Dashboard
  - AssemblyLine Scheduler
- Enhancements, including, but not limited to:
  - Server, Server API, and Server components
  - Parsers, Connectors, and Function Components

  The following sections provide access to summaries of IBM Tivoli Directory Integrator version 7.1.1 new and enhanced features:
  - "Overview of IBM Tivoli Directory Integrator version 7.1.1"
  - "Deprecated or removed functionality and components" on page 242
  - "Configuration Editor" on page 243
  - "Server, Server API, and Server Components enhancements" on page 244
  - "Javadocs, tooling and scripting" on page 245
  - "Administration and Monitoring Console and Action Manager" on page 243
  - "Installation enhancements" on page 244
  - "Password plug-in enhancements" on page 246
  - "Revised logging mechanism" on page 246

Overview of IBM Tivoli Directory Integrator version 7.1.1

Review this section to see a high-level overview of IBM Tivoli Directory Integrator functionality.

IBM Tivoli Directory Integrator (referred to as TDI hereafter) is a tool for real time synchronization of repositories of data, with a special focus on identity data, including directories, databases, and operating system repositories.

As the schema for identity data varies between repositories, TDI also performs transformations on the identity records as they are synchronized. Attribute flow rules and data transformations require configuration, and TDI provides a development environment to specify these using scripting.

TDI is more a development tool than an off-the-shelf end user product. It is also bundled with products such as Tivoli Identity Manager (TIM), IBM Tivoli Directory Server (TDS), Tivoli License Manager (TLM), TAMeb, Lotus Workplace, Lotus Domino, WebSphere EPCIS (RFID), and CCMDB to extend the reach/functionality of those products.

The TDI 7.1.1 release focuses on the following themes as described in marketing requirements:

Tivoli integration

- Reduces Tivoli development and deployment costs
- Increases customer portfolio value
- Increases bundle value

Reduce entrance hurdle

- Easier to demo and test, therefore to sell
- Facilitate enterprise uptake (up-sell)

Enable viral marketing

- Increases up-sell opportunities
- Increases bundling

Product improvements have been implemented in the following areas:

- Browser based Deployment Dashboard
- Creation of simple AssemblyLine in a browser
- Data integration components
- REST API interface
- Eclipse tooling improvements
- New Connectors, Parsers, and Function Components
- Miscellaneous small Component/Connector level enhancements based on requirements from external databases

Deprecated or removed functionality and components

Use this section to see an overview of functionality or components either deprecated or removed from IBM Tivoli Directory Integrator V7.1.1.

The term *deprecated* indicates that the item is marked for removal in a future version of Tivoli Directory Integrator.

Items deprecated in V7.0, and continues to be deprecated in V7.1.1 are:

- **ListenMode=true** in the TCP Connector. Use TCP Server Connector instead.
- **Need Client Authentication over SSL** parameter in TCP Connector.
- **Connection Backlog** parameter in TCP Connector.
- XMLToSDO Function Component
- SDOToXML Function Component
- Direct usage of the Memory Queue Function Component. Use the Memory Queue Connector, or use the system object directly to create a new pipe.
- Old HTTP Client Connector
- Old HTTP Server Connector

Item deprecated in V7.1, and continues to be deprecated in V7.1.1 is:

- IBM WebSphere® MQ Everyplace® (MQe). This lightweight message queue will be removed from the future version of IBM Tivoli Directory Integrator, and a suitable replacement will be bundled.

Similar to V7.1, the following operating system is not supported in IBM Tivoli Directory Integrator V7.1.1:

- HP-UX v11 PA-RISC

Note: HP-UX v11 Integrity continues to be supported with regard to the Tivoli Directory Integrator Server and the Administration and Monitoring Console. The Configuration Editor and Password Synchronization plug-ins are not supported.

Configuration Editor

Use this section to understand the newly designed Eclipse Configuration Editor, and review improvements for version 7.1.1.

Configuration Editor overview

IBM Tivoli Directory Integrator develops fashion solutions that emerge from the integrated development environment (IDE) of the Configuration Editor. Solutions designed using the Directory Integrator's Configuration Editor produce runtime configurations that can be run by the IBM Tivoli Directory Integrator Server. The Configuration Editor uses Eclipse projects as the repository for a single runtime configuration that is built from the various files in the project.

The Configuration Editor for IBM Tivoli Directory Integrator version 7.1.1 is an Eclipse plug-in. Pre-v7.0 releases of the IBM Tivoli Directory Integrator were standalone Java applications based on the Swing framework.

The Eclipse Configuration editor reorganizes how the Configuration Editor manages resources within a project.

New features of the Configuration Editor

The following section shows highlights of the key changes implemented in the IBM Tivoli Directory Integrator Configuration Editor version 7.1.1.

- The IBM Tivoli Directory Integrator Configuration Editor has the following new and enhanced features:
  - AssemblyLine Sequencer - this new feature facilitates stringing multiple AssemblyLines together with a minimum of conditional behavior. It is implemented as a dynamically constructed AssemblyLine using a customized user interface, where developers can create and edit the sequences. Sequences can be launched and managed just like a normal AssemblyLine.
  - TDI Scheduler - using this new feature, you can schedule the execution of AssemblyLines.

Administration and Monitoring Console and Action Manager

Use this section to understand the newly designed Administration and Monitoring console, deployed in the Integrated Solutions Console, and review improvements for the Action Manager.

Administration and Monitoring Console overview

The Administration and Monitoring Console (AMC) is a web-based application for monitoring and managing remote TDI solutions. Use the Administration and Monitoring Console to start, stop, and manage Configs and AssemblyLines remotely. IBM Tivoli Directory Integrator also ships an Action Manager with the Administration and Monitoring Console. The Action Manager is a stand-alone Java application that interacts with the Administration and Monitoring Console database and uses the Directory Integrator remote server API to manage remote AssemblyLines. The current Action Manager, bundled with IBM Tivoli Directory Integrator 7.1.1 Administration and Monitoring Console, supports TDI 7.1.1, TDI 7.1, TDI 7.0, TDI 6.1, TDI 6.1.1 and TDI 6.0. Note that the Action Manager for IBM Tivoli Directory Integrator 7.1.1 supports Directory Integrator versions 6.1 and 6.0 with some restrictions. Versions prior to 6.0 are not supported.

The IBM Tivoli Directory Integrator 7.1.1 Administration and Monitoring Console user interface is deployed into the Integrated Solutions Console (ISC). The Integrated Solutions Console is a GUI framework for deploying portlet-based administrative consoles. The Integrated Solutions Console comes in a standard edition (SE) and in an advanced edition (AE).

There are no new Administration and Monitoring Console features, apart from bug fixes, implemented in IBM Tivoli Directory Integrator 7.1.1.

Action Manager

Action Manager provides a framework for configuring actions to occur when certain conditions, defined in triggers, are satisfied in the Administration and Monitoring Console. There are no specific new Action Manager features, apart from bug fixes, implemented in IBM Tivoli Directory Integrator version 7.1.1.

Server, Server API, and Server Components enhancements

Use this section to see changes to the IBM Tivoli Directory Integrator Server, Server API, and Server Components for version 7.1.1.

IBM Tivoli Directory Integrator Server, Server API, and Server Components

The IBM Tivoli Directory Integrator server is an application that performs services for connected clients as part of client-server architecture. Server applications are programs that accept connections so that they can answer service requests by sending back responses. The server application programming interface (API) is a way for clients to access server functionality programmatically. Through the server API, a client can instruct the Directory Integrator server to perform tasks such as run an AssemblyLine or shut down the server. The clients of the server API can be internal or external. Processes of the internal server run in the process of the IBM Tivoli Directory Integrator server. Processes of the external server are separate programs running on a remote server.

The following enhancements accompany IBM Tivoli Directory Integrator version 7.1.1 servers:

- Support for automatic failover to an alternate connector if there is a connection problem
- A service that optionally verifies the properties that exist at run time
- Enhancement to REST interface to provide access to all of the methods of local APIs
- Apache ActiveMQ support for light weight MQ needs
- Solution Directory of the user is the default location of System Store database, in network mode. For more information, see the “Default location of System Store” section in “System Store” chapter of *IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide*

Installation enhancements

Use this section to understand the newly designed installation features for IBM Tivoli Directory Integrator version 7.1.1.

Rewrite TDI Installer

Major features of the new and enhanced installation for IBM Tivoli Directory Integrator version 7.1.1 are captured in the following improvements:

- IBM JRE 6.0 SR9 is used for the installer and for all TDI components.
- Modified updateInstaller.jar file is used to support rollback of fix pack installation.
- 64-bit JVM is bundled for 64-bit AIX platforms.

Parsers, Connectors and Function Components

Use this section to understand the new and enhanced Parsers, Connectors, and Function Components.

New Parsers, Connectors and Function Components

IBM Tivoli Directory Integrator version 7.1.1 increases functionality by providing new functional components, as well as enhancements to existing components (Connectors, Parsers, and Function Components). The following Connectors and Parsers are new in IBM Tivoli Directory Integrator version 7.1.1.

- TPAE Connectors
  - Simple Tpae IF Connector - reads from and writes data to the Maximo Integration Framework.
  - Tpae IF Connector - works with hierarchical entries to read/write complete Maximo Object Structure once.
  - Tpae IF Change Detection Connector - receives change notifications on a configurable TCP port for HTTP requests, from Maximo based systems.
- CCMDB Connector - reads, writes, deletes, or searches configuration items (CIs) and relationships between them in the IBM Tivoli Change and Configuration Management Database (CCMDB) via JDBC.
- Deployed Asset Connector - integrates data with Tivoli Asset Management for IT (TAMIT) database via JDBC.
- TADDM Connectors:
  - TADDM Connector - communicates with the Tivoli Application Dependency Discovery Manager (TADDM) server using TADDM Java APIs (TADDM SDK).
  - TADDM Change Detection Connector - directly detects changes from the TADDM database using TADDM Java APIs (TADDM SDK).
- File Management Connector - reads and modifies file system structures and file system metadata available on the system it runs on.
- LDAP Group Member Connector - retrieves the members of LDAP groups. This component returns the user entries of group members, and not the group entries themselves. You can access information about the containing group and the parent/ancestor groups via properties (Link).
- File Transfer Function Component - can be used to transfer files from the specified source system to the target system.
- JSON Parser - reads and writes entries using the JavaScript Object Notation (JSON) format.

Enhanced Parsers, Connectors and Function Components

The following Connectors, Parsers and Function Components are improved in IBM Tivoli Directory Integrator version 7.1.1.

- The FTP Client Connector has been enhanced to support explicit FTPS mode.
- The IT Registry CI and Relationship Connector has been enhanced to support registration of Abstract Resources.
- The existing File System Connector has been renamed to File Connector.

Javadocs, tooling and scripting

Use this section to understand the newly designed coding guidelines for Javadocs and improve the ability of IBM Tivoli Directory Integrator to work with PKI encryption.

Javadocs, tooling and scripting enhancements

- Browser-based Deployment Dashboard to configure and manage TDI solutions.
- Enhanced stylesheet to generate verbose AL reports. The verbose AL reports provide more information about the AssemblyLine.
- JavaScript Editor has been enhanced to support Function hints.

Java Virtual Machine

Tivoli Directory Integrator V7.1.1 is now deployed on Java 6, which is J2SE 6.0 SR9.

Password plug-in enhancements

Use this section to see the enhanced capabilities of the Password plug-ins.

Plug-in enhancements

The following enhancements have been implemented to the IBM Tivoli Directory Integrator Password plug-ins:

Enhanced Java Proxy to support provisioning of custom attributes in password synchronization message

> Normally, the synchronization data includes only the user name, password values and the types of the password change (for example, add, delete, or update). A few possible options for the content of a custom attribute are, hard-coded string, hostname or IP address of the system, where the Password Synchronizer is deployed, or the timestamp when the Java Proxy has received the notification. Java Proxy has been enhanced to support custom attributes in password synchronization data.

Revised logging mechanism

Use this section to understand the new logging mechanism designed for logging messages when startAL() script is run from the Configuration Editor.

For detailed information, see the “Known limitations” section in “Configuration Editor” chapter of *IBM Tivoli Directory Integrator V7.1.1 Problem Determination Guide*.

Appendix B. IBM Tivoli Directory Integrator terms

Action Manager (AM)

Action Manager is a stand-alone Java application used to configure failure-response behavior for IBM Tivoli Directory Integrator solutions. AM executes *rules* defined with AMC v3. An AM rule consists of one or more *triggers* that define a failure situation, such as the termination of an AL that should not stop running, or if an AL has not been executed within a given time period, and so forth Furthermore, each rule also defines *actions* to be carried out in case of this failure. Actions include operations like sending events or e-mail, starting ALs (locally or remotely) and changing configuration settings. Action Manager requires IBM Tivoli Directory Integrator v7.1.1 and AMC v3.

Accumulator

A special object that can be set in a Task Call Block (TCB) for use when starting another AssemblyLine either using a scripted call, or a component like the AssemblyLine Connector or the AssemblyLine FC. The Accumulator is either a collection of Work Entry objects handled by the called AL, or it is a component that is called to output each Entry. Accumulator handling is done at the end of each AssemblyLine Cycle.

AES Shorthand for Advanced Encryption Standard. AES is an encryption algorithm for transmitting sensitive, but unclassified, content by U.S. Government agencies.

Adapter

Adapter is a word used in many contexts and with different meanings. A *TDI Adapter* refers to an AssemblyLine that is packaged as a single Connector. Creating a TDI Adapter requires setting up an AssemblyLine that is written to perform, and expose, one or more business-related tasks. Each task is defined as an AssemblyLine Operation (for example, "EnableAccount", or "ReturnGroupMembers"). This AL can then be *published* for sharing, and can be leveraged by the AssemblyLine Connector that offers mode settings reflecting these operations[12].

AL Shorthand for AssemblyLine.

Administration and Monitoring Console (AMC)

AMC is a browser-based console for managing and monitoring solutions. AMC Version 3, which is part of IBM Tivoli Directory Integrator V7.1.1, runs on the WebSphere Application Server (enterprise and express versions), as well as Tomcat. Each AMC version is designed to work with a specific release of TDI and may be incompatible with other versions. AMC V.3 is designed for IBM Tivoli Directory Integrator V7.1.1; however, it also works with TDI 6.1 and TDI 6.0 (albeit with some restrictions). AMC V.2 works with TDI V6.0 and AMC V.1 runs with TDI V5.2.

API Application Program Interface. A way of programmatically (local or networked) calling another application, as opposed to using a command-line or a shell script.

Appender

Appender is a Log4J term (a third party Java library) for a module that directs log messages to a certain device or repository. In IBM Tivoli Directory Integrator you control logging for your AssemblyLines by creating and configuring *Appenders*, either under the Logging tab of a specific AL, or under **Solution Logging and Settings** > **Log** in the Config Editor to control how all AssemblyLines in a project do their logging.

AssemblyLine (AL)

The basic *unit-of-work* in a Tivoli Directory Integrator solution. Each AL runs as a JVM thread in the Server and is made up of a series of AssemblyLine components (one or more Connectors, Functions, Scripts, Attribute Maps and Branches) linked together and driven by the built-in workflow of the AssemblyLine.

[12] AL Operations are also accessible using the AssemblyLine Function Component.

AssemblyLine Component

This term denotes an TDI component used to construct AssemblyLines. The possible Components are:

- Connectors
- Function Components
- Script Component
- Attribute Map Component
- Branches (including Loops and Switches)

The components list in an AssemblyLine is divided into two sections: *Feeds* where the Work Entry for each AL cycle is created from input data by a Connector in Iterator or Server mode, and the *Flow* section that holds the Connectors (in any mode except Server), Functions, Attribute Maps and Scripts providing the additional data access and processing.

AssemblyLine Operation

A business task that is implemented by an AssemblyLine and published using its Operations tab. Each Operation can have its own Input and Output Attributes Maps for defining the parameters expected when this Operation is invoked (Input Map), as well as those returned (Output Map). This is also called the *Schema* of the Operation.

AssemblyLine Phases

An AssemblyLine goes through three phases:

Initialization

At this point the TDI Server uses the "blueprint" for the AssemblyLine in the Config to create the various components as well as set up the AL environment, including processing the TCB, starting the script engine of the AL and invoking theProlog Hooks of the AL. All components that are configured for Initialization At Startup are initialized at this point causing their Prolog Hooks to get run as well.

Cycling

The AL workflow drives each of its components in turn, starting each cycle by invoking the On Start of Cycle Hook. Then the currently active *Feeds* Connector reads in data, creates the Work Entry and passes it to the *Flow* section. The Work Entry is passed from component to component until the end of *Flow* is reached, at which time control is returned to the start of the AssemblyLine again[13]. Cycling continues until an unhandled error occurs or there is no more data available, for example, when the Iterator reaches End-of-Data.

Shutdown

When cycling stops then the AssemblyLine goes into Shutdown phase: Epilog Hooks are called and all initialized components are closed down, which flushes output buffers and executes their Epilog Hooks as well. Finally, the AssemblyLine closes down its environment and its thread terminates.

AssemblyLine Pool

A collection of AL *Flow* sections that can be configured to allow a Server mode Connector to service more clients. Available for ALs that use Server mode Connectors and is set up in the AssemblyLine Settings window of the AssemblyLine.

Attribute

Part of the TDI Entry data model. Attributes are carried by Entry objects (Java "buckets," like the Work Entry) and they can hold zero or more *values*. These *values* are the actual data values read from, or written to, connected systems, and are represented in TDI as Java objects.

[13] If the current cycle was fed by a Server mode Connector, then the reply is created by the Output Map of that Server mode Connector, and sent to the client.

Attribute Map (AttMap)

An Attribute Map is a list of rules (individual Attribute mapping instructions) for creating or modifying Attributes in an Entry object typically based on the values of Attributes found in another Entry object. Components like Functions and Connectors have an Input Map for taking data read into local cache (the conn Entry) and use this to define Attributes in the Work Entry. These components also have an Output Map that takes Attributes carried by the AssemblyLine (in its Work Entry) and use this to set up the conn Entry that is used by the output operation of the component. Attribute Map components use the Work Entry as both the source and target of the mappings.

Attributes can be mapped in one of three ways: Simply (copying values between Attributes), Advanced (using a snippet of JavaScript), or with a TDI Expression.

Attribute Map component

A free-standing list of individual Attribute mappings that take values from the Work Entry and use them to create and update other Attributes in the Work Entry. They can be tied to Connector and Functions to define their Input or Output Maps. Note that Input and Output Maps can be copied to the library as AttMap components for reuse.

Best Practices

Recommended methodology and techniques for working with TDI. These include the ABCs: Automation, Brevity and Clarity:

Automation

Use the automated features of TDI in preference to your own custom scripted logic whenever possible; for example, using Branches/Loops instead of extensive scripting in Hooks. Not only does this make your solution easier to read, maintain, and can step through with the AL Debugger, but your solution benefits directly as built-in logic is strengthened and extended with each new release.

Brevity

Keep your AssemblyLines as short and simple as possible, as well as your script snippets. Break complex logic into simpler patterns that can be tested individually and reused in other solutions.

Clarity

Choose legibility over elegance. Write solutions for others to read and maintain.

Branches

A construct used to control the flow of logic in an AssemblyLine. IBM Tivoli Directory Integrator V7.1.1 provides three types of Branches:

- Simple Branches (IF, ELSE-IF and ELSE)
- Loops (Connector-based, Attribute-based or Conditional)
- Switches (for example, switching on the Work Entry delta operation code, or the Operation an AL is called with).

CBE Common Base Event. A term used in the Common Base Infrastructure. See "Common Base Event" in the chapter about the CBE Generator Function Component in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

CEI The IBM Common Event Infrastructure. See "The Common Event Infrastructure", in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Change Detection Connector (CDC)

A Connector that returns changes made in the connected system. Typically, a CDC can be configured to return only a subset of Entries: new, modified, deleted, unchanged or a combination of these. Some CDCs provide only the changed Attributes in the case of a modified

Entry, while others return them all. Change Detection Connectors also tag the data with special *delta operation codes* to indicate what has changed, and how.[14]

CLI Command-line Interface, such as the `tdisrvctl` utility.

cipher A cipher is any method of encrypting text to hide its readability and its meaning. The resulting encrypted text message is called ciphertext.

ciphertext
Ciphertext is encrypted text, the result of applying a cipher, or an encryption.

Components
The architecture of IBM Tivoli Directory Integrator is divided into two parts: generic functionality and technology-specific features. Generic functionality is provided by the TDI *kernel*, which provides automated behaviors to simplify building integration solutions. The kernel also lets you extend or override these behaviors as desired, as well as doing the housekeeping for your solution: logging and tracing, Hooks for error handling, API and CLI access, and so forth. Technology-specific "intelligence" is handled by helper objects called *components*, such as Connectors, Functions, Branches, Scripts and Attribute Map components. Components provide a consistent and predictable way to access heterogeneous systems and platforms, and the kernel lets you "click" together components to build AssemblyLines.

Compute Changes
A special feature of the Connector Update mode that instructs the Connector to compare the Attributes about to be written to the connected system with those that exist in this data source already; in other words, it compares the value of each Attribute in the conn Entry (the result of the Output Map) with the corresponding ones found during the Update mode *lookup* operation, which is stored in the current Entry.

Config or Config File
A collection of AssemblyLines and components that comprise a solution. A Config is stored in XML format, typically in a Config file and is written, tested and maintained using the Config Editor.

Config Editor (CE)
The graphical development environment used to write, test and maintain Configs. Configs are stored in XML format and are deployed by assigning them to one or more IBM Tivoli Directory Integrator Servers to execute.

Config Instance
A copy of a TDI Config that is running on a Server. Typically loaded only once on a given Server, TDI allows you to start the same Config multiple times if desired. Each running copy is given its own context and can be accessed individually through the API.

Config View
This term is used in the context of AMC to describe how a particular Config appears in the management screens of AMC. A Config View is a selection of the AssemblyLines and properties that are to be visible onscreen (user- or role-based), providing solution-oriented Config administration and management. Config Views can be combined to define a Monitoring View in AMC.

conn Entry
This is the local Entry object maintained by a Connector or Function. The conn Entry is used as a local cache for read and write operations, and data is moved between this cache and the Work Entry of the AssemblyLine using Attribute Maps (specifically, Input and Output Maps).

Connector
One of the component types available in TDI to build AssemblyLines. Connectors are used to

[14] For LDAP there is also a special kind of modify operation where the directory entry has been moved in the tree: *modrdn*, that is, a "renamed" entry.

abstract away the technical details of a specific data store, API, protocol or transport, providing a common methodology for accessing diverse technologies and platforms.

Unlike the other components, Connectors can perform different tasks based on their *mode* setting (for example, Iterate, Delete, Server and Lookup). Modes are provided by the AssemblyLine component part of the Connector. However, the list of modes supported is dependent on the Connector Interface.

Connector Interface

When a component is used in an AssemblyLine, a distinction must be made between the *Connector Interface* (CI), containing the "intelligence" for working with a connected system, for example, LDAP, JDBC, Lotus Notes®, and the *AssemblyLine Connector*. [15] This latter object is the AL wrapper that allows the CI to be plugged into an AssemblyLine and provides them with a consistent set of generic features, like input or output maps, Link Criteria, Hooks and the Delta Engine. See "Objects", and "Connectors" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide* for more information.

Connector Pool

Unlike the AssemblyLine Pool feature available to ALs using Server mode Connectors, a Connector Pool is a global collection of pre-initialized Connectors that can be used in multiple ALs. Note that the Connector Initialization setting "Initialize and terminate every time it is used" means that no AssemblyLine gains exclusive rights to a pooled Connector, giving you detailed control over resources used by your solution.

current Entry

This Entry object is local to a Connector Interface (just like the conn Entry) and contains the Attributes read in from a *lookup* operation (for example, as carried out by Lookup, Update and Delete modes). It is used to provide the Compute Changes feature.

Delta Engine

Available for Connectors in Iterator mode, the Delta Engine provides functionality for detecting changes in data sources that do not offer any changelog or change notification features. See Delta Operation Codes, as well as "Delta mode" on page 12 for more information.

Delta mode (for Connectors)

This Connector mode is used to the apply changes specified with delta operation codes in the Work Entry, and to do so as efficiently as possible by performing incremental modifications. Note that Delta mode is only available for the LDAP and JDBC Connectors, and does not work with Entries without a valid delta operation code. See "Delta mode" on page 12.

Delta Operation Codes

These are special values assigned to Entries, Attributes and their values to reflect change information detected in some data source. An Entry that has delta codes assigned is called a *Delta Entry*, and these are only returned by a limited set of components: Change Detection Connectors, the Delta Engine and the DSML and LDIF Parsers.[16] Delta Operation Codes can be queried and used in Branch Conditions or your own JavaScript code, and are used by Delta mode to apply all types of changes to target systems as efficiently as possible.

Derby

Apache Derby (previously known as Cloudscape) is a small footprint relational database implemented entirely in Java. Derby V10.2 is included as the default System Store for TDI.

DES

Short for Data Encryption Standard. DES is a widely-used method of data encryption using a secret key. DES is superseded by the Advanced Encryption Standard (AES).

Distinguished Name (DN)

An LDAP term that refers to the fully qualified name of an object in the directory, representing

[15] Functions are similar to Connectors in that they are divided into two parts: the Function Interface and the AssemblyLine Function. Unlike Connectors, Functions have no mode setting.

[16] Note that these Parsers only return Delta Entries if the DSML or LDIF entries read contain change information.

the *path* from the root to this node in the directory information tree (DIT). It is usually written in a format known as the User Friendly Name (UFN). The dn is a sequence of *relative distinguished names* (RDNs) separated by a single comma (,).

ECB Short for Electronic Code Book. Electronic Code Book (ECB) is a method of operation for a block cipher. In an ECB, each possible block of plaintext has a defined corresponding ciphertext value and the other way around. The same plaintext value always results in the same ciphertext value. Electronic Code Book is used when a volume of plaintext is separated into several blocks of data, each of which is then encrypted independently of other blocks. Moreover, Electronic Code Book can create a separate encryption key for each block type.

Entry An Entry is a TDI object used to carry data, and forms the core of the TDI Entry model. The Entry object can be thought of as a "Java bucket" that can hold any number of Attributes, which in turn carry the actual data values read from, or written to connected systems. Each Entry corresponds to a single row in a database table or view, a record from a file or an entry in a directory (or similar unit of data), and there are a number of named Entry objects available in the system. The Work Entry and conn Entry are the most commonly used ones, but there is also a current Entry available in some Connector modes, an error Entry that contains the details of the last exception that occurred, and an Operation Entry (Op-Entry) for accessing details of an AL operation.

Epilog A set of Hooks that, if enabled, are run during the AssemblyLine Shutdown phase. Note that the shutdown of components occurs between the two AL Epilog Hooks, which means that the Epilog Hooks of these components are all completed before the AssemblyLine Epilog - After Close Hook is called.

Error Entry
An Entry object that is created by an AssemblyLine during initialization, and contains Attributes like "status", "connectorname" (applies for all types of components) and "exception".[17] See also Error Handling.

Error Handling
Error Handling in TDI is based on the concept of *exceptions*. Exceptions are a feature of a programming language, like Java, C and C++, that lets you build error handling like a wall around your program. It also lets you fortify smaller parts within any wall, so you can add specific handling where necessary. TDI leverages the power of exception handling so that you can design the error handling in your solution the same way.

First you have the On Failure Hook of the AssemblyLine, which is called if the AL stops due to an unhandled exception.[18] This is the outer line of defense.[19] The next level is a component, given that it provides Error Hooks. Connectors actually provide two levels of handling: the mode-specific Error Hook, as well as the Default On Error (same goes for Success Hooks as well).

Finally, in your JavaScript code you can do exception handling yourself. Use the `try-catch` statement, for example:

```
try {
       myObj = someFunctionCallThatCanThrowAnException();
} catch ( excptThrown ) {
       task.logmsg("**ERROR - The call failed: " + excptThrown );
}
```

[17] The "exception" Attribute holds the actual Java exception object, in the case of an error – in which case the "status" Attribute would also be changed from a value of "ok" to "error" and "message" would contain the error text.

[18] An unhandled error is one that is *caught* in an enabled Error Hook (no actual script code is necessary). If you wish to escalate an error to the next level of error handling logic, you need to re-`throw` the exception:

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

[19] If you want to share this logic (or that in any Hook) between AssemblyLines, implement it as a function stored in script and then include them as a Global Prolog for the AL.

ERP Enterprise Resource Planning, usually indicates a software suite of programs that aims to manage enterprise resources, usually after heavy customization by the software vendor.

Exception
See Error Handling.

External Properties
A type of Property Store that uses a flat file for storing configuration settings (like passwords and other component parameter settings) outside the Config itself.

Feeds The first section of an AssemblyLine and can only hold Iterator and Server mode Connectors. The Feeds section is where the Work Entry is created from data retrieved from a connected system or client. The Feeds section is like a built-in Loop that drives the Flow section components list, once for each Entry read.

FIPS Short for Federal Information Processing Standard. TDI uses FIPS 140-2, a standard that defines requirements for cryptographic modules that handle sensitive information.

Flow The second (and usually the main) section of an AssemblyLine and holds a list of components; any type, except Connectors in Server mode. The Flow section receives a Work Entry from the currently active Feeds Connector and passes it from component to component for processing.

Function Component (FC)
One of the component types available in TDI to build AssemblyLines. Functions are used to abstract away the technical details of a specific service or method call. Typical examples are the AssemblyLine FC used to execute ALs and the Java Class FC that lets you browse jar files and call class methods. Unlike Connectors, FCs do not have mode settings.

Global Prolog
A Script component that is defined in the Scripts library folder of the workspace, and which is configured to be executed when an AssemblyLine starts up. The simplest way to do this is to select which Scripts to use with the "Include Addition Prologs - Select" button. Note that Global Prologs are executed before the Prolog Hooks of the AssemblyLine.

GUI (`ibmditk` or `ibmditk.bat`)
The term "TDI GUI" is sometimes used to refer to the Config Editor.

Hook This is a waypoint in the built-in workflow of the AssemblyLine, or of a Connector or Function, where you can customize behavior by writing JavaScript. In a Connector, the Hooks available are also dependent on the mode setting.

HTML
Hyper**T**ext **M**arkup **L**anguage. a more or less standardized way of describing and formatting a page of text on the Web. Different manufacturers' interpretations of the standard are often the cause of different renderings of a given page on various Web browsers.

HTTP **H**yper**T**ext **T**ransfer **P**rotocol. The protocol in use for the Web, another protocol on top of TCP.

IEHS IBM Eclipse Help System. Used to host the IBM Tivoli Directory Integrator documentation locally. The documentation hosted by IBM in the Documentation Library also uses IEHS.

Initial Work Entry (IWE)
An Entry that is passed into an AssemblyLine by the process that called it (for example, an AssemblyLine Connector or Function, or by using script calls like `main.startAL()`. Note that the presence of an IWE causes any Iterators in the Flow section to skip on this cycle.

Iterator
A Connector mode[20] that first creates a data result set (for example, by issuing a SQL SELECT statement, a LDAP search operation, opening a file for input) and then returns one Entry at a time to the AL for processing. Iterators can reside in the AssemblyLine Feeds section where they

[20] Connectors running in Iterator mode are often referred to as "Iterators".

drive data to Flow components. If they are placed in the Flow section then they still retrieve the next Entry from their result set for each AL cycle, but they do not *drive* AL cycling in this case.

IU Installation Unit. A term specific to Solution installation (SI). Each major component of the product is broken into separate IUs - for ease of maintenance, installation and updates.

Java Virtual Machine or JVM
IBM Tivoli Directory Integrator runs inside what is known as a Java Virtual Machine. The JVM runs programs written in the Java language, has its own memory management and is in most respects a computer within the computer.

Javadocs
A set of low-level API documentation, embedded in the source code of the product, and extracted by means of a special process during the build of the product. In IBM Tivoli Directory Integrator the Javadocs can be viewed by selecting **Help** > **Javadocs** from the Config Editor.

```
Not currently visible in TDI V7.1.1.
```

JavaScript
The language you can use to fine tune the behavior of your AssemblyLines. IBM Tivoli Directory Integrator V7.1.1 uses the IBM JSEngine, version 2.0.

JMS **J**ava **M**essaging **S**ervice. A standard protocol used to perform guaranteed delivery of messages between two systems.

JNDI **J**ava **N**aming and **D**irectory **I**nterface. See "JNDI Connector", in the *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

Link Criteria
Link Criteria represent the matching rules defined for a Connector in Update, Lookup or Delete, and they must result in a single entry match in the connected system; otherwise either an Not Found or Multiple Found exception occurs. Note that a Lookup Connector tied to a Loop is an efficient way of dealing with lookup operations where no match (or multiple matches) are expected.

LDAP **L**ightweight **D**irectory **A**ccess **P**rotocol. This protocol, which uses TCP, provides an easier way of accessing a name services directory than the older Directory Access Protocol. For example, LDAP is used in querying the IBM Tivoli Directory Server.

Memory Queue (MemQ)
The MemQ is a TDI object that lets you pass any type of Java object (like Entries) between AssemblyLines running on the same server. This feature is usually accessed through the MemQueue Connector (or the deprecated Memory Queue FC). See also System Queue for more on how to pass data between running ALs.

Message Prefix
All error messages and Info messages in IBM Tivoli Directory Integrator are prefixed with a unique message prefix. The prefix assigned to TDI is **CTGDI**.

Mode Connectors have a mode setting that determines how this component participates in AssemblyLine processing. In addition to any custom modes (implemented through Adapters) there is a set of standard modes:

- Iterator
- AddOnly
- Lookup
- Update
- Delete
- CallReply
- Server

- Delta

Dependent on the features provided by the underlying system or functionality built into the Connector, the list of modes supported by the different Connectors varies. See “Connector modes” on page 5, and "Connectors" in *IBM Tivoli Directory Integrator V7.1.1 Reference Guide* for more information about Connector modes.

Null Value Behavior
Refers to how TDI deals with Attribute mappings that result in NULL values. Null Behavior configuration can be done for a Server by setting Global/Solution properties. These Server-level settings can be overridden for an Attribute Map by clicking the **Null** button in the button bar at the top of the map; or for a specific Attribute using the **Null** button in the Details Window for its mapping.

TDI lets you both configure what constitutes a NULL value situation (for example, missing values, empty string or a specific value) as well as how to handle this.

Op-Entry (Operation Entry)
An entry that contains information about the Operation for the currently executing AL. An Op-Entry persists its value over successive cycles for the same AL run and is available for scripting using the `task.getOpEntry()` method.

Parameter Substitution
A way of specifying patterns based on Java MessageFormat class - for simpler and quicker editing. Available in various places in IBM Tivoli Directory Integrator, wherever properties are used.

Parser TDI components used to interpret or generate the structure for a byte stream. Parsers are used by attaching them to a Connector that reads or writes byte streams, or to a Function component like the Parser FC that is used to parse data in the Work Entry.

Persistent Object Store
See System Store.

Persistent Parameter Store
See Property Store.

plaintext
Plaintext is unencrypted text. In cryptography, plaintext is ordinary readable text before being encrypted into ciphertext or after being deciphered.

Prolog A set of Hooks that, if enabled, are run during the AssemblyLine Phases. You can also define Global Prologs: scripts that are run before either of the AL Prolog Hooks. Note that the "At Startup" initialization of components occurs between the two AL Prolog Hooks, which means that the Prolog Hooks of these components are all completed before the AssemblyLine Prolog - after the Initialization Hook is called. See also Epilog.

Properties
This term refers to values maintained in a Property Store and used to configure AssemblyLine and Component settings at run-time. [21]

Property Store
This is a feature for reading and writing all types of properties. This includes:

- Java-Properties, which are settings of the JVM.
- Global-Properties, IBM Tivoli Directory Integrator Server settings that are kept in a file called `global.properties` in the `etc` folder of your installation directory.
- Solution-Properties, which typically override Global-Properties and are found in a file in your solution directory called `solution.properties`.

[21] Note that an Entry object can also hold *properties* (in addition to Attribute and delta operation codes) and these can be accessed using the `getProperty()` and `setProperty()` methods of the Entry class.

- System-Properties, for keeping custom property settings (uses the System Store).

In addition, you can define your own Property Stores using a Property Connector. The Property Store feature also lets you designate one of your Property Stores as a *Password Store*, giving you automatic protection of sensitive configuration details.

Raw Connector
Deprecated term; this is now called the Connector Interface and refers to the part of an AL Connector that contains the logic needed to access a specific API, protocol or transport.

Relative Distinguished Name (RDN®)
In LDAP terms the name of an object that is unique relative to its siblings. RDNs have the form *attribute name=attribute value*. For example,

```
cn=John Doe
```

Resource Library
A simple method for sharing AssemblyLines and components between Configs. In the Config Editor, the Resources folder appears just below the AssemblyLines folder in the workspace.

RMI **R**emote **M**ethod **I**nvocation; a way of making procedure or method calls on a remote system using a network communication channel. In IBM Tivoli Directory Integrator, used by the Remote API functionality.

RSA RSA is an internet encryption and authentication system that uses an algorithm developed by Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman. The encryption system is owned by RSA Security. RSA is an algorithm for public-key cryptography, suitable both for signing and for encryption.

Sandbox
The feature of the IBM Tivoli Directory Integrator that enables you to record AssemblyLine operations for later playback without any of the data sources being present. See “Sandbox” on page 193.

SAP Used to stand for "Systeme, Anwendungen, Produkte" (Systems, Applications, Products) but today, the abbreviation just stands for itself. A large, German provider of an integrated suite of ERP applications. Mostly known for its R/3 distributed ERP software suite, but also known for its mainframe-based R/2 software.

Script Component (SC)
A Script Component is a block of JavaScript that is stored as a single component in TDI. In addition to appearing in the Scripts library folder of the workspace[22], Script Components can be dropped anywhere in the Flow section of an AssemblyLine.

Script Engine
The component that interprets the Java scripts written inside a TDI Config. IBM Tivoli Directory Integrator V7.1.1 uses the IBM JSEngine 2.0, which replaces Rhino from earlier releases.

Schema
Schema, unfortunately, can mean different although related things, depending on context. In a relational database context, a Schema is the collection of tables and objects a user has defined and owns (including content); and each table in a schema is described by a Data Definition. In an LDAP context, the Schema is the actual layout of the LDAP database, with its attributes and objects.

In addition, Connectors and Functions can have Input and Output schemas that represent the data model discovered in a connected system. Furthermore, an AssemblyLine Operation can have an Input and Output schema as well.

[22] In order to be used as Global Prologs (which are executed at the very start of Assemblyline Initialization) the Script must be in the *Scripts* library folder and selected for inclusion in the Config tab of an AssemblyLine.

In a product like TDI, which can access both relational databases as well as LDAP databases, the word Schema can therefore mean different things, depending on where it is used.

Script Connector
A Script Connector is a Connector where you write the *Interface* functionality yourself: It is empty in the sense that, in contrast to an already-existing Connector, the Script Connector does not have the base methods `getNextEntry()`, `findEntry()` and so forth implemented. Not to be confused with the Script Component.

Server (`ibmdisrv` or `ibmdisrv.bat`)
This is the part of TDI that is used to deploy and execute Configs.

Server (mode)
This is a Connector mode used for providing a request/response service (like an HTTP server). This mode also provides an AssemblyLine Pool feature to enable support for more connections or traffic.

Solution Directory
The directory in which you store your Config files, Derby databases, properties files, keystores and so forth. The solution directory is selected when you install TDI, and the filepaths used in your solution can be relative to this folder. The solution directory can be explicitly specified when you start the Config Editor or Server using the **-s** commandline option. Note that the counterpart of `global.properties` is kept in this folder and called `solution.properties`, unless your solution directory is the same as your installation directory.

SI Solution Installer. A common IBM utility for installation of many IBM products. The IBM Tivoli Directory Integrator 6.1.1 installer is one such product.

SSL Secure Socket Layer; a protocol used in Internet communications to encrypt data such that if someone where to eavesdrop on the packets going back and forth he would not be able to see what the packets contain. The protocol was invented by Netscape; and you can see if a Web page uses the SSL protocol to talk to the Web server if it has the 'https//' prefix instead of 'http'. SSL is not limited to Web pages; in fact, IBM Tivoli Directory Integrator uses it (if configured that way) to talk between different Servers and AssemblyLines if network access is called for.

State Defines the *level of participation* for an AssemblyLine component. It can be in either *Enabled* State, which means it participates in AL processing, or *Disabled* in which case the component is not used in any way.

Connectors and Functions can be set to a third State: *Passive*. Passive State causes the component to be initialized and closed during the Assemblyline Initialization and Shutdown phases, but never used during AL cycling. However, you can drive these components manually through script calls.

System Queue
A built-in queue infrastructure to facilitate the guaranteed delivery of messages between AssemblyLines, even running on different TDI Servers. By default, the System Queue uses the bundled MQe (WebSphere MQ Everyplace), but can be configured to leverage other JMS-compliant messaging systems. TDI provides a SystemQueue Connector to help you leverage this feature.

For more information about the System Queue and how to enable it, see the "System Queue" chapter in the *IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide*.

System Store
Called the Persistent Object Store, or POS in earlier TDI versions, the System Store is a relational database used to store state information, like Delta Tables (used by the Delta Engine) or Iterator state for Change Detection Connectors. It also provides the User Property Store which is accessible through the `system.setPersistentObject()`, `system.getPersistentObject()` and `system.deletePersistentObject()` methods. In the current implementation, the Derby product (previously known as CloudScape) is used. See http://db.apache.org/derby for more details.

Task By convention, all threads (AssemblyLines, EventHandlers and so forth) are referred to as *tasks* and are accessible from script code using the pre-registered **task** variable.

Note: EventHandlers are no longer a feature in IBM Tivoli Directory Integrator V7.1.1; they were, however, part of previous releases.

Task Call Block
A Java structure used to pass parameters to and from AssemblyLines. Often referred to by its abbreviation: **TCB**.

TCP **T**ransmission **C**ontrol **P**rotocol; a level 4 (transmission integrity) protocol usually seen in combination with its layer 3 (routing) Internet Protocol as in TCP/IP. A stack of protocols designed to achieve a standardized way of communicating across a network, be it local (as in on the premises) or over long distances. Originally invented and specified by DARPA, the US Defense Advanced Research Projects Agency. Successor to ARPANET, which was a network of a small number of universities and the US Department of Defense, the civil side of which was managed by the Stanford Research Institute (SRI). TCP is related to UDP.

TDI Unofficial monicker for this product, IBM **T**ivoli **D**irectory **I**ntegrator.

TMS XML
Tivoli Message Standard XML. A Tivoli standardized way of formatting messages. Each message is prefixed by a unique TMS code, which can be looked up in the Message Guide for explanation and user response. A code that ends in an "E" indicates an Error, "W" indicates a Warning and "I" indicates an Information message. All Tivoli messages issued by TDI start with the unique identifier of this product, which is "CTGDI".

Tombstone
A record or trace showing that an AssemblyLine, an EventHandler or Config has terminated. Configured through the Tombstone Manager in the CE. The trace includes a timestamp and the AL exit status. The Tombstone Manager creates a tombstone for each AssemblyLine as it terminates.

TWiki TWiki is a flexible and easy-to-use enterprise collaboration system software. Its structure is similar to the WikiPedia, except that is not linked into that. It is rather meant as an independent community resource for a group of people with common interest. There is one for IBM Tivoli Directory Integrator, at http://www.tdi-users.org.

Note: The TWiki site is a volunteer effort, and is not an official Tivoli support forum. If you need immediate assistance, contact your local Tivoli support organization.

Update
One of the standard Connector modes. Update mode causes the Connector to first perform a lookup for the entry you want to update,[23] and if found it modifies this entry. If no match is found then a new entry is added instead. See also Computed Changes.

UDP **U**ser **D**atagram **P**rotocol. A protocol used on top of the Internet Protocol (IP) which, unlike TCP does guarantee that the packet of data sent with it reaches the other end. Also see TCP.

URL **U**nified **R**esource **L**ocator. A way of defining where a resource is, be it a fileserver or a HTML page on the Web.

User Property Store
See “User Property Store” on page 209.

Value (data values and types)
See Entries, and Attribute.

[23] Data is read into both the `conn` and `current` Entry objects. After the Output Map, the contents of `conn` are now the Attributes to be written. The original entry data is still available in `current`.

WikiPedia

A Web-based worldwide encyclopedia, where (registered) users can add articles or pictures, edit them, browse them, search for applicable content, and so forth. For IBM Tivoli Directory Integrator there is one similar in functionality but not linked into the WikiPedia, a "TWiki" at http://www.tdi-users.org. The TWiki is a groupware product.

Work Entry

An Entry object that is used by the AssemblyLine to carry data from component to component.[24] This object can be accessed in script code using the pre-defined variable `work`. The Work Entry is typically built by a Server or Iterator mode Connector in the Feeds section before being passed to the AL Flow section. You can also have an Initial Work Entry (IWE) passed in if the AL was called from another process; or you can create it in the Prolog by using `task.setWork()`:

```
init_work = system.newEntry(); // Create a new Entry object
init_work.setAttribute("uid", "cchateauvieux"); // populate it
task.setWork(init_work); // make it known as "work" to the Connectors
```

Note that an Iterator in the Feeds section does not return any data if the Work Entry is already defined at this point in the AL. So if an IWE is passed into an AssemblyLine, any Iterators in the Feeds section simply pass control to the next component in line. It is also the reason why multiple Iterators in the Feeds section run sequentially, one starting up when the previous one reaches End-of-Data.

XML The **X**tensible **M**arkup **L**anguage. A general purpose markup language (See also HTML) for *creating* special-purpose markup languages, and also capable of describing many types of data. IBM Tivoli Directory Integrator uses XML to store Config files.

[24] Note that the "Work Entry" window shown in the Config Editor is actually a list of all Attributes that appear in Input Maps or in the Loop Attribute field of Loops in the AssemblyLine.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law :

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information in softcopy form, the photographs and color illustrations might not be displayed.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Index

Printed in USA

SC27-2706-01