Directory Integrator
Version 7.0

Getting Started Guide

GI11-8185-00

Directory Integrator
Version 7.0

Getting Started Guide

GI11-8185-00

Notices

Before using this information and the product it supports, read the general information under "Notices" on page 77.

Product Version 7.0

This edition applies to version 7, release 0 of the IBM Tivoli Directory Integrator and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

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.

Who should read this book?

Read this book if you are a systems administrator, user, or anyone interested in learning more about IBM Tivoli Directory Integrator.

This book is intended for system administrators and users and anyone interested in learning more about IBM Tivoli Directory Integrator.

This book is also intended for those responsible for the development, installation and administration of solutions using theTivoli Directory Integrator. The reader should be familiar with the concepts and the administration of the systems that the developed solution will connect to. Depending on the solution, these could include, but are not limited to, one or more of the following products, systems and concepts:

- IBM Tivoli Directory Server
- IBM Tivoli Identity Manager
- IBM Java Runtime Environment (JRE) or Sun Java Runtime Environment
- Microsoft Active Directory
- Windows and UNIX operating systems
- Security management
- Internet protocols, including HyperText Transfer Protocol (HTTP), HyperText Transfer Protocol Secure (HTTPS) and Transmission Control Protocol/Internet Protocol (TCP/IP)
- Lightweight Directory Access Protocol (LDAP) and directory services
- A supported user registry
- Authentication and authorization concepts
- SAP ABAP Application Server

Publications

Read the descriptions of the IBM Tivoli Directory Integrator V7.0 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 V7.0 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 V7.0 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 V7.0 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 V7.0 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 V7.0 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 V7.0 Messages Guide
Provides a list of all informational, warning and error messages associated with the Tivoli Directory Integrator.

IBM Tivoli Directory Integrator V7.0 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 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 V7.0 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 V7.0 uses the JNDI client from Sun Microsystems. For information about the JNDI client, refer to the *Java Naming and Directory Interface™ Specification* on the Sun Microsystems Web site at http://java.sun.com/j2se/1.5.0/docs/guide/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 InstallShield Multiplatform (ISMP) 11.5 wizard.

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. Introduction

This book is a simple introduction to a simple system. Make no mistake; the word *simple* is used here in its most positive and powerful context, because the best way to wrap your mind around a complex problem is to simplify it; Break it down into more manageable pieces and then master those constituent parts. Divide and conquer. This is a technique you instinctively use to solve everyday problems, and which is equally relevant for engineering information exchange across an office, an enterprise or the globe.

IBM Tivoli Directory Integrator[1] is designed and built on the premise that even the most complex integration problems can be decomposed down into three basic parts:
- The systems involved in the communication – also called *data sources*,
- The *data flows* between these systems,
- The *events* that trigger the data flows.

With IBM Tivoli Directory Integrator you translate this atomic understanding of the integration problem directly into a solution, building it incrementally, one flow at a time, with continuous feedback and verification. This approach makes integration projects easier to estimate and plan, sometimes reducing this effort to the counting and costing the individual data flows to be implemented. Completing a task in runnable steps also allows you to regularly demonstrate progress to stakeholders.

IBM Tivoli Directory Integrator further accelerates development by abstracting away the technical differences between your data sources, allowing you to spend more time concentrating on the business requirements.

Leveraging the power of Eclipse, the IBM Tivoli Directory Integrator development environment is both comprehensive and extensible. Integration projects result in libraries of components and business logic that can be quickly reused to address new challenges. As a result, teams across your organization can share IBM Tivoli Directory Integrator assets, resulting in independent projects – even point solutions – that immediately fit into a coherently integrated and managed infrastructure.

This document gives you an introduction to the simplify and solve methodology described above. You will also take your first steps toward tapping into the elegant simplicity of the Tivoli Directory Integrator toolset, specifically these two programs:
- The development environment, called the *Configuration Editor*, or 'CE' for short,
- The run-time engine, simply referred to as the *Server*.

You will assemble your Tivoli Directory Integrator solutions with the CE, while one or more Servers are used to power them. These programs work in concert, making the user experience seamless, and even allowing you to work across platforms; for example, developing on your laptop while testing and debugging solutions running remotely on a mainframe.

Scripting in JavaScript

As mentioned above, IBM Tivoli Directory Integrator lets you rapidly assemble integration solutions. However, in order to extend built-in automated functionality with your own custom processing and flow behavior, you will need to write snippets of script.

1. Don't let the name fool you; Tivoli Directory Integrator is not limited to directory work, and supports all major data stores, transports, protocols and APIs – including of course LDAP directories.

Scripting is done in JavaScript, and Tivoli Directory Integrator includes the IBM JSEngine to provide a fast, reliable scripting environment. As a result, you will need to use and understand the core JavaScript language. There are several good online and hardcopy resources for learning JavaScript. Check the Tivoli Directory Integrator newgroups and websites for recommendations and links.

For more information about scripting in IBM Tivoli Directory Integrator, see the *IBM Tivoli Directory Integrator V7.0 Users Guide*.

Installing IBM Tivoli Directory Integrator

Tivoli Directory Integrator installs in a few minutes and you can begin building, testing and deploying solutions immediately. It runs on a wide variety of platforms, including Microsoft Windows, IBM AIX®, IBM System z®, and a number of UNIX and Linux environments.

There are three paths of interest when installing Tivoli Directory Integrator, and the installer will ask you to specify the first two:

1. The *Installation Directory*, where the program files are kept, along with the batch-files or scripts used to launch the various tools.
2. The *Solution Directory*, often abbreviated 'SolDir', which is the current folder whenever you run Tivoli Directory Integrator. You will notice that the startup batch-files and scripts for the Config Editor development environment (`ibmditk`) and the Server (`ibmdisrv`) both start with a command to change directory to the Solution Directory. As a result, all relative paths used in your solution will be expanded from your Solution Directory.
3. The *workspace* folder. This is where your project and resource[2] files are kept. This will default to a folder named "`workspace`" in your Solution Directory.

For more information about installing the Tivoli Directory Integrator, see "Tivoli Directory Integrator installation instructions" in the *IBM Tivoli Directory Integrator V7.0 Installation and Administrator Guide*.

Installing the tutorial files

The tutorial exercises in this book require supporting data files that are located in the `examples/Tutorial` sub-folder of the Tivoli Directory Integrator installation directory. For example, a standard Windows installation would place these files in the following directory:

```
C:\Program Files\IBM\TDI\V7.0\examples\Tutorial
```

The 'Tutorial' directory should contain the following files:

- `CreatePhoneDB.assemblyline`
- `index.html`
- `OtherPage.html`
- `People.csv`
- `PhoneNumbers.xml`
- `readme.txt`
- `Return web page.script`

Note: As mentioned in the previous section, the installer will ask you to specify the location of your Solution Directory. This is where your project and resource files will be stored, and it will typically be a sub-directory called `My Documents\TDI` under your home area.

Copy the `Tutorial` folder to your Solution Directory in order to make it more readily accessible from the Configuration Editor tooling.

2. These terms are explained in Chapter 2, "Introducing IBM Tivoli Directory Integrator," on page 9

Simplify and solve

This section helps you to understand your starting place when designing a data integration solution. Although the design strategy is incremental, it is suitable for any size of integration and systems deployment project, including large ones.

Use a strategy like this one for designing a step-by-step data integration solution using IBM Tivoli Directory Integrator:

- Reduce complexity by breaking the problem up into smaller, manageable pieces.
- Start with a portion of the overall solution, preferably one that can be completed in a week or two.
- Start with a portion of the overall solution that can be put into production all by itself.

How do you eat an elephant?

The answer is, one bite at a time. This is also the best approach for digesting large integration and systems deployment projects. The key to success is to reduce complexity by breaking the problem up into smaller, more manageable pieces. Once this is done, you then begin work on a portion of the overall solution, preferably one that can be deployed independently. That way, it's already providing return on investment while you tackle the rest.

After isolating the piece you are going to work with, simplify it further by focusing on the basic units of communication: the data flows themselves. You are now poised to start the implementation. Integration development is done using the IBM Tivoli Directory Integrator Configuration Editor (abbreviated as 'CE') through a series of try-test-refine cycles, making the process an iterative and even exploratory one. This not only helps you to discover more about your own installation, but also lets you evolve your integration solution as your understanding of the problem set and its impact on your infrastructure grows.

Related topics

See the following topics for an explanation of how Tivoli Directory Integrator allows you to transform data using AssemblyLines.

- "Kernel/Component Architecture"
- "Entry-Attribute-value data model" on page 4
- "Data flows = AssemblyLines" on page 5

Kernel/Component Architecture

A fundamental quality of the Tivoli Directory Integrator is its kernel/component design.

The term *kernel* here refers to the rapid integration development (RID) framework that allows you to quickly assemble your integration solutions and provides automated execution logic to drive them. Features that you would otherwise need to hand-code (and are therefore often neglected) like log/trace modules, connection recovery, change detection, error handling and an external management API are immediately available to even the simplest data flow.

In addition to this generic kernel functionality, Tivoli Directory Integrator provides a set of data source-specific components: helper objects that abstract away the technical details of interacting with your data sources. The two types of components that you will use the most are *Connectors* and *Parsers*.

Connectors provide connectivity to a wide variety of data sources, as well as inherent handling of structured data regardless of its underlying organization. Some Connectors also serve as event-handlers, for example binding to IP ports and waiting for incoming connections, or 'listening' for changes to occur in directories, databases or files.

Parsers on the other hand are used to deal with unstructured data – that is, bytestreams, like those found in files, POP3/SMTP email, MQ messages and data streaming across IP ports.

Tivoli Directory Integrator provides an extendable library of Connectors and Parsers, each designed to work with a specific system, service, API, transport or format. The interchangeable nature of Tivoli Directory Integrator components allows you to build a solution based on test data – for example, text files – and then simply swap out the Connectors used in order to point your solution at live sources for verification and deployment.

Furthermore, Tivoli Directory Integrator components are straightforward to use, as well as easy to build and extend. You can augment your library to deal with custom data sources and services by downloading new components from a community website, writing your own components in Java, or by interactively building and testing them using script directly in the CE.

Entry-Attribute-value data model

The way data is organized and stored differs greatly from system to system:

- Databases store information in rows, typically with a fixed number of columns, each carrying a single value for that record;
- Directories maintain object-oriented entries that can contain a varying number of attributes. These in turn hold zero, one or multiple values[3];
- Lotus® Domino databases contain Documents that are made up of Fields, each of which can be defined as single- or multi-valued;
- Still other systems represent their data content as nodes, objects, records, formatted byte streams or key-value sets.

In order for communication to meaningful to all participants, data formats have to be compatible or they must be translated to suit each system involved. This is called *data marshalling* and is often the first hurdle an integration specialist faces – and one which can quickly consume a sizeable chunk of project resources to overcome. IBM Tivoli Directory Integrator Connectors handle this for you by automatically converting source-specific types to a consistent, canonical representation. Individual data values are translated to relevant Java objects, with comparable native types being represented in the same way. For example, lines read from files, LDAP string attributes, Domino text fields and RDBMS CHAR and VARCHAR columns are all converted to `java.lang.String` by their respective Connectors.

These marshalled values are then accumulated into Attributes: specialized Java objects defined by Tivoli Directory Integrator. As noted above, some sources permit only a single value per column or field, while others allow several values to be stored under the same attribute name. The Tivoli Directory Integrator Attribute supports both single-valued and multi-valued implementations, and can even hold no values at all if necessary, for example when representing a nullable column in a database.

All the Attributes that make up a single unit of data (that is, record, message, document, and so forth) are collected in another Tivoli Directory Integrator object called an *Entry*. An Entry can hold any number of Attributes, or none at all.

3. Consider for a moment the fact that you probably have multiple email addresses, all of which can be stored in the multi-valued attribute entitled 'mail' in your company's employee directory

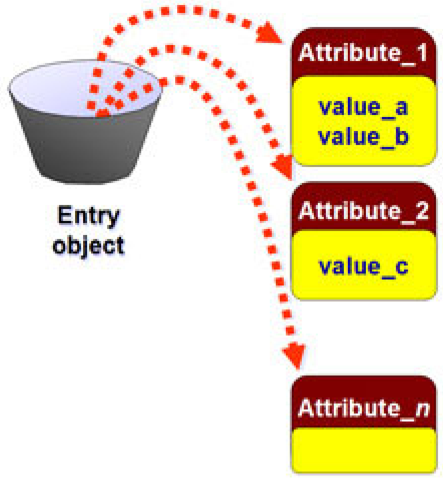


Figure 1. The Entry-Attribute-value data model

Each data flow has a primary Entry 'bucket' called its Work Entry. Whenever a Connectors reads in data, it creates Attributes and puts these in the Work Entry. Any Connector configured for output uses Attributes already found in the Work Entry to drive changes to target systems.

This two-stage approach provides for almost unlimited flexibility in how data is transferred, transformed, filtered and enriched. It also means that you can initially build your data flow entirely with input Connectors and then interactively examine the data with the CE as it is read and manipulated before you even have to consider connections to output systems.

As you will see later on, the Tivoli Directory Integrator Entry handles complex hierarchical data just as easily as it does flat schema.

Data flows = AssemblyLines

Each data flow in your solution is implemented as an IBM Tivoli Directory Integrator *AssemblyLine*, also abbreviated as 'AL' in this and other literature.

ALs are ordered lists of components forming a single, continuous path from input sources to targets. Built-in behavior provided by the kernel ties the components together and passes data carried in the Work Entry from one to the next.

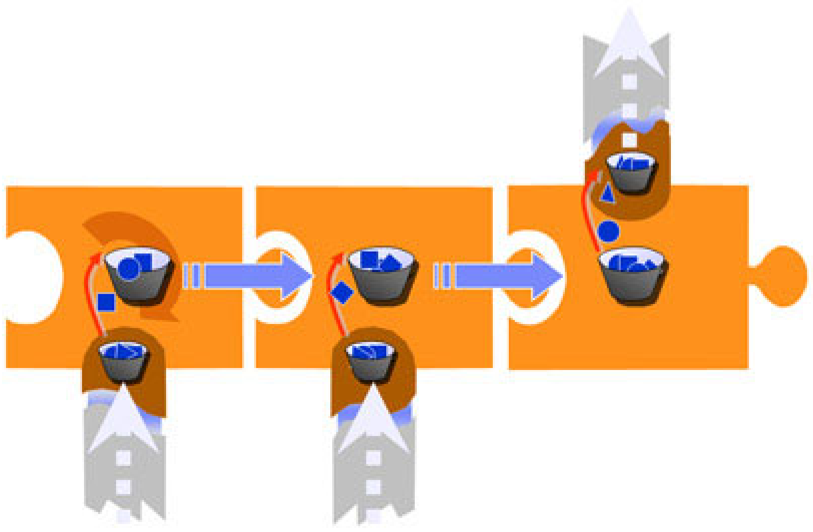


Figure 2. Data flowing down an AssemblyLine

It's said that a picture is worth a thousand words, and the diagram above is no exception. The three puzzle pieces represent Connectors linked together to form an AssemblyLine. The darker 'stem' of each puzzle piece highlights the data source specific part of the Connector – that is, the *interface* to the connected system – known as the *Connector Interface* (abbreviated as 'CI'). The lighter colored remainder of each puzzle piece depicts the generic functionality of the kernel that makes all components work in a similar and predictable fashion, enabling them to be linked together and providing automated patterns of behaviors with control points for customization[4].

This picture illustrates a few more important concepts. For example, in addition to the Work Entry shown above flowing from component to component down the AssemblyLine, there is an additional Java "bucket" nestled in each of the Connector Interfaces. Each local Entry object is used to cache data during read and write operations performed by that CI, and is called its *Conn Entry*.

Now notice the curved arrows illustrating data flowing between the various Conn Entries and the AL's Work Entry. These are *Attribute Maps* and each one represents a set of rules for data movement and transformation on its way either in or out of the AL. Those that lift data from a Conn Entry into the Work Entry are named *Input Maps* since they determine what data is brought into the AssemblyLine. The arrow in the rightmost puzzle piece that shows data moving in the other direction – from the Conn Entry to the Work Entry – is called an *Output Map*.

Since there is only one Work Entry at any time, you can deduce that AssemblyLines process one item at a time: for example, one database row, directory entry, MQ message, and so forth. This is another important aspect of IBM Tivoli Directory Integrator, and although an AssemblyLine can cycle hundreds or even thousands of Entries per second[5], it's an important consideration when designing your solution. It is of course possible to spread work across multiple AssemblyLines, and you will find this and other techniques for optimizing AL performance in other Tivoli Directory Integrator literature.

Getting Started

A good start for any integration project is to make a diagram of the problem at hand.

4. As you can see, every AssemblyLine component reflects the kernel/component architecture of Tivoli Directory Integrator. If you decide to make your own component, it is only its interface that you have to implement. The AL "wrapper" and its wealth of built-in functionality are available automatically, courtesy of the Tivoli Directory Integrator kernel.

5. Performance will depend on the design and complexity of the AssemblyLine and the configuration of the machine running the Server.

Using a pencil and a piece of paper, sketch out the desired flows in broad strokes. This exercise not only helps you to visualize the scope of the task, it serves as a blueprint for implementing these flows in IBM Tivoli Directory Integrator.

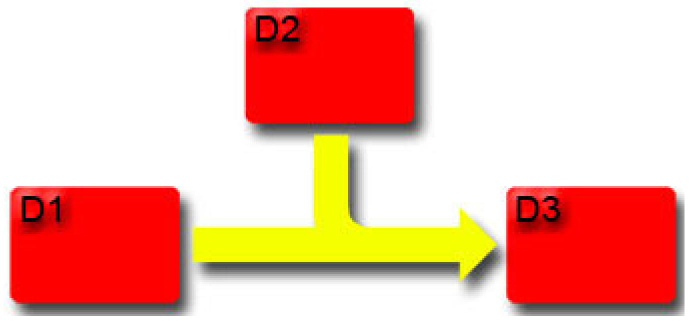


Figure 3. Tutorial scenario

The first step in creating a Tivoli Directory Integrator solution is translating data flows between data sources into AssemblyLines made up of Connectors. The Tivoli Directory Integrator mantra of 'simplify and solve' prescribes building your solution incrementally, starting as simple as possible.

To illustrate this, consider the example scenario you will use for your first AssemblyLine. This integration task involves three data sources, labeled D1, D2 and D3. The desired solution is to migrate the contents of D1 to D3, augmenting this data with values found in D2. Translating this requirement to an AssemblyLine, you end up with three Connectors, one for each data source:

1. the first Connector to *iterate* through D1, feeding this data into the flow;
2. followed by a second Connector that *looks up* related records in D2 and merges these values with those coming from D1;
3. finally a third Connector configured to *add* these augmented records to D3.

Instead of attacking the entire problem at once, Tivoli Directory Integrator allows you to simplify the task by starting with only two Connectors: one that reads the contents of D1 into the AL and another to write these values to D3. Once this minimal AssemblyLine is working properly, it can then be extended with the Connector into D2 to join in additional Attributes. This is precisely how you will create your first IBM Tivoli Directory Integrator solution, and the steps to guide you through this process comprise the remainder of this guide.

Chapter 2. Introducing IBM Tivoli Directory Integrator

This section provides information useful in understanding Tivoli Directory Integrator essentials, as well as a set of tutorial exercises to give you hands-on experience with the development environment.

Your first step in getting to know the product is to start the Tivoli Directory Integrator development tooling, known as the Configuration Editor, or CE for short.

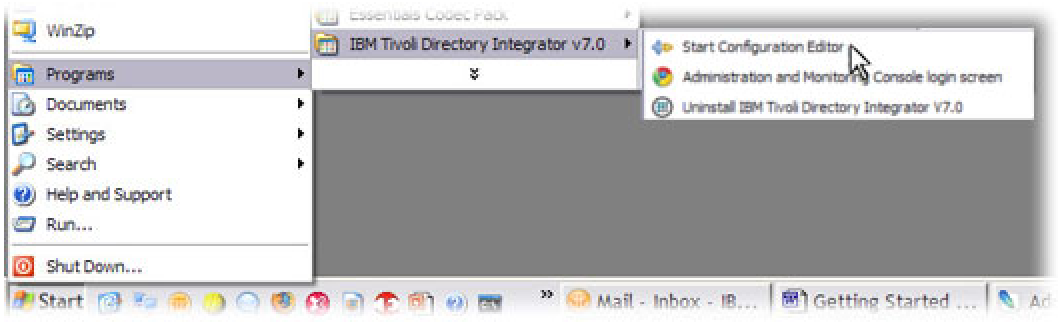
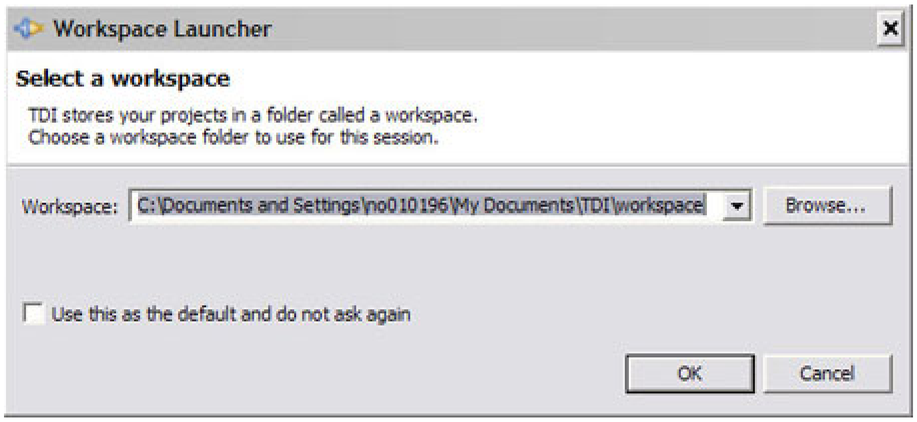

WinZip
Programs
Documents
Settings
Search
Help and Support
Run...
Shut Down...
Start
Essentials Codec Pack
IBM Tivoli Directory Integrator v7.0
Start Configuration Editor
Administration and Monitoring Console login screen
Uninstall IBM Tivoli Directory Integrator V7.0
Mail - Inbox - IB...
Getting Started ...

Figure 4. Starting the Configuration Editor

The first time that you start up the CE, you will see get this dialog for specifying your workspace.



Workspace Launcher

Select a workspace

TDI stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

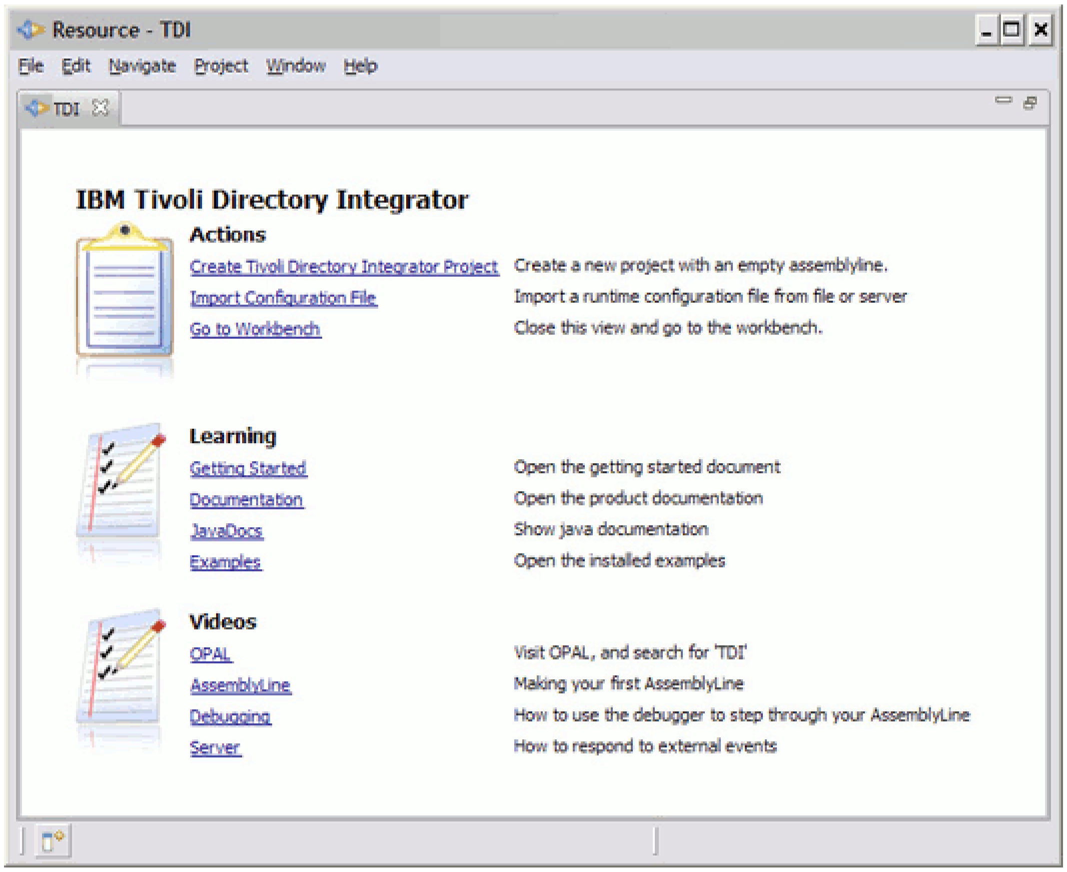
Workspace: C:\Documents and Settings\no010196\My Documents\TDI\workspace

☐ Use this as the default and do not ask again

Figure 5. Selecting your workspace

Your workspace is where the Configuration Editor will store your project files, including components and AssemblyLines, and it is typically located under your Solution Directory.

Once you are happy with the location of your workspace press the **OK** button. Now the Welcome Screen will appear.



IBM Tivoli Directory Integrator

Actions

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

Videos

OPAL	Visit OPAL, and search for 'TDI'
AssemblyLine	Making your first AssemblyLine
Debugging	How to use the debugger to step through your AssemblyLine
Server	How to respond to external events

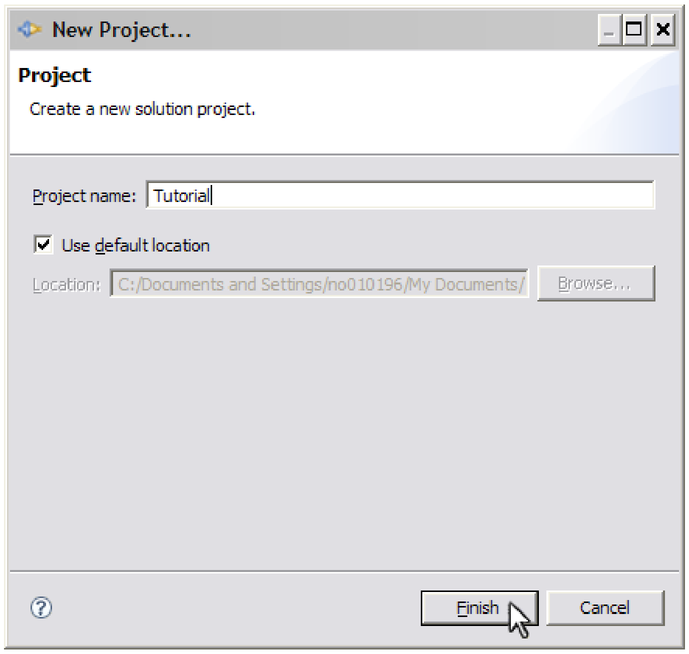
Figure 6. Configuration Editor Welcome Screen

The Welcome screen offers a number of quick-start links[6].

Whenever you build, test or modify integration solutions with Tivoli Directory Integrator, you are working within a project. Projects are collections of AssemblyLines and their constituent components, and each project appears in its own sub-folder of your workspace. The AssemblyLines and components that make up a project are stored as individual files, which in turn are located in sub-directories of the project folder.

Select the topmost link (*Create Tivoli Directory Integrator Project*) to set up your first project. You must now give your new project a name. Call it 'Tutorial' and press **Finish**.

6. You can return to this screen at any time by selecting **Help** > **Welcome** in the Main Menu.


New Project...

Project

Create a new solution project.

Project name: Tutorial

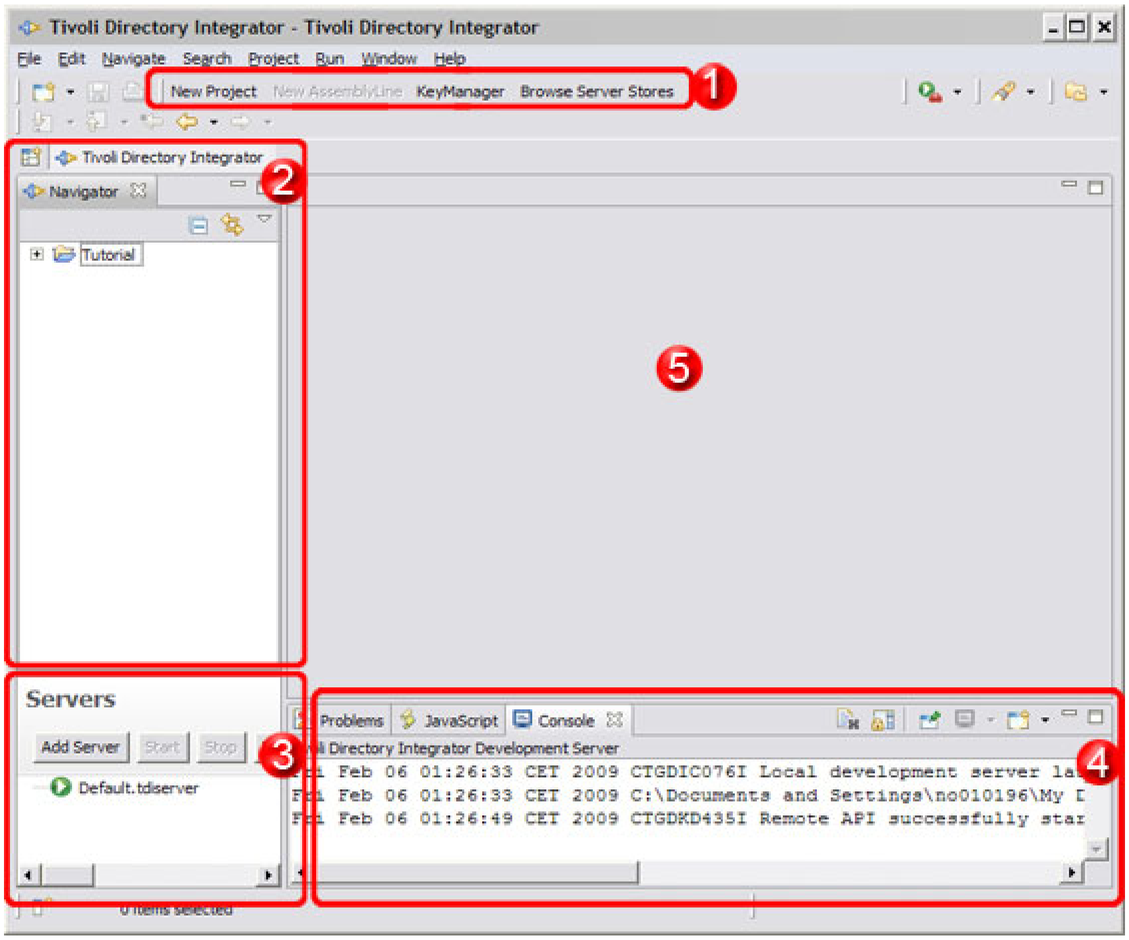
☑ Use default location

Location: C:/Documents and Settings/no010196/My Documents/

Figure 7. Naming your new project

You will now see the main development work area. The panels here can all be resized, and you can decide how the screen is organized. What you see on screen here is the default 'TDI' Perspective[7].

7. A *Perspective* is simply an organization of the development environment panels. If you have made changes to layout and want to return to the default TDI Perspective, simply click on **Window** in the topmost menu and select the **Reset Perspective** option.


Tivoli Directory Integrator - Tivoli Directory Integrator
File Edit Navigate Search Project Run Window Help
New Project New AssemblyLine KeyManager Browse Server Stores
1
Tivoli Directory Integrator
Navigator
2
Tutorial
5
Servers
Add Server Start Stop
3
Default.tdiserver
Problems JavaScript Console
4
0 items selected

```
li Directory Integrator Development Server
Fri Feb 06 01:26:33 CET 2009 CTGDIC076I Local development server la
Fri Feb 06 01:26:33 CET 2009 C:\Documents and Settings\no010196\My D
Fri Feb 06 01:26:49 CET 2009 CTGDKD435I Remote API successfully star
```

Figure 8. Config Editor main screen

This is the main screen where you will spend most of your time when working with Tivoli Directory Integrator. Without going into the details of all the navigational elements here[8], let's look at the numbered areas highlighted in the above screenshot:

1. In the middle of the main button row is a set of shortcuts for creating new Projects, and if a Project is selected in the Navigator, for creating new AssemblyLines in it. There is also a button for launching the Tivoli KeyManager tool to work with certificate key- and truststores; as well as a **Browse Server Stores** button for retrieving the various property settings from the Tivoli Directory Integrator Server associated with this Project.
2. This is the *Navigator* panel and provides a tree-view of your development assets. Your new 'Tutorial' Project should appear here.
3. The Servers panel displays the status of all configured Servers. You can see by the arrow icon next to 'Default.tdiserver' that this Server has been started for you. This panel also provides buttons for defining new Servers, Starting and Stopping your Servers, as well as for refreshing the list and view a Server's log[9].

8. As with most Eclipse-based applications, there will be a number of ways to perform the same operation. The *IBM Tivoli Directory Integrator V7.0 Users Guide* describes all the various options and panels available.

9. If for some reason your server has not been started correctly, open 'TDI Servers' and double-click on 'Default.tdiserver'. This opens up the associated Server Document. Make sure that the Installation and Solution Directory settings are correct and then press the **Create Solution Directory** option at the top of this panel. If this does not correct the problem, then contact support.

Note that whenever you launch an AssemblyLine, both the Config Instance[10] and the AL also show up in this panel.

4. Here you will see a set of tabs with the currently selected tab showing console output coming from your Server. The messages displayed here now tell you that your Server is running and that its API is initialized and ready for use.
5. The gray area in this screenshot is where *editor* panels appear as you create and open AssemblyLines and components. Each type of resource (Connector, Parser, AssemblyLine, and so forth) has its own specially designed editor.

Creating your first AssemblyLine

Returning to the example scenario outlined in the introduction, you will now create an AL that migrates information from D1 to D3, ignoring for the moment the joining of data from D2.

Figure 9. Simplified scenario diagram with just two data sources

The 'Tutorials' folder (that you should have copied from *TDI installation directory*/`examples` to your Solution Directory) contains a file named People.csv:

```
First;Last;Title
Bill;Sanderman;Chief Scientist
Mick;Kamerun;CEO
Jill;Vox;CTO
Roger
Gregory;Highpeak;VP Product Development
Ernie;Hazzle;Chief Evangelist
Peter;Belamy;Business Support Manager
```

You can see from the above listing that this is in *character separated value* format (CSV). This file represents our D1 input data source. Your AL will extract this data and transfer it to an XML document which will be our D3 output target.

Click on **New AssemblyLine** in the topmost toolbar and call the new AL 'CSV2XML'.

10. Whenever the Tivoli Directory Integrator Server loads a Config, it creates a *Config Instance* that encapsulates the AssemblyLines of that project and allows these to run in their own contained environment. This means that you can load the same Config multiple times on the same Server, resulting in separate Config Instances all containing the same set of ALs without these interfering with each other.

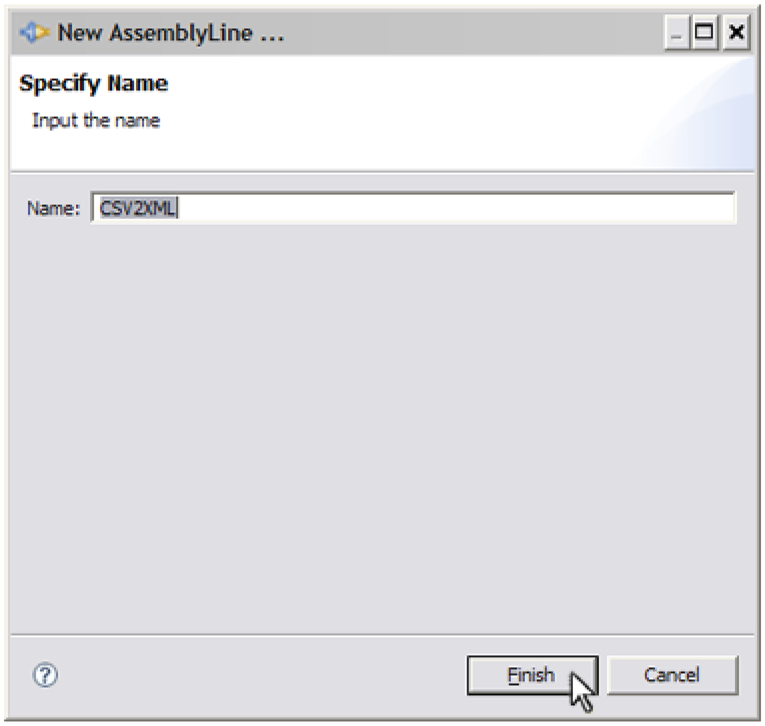


Figure 10. New AssemblyLine dialog box

Now press the **Finish** button to open the AL in an AssemblyLine editor tab.

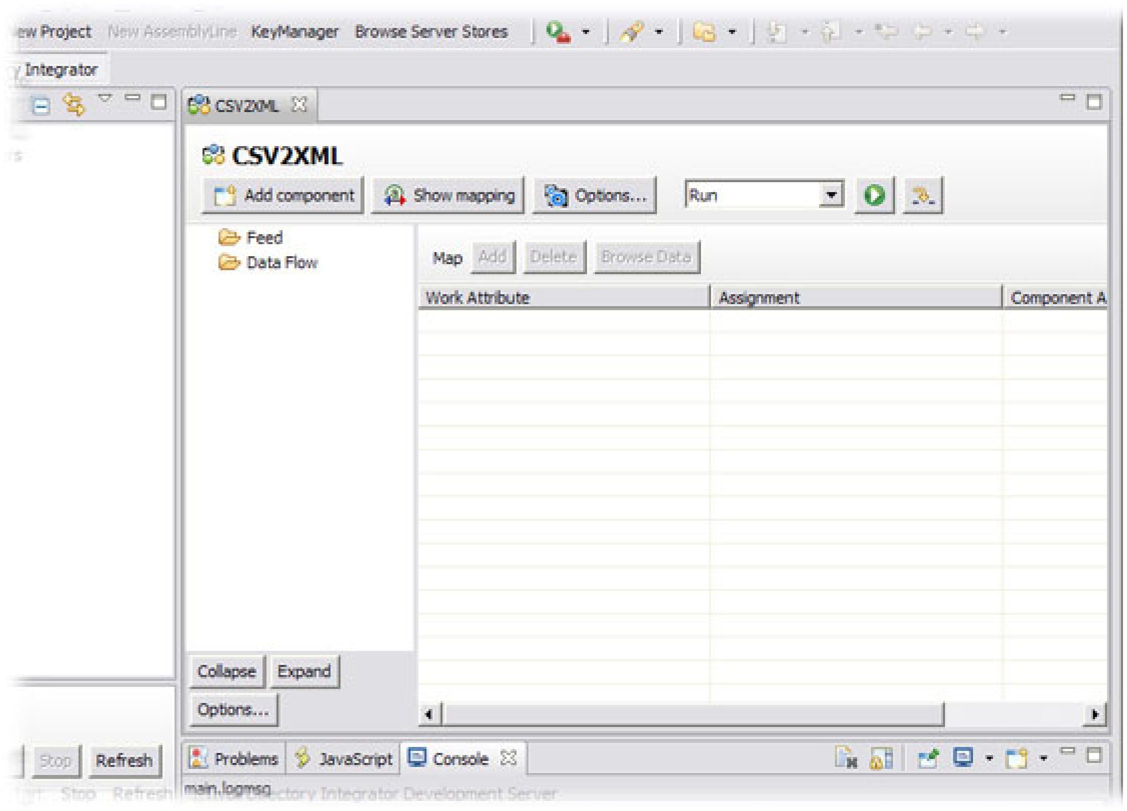


Figure 11. Empty AssemblyLine editor

The left part of the AL editor contains the list of components that make up this AssemblyLine and is empty right now except for the section names: *Feed* and *Data Flow*. The right-hand area displays all Attributes being mapped in and out of the AL.

To understand these AssemblyLine sections, consider for a moment what we want this new AL to do: *For each line in the CSV file, create a new node in the XML document*. This looping behavior is provided for you automatically by the Tivoli Directory Integrator kernel, driving components listed under the AL *Data Flow* section as long as there is input data coming from Connectors in the *Feed* section[11].

Let's take advantage of this functionality by adding a Connector to the Feed section to read in our CSV input file. Do this by right-clicking on the *Feed* section folder and selecting **Insert Component...**

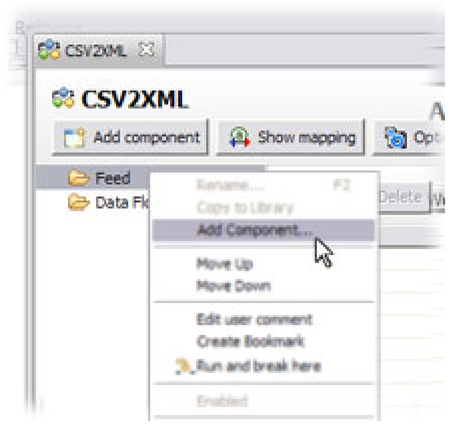


Figure 12. Inserting a new component

You will be presented with the **Choose Component** wizard.

11. Note that only one *Feeds* Connector will be delivering data to the AL at a time. If you put more than one Iterator Connector here then the topmost one will empty first before the next one in line begins reading from its source.

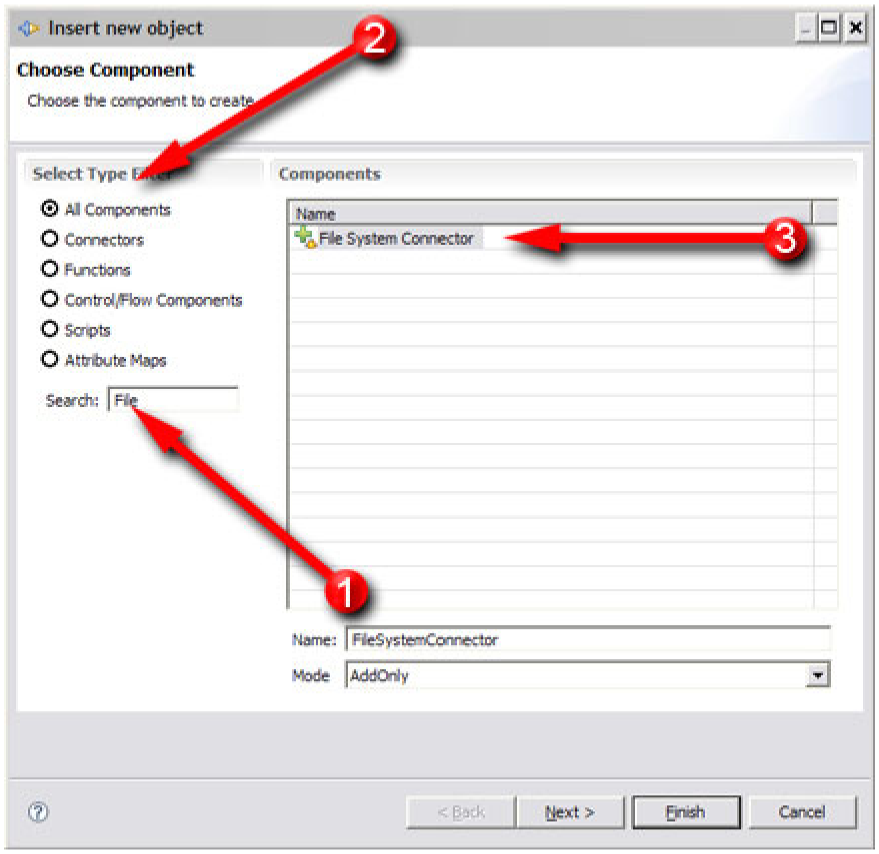


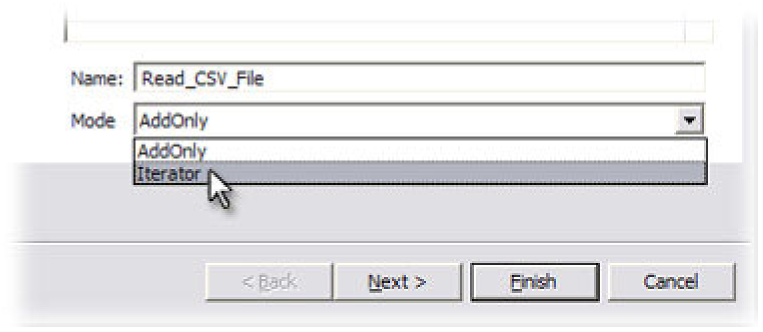
Figure 13. Choosing the component

This dialog gives you a couple of options to find and select the component you want:

1. Start typing any part of the name of the component in the text field and the selection list to the right is filtered accordingly. For this example, type "file".
2. You can optionally limit the selection list to include only a single type of component – Connectors, Parsers, Scripts, and so forth.
3. Locate and select the desired component in this list. In our example this will be 'File System Connector'.

The new Connector is automatically named 'FileSystemConnector' for you. Change this to 'Read_CSV_File' so that it has more meaning in the context of your solution[12] and then select **Iterator** from the **Mode** drop-down.

12. Although you can name Connectors as you like, it is recommended that you name them in the same way that you would a script variable: start with a letter, followed with any number of letters, digits and underscore characters. This is because all AL components are automatically registered as script variables, making it easier if you later want to reconfigure and drive them from your script code.

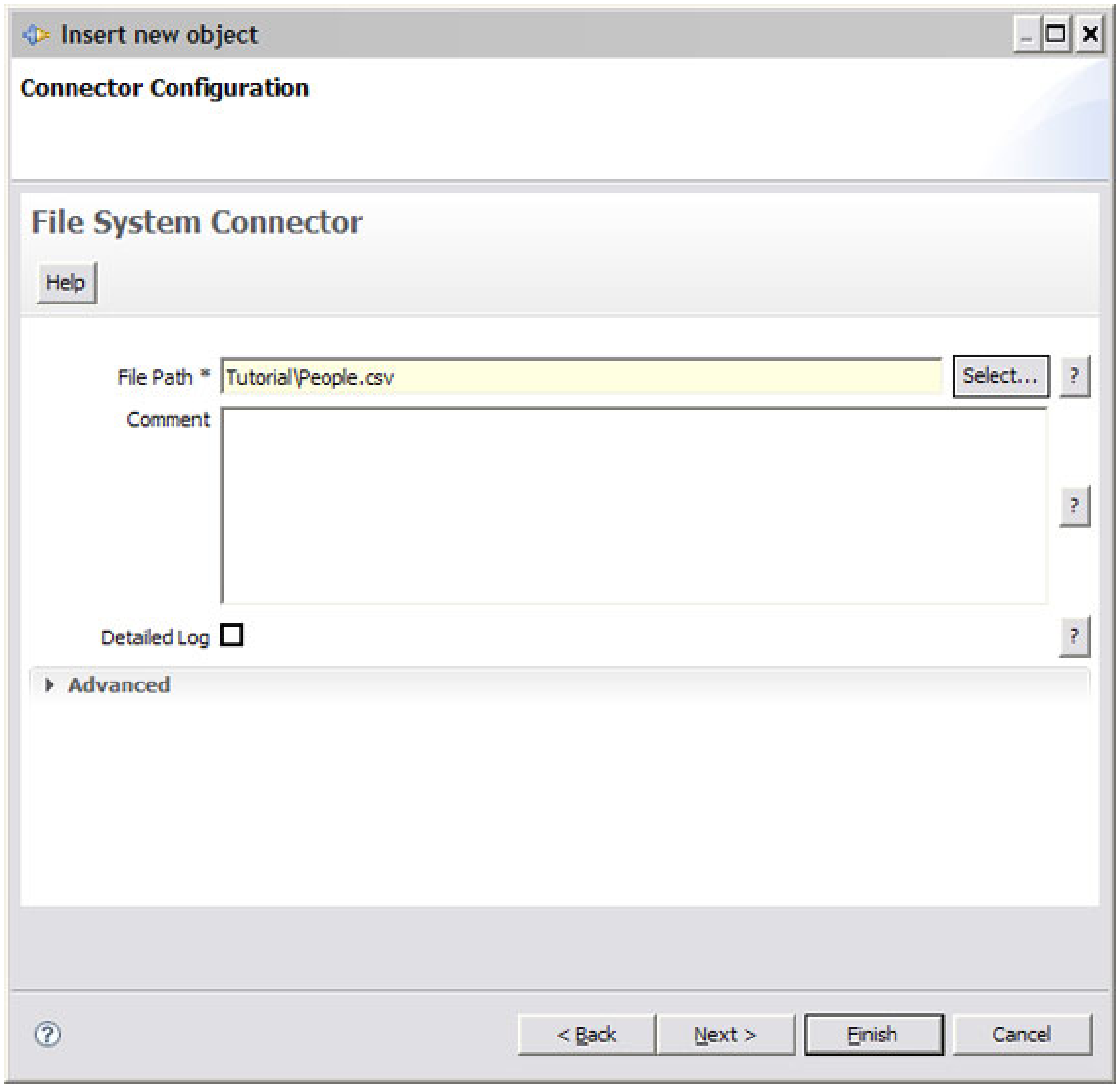

Name: Read_CSV_File

Mode AddOnly

Figure 14. Renaming the Connector and changing its mode

It's the Mode setting of a Connector that tells the built-in AL execution logic what role this component plays in the flow. Iterator Mode results in the *for-each* behavior you need in order to drive the data from the CSV file, one entry at a time, to the components you will add to the *Data Flow* section.

Now press the **Next** button to continue on to the configuration panel for the selected Connector.



File Path * Tutorial\People.csv

Comment

Detailed Log ☐

Figure 15. File System Connector Configuration panel

Each component provides its own set of configuration parameters. The ones shown onscreen now are for the File System Connector and it has only one required parameter: **File Path**. Type in the path to the `People.csv` file – either the full path, or the relative path from your Solution Directory as shown in the screenshot above[13] – or press the **Select** button to bring up a file browser to locate this file.

Because a formatted text file is a byte stream and not a structured data source like a database or directory, you must set up a Parser to interpret the formatting of the stream as it is read. Tivoli Directory Integrator provides a powerful and versatile Data Browser feature for interactively testing your Connector/Parser selection and configuration.

We'll take a look at this in a moment, but first you need to complete this wizard by pressing **Next** again and proceeding to **Parser Configuration**. Here you click on the **Select Parser** button.

13. This technique makes your solution easier to move and share since all you have to do is specify the desired Solution Directory and all relative paths will work unaltered.

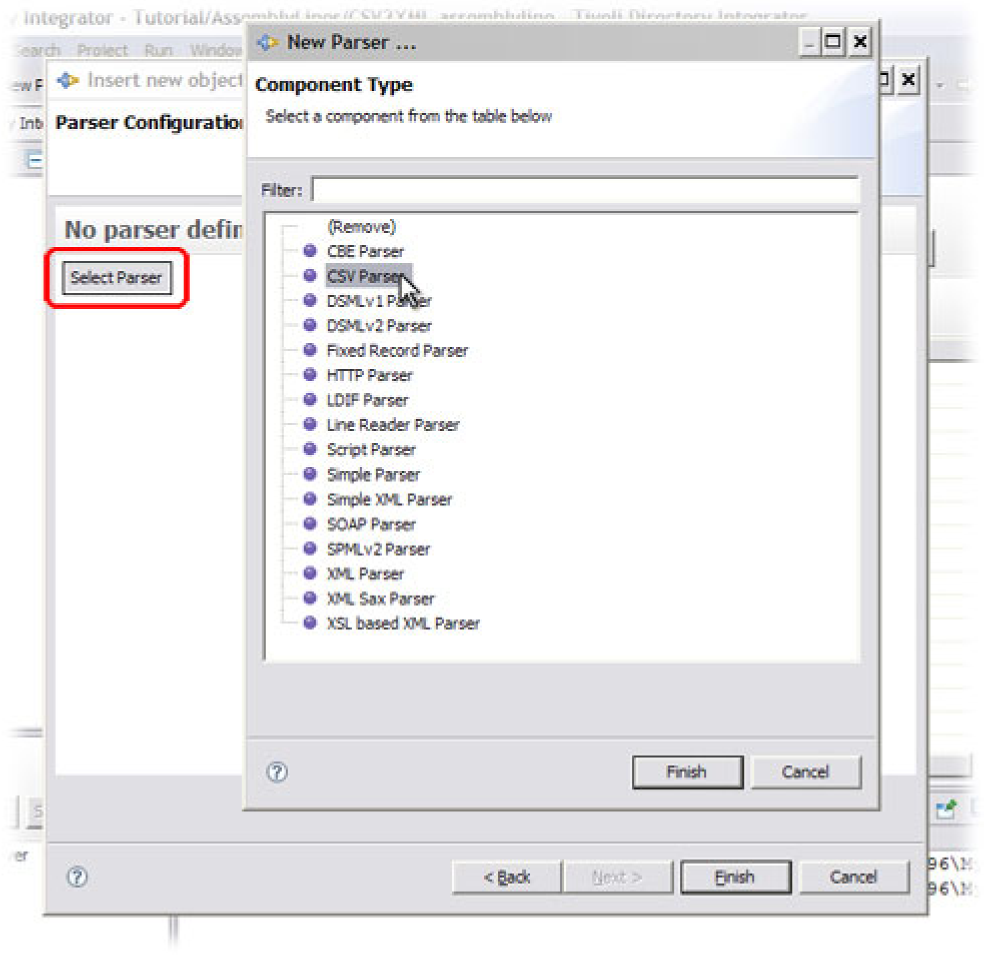
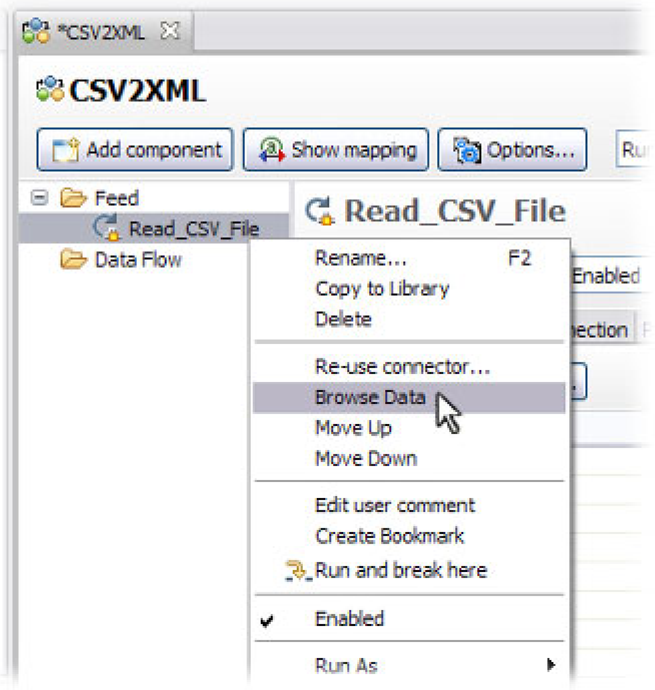


Figure 16. Selecting Parser during Insert new object

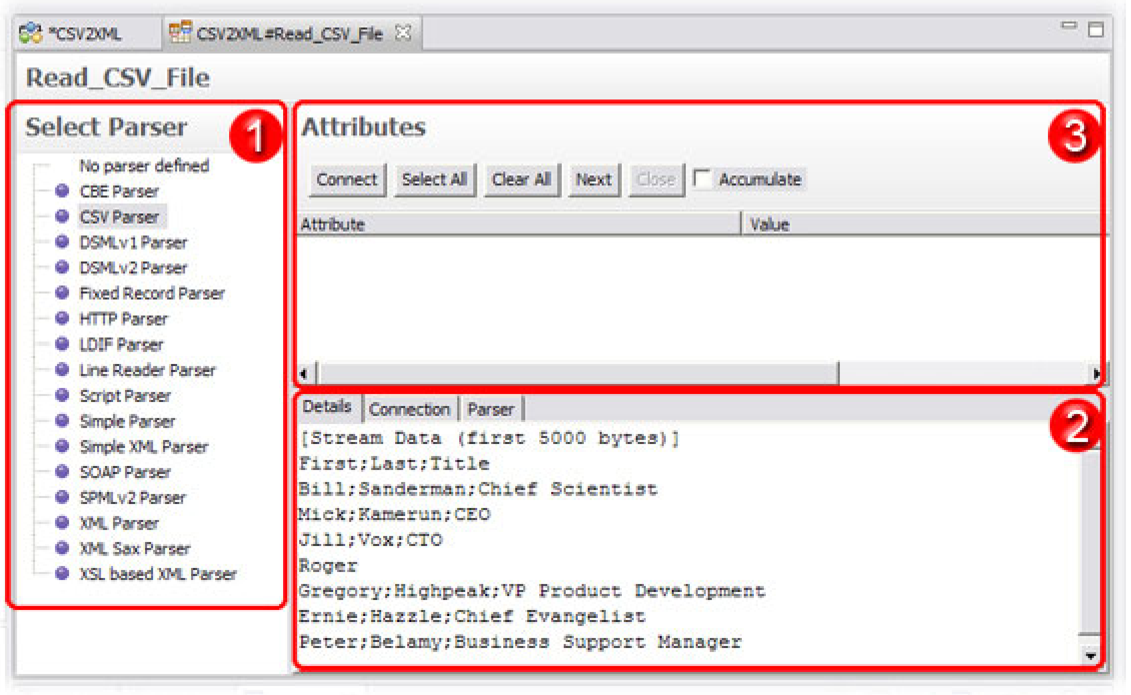
Select the 'CSV Parser' and press **Finish** to complete Parser selection. You will now see the Parser Configuration panel. Since you don't need to change the default settings, simply press Finish again to complete the wizard.

The next step is to have your Connector *discover* the schema of your input source in order to map these values into your AssemblyLine. This is where the Data Browser comes in handy[14]. Start with the Data Browser by right-clicking on your new Iterator Connector in the AssemblyLine Components tree and selecting **Browse Data** from the context menu.

14. Since you know the file is in CSV format, the quickest approach would be to just click on the **Connect** and **Next** button in the Schema area of the Iterator Connector. Then you drag discovered Attributes into the Input Map as desired. The Data Browser is useful when you are unsure of the format. But I still thought you ought to try it :)



This will open the Data Browser in a new editor tab.

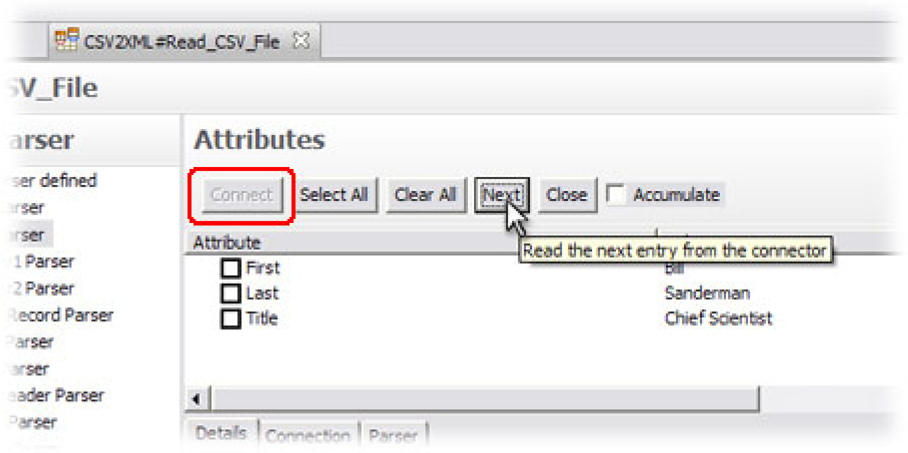


```
[Stream Data (first 5000 bytes)]
First;Last;Title
Bill;Sanderman;Chief Scientist
Mick;Kamerun;CEO
Jill;Vox;CTO
Roger
Gregory;Highpeak;VP Product Development
Ernie;Hazzle;Chief Evangelist
Peter;Belamy;Business Support Manager
```

Figure 17. The Data Browser

The area labeled 1 in the above screenshot is for choosing – and changing – the selected Parser. Area 2 provides a Details tab that shows you the raw byte stream that will be parsed. There are also tabs for changing the Connector's Connection parameters, as well one for configuring the chosen Parser.

The last section of this dialog (#3 in the screenshot) is for connecting to the data source and discovering which Attributes are available. Do this now by first pressing **Connect** and then the **Next** button.

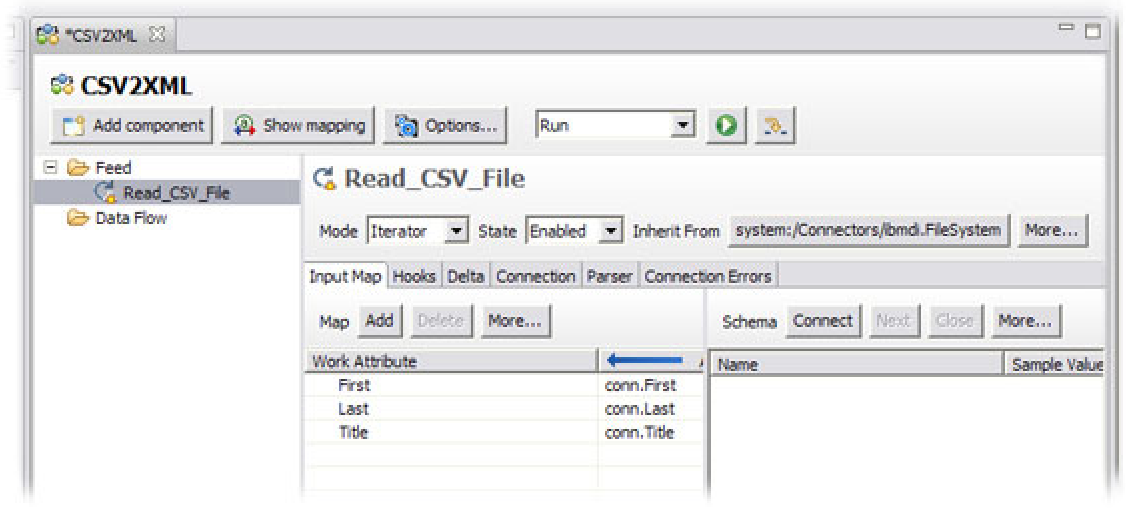


Attribute	
First	Bill
Last	Sanderman
Title	Chief Scientist

Figure 18. Interactively discovering schema by browsing live data

You have now discovered the *schema* of this file. Select the Attributes you want to map in, which in this case is all of them, by either selecting the checkbox next to each one, or using the **Select All** button.

Use the Ctrl-W shortcut to close the Data Browser tab, or simply click on the 'Close' symbol (X) at the right edge of the tab and return to the AssemblyLine editor where your AL should look like the screenshot below[15].



Mode: Iterator; State: Enabled; Inherit From: system:/Connectors/ibmdi.FileSystem

Work Attribute	
First	conn.First
Last	conn.Last
Title	conn.Title

Figure 19. AL with Iterator Connector in place

Details for the selected component are shown to the right of the AL component list, including the three mapping rules you just set up in the Input Map. Each Attribute Map item has an **Assignment**, which is a snippet of script that is evaluated in order to set the value (or values) of the target Attribute.

Before continuing, take a moment to reflect on these *Assignments*: You will recall from the "Entry-Attribute-value data model" on page 4 section that the AssemblyLine has a globally available

15. If for some reason your Connector is in the *Data Flow* section, simply drag it up to *Feed*. If the mode setting is not Iterator then right-click on the Connector, select **Mode** and then choose **Iterator**.

Work Entry that carries all data being transported down the AL. This object is referenced in script code by using the pre-registered script variable `work`. In addition, the Interface of every Connector has its own *Conn Entry* that is used as a cache for reads and writes. This component-specific object is accessed from script through the pre-registered variable `conn`[16]. To illustrate, consider the first mapping rule. It creates an Attribute in the Work Entry named 'First'. Its value is derived from the following assignment:

```
conn.First
```

This shorthand notation references the Attribute called 'First' that was just read into the `conn` Entry, and its values are used to populate the new Work Entry Attribute. A comparable assignment script would be:

```
   return conn.getAttribute("First");
```
[17]

Getting back to the exercise, you now you need to add your output Connector for creating the target XML document (data source D3). This time try using the **Add component** button at the top of the AssemblyLine Components panel.

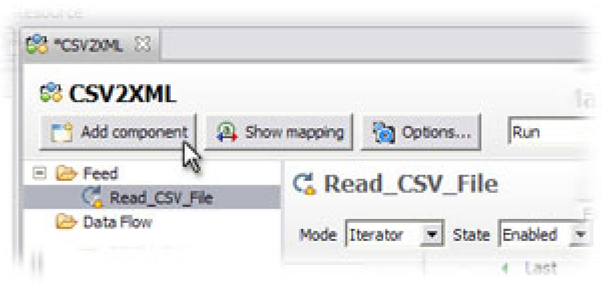


Figure 20. Add component button

Again choose the File System Connector, renaming it to 'Write_XML_File'. Leave the Mode setting as **AddOnly** and then press **Next**.

In the Connector Configuration panel, set the **File Path** parameter to write to a file called `Output.xml` in the Tutorial folder. Then choose 'XML Parser' in the next Wizard panel. Now you can press **Finish** since you don't need to change the XML Parser configuration. Note that in the case of your output Connector, you can't do Schema Discovery since there is no `Output.xml` file to discover from.

16. The `conn` variable is only available for limited periods, as shown in the Tivoli Directory Integrator Hook Flow Diagrams. Outside this scope it is still accessible by querying a component for its conn Entry.

17. For users familiar with version 6.x and earlier, you can also use the pre-7.0 syntax:

```
ret.value = conn.getAttribute("First")
```

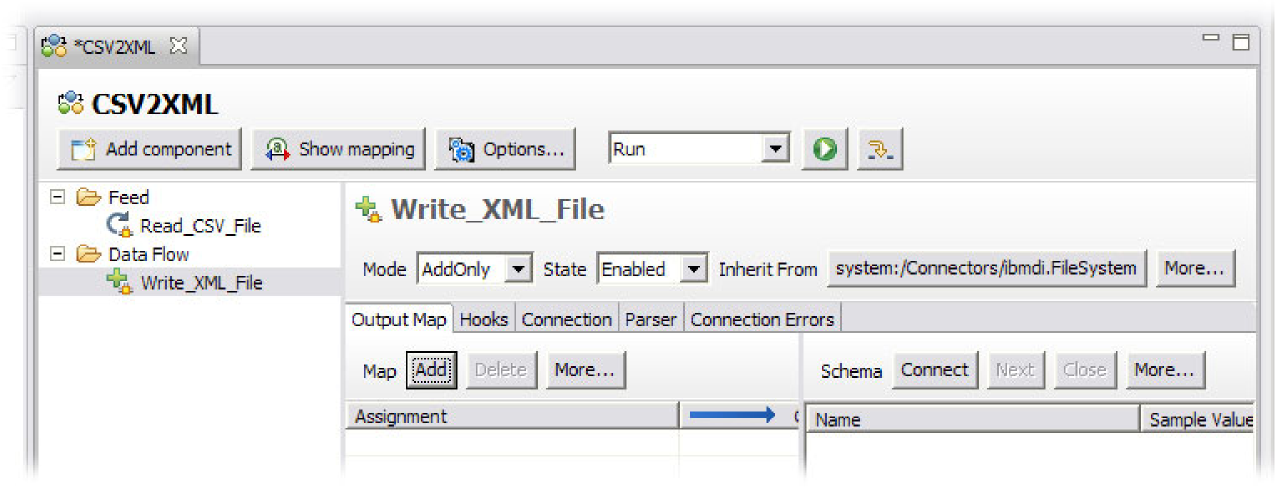
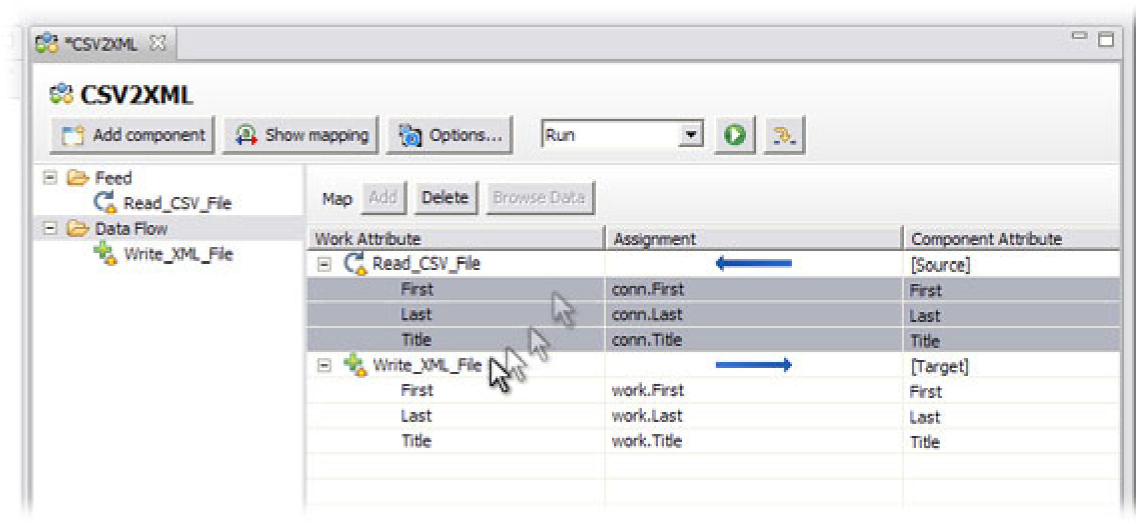


Figure 21. AL with two Connectors in place

You may have noticed that when you select a component, its details appear in the right part of the editor screen. Whenever you select either the 'Feed' or the 'Data Flow' folder, you are presented with the overview of all **Attribute Maps** for this AssemblyLine. This is a handy display for copying your input Attributes to the Output Map of your latest Connector, so bring up this screen now by clicking on either 'Feed' or 'Data Flow'.

Here you see the list of Attributes (three in total) that are being brought into your AL by the Iterator-mode Connector. Select these Input Map Attributes[18] and drag them down to the Output Map of the 'Write_XML_File' Connector, completing the data flow.



Work Attribute	Assignment	Component Attribute
Read_CSV_File		[Source]
First	conn.First	First
Last	conn.Last	Last
Title	conn.Title	Title
Write_XML_File		[Target]
First	work.First	First
Last	work.Last	Last
Title	work.Title	Title

Figure 22. Dragging Attributes to the Output Map

Notice how the Assignment is automatically converted from input format to output. For example, the first map item in the Input Map of your 'Read_CSV_File' Connector will create an Attribute in the Work Entry named 'First' to hold any values found in `conn.First` (that is, the Attribute called 'First' that was read into the Conn Entry). When you drag this input mapping rule to an Output Map then its

18. You can Control-click to select multiple, or use Shift-click to select a range.

assignment is changed so that the value now comes from the Work Entry instead, and it is creating a target Attribute in the Connector's cache (the Conn Entry).

If you want to change the source for any mapping rule then edit the assignment. In order to change the name of the Attribute being mapped to, simply right-click and rename it. Do this now for the first two Output Map rules.

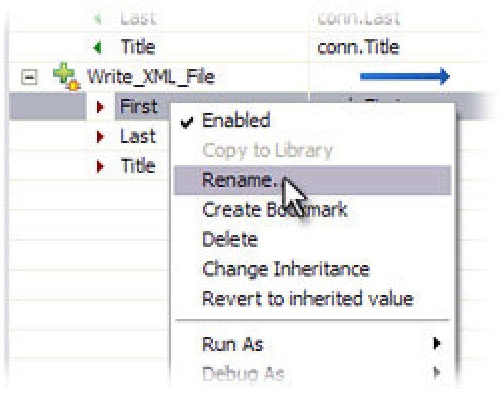


Figure 23. Renaming an Attribute Map rule

Change 'First' to 'FirstName' and 'Last' to 'LastName'.

Now add a new map item to this Output Map by right-clicking on the 'Write_XML_File' Output Map itself and choosing **Add Attribute**.

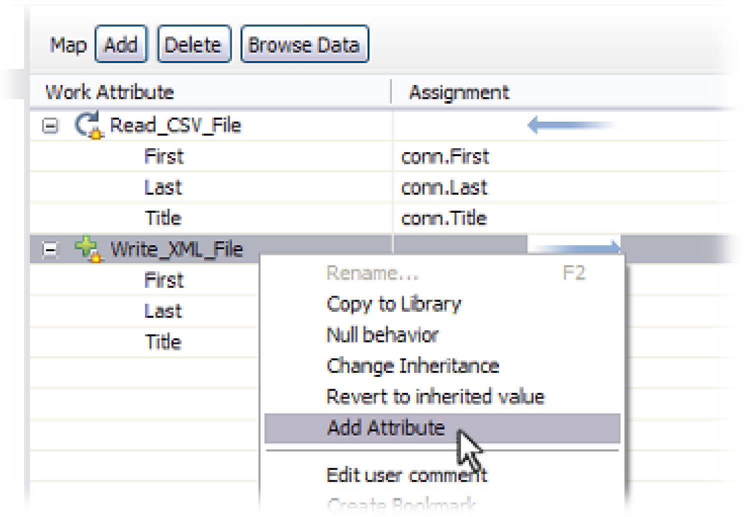


Figure 24. Adding the 'FullName' Attribute to the Output Map

Call the target of this new mapping rule 'FullName', press **OK** and then double-click on it to edit its assignment.. This opens up the Script editor panel and presents you with a default assignment script: `work.FullName`. Of course there is no 'FullName' Attribute in the Work Entry, so this map will not be able to set any values. Instead, you must *compute* this value by changing the script so that it concatenates the First and Last Attributes, leaving a single space between these values:

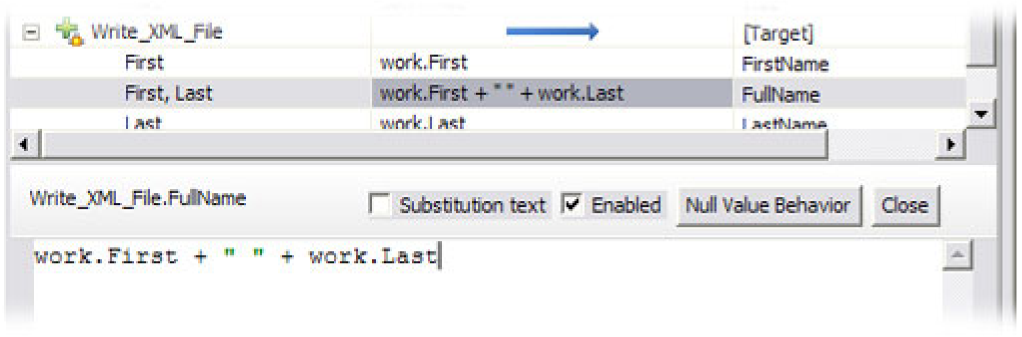


Figure 25. Editing the assignment

The script should read as follows:

```
work.First + " " + work.Last
```

Note that no terminating semi-colon is required for one-liner Attribute Map assignment scripts like this[19]. Press the **Close** button at the top-right of the Script editor panel when you are done.

Running your Assemblyline

It's now time to test your AL.

You do this by pressing the **Run** button at the top of the AssemblyLine Editor.

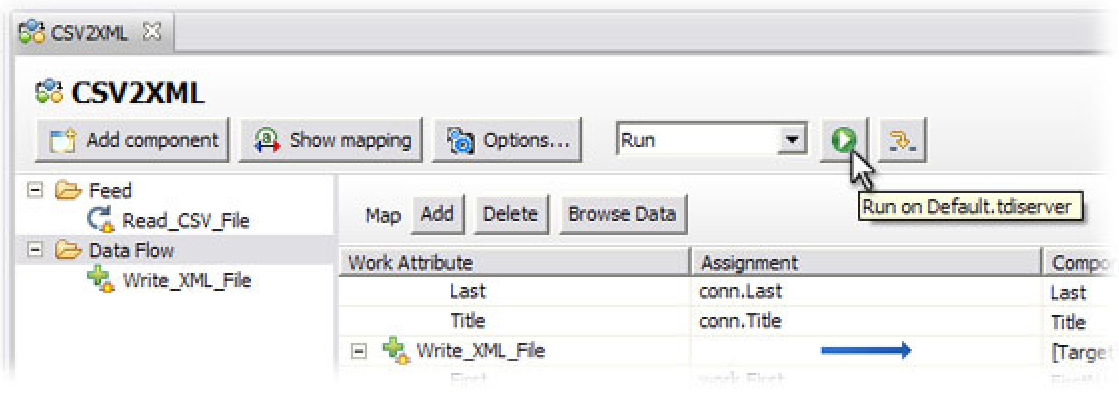
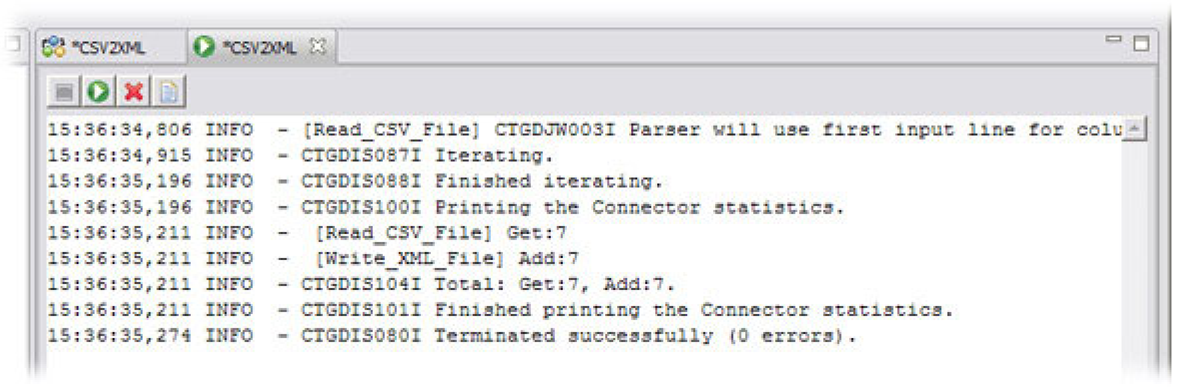


Figure 26. The Run button

You will now see a new tab open with a *Run window* showing the log output coming from your AssemblyLine.

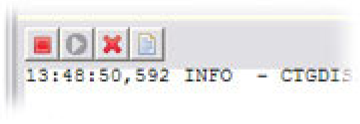
19. Pre-7.0 syntax is also supported so map assignment scripts can still start with "ret.value =".



```
15:36:34,806 INFO  - [Read_CSV_File] CTGDJW003I Parser will use first input line for colu
15:36:34,915 INFO  - CTGDIS087I Iterating.
15:36:35,196 INFO  - CTGDIS088I Finished iterating.
15:36:35,196 INFO  - CTGDIS100I Printing the Connector statistics.
15:36:35,211 INFO  -  [Read_CSV_File] Get:7
15:36:35,211 INFO  -  [Write_XML_File] Add:7
15:36:35,211 INFO  - CTGDIS104I Total: Get:7, Add:7.
15:36:35,211 INFO  - CTGDIS101I Finished printing the Connector statistics.
15:36:35,274 INFO  - CTGDIS080I Terminated successfully (0 errors).
```

Figure 27. Log Output from the AssemblyLine run

The CE actually took your AssemblyLine with all its components and exported a *Config* – an XML document that defines work assigned to a Tivoli Directory Integrator Server. It then piped this Config to the Server and instructed it to run your AL, capturing all log output for display onscreen.



```
13:48:50,592 INFO  - CTGDIS
```

Figure 28. Button bar for the Log Output window

The Run window includes a button bar with options to stop the AL, restart it once it has stopped and to clear the log contents. The rightmost button opens the current log output in an external editor.

The log output of an AssemblyLine ends in statistics for all components involved, which in your case is just two Connectors. From the above information it's clear that seven entries were read from the CSV file and seven nodes written to the XML document.

You could locate your output file on disk and open it in a browser window to confirm your results. You can also use the Data Browser by right-clicking on your output Connector (Write_XML_File) and selecting **Browse Data**.

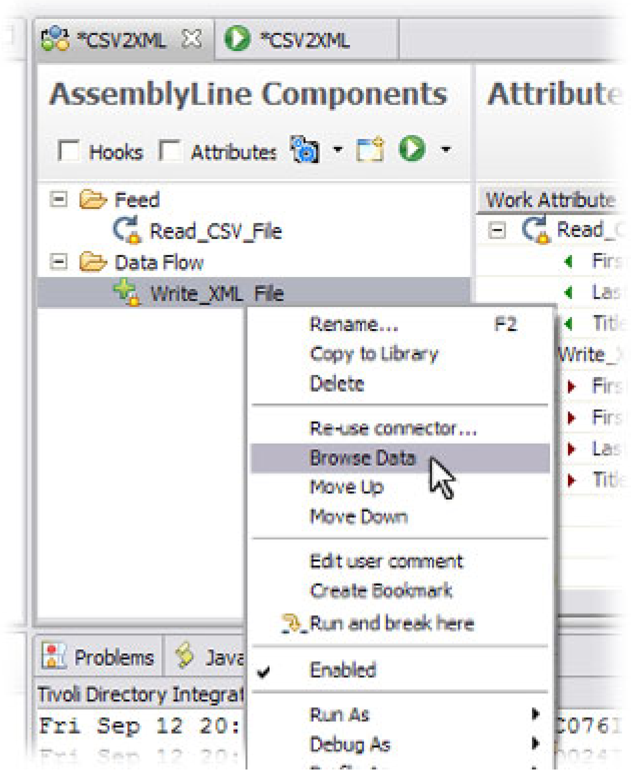
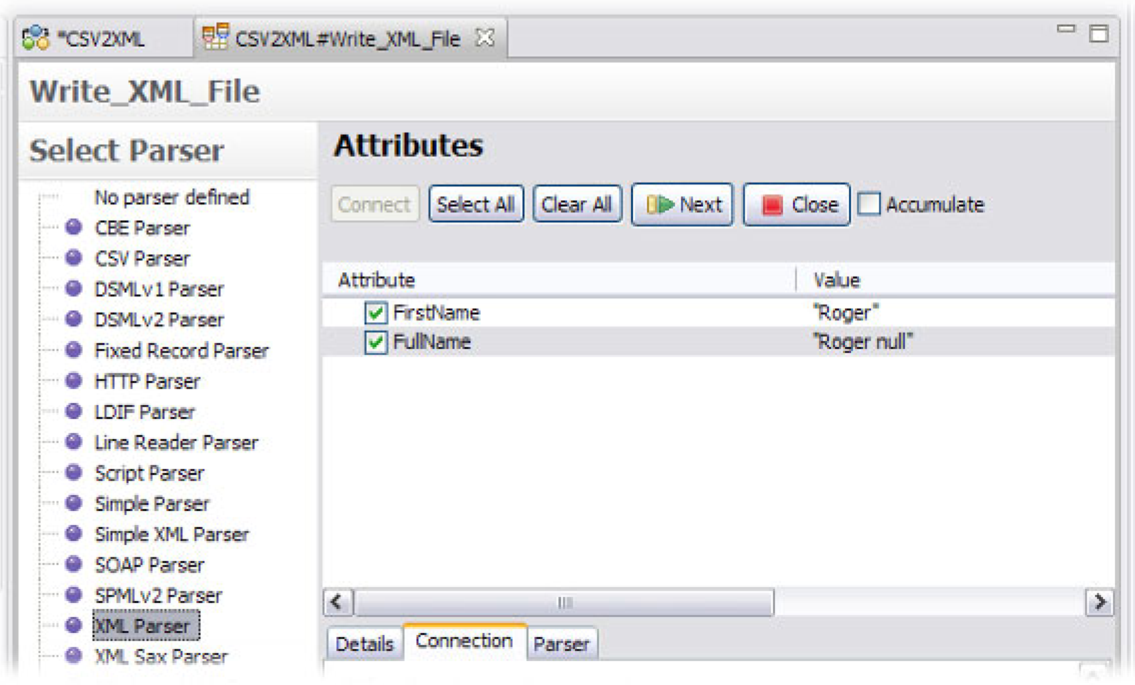


Figure 29. Browsing Data created by an Output Connector

This brings up the Data Browser for the chosen Connector, configured and ready to go. Press the **Connect** button and then **Next** to read and display the output data.



Attribute	Value
FirstName	"Roger"
FullName	"Roger null"

Figure 30. Browsing the resulting XML

The output XML should be an accurate representation of the input values, plus your mapping logic; everything looks good apart from the third entry (Roger), which is missing the 'LastName' and 'Title', and has this computed 'FullName' value: `Roger null`.

If you examine the input data more closely then you'll see that one of the CSV lines is incomplete:

```
First;Last;Title
Bill;Sanderman;Chief Scientist
Mick;Kamerun;CEO
Jill;Vox;CTO
Roger
Gregory;Highpeak;VP Product Development
Ernie;Hazzle;Chief Evangelist
Peter;Belamy;Business Support Manager
```

Missing and invalid input data is a common phenomena; your solution will need to be prepared to either filter out or correct this during processing.

Null Behavior: Dealing with missing Attributes/values

To deal with missing values you can either use the built-in Null Behavior feature of Attribute Maps, or you can detect and handle this yourself.

Let's start by configuring Null Behavior. Do this by right-clicking on the 'Read_CSV_File' Connector in the Attribute Maps panel (not in the AL components tree-view) and selecting **Null Behavior** from the context menu[20].

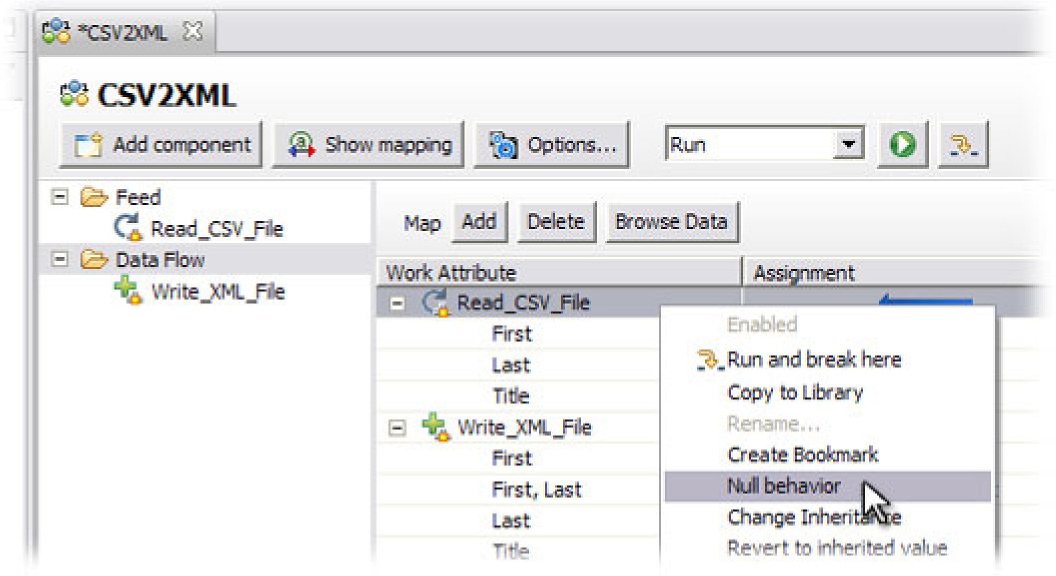
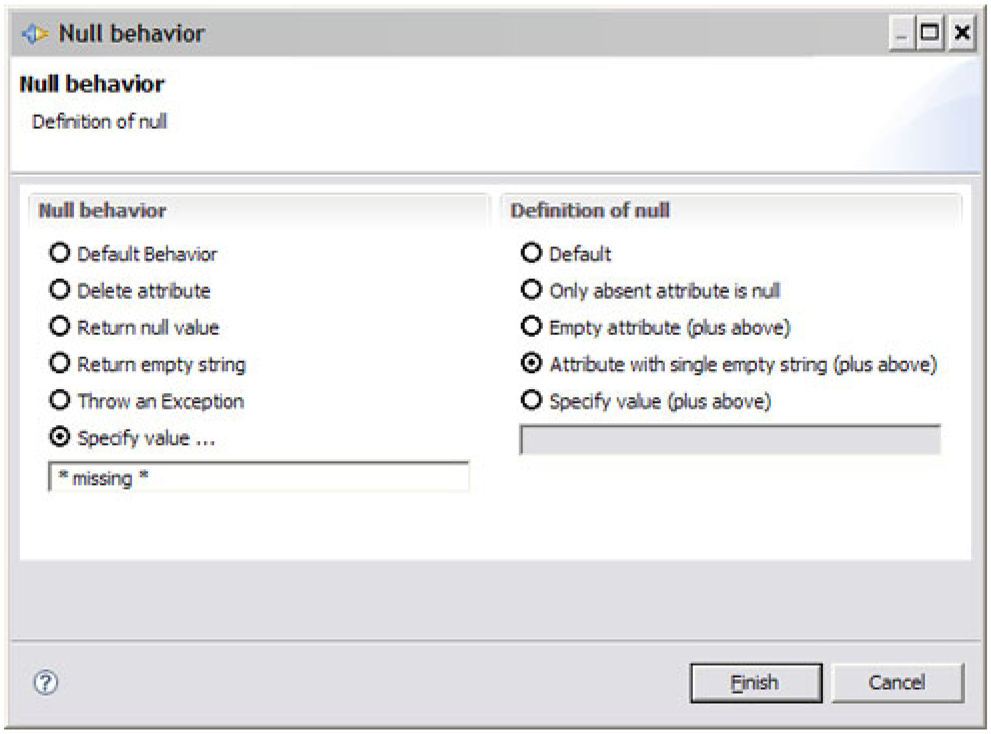


Figure 31. Null Behavior button for AL-level configuration

This brings up the Null Behavior dialog where you can configure both how *null* is defined – which can vary depending on the type of source you are reading from – as well as how this situation should be handled. By default, *null* means that an Attribute is missing, and the default handling is to remove this Attribute from the mapping operation. The end result of this is that no Attribute with the specified name will be found in the receiving Entry.

20. Or you can select the Connector itself in the AL component list; press the **More...** button at the top of the Input Map and choose 'Null Behavior' there.

Once in the Null Behavior dialog, use the radio buttons on the right to define *null* as an empty string value, and then those on the left to specify a default value of "* missing *" to be returned in this case.


Null behavior

Null behavior

Definition of null

Null behavior

- ○ Default Behavior
- ○ Delete attribute
- ○ Return null value
- ○ Return empty string
- ○ Throw an Exception
- ● Specify value ...

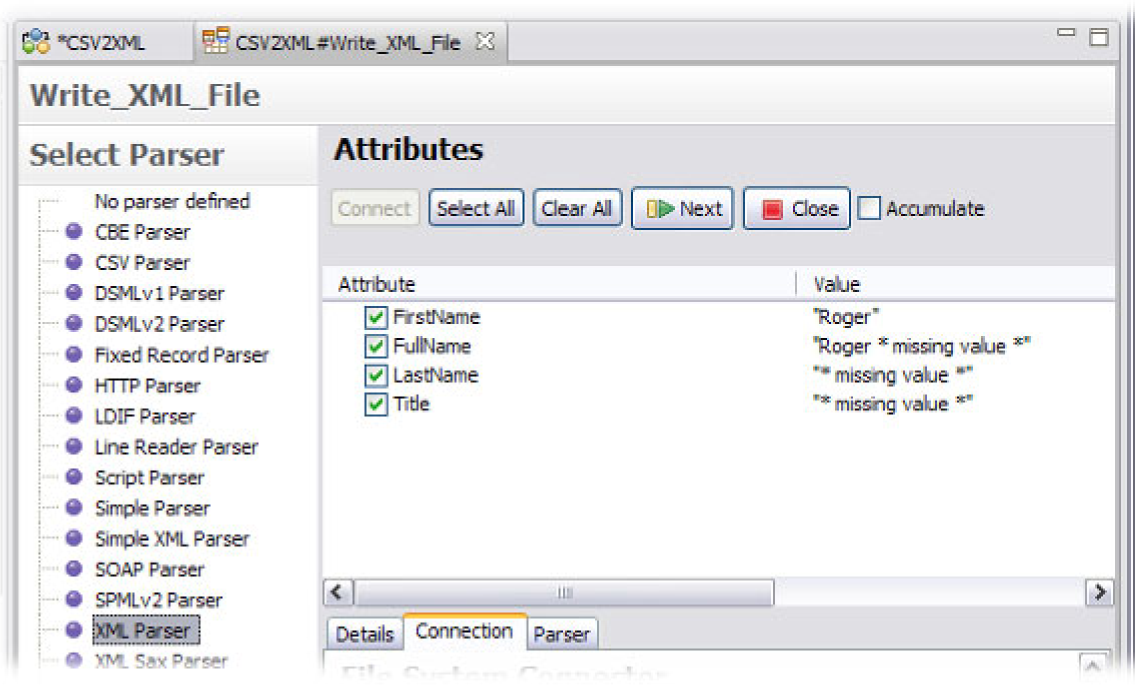
  * missing *

Definition of null

- ○ Default
- ○ Only absent attribute is null
- ○ Empty attribute (plus above)
- ● Attribute with single empty string (plus above)
- ○ Specify value (plus above)

Figure 32. Null Behavior configuration dialog

Re-run your AssemblyLine and refresh the browser window displaying the contents of `Output.xml`. The entry for 'Roger' should now have the special *null* value ("* missing *") for 'Title' and 'LastName'.



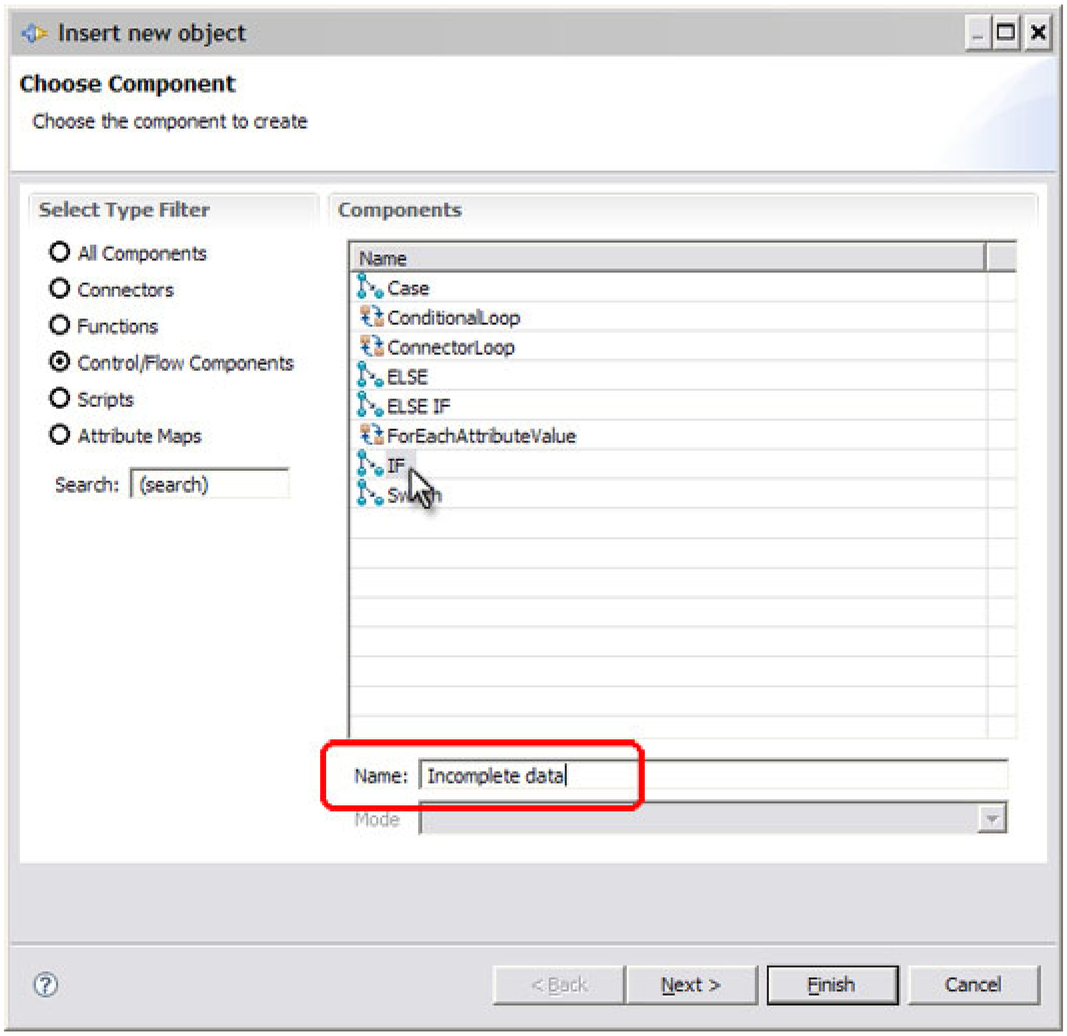
Attribute	Value
✓ FirstName	"Roger"
✓ FullName	"Roger *missing value *"
✓ LastName	"* missing value *"
✓ Title	"* missing value *"

Figure 33. Result in the XML output of Null Behavior settings

This is somewhat better – at least it's a conscious choice. However, sometimes an entry is just too incomplete to continue processing. In our scenario you need at least values for 'FirstName' and 'LastName' in order to compute 'FullName', so you will now add filtering logic to your AssemblyLine to ensure that all entries fulfill this requirement.

Start by clicking the **Insert Component** button again and then choosing **Control/Flow Components** from the radio buttons at the left. Then select 'IF' in the list, name it 'Incomplete data'[21] and press **Finish**.

21. You were previously encouraged to name Connectors as you would a script variable. This also applies to Function components. However, it is less important for Attribute Map components, Branches, Loops and Scripts, since these are seldom accessed directly from script. In this guide higher priority has been given to making the AssemblyLine easy to read.



( ) All Components
( ) Connectors
( ) Functions
(●) Control/Flow Components
( ) Scripts
( ) Attribute Maps

Name
Case
ConditionalLoop
ConnectorLoop
ELSE
ELSE IF
ForEachAttributeValue
IF
Switch

Name: Incomplete data

Figure 34. Selecting the IF branch component

Now drag this IF branch above your output Connector. The IF branch editor area lets you add your conditions.

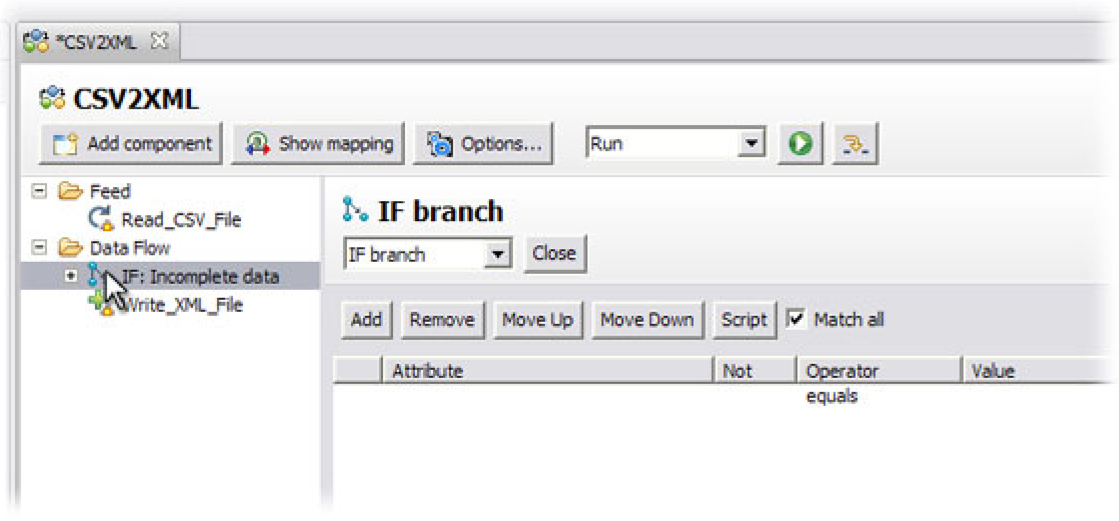
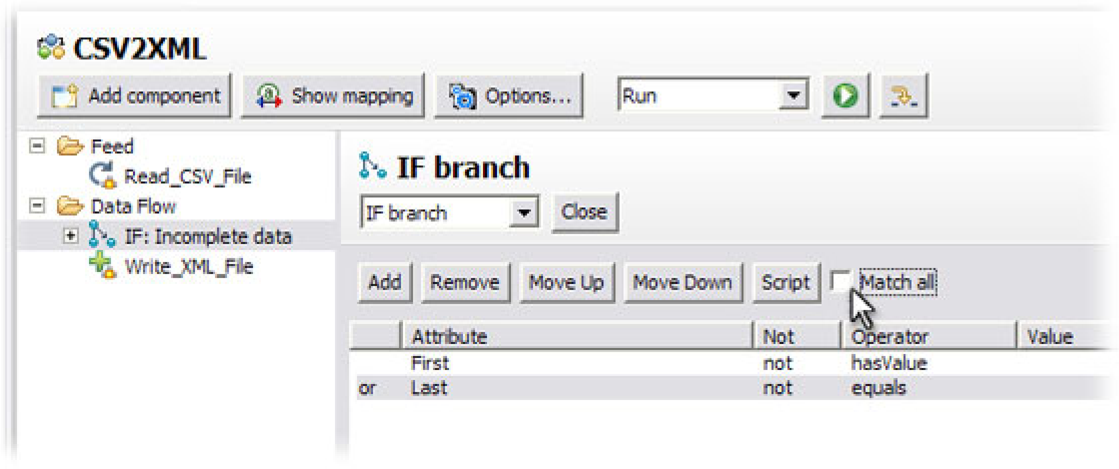


Figure 35. Editing conditions for the IF branch

Here you have the option of adding simple conditions or writing a snippet of Script – or both. Note the **Match All** checkbox that decides whether your Conditions (simple and scripted) are evaluated with an implied *OR* between them when this is unchecked, or with *AND* when it is selected.

Add a simple condition by pressing the **Add** button. Then pick the 'First' Attribute from the leftmost drop-down and then the 'has value(s)' operator. Negate this condition by toggling the **not** column value. Nothing needs to be specified in the right-most field when using the 'has value(s)' operator. Now add a similar *has no values* condition for the Attribute named 'Last' as well. Finally, make sure the **Match All** checkbox is unchecked – which implies *Match Any* – so that the lack of values for either Attribute will trigger this branch.



	Attribute	Not	Operator	Value
	First	not	hasValue	
or	Last	not	equals	

Figure 36. Adding simple Conditions to the IF branch

An IF Branch diverts the execution flow of the AssemblyLine whenever the specified Conditions evaluate to *true*. In your case, processing will continue to those components placed under the branch if either the 'First' or 'Last' Attributes do not have values assigned to them – or even if either does not exist in the Work Entry. In other words, the 'has value(s)' operator also includes the check for 'exists'.

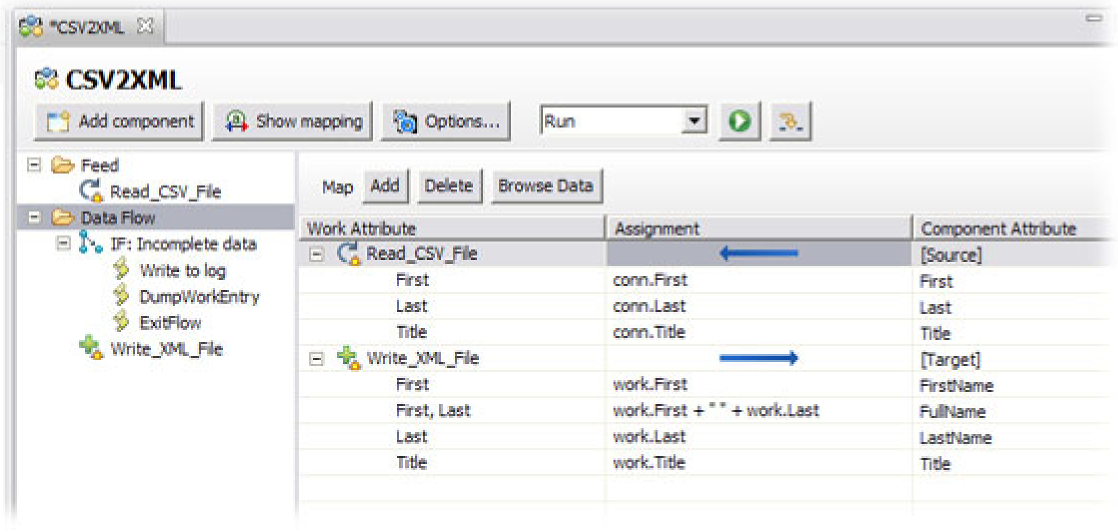
Now expand the IF branch and double-click on the placeholder displayed under it to insert a component here. In the **Choose Component** dialog, click on the radio button for **Scripts**, select the one called 'Script' and press **Finish**. Rename this Script component (also called an 'SC') to 'Write to log' and then enter the following snippet of JavaScript in it[22]:

```
task.logmsg("*** Skipping incomplete entry");
```

The *task* variable used here references the AssemblyLine itself, and it provides you with a number of useful functions like the `logmsg()` method. This scripted call causes the specified text message to be written to your log output.

To make log output even more informative, let's include the current contents of the Work Entry as well. Do this by right-clicking on the IF branch, selecting **Insert Component...** and again choosing the radio button for **Scripts**. This time select the pre-defined Script component labeled 'Dump Work Entry'.

Finally, you must instruct the AL to stop the current cycle at this point and return control to the Iterator Connector so that it can read in the next CSV line, effectively filtering the current Entry from the XML output. Unless you specify this behavior yourself then control will continue to the first component after the IF branch. So once again you must right-click on your IF branch and then on **Insert Component...** Choose **Scripts** and then select the SC called 'Exit Flow'. Now your AL should look like this:



Work Attribute	Assignment	Component Attribute
Read_CSV_File	←	[Source]
First	conn.First	First
Last	conn.Last	Last
Title	conn.Title	Title
Write_XML_File	→	[Target]
First	work.First	FirstName
First, Last	work.First + " " + work.Last	FullName
Last	work.Last	LastName
Title	work.Title	Title

Figure 37. Your first complete AssemblyLine

Before running your AssemblyLine again you will need to open the Null Behavior dialog once more and restore both the default definition and behavior selections – otherwise your IF Branch conditions will never evaluate to *true*.

22. The Script editor provides a feature called *code-completion* that shows which options you have. For example, type "tas" in the Script editor and then press the Ctrl + Space key combination to open the code-completion drop-down, which should provide a single option: **task**. If you press Enter then this choice is selected and entered in your script. Now type the period key (.) so your script becomes "task." and wait just a moment; You will see a new code-completion drop-down appear, this time with a list of all the methods and properties that you can access in the `task` object.

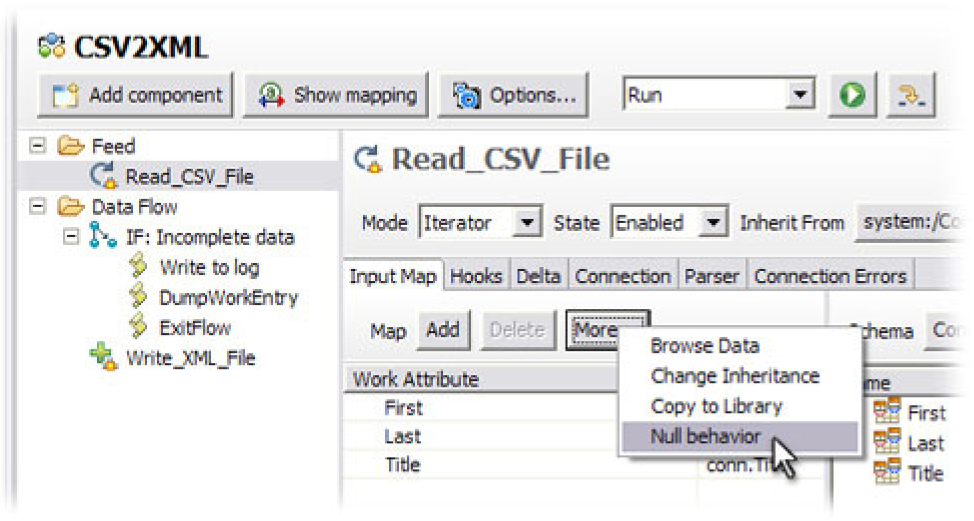
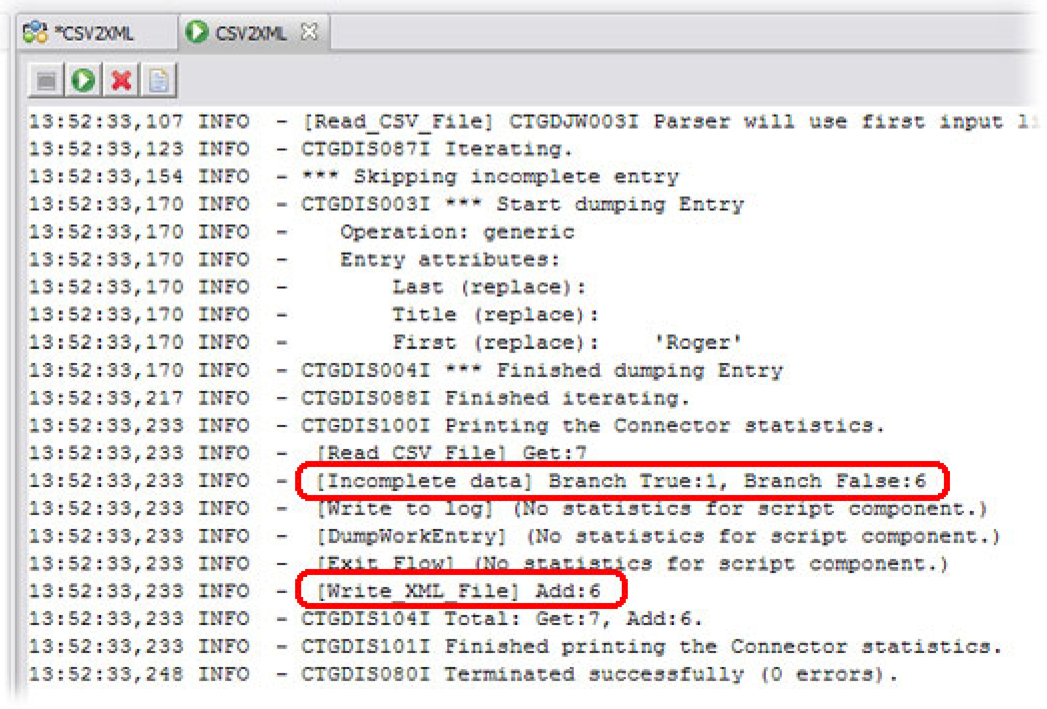


Figure 38. Resetting Null Behavior for the AssemblyLine

Now when you run your AssemblyLine again you will see your message followed by the Work Entry dump:



```
13:52:33,107 INFO  - [Read_CSV_File] CTGDJW003I Parser will use first input li
13:52:33,123 INFO  - CTGDIS087I Iterating.
13:52:33,154 INFO  - *** Skipping incomplete entry
13:52:33,170 INFO  - CTGDIS003I *** Start dumping Entry
13:52:33,170 INFO  -    Operation: generic
13:52:33,170 INFO  -    Entry attributes:
13:52:33,170 INFO  -        Last (replace):
13:52:33,170 INFO  -        Title (replace):
13:52:33,170 INFO  -        First (replace):    'Roger'
13:52:33,170 INFO  - CTGDIS004I *** Finished dumping Entry
13:52:33,217 INFO  - CTGDIS088I Finished iterating.
13:52:33,233 INFO  - CTGDIS100I Printing the Connector statistics.
13:52:33,233 INFO  -  [Read_CSV_File] Get:7
13:52:33,233 INFO  -  [Incomplete data] Branch True:1, Branch False:6
13:52:33,233 INFO  -  [Write to log] (No statistics for script component.)
13:52:33,233 INFO  -  [DumpWorkEntry] (No statistics for script component.)
13:52:33,233 INFO  -  [Exit Flow] (No statistics for script component.)
13:52:33,233 INFO  -  [Write_XML_File] Add:6
13:52:33,233 INFO  - CTGDIS104I Total: Get:7, Add:6.
13:52:33,233 INFO  - CTGDIS101I Finished printing the Connector statistics.
13:52:33,248 INFO  - CTGDIS080I Terminated successfully (0 errors).
```

Figure 39. Log output with your messages and Work Entry dump

From the statistics you can see that your IF 'Data incomplete' branch was true once, resulting in only six nodes being added to your XML output. If you again refresh the Data Browser window then you will see that 'Roger' is indeed gone.

Debugging your AssemblyLine

One of the most powerful features of Tivoli Directory Integrator is its built-in AssemblyLine Debugger that allows you to walk through the execution of your AL, viewing and even modifying data in-flight.

Let's step through your first AssemblyLine by clicking the **Start Debug** session button.

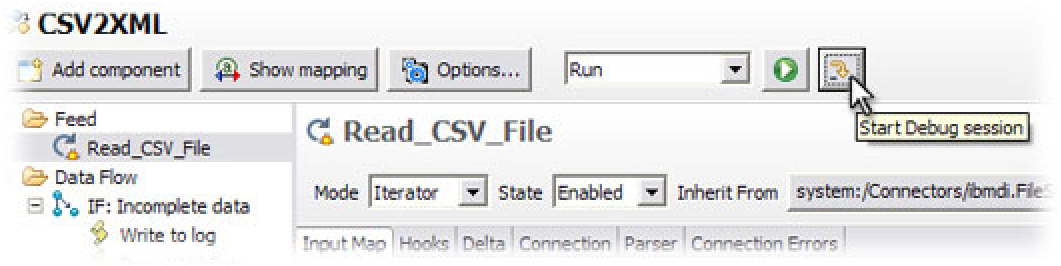
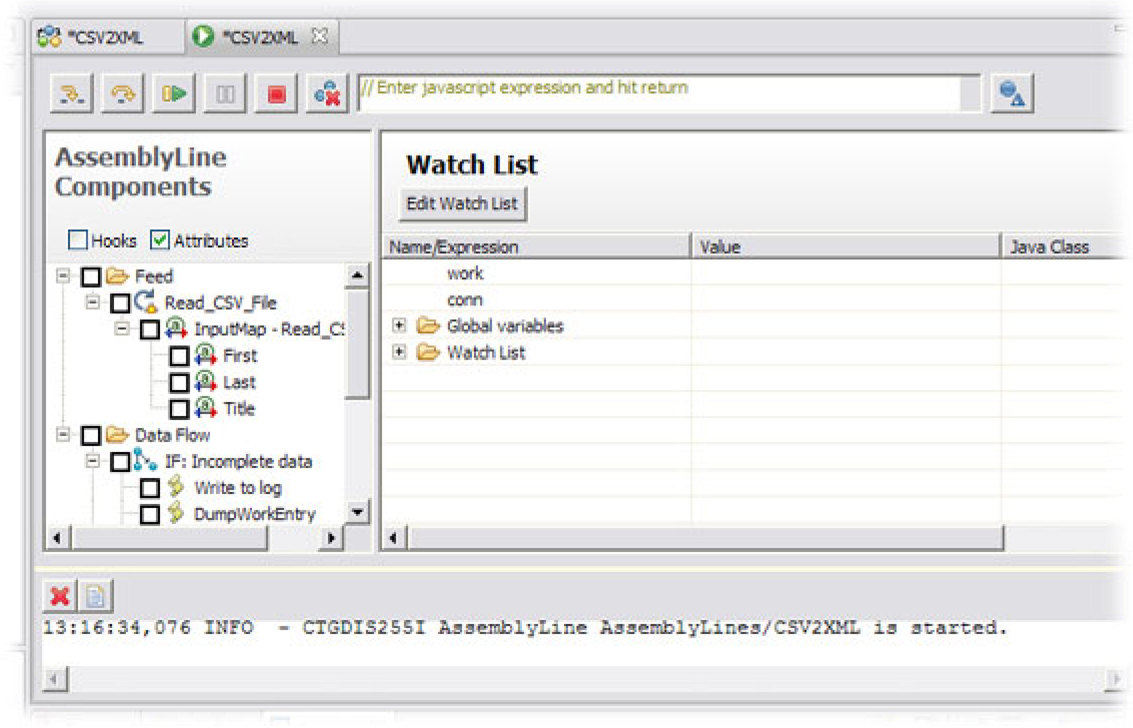


Figure 40. Starting the AL in Step (Paused) mode

Instead of the standard Run window, you will find yourself in the AL debug window.



```
13:16:34,076 INFO  - CTGDIS255I AssemblyLine AssemblyLines/CSV2XML is started.
```

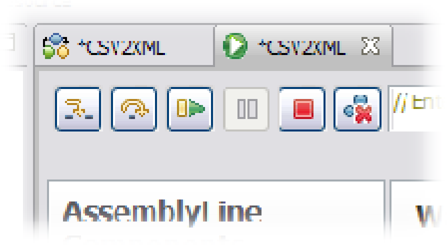
Figure 41. AssemblyLine Debugger

Along the bottom is the log output which is identical to what you saw in the standard Run window. The rest of this window is divided into two parts: the **AssemblyLine Components** list, and **Watch List**.

The **AssemblyLine Components** area shows your AL components along with the Hooks that let you influence the underlying automated workflows of Tivoli Directory Integrator. Each Hook is a script container that is executed at a specific point in the built-in workflow. The **Show disabled hooks** checkbox is used to display all Hooks, including those that are not currently enabled, allowing you to step through these as well. Leave this checkbox open for now. Next to each item in this tree-view is a **Breakpoint** checkbox that you can use to signal where execution should stop.

The **Watch List** displays the value of Attributes, script variables and Script expressions of your choosing. By default, this list contains the work Entry, and when available, the conn Entry as well. There is also a folder labeled 'Global variables' that contains all the script variables registered in the Script Engine of the running AssemblyLine.

If you now raise your attention to just above the **AssemblyLine Components** label, you'll see six buttons.



These have the following functions, starting from the left:

- **Step into** which move execution to the next item in the **Components and Hooks** tree-view. This includes each Attribute Map item and even each line of underlying script. If you have enabled Hooks, or have the **Show disabled hooks** options selected, then you will see control move from Hook to Hook as well.

  Use this button to step into the script of an Attribute Map assignment, SC or Hook.
- **Step over** takes you to the top of the next component, or Attribute Map item.
- **Continue** causes your AssemblyLine to run until the next Breakpoint.
- **Pause** is for pausing your AL and returning control to you in the Debugger.
- **Stop** causes AL processing to end.
- **Remove all breakpoints** unchecks all breakpoints you may have set throughout your AL.

Start the execution of your AssemblyLine by pressing the **Step into** button – the first one on the left. You should see 'Read_CSV_File' highlighted[23]. If you now press **Step into** then the CE will walk you through the mapping of each Attribute, starting with 'First'. Using the **Step into** button will even walk you through the Assignment script, line by line.

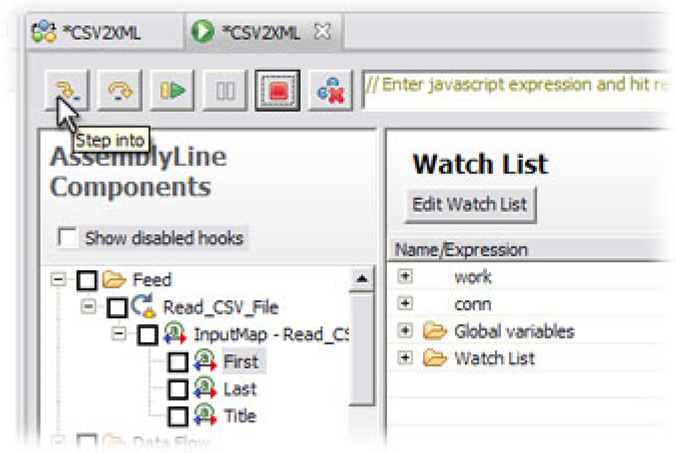


Figure 42. Stepping into the Attribute Map

Notice how a script window opens up to allow you to step through each line of JavaScript as it is executed. This is true for Attribute Maps, Script Components (SCs) and Hooks, and even scripted Connectors, Functions and Parsers.

23. If you had selected the **Show disabled hooks** checkbox, then the Debugger would have stepped through the Prolog Hooks of the AssemblyLine and both of your Connectors. Try this yourself to see how AL startup works.

Now set a check in the box next to the SC labeled 'Write to log' – the first one under your IF branch – and then press the **Continue** button.

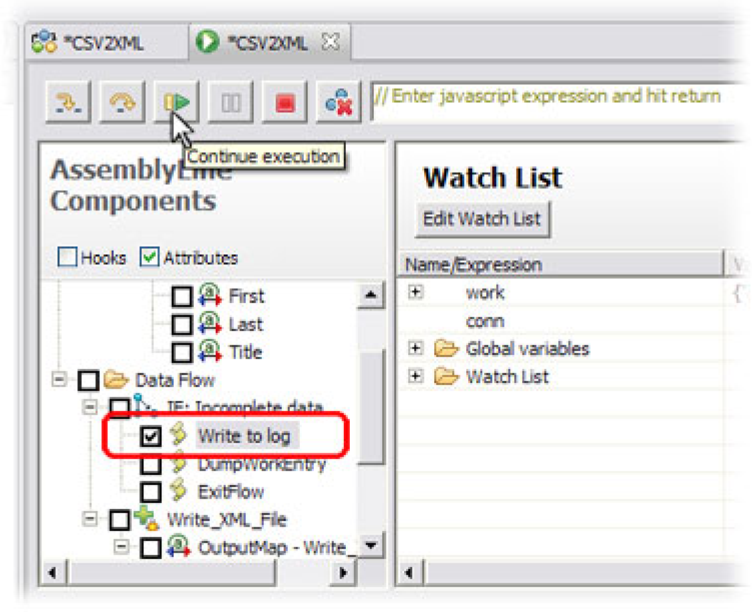
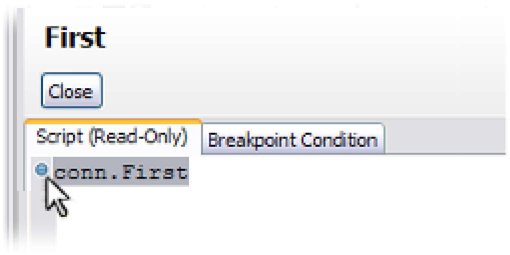


Figure 43. Set Breakpoint and Continue execution

The execution of your AL will proceed until the IF branch evaluates to *true* and processing reaches your Breakpoint, at which time a Script Quick Editor window appears for stepping through your script and control is returned to you. There is a tab labeled 'Breakpoint Condition' with a script editor where you can write a snippet of script that evaluates to *true* or *false* (like a branch Condition). If you enter a Conditional expression here then this Breakpoint will only be active when the Condition is *true*[24]. You will do this later in the tutorial exercise.

Whenever you are stepping through or viewing script in the Script Quick Editor, you can double-click to the left of any line of script in order to set a Breakpoint there.

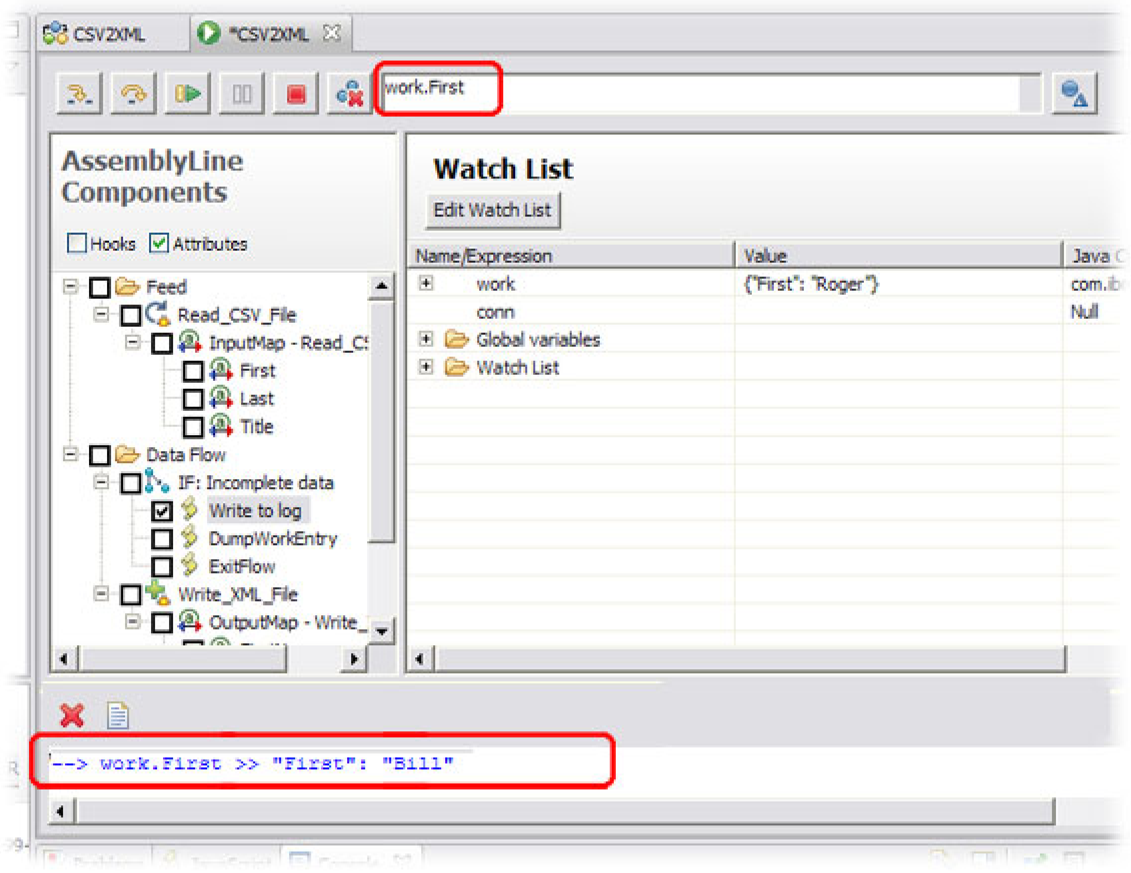


```
conn.First
```

Figure 44. Setting Breakpoints in your scripts

Another powerful debugging tool is the *Script Commandline* at the top of the Debugger screen, which allows you to interactively enter and execute script. Try this out while your AL is paused at the 'Write to log' Script component by first setting your cursor in the text field to the right of the Step buttons, entering the script snippet `work.First` and then pressing the ENTER key on your keyboard. Remember that the names of Attributes are case-sensitive when using this shorthand notation.

24. Conditional Breakpoints make it easy to look for specific values in Attributes or script variables. Double-click on any item in the Components and Hooks tree-view in order to add, edit or delete a Condition for that Breakpoint.



```
work.First
```

Name/Expression	Value	Java C
work	{"First": "Roger"}	com.ib
conn		Null
Global variables		
Watch List		

```
--> work.First >> "First": "Bill"
```

Figure 45. Evaluating script interactively in the running AL

Pressing the ENTER key causes your script snippet to be sent to the Server where it is executed in the context of your running AssemblyLine. Results from this statement are then displayed in the Log Output window. In your case the result was to return the Work Entry Attribute named 'First' with the value 'Roger'[25].

The Script Commandline also lets you modify data, for example using the `work.setAttribute()` method or by directly assigning values to script variables:

```
work.First = "Roger is incomplete";
```

From this interactive script area you can also invoke Script functions, load jar files, instantiate objects and call Java library functions – in short, anything that you can do from script from within your AssemblyLine. The Script Commandline also provides a history drop-down so that you can browse and re-use previously entered snippets.

25. You may also have noticed that the name of an Attribute is not case-sensitive. As such, `work.first` and `work.First` both evaluate to the same output.

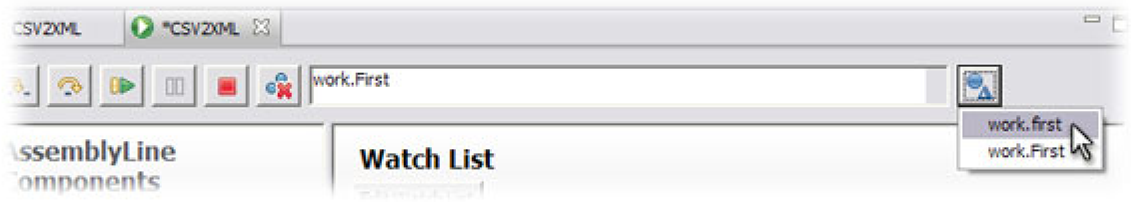
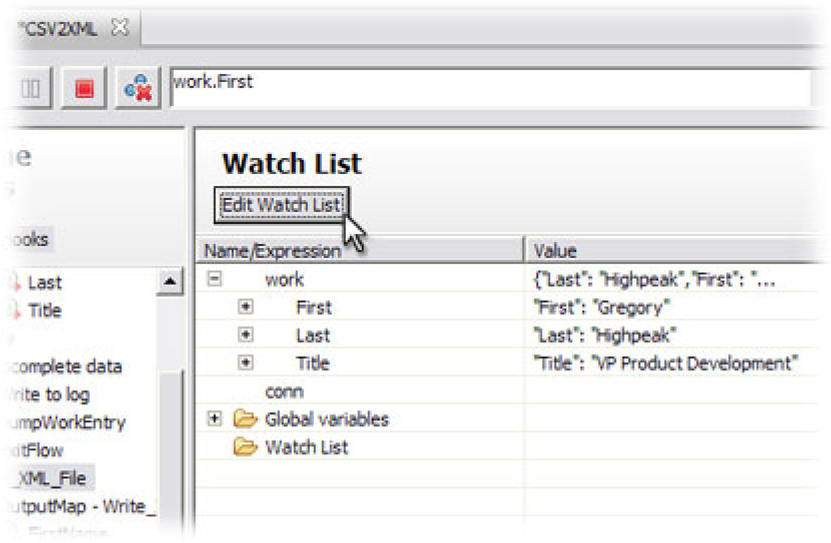


Figure 46. Script Commandline history drop-down

Finally, there is a button labeled **Edit Watch List** at the top of the 'Watch List' display for adding items to your *Watch List*. The Watch List is a series of JavaScript variables and expressions that are automatically updated in the display each time the Debugger pauses to give you control.



Name/Expression	Value
work	{"Last": "Highpeak","First": "...
First	"First": "Gregory"
Last	"Last": "Highpeak"
Title	"Title": "VP Product Development"
conn	
Global variables	
Watch List	

Figure 47. Editing the Watch List

It is highly recommended that you spend some time to familiarize yourself with the AssemblyLine Debugger. Not only does it provide unique insight into how your AL operates, including all the built-in workflows provided by the Server kernel, but it will help you validate your own implementation and assumptions about your data.

Looking up data from a sequential source

Continuing with our tutorial scenario, the next step is to add the lookup from D2.

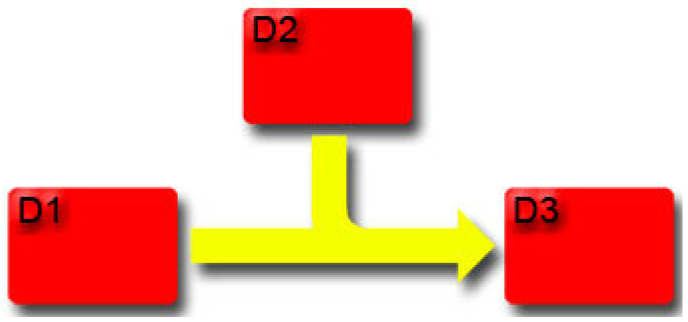
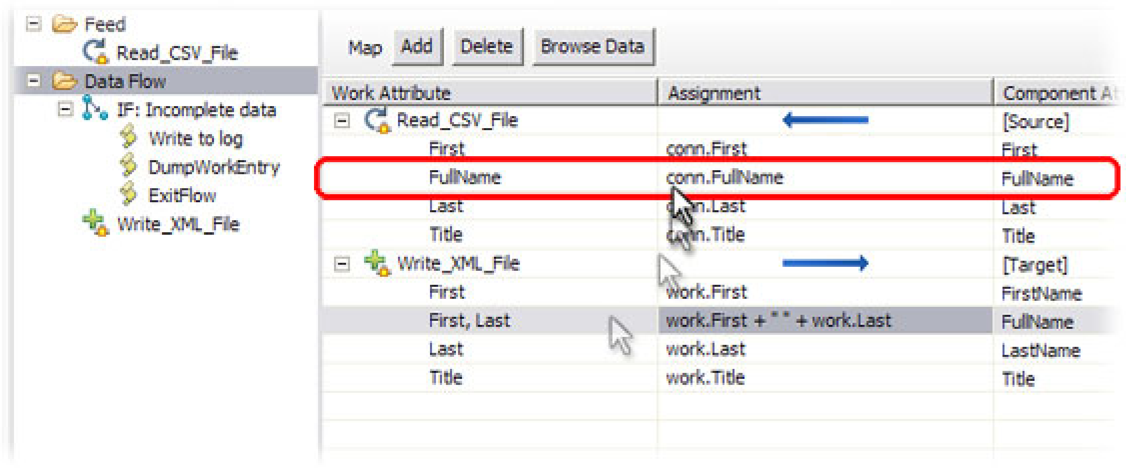


Figure 48. The scenario flow diagram

The Tutorial directory contains a file called `PhoneNumbers.xml` that will serve as your D2 data source. This file holds a series of XML entries, each with two attributes: 'User' and 'telephoneNo'.

Your job will be to include 'telephoneNo' as part of the data written to the output XML document. Since you can't randomly access this text file to do a Lookup as you could for a database or directory, the correct telephone number for each CSV entry will be found by looping through the file and comparing 'User' with 'FullName' coming from the current CSV entry.

However, 'FullName' is first being computed in the Output Map of the 'Write_XML_File' Connector – in other words, too late to do the comparison. That means you must move this computed Attribute from the Output Map of 'Write_XML_File' to the Input Map of 'Read_CSV_File'. Do this by first dragging the Attribute Map item up from one map to the other.

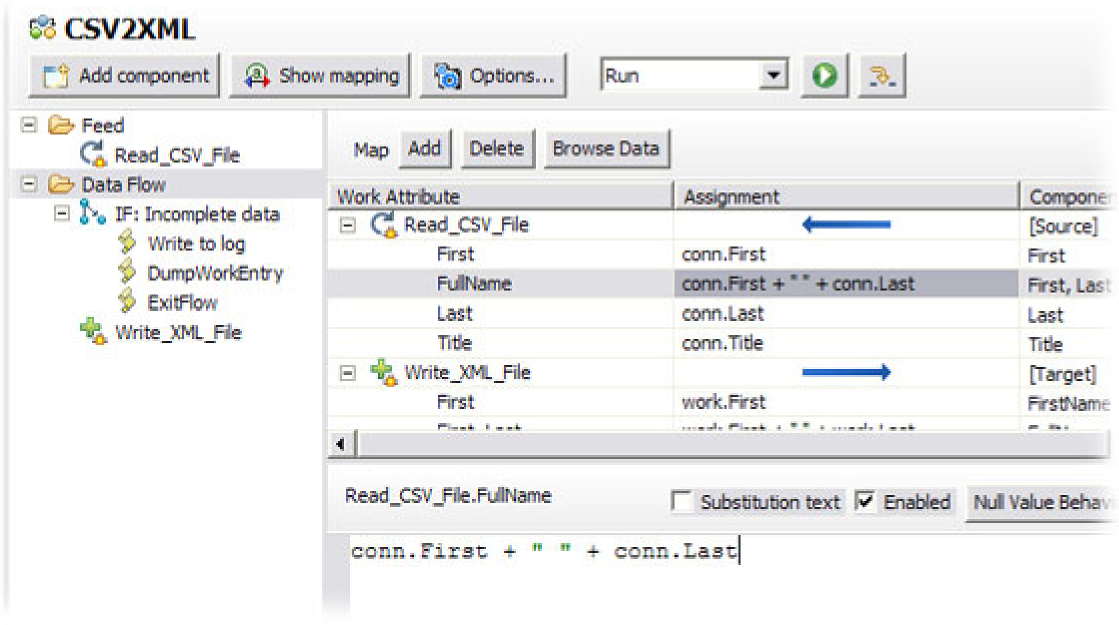


Work Attribute	Assignment	Component A
Read_CSV_File		[Source]
First	conn.First	First
FullName	conn.FullName	FullName
Last	conn.Last	Last
Title	conn.Title	Title
Write_XML_File		[Target]
First	work.First	FirstName
First, Last	work.First + " " + work.Last	FullName
Last	work.Last	LastName
Title	work.Title	Title

Figure 49. Dragging 'FullName' to the Input Map of your Iterator Connector

Now there will be a 'FullName' Map item in both maps. You need to adjust the Input Map assignment since you are now mapping from the Conn Entry to Work, instead of the other way around as is the case for an Output Map. Do this by double-clicking on 'FullName' under 'Read_CSV_File' and changing the assignment to be:

```
conn.First + " " + conn.Last
```


CSV2XML

Add component | Show mapping | Options... | Run

- Feed
  - Read_CSV_File
- Data Flow
  - IF: Incomplete data
    - Write to log
    - DumpWorkEntry
    - ExitFlow
  - Write_XML_File

Map | Add | Delete | Browse Data

Work Attribute	Assignment	Compone
Read_CSV_File	←	[Source]
First	conn.First	First
FullName	conn.First + " " + conn.Last	First, Last
Last	conn.Last	Last
Title	conn.Title	Title
Write_XML_File	→	[Target]
First	work.First	FirstName

Read_CSV_File.FullName   Substitution text   [✓ Enabled]   Null Value Behav

```
conn.First + " " + conn.Last
```

Figure 50. Editing the assignment for 'FullName'

You can also edit the original 'FullName' Output Map item so that its assignment is simply `work.FullName` since this Attribute will now be available in the Work Entry, thanks to your modified Input Map.

Now re-run your AssemblyLine and check `Output.xml` to make sure it is unchanged. Once you've confirmed this, you will now use a *Loop* component to read through the `PhoneNumbers.xml` file and search for each user's number[26].

Start by adding a new component to the **Data Flow** section, this time choosing the component called *ConnectorLoop* and then naming it 'TelephoneNumber'. A ConnectorLoop is a looping component that uses a Connector to read information from a data source and then cycles all components attached under it once for each entry returned by that Connector. This is similar to the *for-each* behavior of an Iterator Connector in the Feed section, which cycles components in the **Data Flow** section for each entry read.

Drag your new 'TelephoneNumber' ConnectorLoop between the IF branch and the 'Write_XML_File' Connector. Make sure it does not end up *inside* the IF branch.

26. There are three types of Loop components: 1) The *ConnectorLoop*, which lets you cycle on data returned by a Connector in Iterator or Lookup mode. This is the type of Loop you will use in this exercise; 2) the *ForEachAttributeValueLoop*, making it easy to loop through the values of a multi-valued Attribute, such as those you find in systems like Lotus Notes and LDAP Directories; and 3) the *ConditionalLoop*, which uses Simple and scripted Conditions – just like those used by Branches – to control cycling.

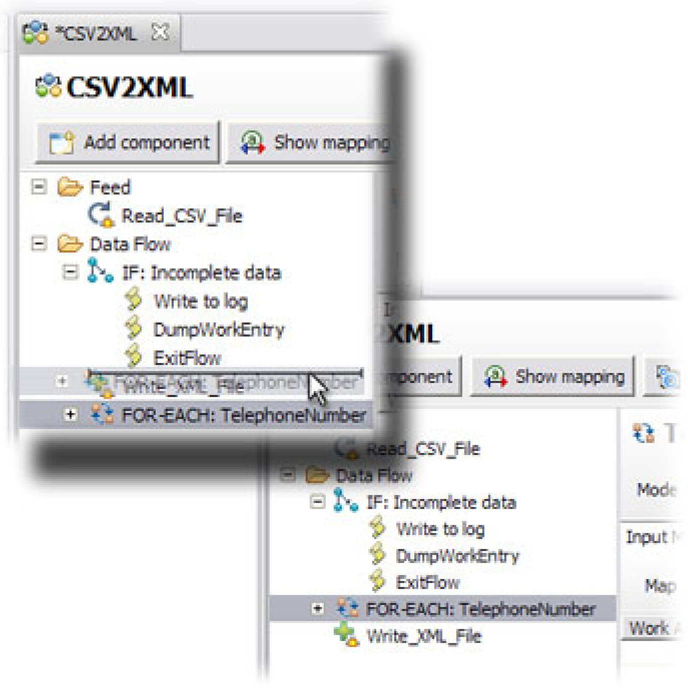


Figure 51. Drag the ConnectorLoop

Select it now to open its editor, which is similar to a Connector editor.

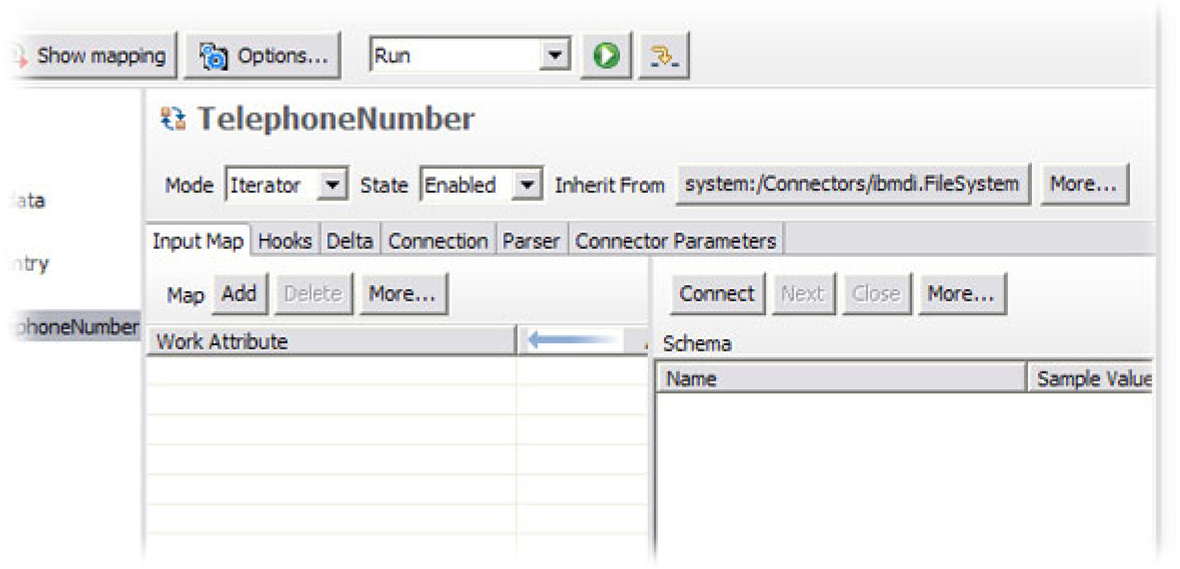


Figure 52. ConnectorLoop Configuration

The main differences are that the **Mode** drop-down will only ever contain **Iterator** and **Lookup** options. Furthermore, there is a **More...** button that provides options for limiting the entries cycled, as well as an **Initialize** drop-down parameter with the three selections:

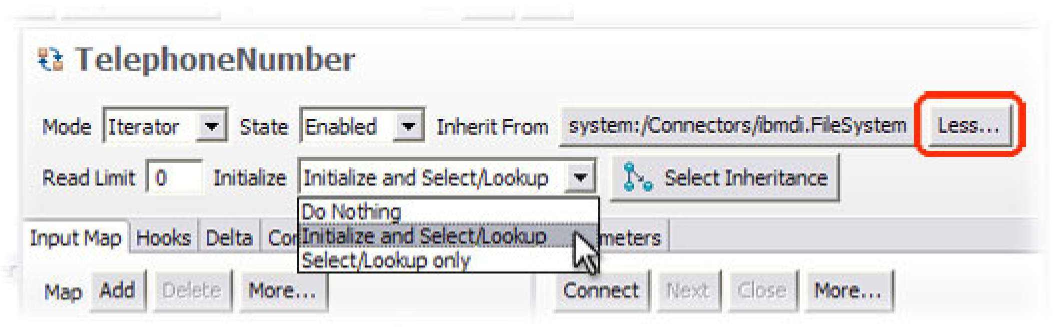

TelephoneNumber
Mode Iterator
State Enabled
Inherit From system:/Connectors/ibmdi.FileSystem
Less...
Read Limit 0
Initialize Initialize and Select/Lookup
Select Inheritance
Do Nothing
Initialize and Select/Lookup
Select/Lookup only
Input Map Hooks Delta Co meters
Map Add Delete More...
Connect Next Close More...

Figure 53. ConnectorLoop Advanced Settings

- **Do Nothing**, which means that when this Loop is reached during AL processing then its embedded Connector will not be initialized in any way;
- **Initialize and Select/Lookup**, to cause the Connector to be initialized whenenever the Loop starts to cycle. Use this option since your ComponentLoop will be reading from a file and you want it start from the beginning each time;
- **Select/Lookup Only**, which is useful when your ConnectorLoop is pointing at a database, directory or some other randomly accessible data source. Re-initializing the connection each time is not necessary in this case. All that must be done is to re-issue the search, which is a *Select* in the case of Iterator mode, and a *Lookup* operation for Lookup mode.

Configure the LoopConnector (which is of type 'FileSystem' by default) to read the `PhoneNumber.xml` file and then select the 'XML Parser'. Now bring up the Attribute Map tab to discover Attributes.

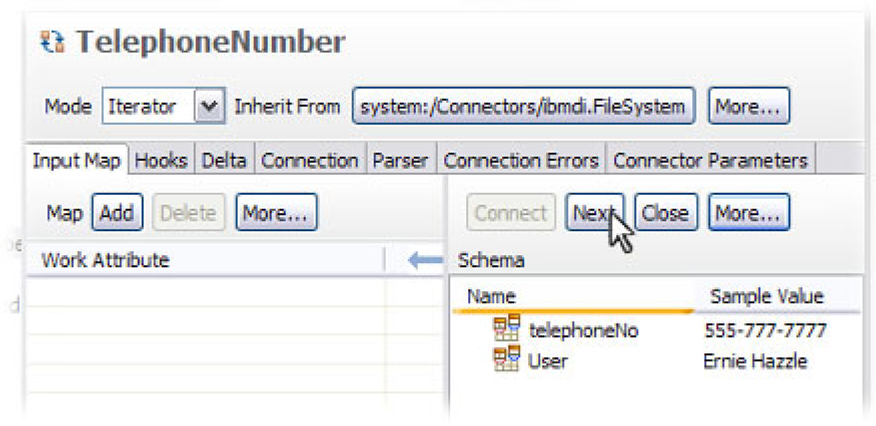
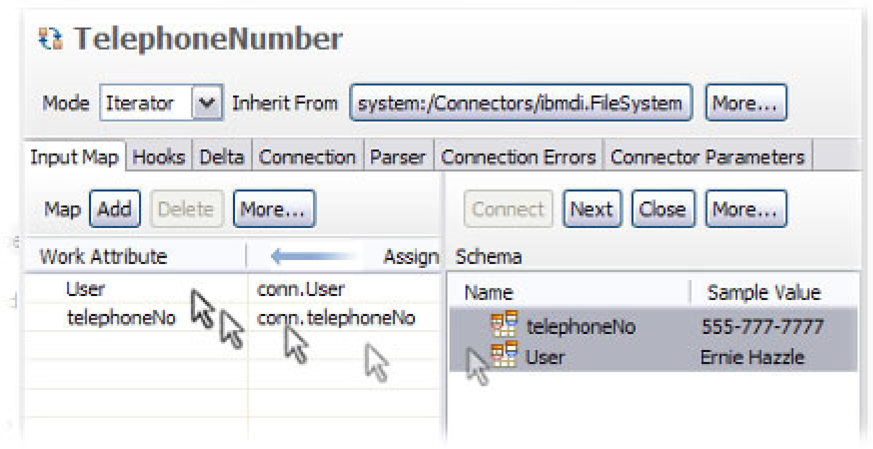

TelephoneNumber
Mode Iterator
Inherit From system:/Connectors/ibmdi.FileSystem
More...
Input Map Hooks Delta Connection Parser Connection Errors Connector Parameters
Map Add Delete More...
Connect Next Close More...
Work Attribute
Schema
Name
Sample Value
telephoneNo 555-777-7777
User Ernie Hazzle

Figure 54. Hierarchical Attributes

You will do your mapping at the Attribute level by selecting the 'User' and 'telephoneNo' in the Input Schema and dragging them to the Input Map.


TelephoneNumber
Mode Iterator
Inherit From system:/Connectors/ibmdi.FileSystem
More...
Input Map Hooks Delta Connection Parser Connection Errors Connector Parameters
Map Add Delete More...
Connect Next Close More...
Work Attribute
Assign
Schema

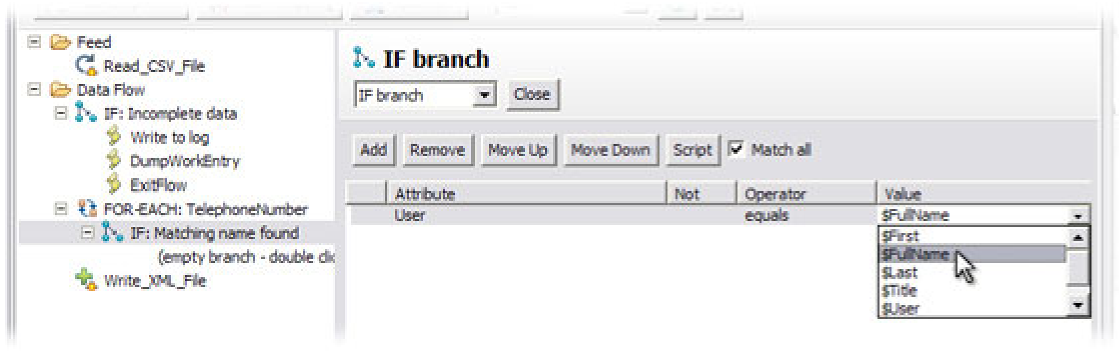
Work Attribute	Assign
User	conn.User
telephoneNo	conn.telephoneNo

Name	Sample Value
telephoneNo	555-777-7777
User	Ernie Hazzle

Figure 55. Dragging from Schema to Attribute Map

As a result, you will have one mapping rule for an Attribute named "User" and one for "telephoneNo".

You can now close the LoopConnector editor and then add an IF branch underneath it by right-clicking on the ConnectorLoop and choosing **Insert Component...**. Call this IF branch 'Matching name found'. Now add a simple condition that checks if 'User' *equals* '$FullName'[27].



Attribute	Not	Operator	Value
User		equals	$FullName

Figure 56. Condition editor for IF branch

Whenever a match is found then you will want the ConnectorLoop to exit with the correct values in the 'User' and 'telephoneNo' Attributes. To do this, add a Script component that you name 'Exit loop' and write the following script into:

```
system.exitBranch("loop");
```

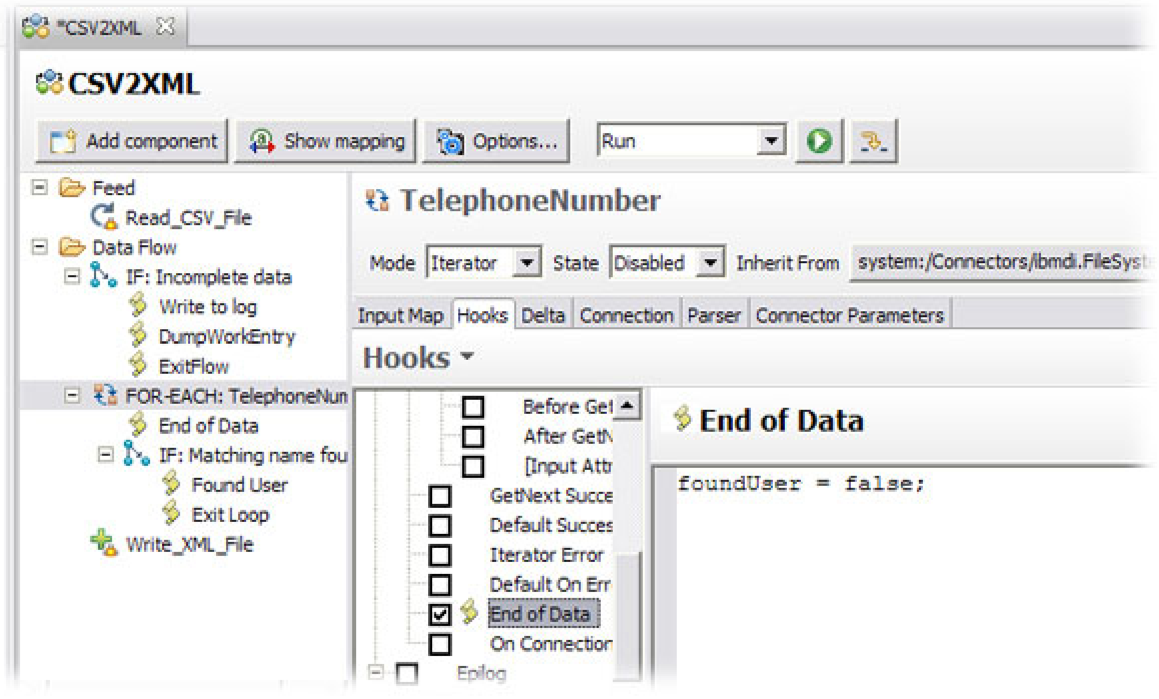
But what happens if the ConnectorLoop reaches the end of `PhoneNumbers.xml` without finding a match? The 'User' and 'telephoneNo' Attributes contain the values read from the last entry in the file, so just checking for empty Attributes won't help. You will need to devise some other way of detecting a failed match.

The answer is to use a script variable as a flag to indicate that a match was found. Do this now by inserting a Script component that you call 'Found user' inside the IF branch, dragging it just before the 'Exit loop' SC. This Script component should contain the following script snippet:

```
foundUser = true;
```

27. The dollar sign is a special character used here to indicate that 'FullName' is not a literal string to match, but rather the value of an Attribute found in the Work Entry.

To indicate that a the end of the input file has been reached without finding a match simply select your ConnectorLoop and open the Hooks tab.



Mode Iterator State Disabled Inherit From system:/Connectors/ibmdi.FileSys

```
foundUser = false;
```

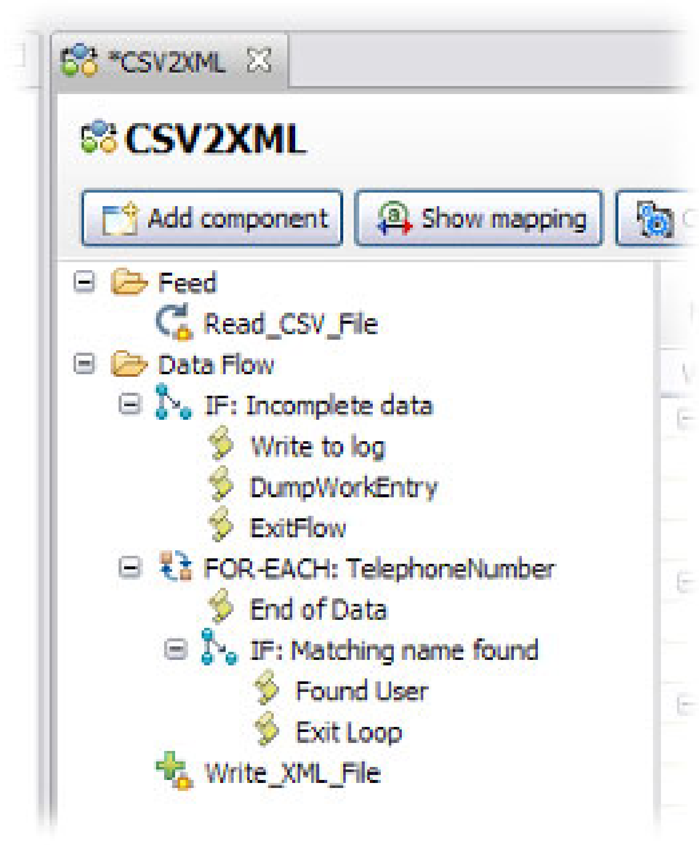
Figure 57. Scripting the End of Data Hook

Select the Hook called 'End of Data' and enter this script.

```
foundUser = false;
```

The 'End of Data' Hook will only be reached if the Connector attempts to read past the last entry in its connected source. In this case, no match has been found.

Now your AssemblyLine should have these components:



```
Feed
    Read_CSV_File
Data Flow
    IF: Incomplete data
        Write to log
        DumpWorkEntry
        ExitFlow
    FOR-EACH: TelephoneNumber
        End of Data
        IF: Matching name found
            Found User
            Exit Loop
    Write_XML_File
```

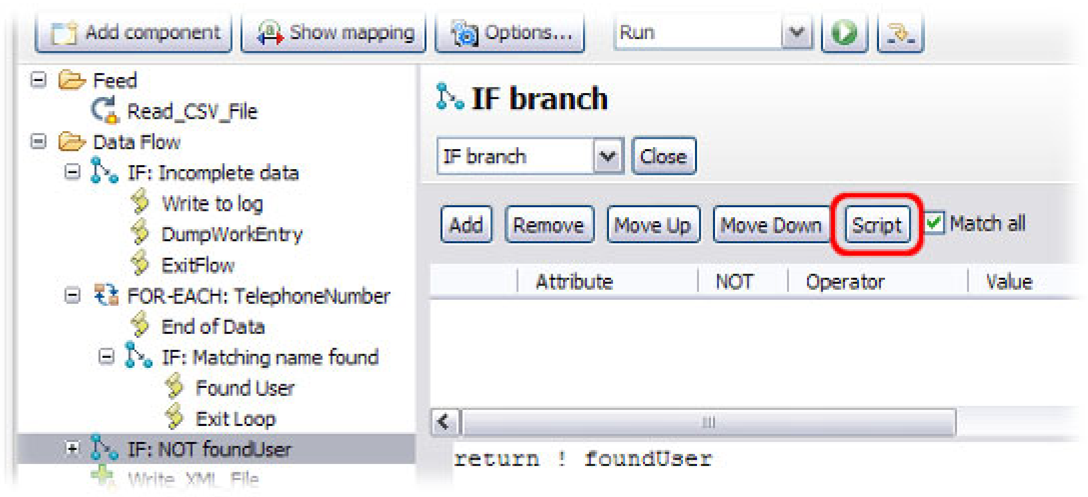
Figure 58. Component list in the AssemblyLine Data Flow section

You should now be able to ascertain whether or not the search was successful by checking your script variable. This is important since whenever no match is found then you must also set a default value for the 'telephoneNo' Attribute; otherwise it will still have the last value read in by the ConnectorLoop.

So add another IF branch immediately following the ConnectorLoop and call it 'NOT foundUser'. Click on the **Script** button in the **IF Branch** details area and enter this script to check the value of your script variable:

```
! foundUser
```

The exclamation mark negates the value of foundUser so if it has been set to *false* in the 'End of Data' Hook of your ConnectorLoop, this branch Condition will evaluate to *true*.



```
return ! foundUser
```

Figure 59. Scripting a Condition for the IF branch

Insert a new component of type 'Attribute Map' underneath it. Call this Attribute Map component 'Set default telephoneNo' to make its function clear in the context of your AssemblyLine. Now use the **Add Attribute** button to create a single Attribute named 'telephoneNo' – the same name as that being returned by your ConnectorLoop. Double-click on this Attribute to set up the assignment script:

```
"N/A"
```

This means that any person read from your CSV input and not found in PhoneNumbers.xml will get a 'telephoneNo' value of "N/A"[28].

Finally, include this new 'telephoneNo' Attribute in the Output Map of 'Write_XML_File' by dragging it there and then making sure the assignment is:

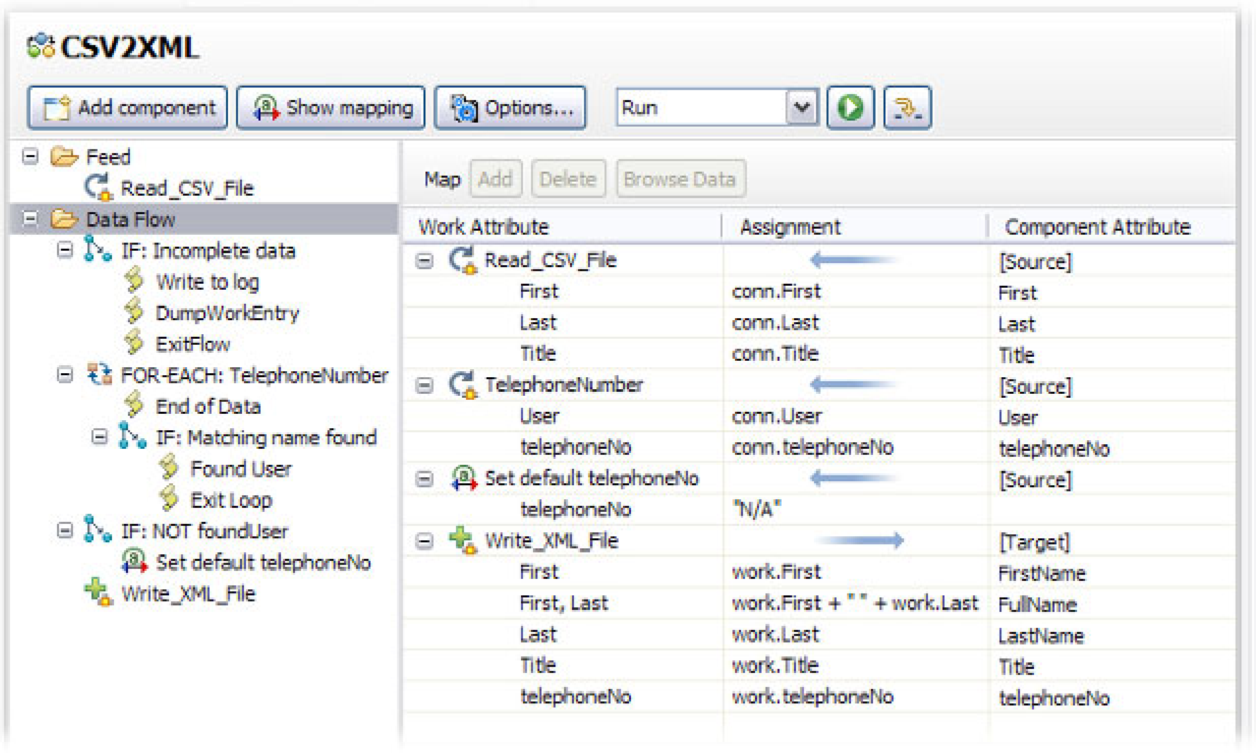
```
work.telephoneNo
```

Your AssemblyLine should now look like this.

28. If you would prefer these users to have no 'telephoneNo' Attribute at all, simply use an assignment that returns no value. This is done by returning the special value null:

```
null
```

This will cause default Null Behavior will remove the Attribute from the Work Entry. As a result, it will not reach the Output Map of your 'Write_XML_File' Connector and therefore not appear in the resulting XML document.



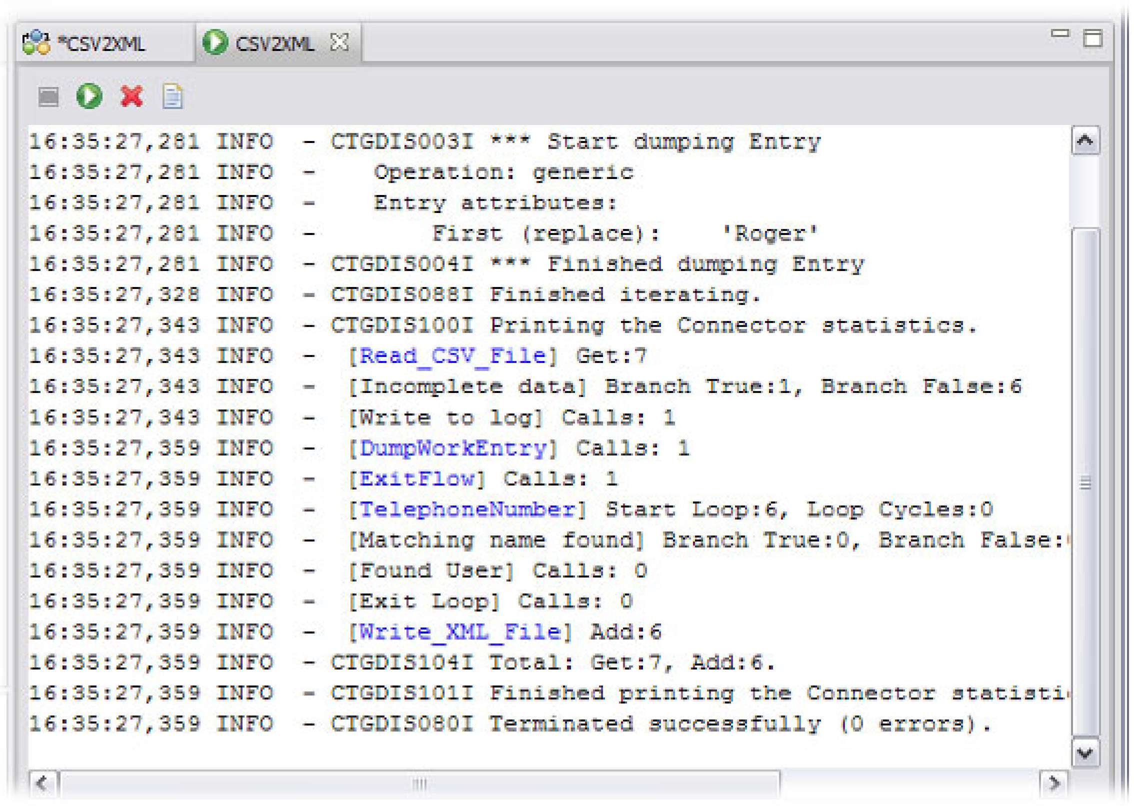
Feed
- Read_CSV_File

Data Flow
- IF: Incomplete data
  - Write to log
  - DumpWorkEntry
  - ExitFlow
- FOR-EACH: TelephoneNumber
  - End of Data
  - IF: Matching name found
    - Found User
    - Exit Loop
- IF: NOT foundUser
  - Set default telephoneNo
- Write_XML_File

Work Attribute	Assignment	Component Attribute
Read_CSV_File	←	[Source]
First	conn.First	First
Last	conn.Last	Last
Title	conn.Title	Title
TelephoneNumber	←	[Source]
User	conn.User	User
telephoneNo	conn.telephoneNo	telephoneNo
Set default telephoneNo	←	[Source]
telephoneNo	"N/A"	
Write_XML_File	→	[Target]
First	work.First	FirstName
First, Last	work.First + " " + work.Last	FullName
Last	work.Last	LastName
Title	work.Title	Title
telephoneNo	work.telephoneNo	telephoneNo

Figure 60. AssemblyLine complete with FOR-EACH Loop

Now run your AL again and examine the log output. Your 'NOT foundUser' branch should have been *true* twice and *false* for the other four entries.

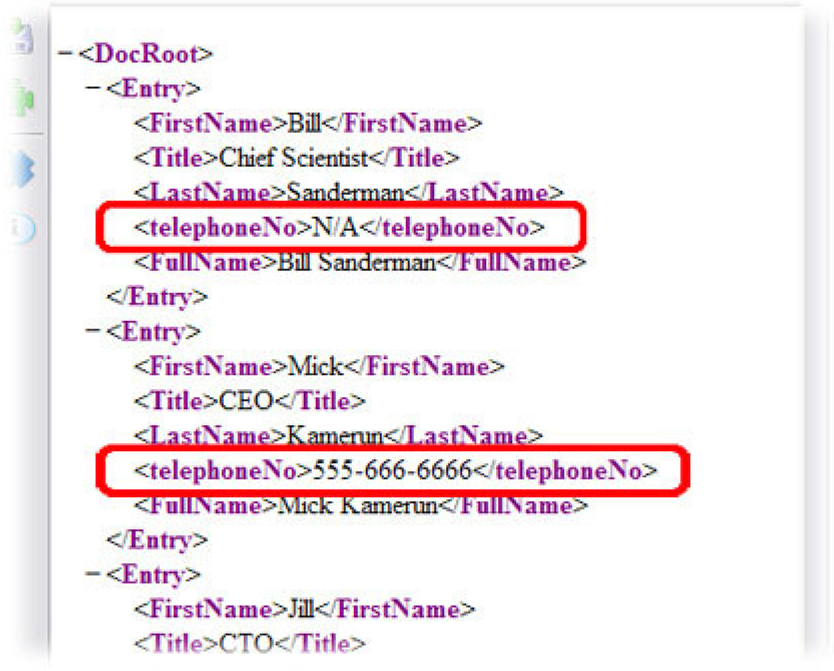


```
16:35:27,281 INFO  - CTGDIS003I *** Start dumping Entry
16:35:27,281 INFO  -     Operation: generic
16:35:27,281 INFO  -     Entry attributes:
16:35:27,281 INFO  -         First (replace):    'Roger'
16:35:27,281 INFO  - CTGDIS004I *** Finished dumping Entry
16:35:27,328 INFO  - CTGDIS088I Finished iterating.
16:35:27,343 INFO  - CTGDIS100I Printing the Connector statistics.
16:35:27,343 INFO  -  [Read_CSV_File] Get:7
16:35:27,343 INFO  -  [Incomplete data] Branch True:1, Branch False:6
16:35:27,343 INFO  -  [Write to log] Calls: 1
16:35:27,359 INFO  -  [DumpWorkEntry] Calls: 1
16:35:27,359 INFO  -  [ExitFlow] Calls: 1
16:35:27,359 INFO  -  [TelephoneNumber] Start Loop:6, Loop Cycles:0
16:35:27,359 INFO  -  [Matching name found] Branch True:0, Branch False:
16:35:27,359 INFO  -  [Found User] Calls: 0
16:35:27,359 INFO  -  [Exit Loop] Calls: 0
16:35:27,359 INFO  -  [Write_XML_File] Add:6
16:35:27,359 INFO  - CTGDIS104I Total: Get:7, Add:6.
16:35:27,359 INFO  - CTGDIS101I Finished printing the Connector statisti
16:35:27,359 INFO  - CTGDIS080I Terminated successfully (0 errors).
```

Figure 61. Log Output with IF branch statistics

Note that some component names are highlighted (blue) in the AL statistics of the log output. If you Ctrl-click on one with left mouse button it opens the selected component up in the AssemblyLine editor.

As a result, your XML output should look like this:



```
- <DocRoot>
  - <Entry>
      <FirstName>Bill</FirstName>
      <Title>Chief Scientist</Title>
      <LastName>Sanderman</LastName>
      <telephoneNo>N/A</telephoneNo>
      <FullName>Bill Sanderman</FullName>
    </Entry>
  - <Entry>
      <FirstName>Mick</FirstName>
      <Title>CEO</Title>
      <LastName>Kamerun</LastName>
      <telephoneNo>555-666-6666</telephoneNo>
      <FullName>Mick Kamerun</FullName>
    </Entry>
  - <Entry>
      <FirstName>Jill</FirstName>
      <Title>CTO</Title>
```

Figure 62. XML output with 'telephoneNo' Attribute

So far, so good. It's now time to try using *Lookup* mode to do the join.

Using Lookup Mode

Rather than modify and potentially mess up your running AssemblyLine, you will make a copy of it and then change the copy instead. Do this by right-clicking on the 'CSV2XML.assemblyline' and selecting **Copy**.

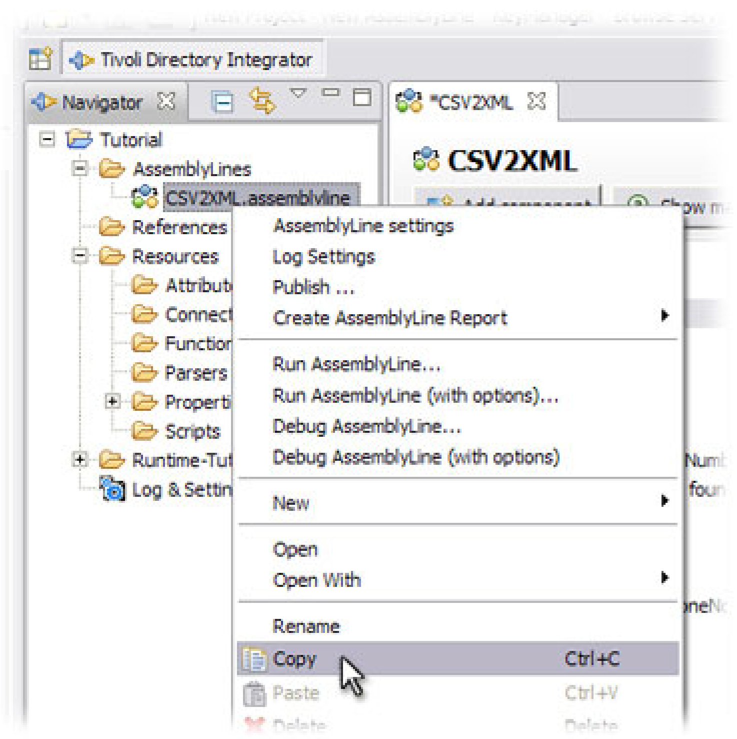


Figure 63. AssemblyLine Copy function

Now right-click on the 'AssemblyLines' folder in the Navigator and select the **Paste** option. Call this new AssemblyLine 'CSV2XML_LookupMode.assemblyline' and then double-click on it to open the AL editor.

You can now remove the ConnectorLoop along with all components under it. Just select it and press the **Delete** key. In its place you will be dragging in a JDBC Connector that you will copy from another AssemblyLine.

But first you must build the database table that this Connector will be reading from; or rather, you must run a pre-built AssemblyLine to build it for you. To do this, use a file browser to navigate to the 'Tutorials' folder and locate the file called `CreatePhoneDB.assemblyline`. Drag it into the CE window on top of the 'AssemblyLines' folder in the Navigator panel.

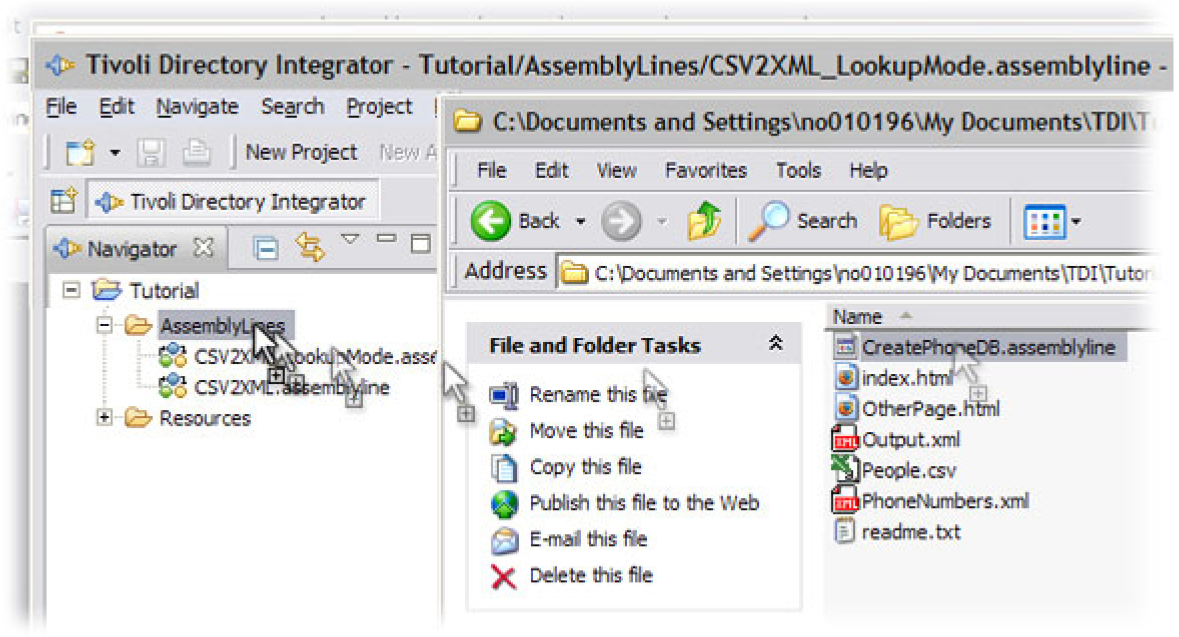

Tivoli Directory Integrator - Tutorial/AssemblyLines/CSV2XML_LookupMode.assemblyline -
File Edit Navigate Search Project
New Project
Tivoli Directory Integrator
Navigator
Tutorial
AssemblyLines
CSV2XML_LookupMode.asse
CSV2XML.assemblyline
Resources
C:\Documents and Settings\no010196\My Documents\TDI\T
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Documents and Settings\no010196\My Documents\TDI\Tutor
File and Folder Tasks
Rename this file
Move this file
Copy this file
Publish this file to the Web
E-mail this file
Delete this file
Name
CreatePhoneDB.assemblyline
index.html
OtherPage.html
Output.xml
People.csv
PhoneNumbers.xml
readme.txt

Figure 64. Copying an AssemblyLine into your project

The AssemblyLine will be imported into your project. Now right-click on it and choose **Run AssemblyLine...**

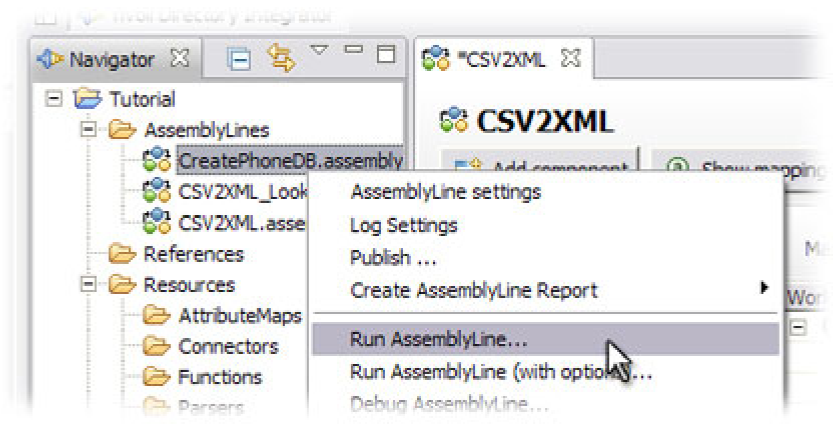


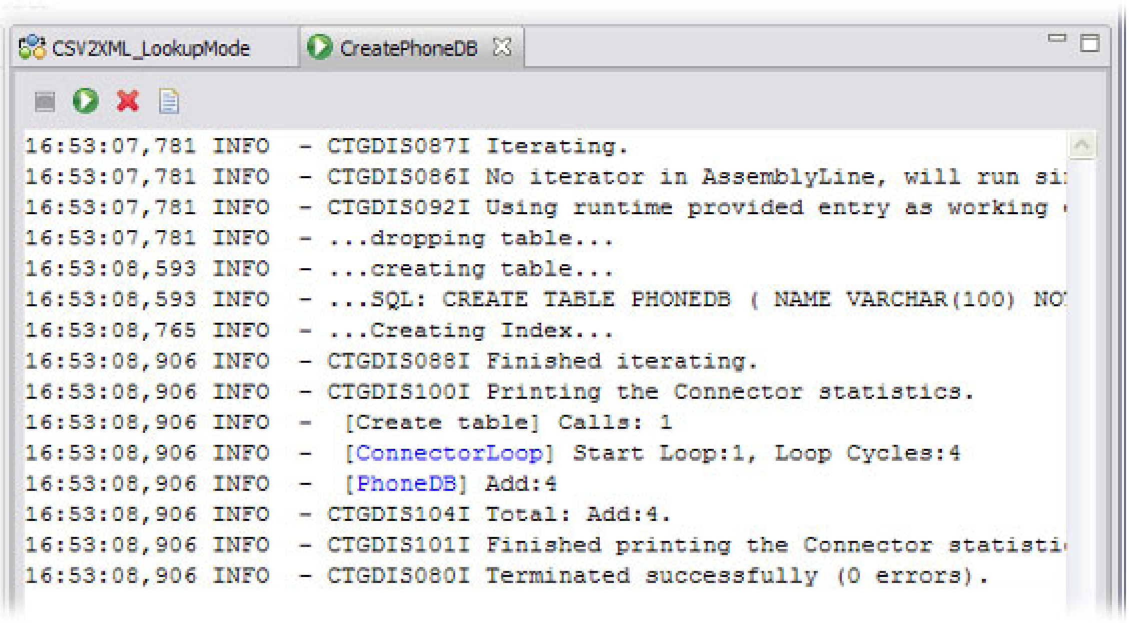
Figure 65. Run the CreatePhoneDB AL

This AssemblyLine will first create a Derby[29] database under your solution directory called 'TutorialDB' and then set up a 'PhoneDB' table. It will then loop through `PhoneNumbers.xml` and load this information into the new table[30].

If all goes well then the log output should look like this:

29. Derby is an open-source relational database, bundled with Tivoli Directory Integrator.

30. Although the AL will not be covered in this guide, it is a good example of advanced scripting techniques used to exploit the data source-specific functionality found in most Connector Interfaces. Feel free to examine and play with this AssemblyLine.



```
16:53:07,781 INFO  - CTGDIS087I Iterating.
16:53:07,781 INFO  - CTGDIS086I No iterator in AssemblyLine, will run si
16:53:07,781 INFO  - CTGDIS092I Using runtime provided entry as working
16:53:07,781 INFO  - ...dropping table...
16:53:08,593 INFO  - ...creating table...
16:53:08,593 INFO  - ...SQL: CREATE TABLE PHONEDB ( NAME VARCHAR(100) NO
16:53:08,765 INFO  - ...Creating Index...
16:53:08,906 INFO  - CTGDIS088I Finished iterating.
16:53:08,906 INFO  - CTGDIS100I Printing the Connector statistics.
16:53:08,906 INFO  -  [Create table] Calls: 1
16:53:08,906 INFO  -  [ConnectorLoop] Start Loop:1, Loop Cycles:4
16:53:08,906 INFO  -  [PhoneDB] Add:4
16:53:08,906 INFO  - CTGDIS104I Total: Add:4.
16:53:08,906 INFO  - CTGDIS101I Finished printing the Connector statisti
16:53:08,906 INFO  - CTGDIS080I Terminated successfully (0 errors).
```

Figure 66. Log output from the 'CreatePhoneDB' AssemblyLine

The 'CreatePhoneDB' AssemblyLine has a JDBC Connector that is already configured and ready to use. You are going to copy this component to your Project Resource library (specifically, to the "Connectors" folder) and then reuse it in your AL.

Open up the 'CreatePhoneDB' AssemblyLine, grab the Connector called 'PhoneDB' and drag it to the 'Connectors' folder located under 'Resources' in the Navigator tree.

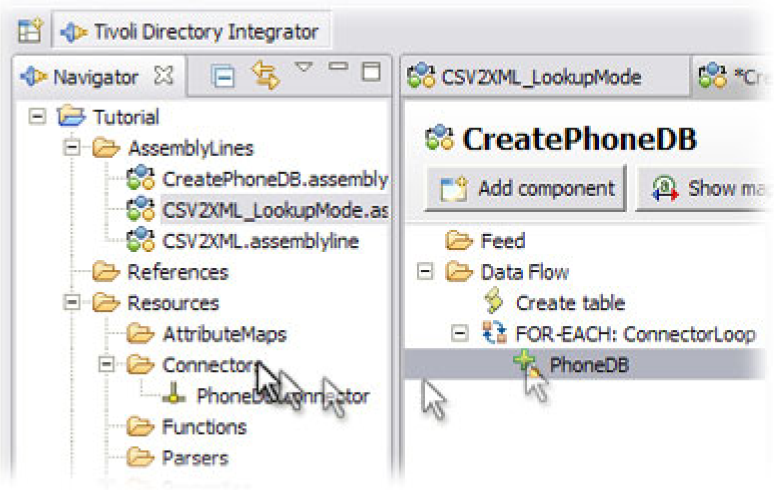


Figure 67. Drag a Connector to Resources

Close 'CreatePhoneDB' so that your own AssemblyLine is visible again. Then drag the new 'PhoneDB.connector' resource into the spot previously occupied by the ConnectorLoop.

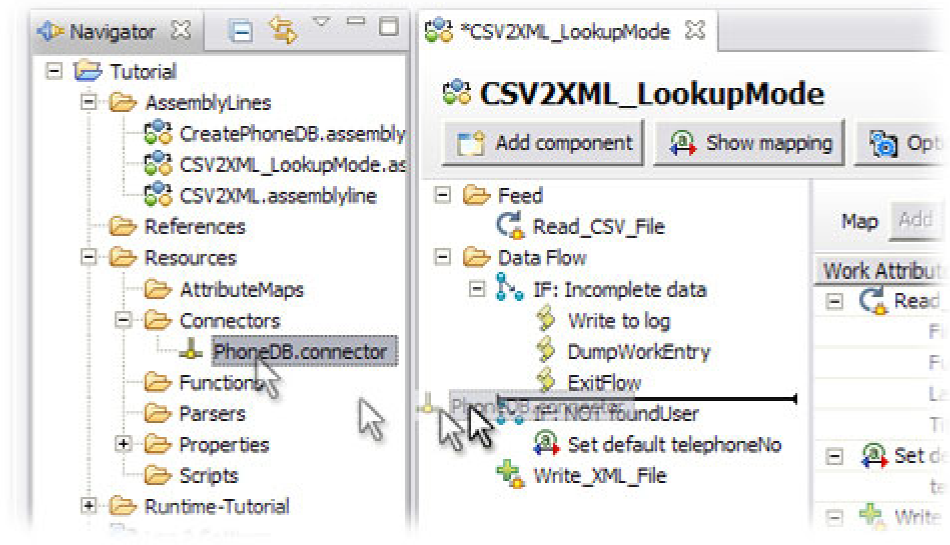


Figure 68. Drag the new resource into your AssemblyLine

Inheritance

Did you notice how this Connector shows up blue in your AssemblyLine? This is because it is now inheriting from your resource library. This means that it will dynamically retrieve configuration settings at run-time from the Connector you dragged. This inheritance feature makes it easy to re-use resources like configured components and scripted logic across multiple AssemblyLines.

You can change the ancestor of a component with the **Inherit From** button at the top of its editor panel. Component tabs, like Config, Delta, Input/Output Maps and Hooks also provide an inheritance option, allowing you to set a different ancestor than that of the component itself.

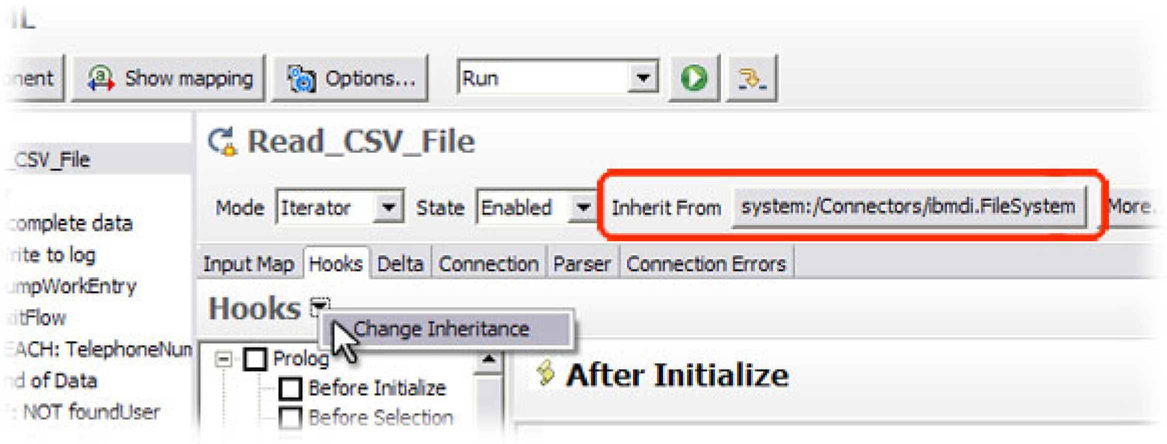


Figure 69. Setting Inheritance for the Hooks tab

Inherited values are displayed in **blue** type, and if you change it then inheritance is broken. Inheritance is restored by using the **Revert to inherited value** option in the Context menus of Attribute Map rules and Hooks.

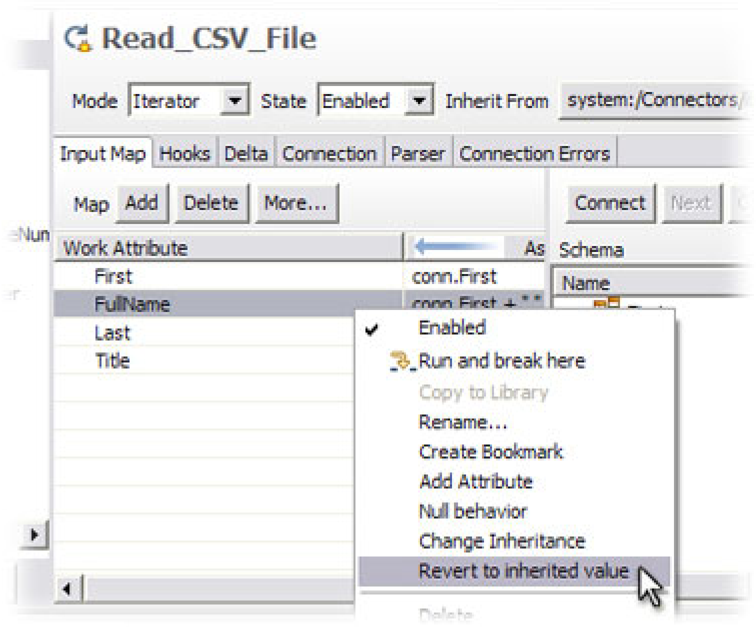


Figure 70. Restoring inheritance for a mapping rule

Returning to the exercise again, first change the **Mode** setting of the new "PhoneDB" Connector now from **AddOnly** to **Lookup**.

Secondly, since the Attribute Map was originally an *Output Map* associated with the previous mode, you will have to discover the input schema by pressing the **Connect** and **Next** buttons above the Connector Schema. The third and last step is to drag the 'PHONE' Attribute from Schema over to the Input Map, giving you a simple map for this value.

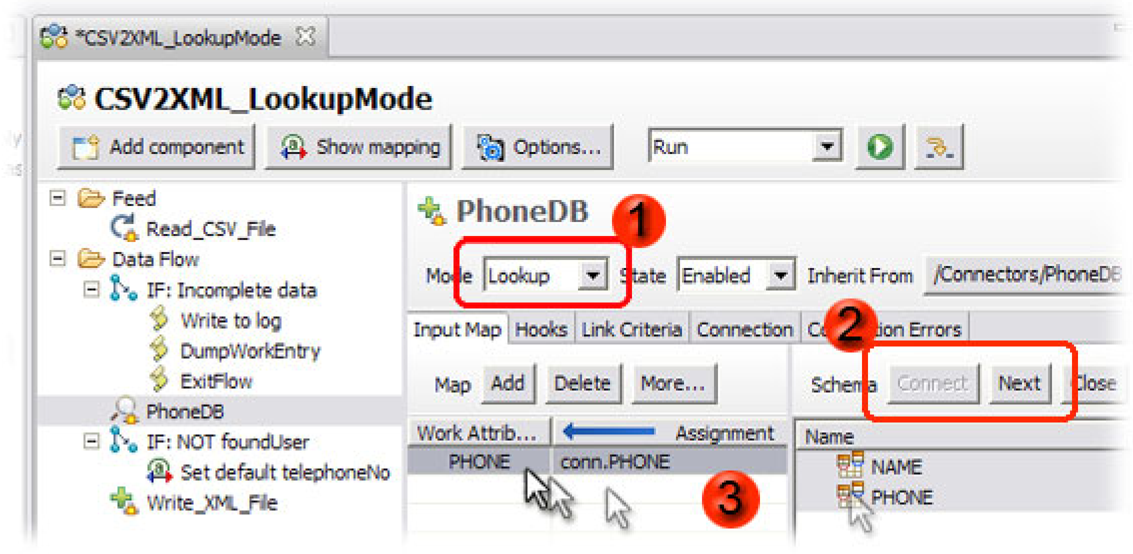


Figure 71. Changing mode, discovering and mapping Attributes

Once you have the Input Map rule in place, it's important that you rename it from 'PHONE' to 'telephoneNo' so that it fits the assignment of the Output Map of your 'Write_XML_File' Connector.

In case you were wondering, you don't have to map in 'NAME' since you already have the name of the user. This field will instead be used when defining the search rule.

Lookup search rules = Link Criteria

When you selected *Lookup* mode then a new tab labeled 'Link Criteria' appeared in the Connector editor. Link Criteria is for defining the matching rules for the search.

You can either define these as simple Link Criteria by using the drop-downs, adding new Link Criteria as needed and setting the **Match Any** checkbox as you would for Conditions. Alternatively, you can select the **Build criteria with custom script** checkbox and instead write a snippet of script that computes the actual search rule, like this example of an LDAP search filter:

```
"(cn=" + work.FullName + ")"
```

Note that this will tie your solution more closely to the data source being searched since you have to write the actual syntax expected by the connected system. In our case it would mean creating a WHERE clause (without the 'WHERE' keyword itself).

In contrast, simple Link Criteria are translated to native search syntax for you by the Connector, so you can switch the Connector Interface without having to redo your Link Criteria.

Simple Link Criteria look similar to Conditions. The first drop-down is populated with the schema you discovered and the second one shows you which Work Entry Attributes are available at this point in your AL. Again, just as with Conditions, the dollar sign is used here to indicate that the value of the named Attribute should be substituted at run-time in order to create the search filter.

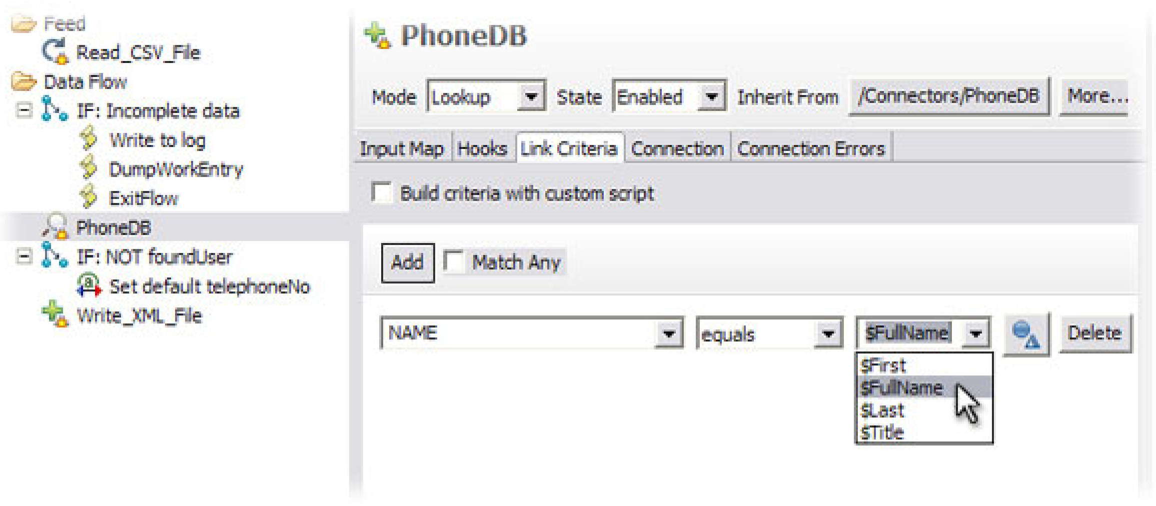


Figure 72. A simple Link Criteria

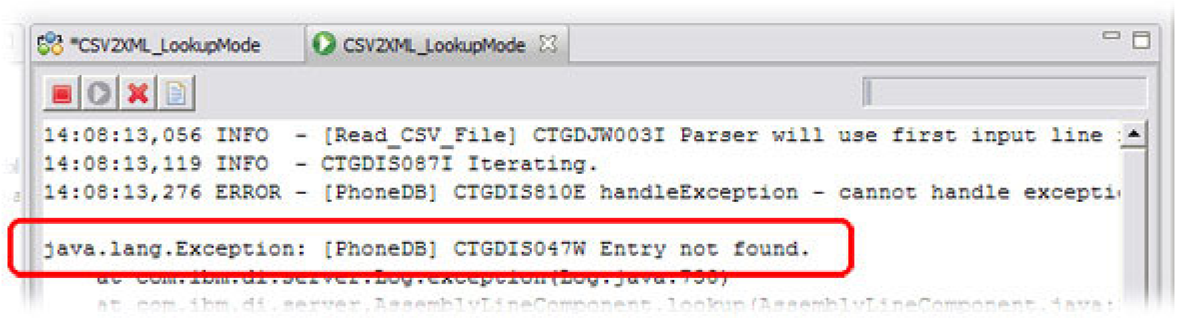
Remember to save your work, for example by pressing CTRL-S[31]. You will want to do this regularly so that you don't loose any work.

It's time to **Run** your AssemblyLine again.

31. If you have deleted AssemblyLines or resources and wish to undo this, right-click on your project in the Navigator panel and select **Restore from Local History...** which will present you with a list of asset versions to restore from. You will of course have to save something first before it shows up in your Local History.

Deciphering Run errors

Do not panic. You should have gotten an error, indicated by the stack dump appearing in the log output. Scroll to the top of the very first stack dump (there should be only one in this case). Here you will see information on both *where* the problem occurred as well as *what* caused it.



```
14:08:13,056 INFO  - [Read_CSV_File] CTGDJW003I Parser will use first input line :
14:08:13,119 INFO  - CTGDIS087I Iterating.
14:08:13,276 ERROR - [PhoneDB] CTGDIS810E handleException - cannot handle excepti

java.lang.Exception: [PhoneDB] CTGDIS047W Entry not found.
    at com.ibm.di.server.Log.exception(Log.java:750)
    at com.ibm.di.server.AssemblyLineComponent.lookup(AssemblyLineComponent.java:
```

Figure 73. Error message in log output

The component name is in brackets ('PhoneDB') followed by the error description that an Entry was not found – in other words, that the Lookup failed.

When a Connector is configured in *Lookup* mode, the system expects to find one and only one matching record when the search is performed. If none are found – or if multiple records match the Link Criteria – then you end up in special *Hooks* that must at least be enabled to prevent the AL from stopping. This behavior is clearly visible in the DataFlow diagrams that are part of the *IBM Tivoli Directory Integrator V7.0 Reference Guide*. Here is an excerpt from the page detailing *Lookup* mode:

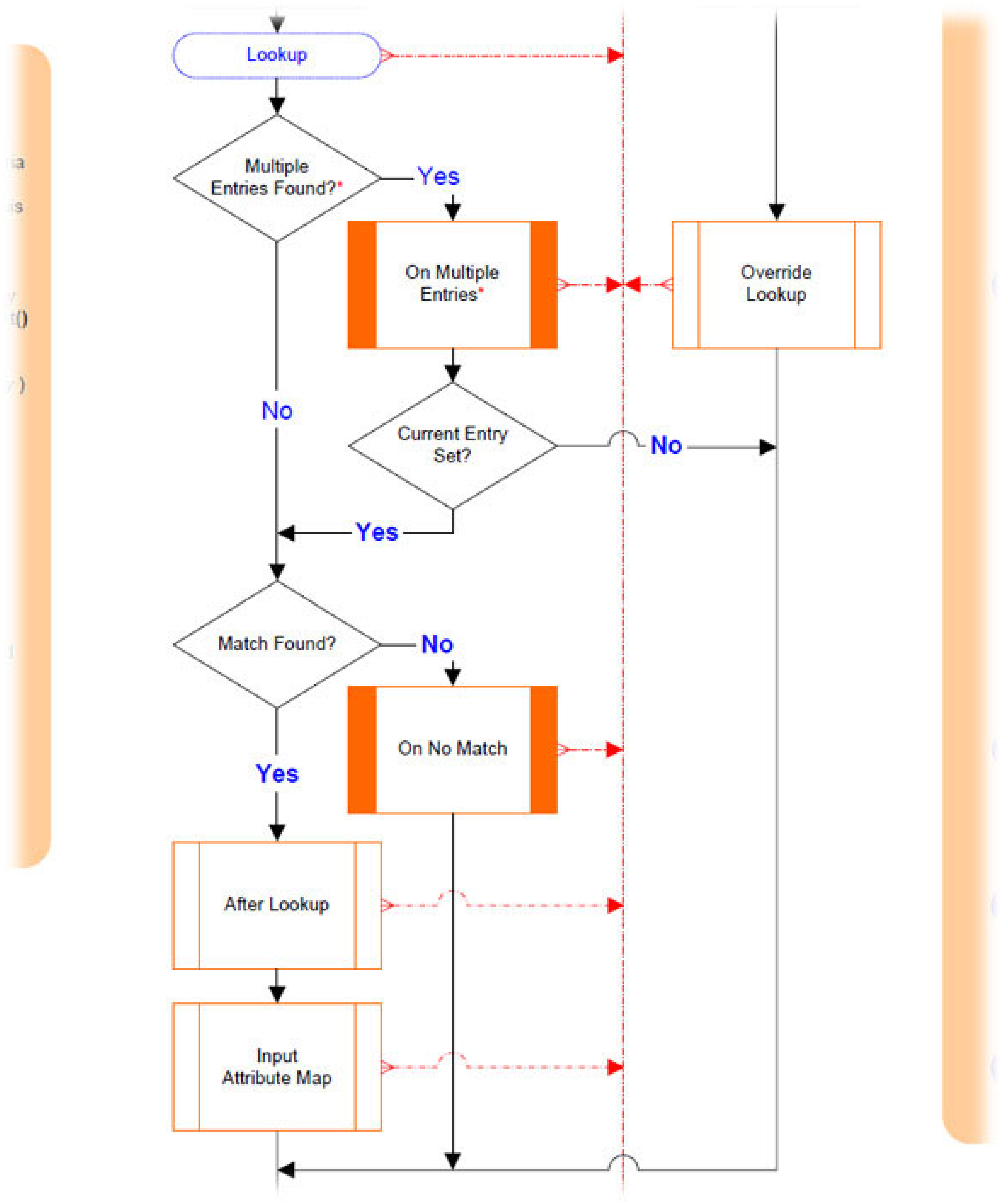


Figure 74. Partial Flow Diagram for Lookup mode

You can take advantage of this behavior to set your `userFound` flag variable. Right-click on the 'PhoneDB' Connector and choose **Hooks...** to open the Hooks editor. Select the 'On No Match' Hook and enter the script code to set `foundUser` to *false*.

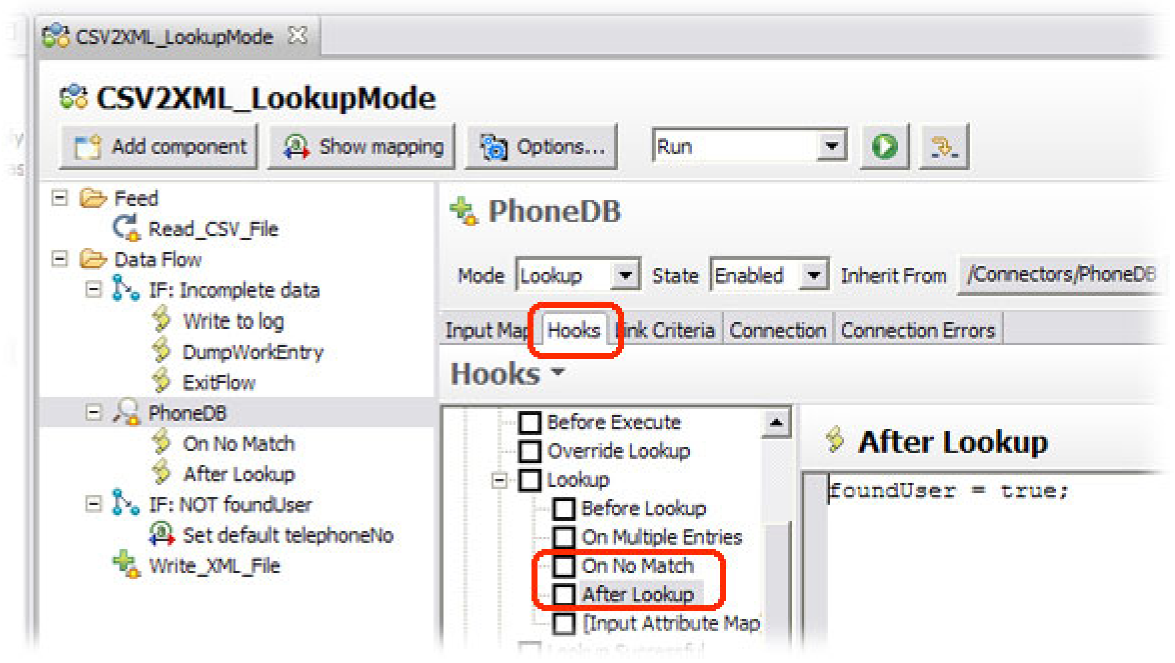
```
foundUser = false;
```

Now choose 'After Lookup' and enter this complimentary Hook script[32]:

32. As you can see from the Flow Diagram, the 'After Lookup' Hooks is only executed if the search results in a single match.

```
foundUser = true;
```

Your AssemblyLine should now resemble the one in this screenshot:



```
foundUser = true;
```

Figure 75. First tutorial exercise completed

It's time to **Run** this AssemblyLine again, and this time it should complete without errors. The `Output.xml` file should be identical to the results you got from your Loop-based AL.

Congratulations! You have just completed your first Tivoli Directory Integrator tutorial exercise. Now it's time discuss how AssemblyLines can be triggered by real-time events.

Chapter 3. Event-driven integration

So far you've been running your AssemblyLines as batch processes, manually starting them each time you want data to flow. You can also run AssemblyLines from the command line by invoking theTivoli Directory Integrator Server, like this[33]:

```
ibmdisrv -c examples/Tutorial/Tutorial1.xml -r CSVtoXML
```

In this way it's a simple task to use scheduling tools (for example, *crontab*) to schedule their operation, or to easily launch them from external applications.

Tivoli Directory Integrator provides a number of features for making your AssemblyLines event-aware, allowing your solutions to handle and respond to a wide variety of real-time triggers.

Examples of these triggers are:
- protocol requests coming over an IP port, like REST calls, SNMP and web services;
- new messages appearing on a queue;
- emails arriving in an Inbox;
- changes to data, for example in files, databases, directories, and Notes databases;
- schedule or timer-based AssemblyLine operations.

This is not a complete list, and you will find both inspiration and guidance in other Tivoli Directory Integrator literature, in the community websites and the newsgroups.

Handling these events in your solution can be done in a number of ways:

Connectors in Iterator Mode

Some Connectors allow you to configure timeout parameters for Iterator Mode. One example is the FileSystem Connector, which can be set up to read through a file to the end and then wait for new information to appear - so-called 'tail read'.

Other Connectors, like those for RDBMS Change detection and LDAP Changelog, work in a similar way. These Connectors allow you to build AssemblyLines that run continuously, waiting for new changes to appear in the connected system.

There is also a Timer Connector that runs in Iterator Mode and can be configured to drive your AssemblyLine at timed intervals according to a scheduling parameter. You will be testing this one shortly.

Tivoli Directory Integrator includes a Web Administration console as part of the standard installation. This browser-based application called the Administration and Monitoring Console (AMC) lets you monitor the health of your AssemblyLines, hot-load Configs to running Servers, start and stop ALs, and configure failure/response behavior to keep your integration solutions highly available. It can also be used to set up schedules for when your AssemblyLines should run. However, the Web Admin tool is beyond the scope of this Guide.

Connectors in Server Mode

A few specialized Connectors, like the HTTP Server Connector and the LDAP Server Connector, allow you to build solutions that process incoming requests from external clients, perform requested actions and reply with appropriate responses. You will be using the HTTP Server Connector in the next exercise.

33. The Tivoli Directory Integrator Server provides a usage message when invoked with no commandline arguments: `ibmdisrv`

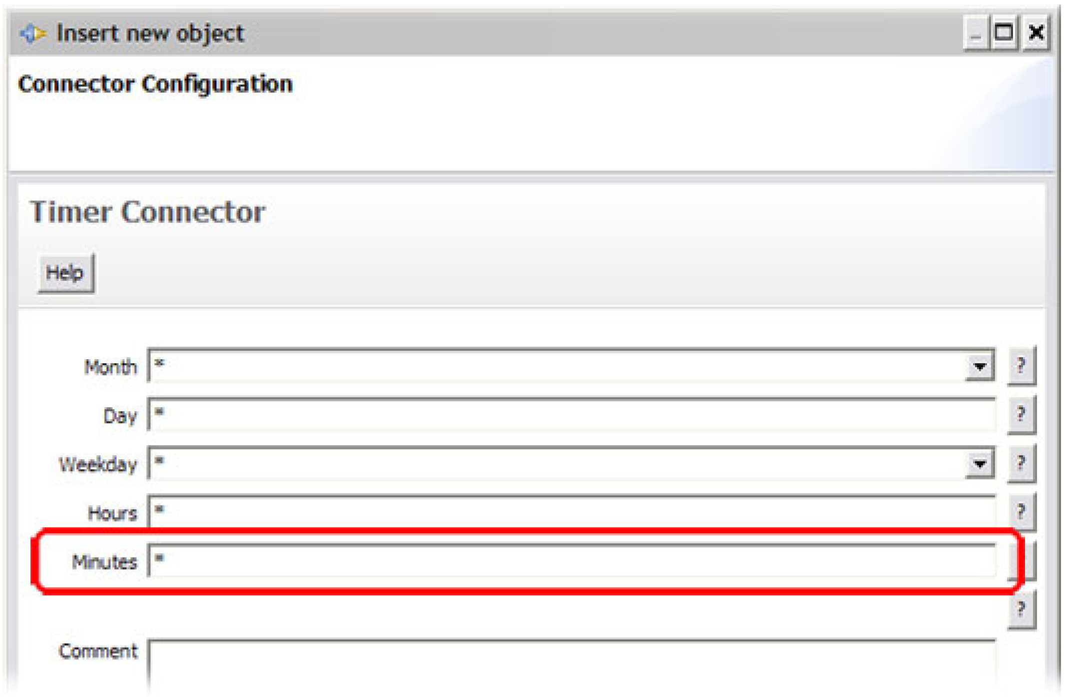
Notifications and properties

Tivoli Directory Integrator has components that can subscribe to Tivoli Directory Integrator notification events, just as there are components (and script calls) for sending these events – even between distinct Tivoli Directory Integrator Servers running on different platforms.

This guide will take you on a closer look at AL scheduling and at Connector Server mode.

Scheduling AssemblyLines

Start by creating a new AssemblyLine that you call 'Scheduler'. Then add a 'Timer Connector' to the *Feed* section and configure it to run every minute by entering the wildcard character (*) in the Minutes parameter.



Month	*
Day	*
Weekday	*
Hours	*
Minutes	*
Comment	

Figure 76. Configure the Timer Connector

The Timer Connector offers a 'Timestamp' Attribute for Input Mapping, although it is not necessary in this exercise.

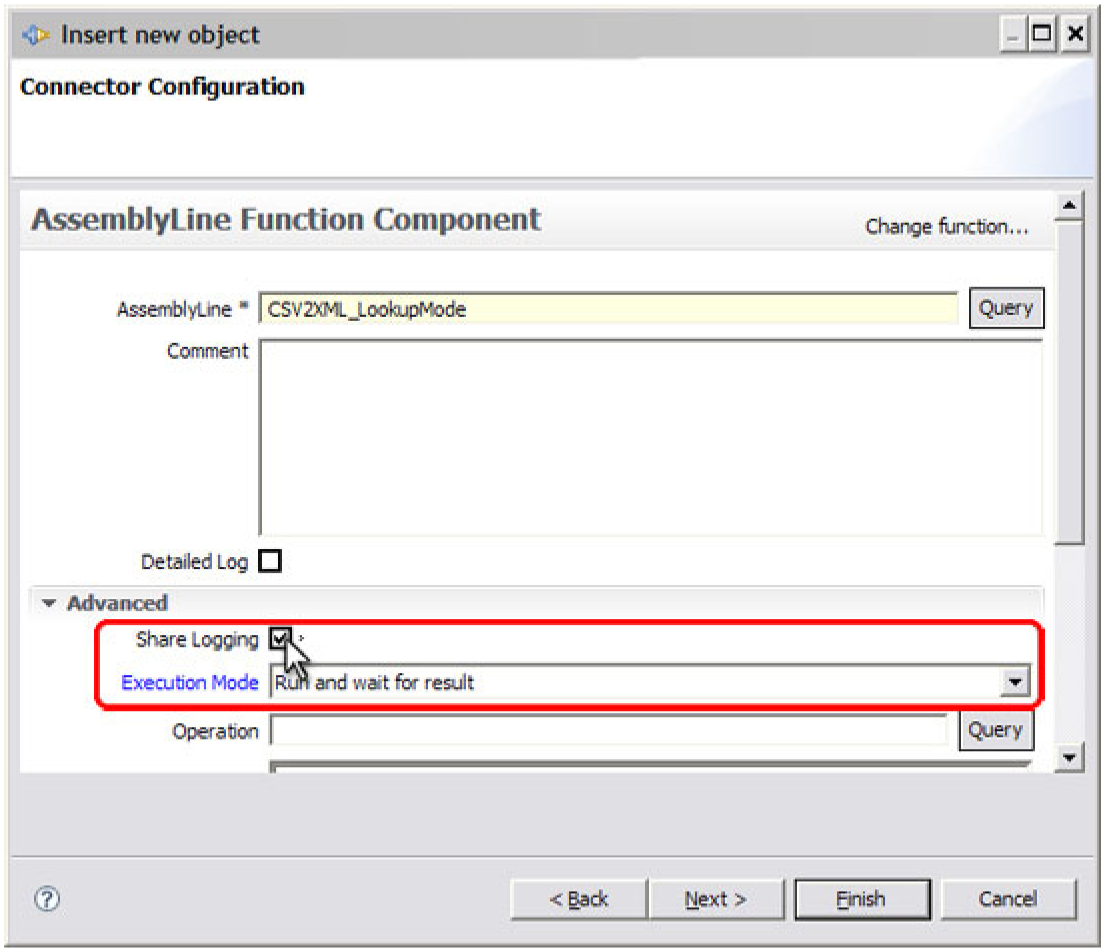
Now add a new component to the Data Flow section, this time selecting the 'AssemblyLine Function Component'[34] (AL FC). The AssemblyLine Function Component lets you invoke another AL as though it were a service call, optionally passing in Attributes via the Output Map of the FC, and mapping in whatever happens to be in the Work Entry when the called AL completes. You will not need to do any mapping in this example[35].

Now configure the AL FC to start your 'CSV2XML_LookupMode' AssemblyLine by using the **Query** button. In the **Advanced** section, enable the checkbox for shared log settings between the calling and called AssemblyLines, and leave the **Execution Mode** set to 'Run and wait for result'. This means that

34. A Function component (or 'FC' for short) is similar to a Connector, but has no Mode setting and always provides both Input and Output Maps. FC's are used to make service requests, using their Output Map to pass parameters to the called service and providing an Input Map for retrieving any response Attributes.

35. Note that if you map out Attributes to a called AssemblyLine then this will disable any *Feed* section Iterator Connectors for the first AL cycle. After the first cycle, however, the Iterator Connector will function as usual.

your 'Scheduler' AL will pause until the called AssemblyLine completes.


Insert new object

Connector Configuration

AssemblyLine Function Component

AssemblyLine *: CSV2XML_LookupMode

Comment:

Detailed Log: ☐

Advanced

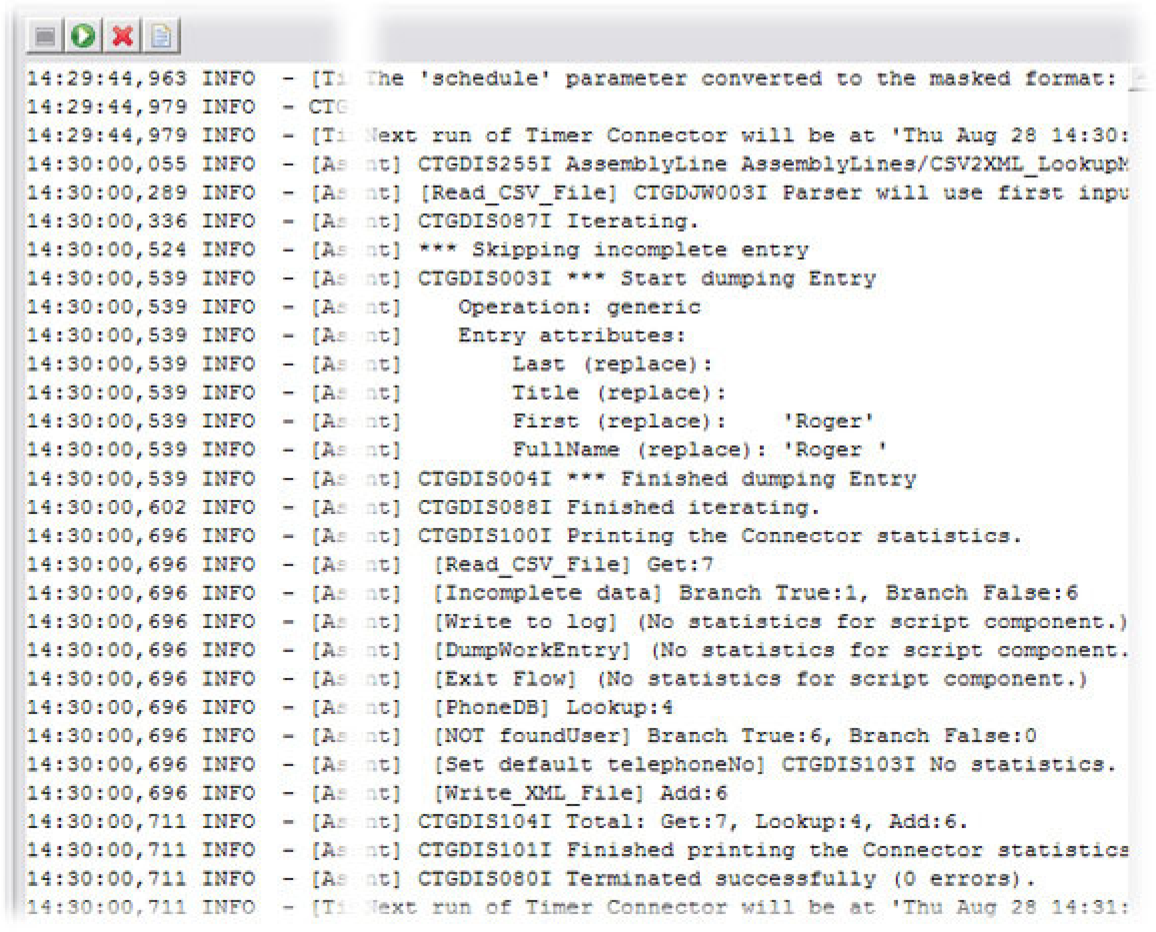
Share Logging: ☑

Execution Mode: Run and wait for result

Operation:

Figure 77. Configure the AssemblyLine Function Component

Your Scheduler AssemblyLine is ready to test. Press the **Run** button and wait for the internal clock of your machine to reach the next whole minute.



```
14:29:44,963 INFO  - [Ti The 'schedule' parameter converted to the masked format:
14:29:44,979 INFO  - CTG
14:29:44,979 INFO  - [Ti Next run of Timer Connector will be at 'Thu Aug 28 14:30:
14:30:00,055 INFO  - [As nt] CTGDIS255I AssemblyLine AssemblyLines/CSV2XML_LookupM
14:30:00,289 INFO  - [As nt] [Read_CSV_File] CTGDJW003I Parser will use first inpu
14:30:00,336 INFO  - [As nt] CTGDIS087I Iterating.
14:30:00,524 INFO  - [As nt] *** Skipping incomplete entry
14:30:00,539 INFO  - [As nt] CTGDIS003I *** Start dumping Entry
14:30:00,539 INFO  - [As nt]    Operation: generic
14:30:00,539 INFO  - [As nt]    Entry attributes:
14:30:00,539 INFO  - [As nt]        Last (replace):
14:30:00,539 INFO  - [As nt]        Title (replace):
14:30:00,539 INFO  - [As nt]        First (replace):    'Roger'
14:30:00,539 INFO  - [As nt]        FullName (replace): 'Roger '
14:30:00,539 INFO  - [As nt] CTGDIS004I *** Finished dumping Entry
14:30:00,602 INFO  - [As nt] CTGDIS088I Finished iterating.
14:30:00,696 INFO  - [As nt] CTGDIS100I Printing the Connector statistics.
14:30:00,696 INFO  - [As nt]  [Read_CSV_File] Get:7
14:30:00,696 INFO  - [As nt]  [Incomplete data] Branch True:1, Branch False:6
14:30:00,696 INFO  - [As nt]  [Write to log] (No statistics for script component.)
14:30:00,696 INFO  - [As nt]  [DumpWorkEntry] (No statistics for script component.
14:30:00,696 INFO  - [As nt]  [Exit Flow] (No statistics for script component.)
14:30:00,696 INFO  - [As nt]  [PhoneDB] Lookup:4
14:30:00,696 INFO  - [As nt]  [NOT foundUser] Branch True:6, Branch False:0
14:30:00,696 INFO  - [As nt]  [Set default telephoneNo] CTGDIS103I No statistics.
14:30:00,696 INFO  - [As nt]  [Write_XML_File] Add:6
14:30:00,711 INFO  - [As nt] CTGDIS104I Total: Get:7, Lookup:4, Add:6.
14:30:00,711 INFO  - [As nt] CTGDIS101I Finished printing the Connector statistics
14:30:00,711 INFO  - [As nt] CTGDIS080I Terminated successfully (0 errors).
14:30:00,711 INFO  - [Ti Next run of Timer Connector will be at 'Thu Aug 28 14:31:
```

Figure 78. Log output from both AssemblyLines

If all goes well then your log output should look like the screenshot above.

The Timer Connector is useful for scheduling AssemblyLine operation in order to keep two or more systems synchronized. This is particularly handy in cases where source data systems do not provide any change notification service that Tivoli Directory Integrator can hook into[36].

Of course, if you are working with an LDAP directory that provides a changelog, like Tivoli Directory Server, or with Active Directory, DB2® or some other major RDBMS, or with Lotus Domino, then Tivoli Directory Integrator has specialized Iterator mode Connectors designed to detect changes in these systems. Only these changes are driven into your AssemblyLine, along with information about what has changed and how, facilitating fast and efficient data synchronization.

Another important feature is the Delta Engine that allows any Connector in Iterator mode to automatically compute change information for any kind of input source, including text files. All these components and features are detailed in the *IBM Tivoli Directory Integrator V7.0 Reference Guide*.

36. Note that this technique is often used for Lotus Connections in order to make these deployments consistent, regardless of the type of system that Profiles data is synchronized from.

Service request AssemblyLines

In this last exercise you will be building a web server AssemblyLine to provide a very simple user interface for launching your 'CSV2XML_LookupMode' AL; In other words, an HTTP service for initiating data transfers.

Start by creating a new AL and call it 'TINA_WebServer'[37]. Then insert a new component, choosing the 'HTTP Server Connector' and pressing **Finish** to end the wizard. Now open its **Input Map** tab.

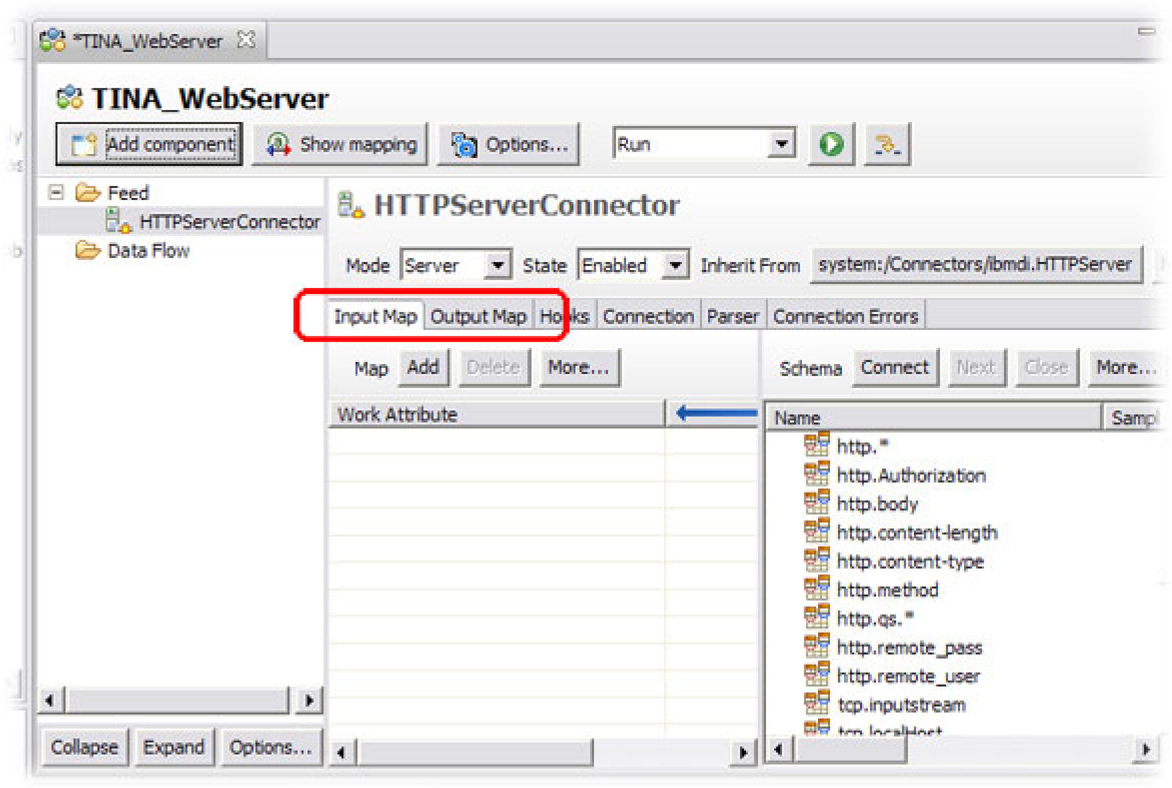


Figure 79. HTTP Server Connector Attribute Map panel

Server Connectors are complimentary to Function components. Whereas FC's make service requests, a Server Connector provides and powers a service. As a result, the Input Map for a Server Connector is used to receive Attributes coming from the client making a request, while the Output Map provides a way to reply. You can also see in the right-hand part of the Attribute Map screen that the Input Schema is already in place. The same is true for the Output Schema. Server Connectors provide this information to help you do your mapping. Note however that some of these Schema Attributes contain wildcard characters, like 'http.qs.*'. This is design-time information telling you to expect any number of incoming Attributes whose names start with 'http.qs.'[38].

Finally, the **Mode** drop-down for all Server Connectors offer both Server and Iterator modes. There is an additional mode (Response) not shown here, bringing the total to three. The Server Connector switches between these modes at various stages in its operation:

1. A Server Connector first starts in *Server* mode, connecting to some resource like an IP port and waiting for incoming client connections;

37. TINA here stands for "TINA Is Not Apache" :-)

38. This particular set of Attributes (http.qs.*) will carry any query string parameters passed into the HTTP call by the client.

2. Once a connection is made, the Connector switches to *Iterator* mode to retrieve client data based on the Input Map and passing this to the Data Flow components;
3. Finally, when the *Data Flow* components are finished executing, the Connector goes into Response mode, using the Output Map to form a reply back to the client.

You don't have to worry about this since it is handled automatically for you. However, if you do any Hook scripting then you will notice that there are three sets of Hooks: *Server*, *Iterator* and *Response*.

Continue with the tutorial exercise by adding an Input Map Attribute item.

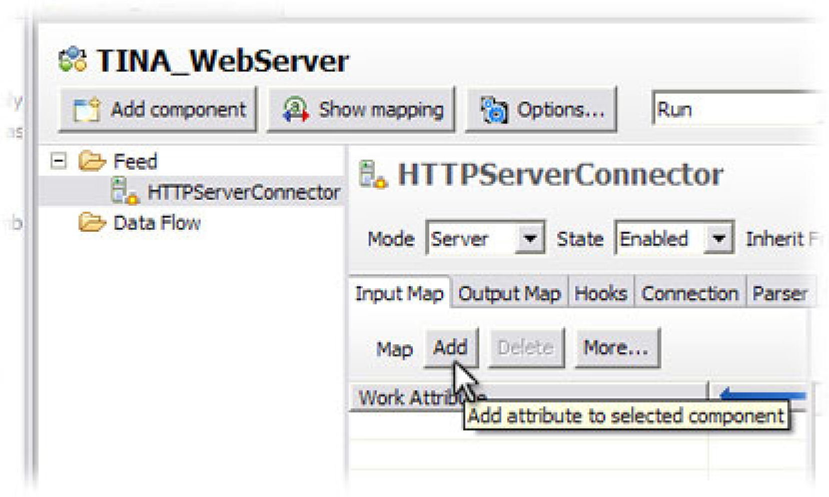
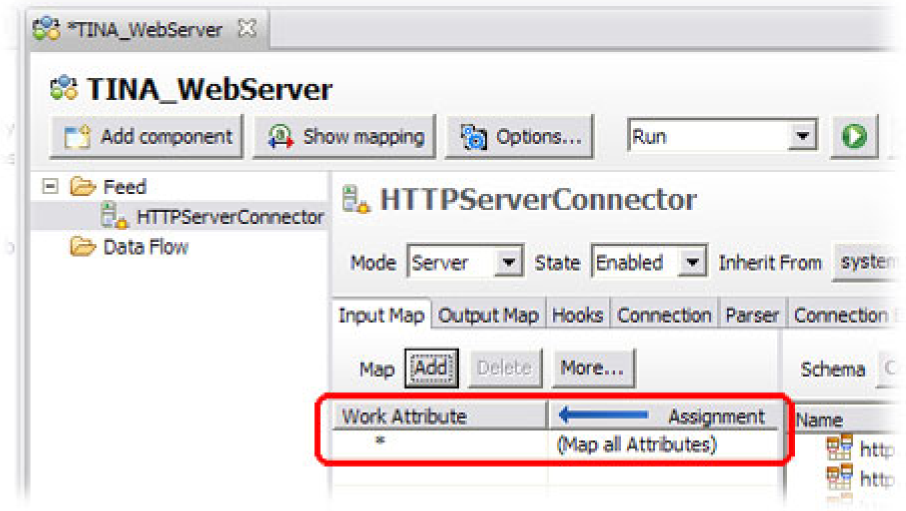


Figure 80. Add Input Attribute Map item

Call this item "*", which is the special wildcard map character that will instruct the Connector to map all of the Conn Entry Attributes into the Work Entry. Your Input Map will look like this:



Work Attribute	Assignment
*	(Map all Attributes)

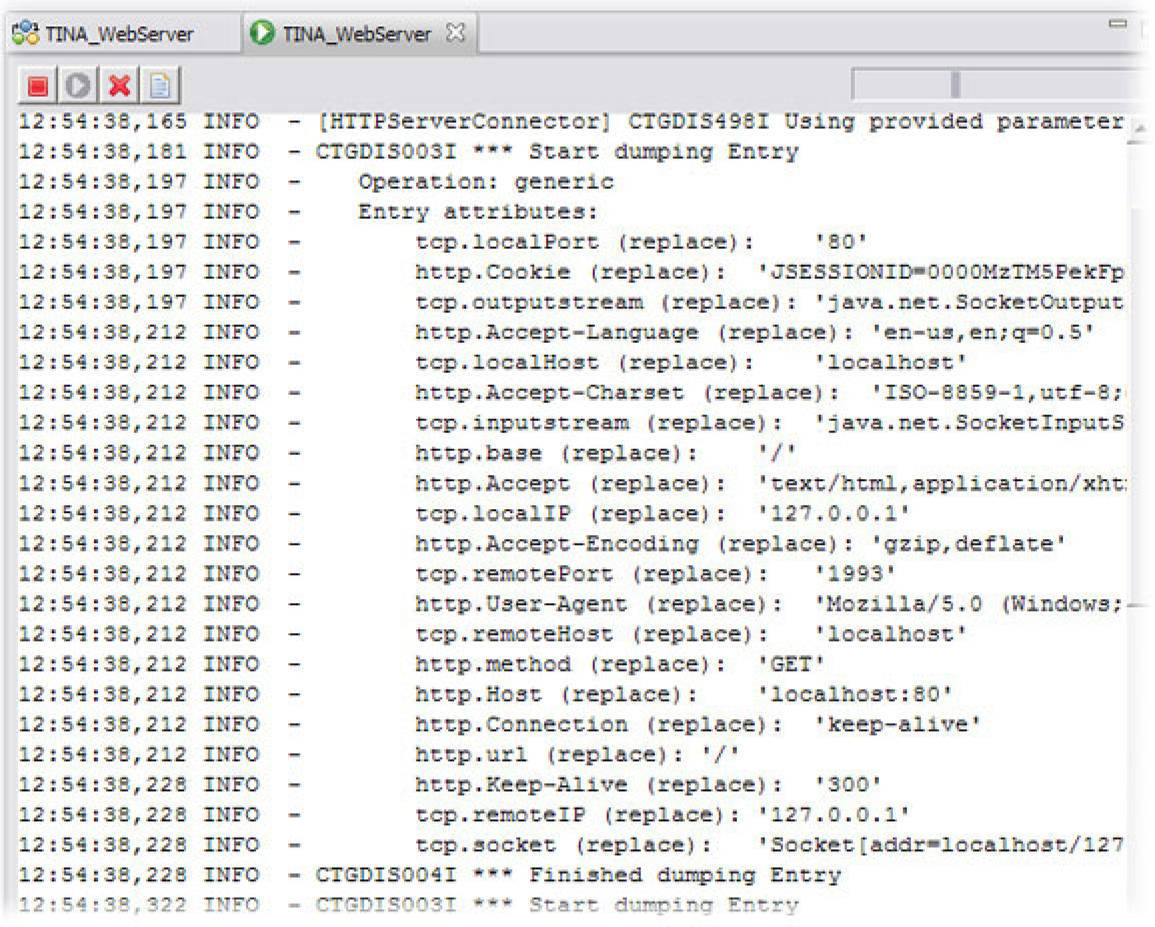
Figure 81. Wildcard map item

Add a wildcard map item to the Output map as well to ensure that any Attributes set up in the Work Entry for the response message will get mapped back to the client.

To test this component, put a 'Dump Work Entry' Script component in the Flow section and then **Run** the AssemblyLine. The log output should display the message that your HTTP Server Connector is listening for HTTP connections on port 80[39]. That means your AL is waiting for clients to connect, which you do now by opening a browser and navigating to the following URL:

```
http://localhost:80
```

Now look at your log output. Here you will see a number of TCP and HTTP header properties that were returned as Attributes.



```
12:54:38,165 INFO  - [HTTPServerConnector] CTGDIS498I Using provided parameter
12:54:38,181 INFO  - CTGDIS003I *** Start dumping Entry
12:54:38,197 INFO  -    Operation: generic
12:54:38,197 INFO  -    Entry attributes:
12:54:38,197 INFO  -        tcp.localPort (replace):    '80'
12:54:38,197 INFO  -        http.Cookie (replace):  'JSESSIONID=0000MzTM5PekFp
12:54:38,197 INFO  -        tcp.outputstream (replace): 'java.net.SocketOutput
12:54:38,212 INFO  -        http.Accept-Language (replace): 'en-us,en;q=0.5'
12:54:38,212 INFO  -        tcp.localHost (replace):    'localhost'
12:54:38,212 INFO  -        http.Accept-Charset (replace):  'ISO-8859-1,utf-8;
12:54:38,212 INFO  -        tcp.inputstream (replace):  'java.net.SocketInputS
12:54:38,212 INFO  -        http.base (replace):    '/'
12:54:38,212 INFO  -        http.Accept (replace):  'text/html,application/xht
12:54:38,212 INFO  -        tcp.localIP (replace):  '127.0.0.1'
12:54:38,212 INFO  -        http.Accept-Encoding (replace): 'gzip,deflate'
12:54:38,212 INFO  -        tcp.remotePort (replace):   '1993'
12:54:38,212 INFO  -        http.User-Agent (replace):  'Mozilla/5.0 (Windows;
12:54:38,212 INFO  -        tcp.remoteHost (replace):   'localhost'
12:54:38,212 INFO  -        http.method (replace):  'GET'
12:54:38,212 INFO  -        http.Host (replace):    'localhost:80'
12:54:38,212 INFO  -        http.Connection (replace):  'keep-alive'
12:54:38,212 INFO  -        http.url (replace): '/'
12:54:38,228 INFO  -        http.Keep-Alive (replace):  '300'
12:54:38,228 INFO  -        tcp.remoteIP (replace): '127.0.0.1'
12:54:38,228 INFO  -        tcp.socket (replace):   'Socket[addr=localhost/127
12:54:38,228 INFO  - CTGDIS004I *** Finished dumping Entry
12:54:38,322 INFO  - CTGDIS003I *** Start dumping Entry
```

Figure 82. TCP and HTTP header properties returned as Attributes

The only Attribute you will be interested in for this exercise is 'http.base' which holds that part of the URL appearing after the host and socket. Specifically, you will be looking for it to contain the text 'RunAL'.

To check for this text, add an IF branch to the Data Flow section of your AL. Call it 'RunAL detected' and set the Condition to be: http.base *contains* 'RunAL'.

If this branch Condition evaluates to *true* then you want to launch your 'CSV2XML_LookupMode' AL. To do this you will re-use the AssemblyLine Function component from your 'Scheduler' AL by first dragging it to **Resources** > **Functions** in the Navigator panel and then back into this AssemblyLine, dropping it on

39. If for some reason this port is already in use, simply open the Configuration panel for the HTTP Server Connector and choose another port.

top of the new IF branch.

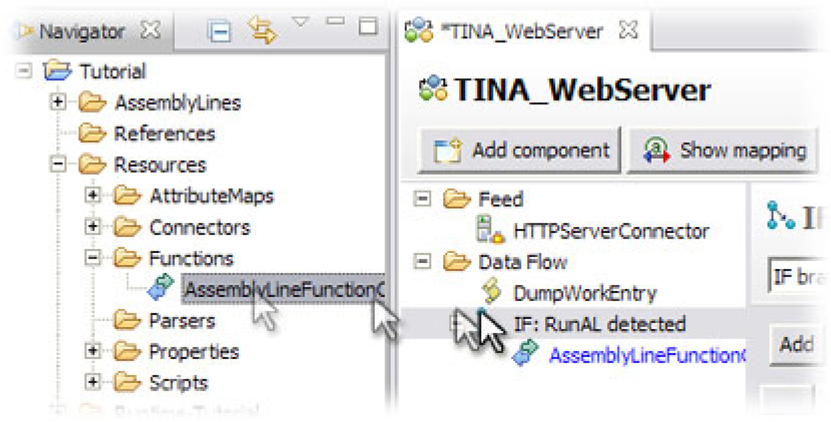
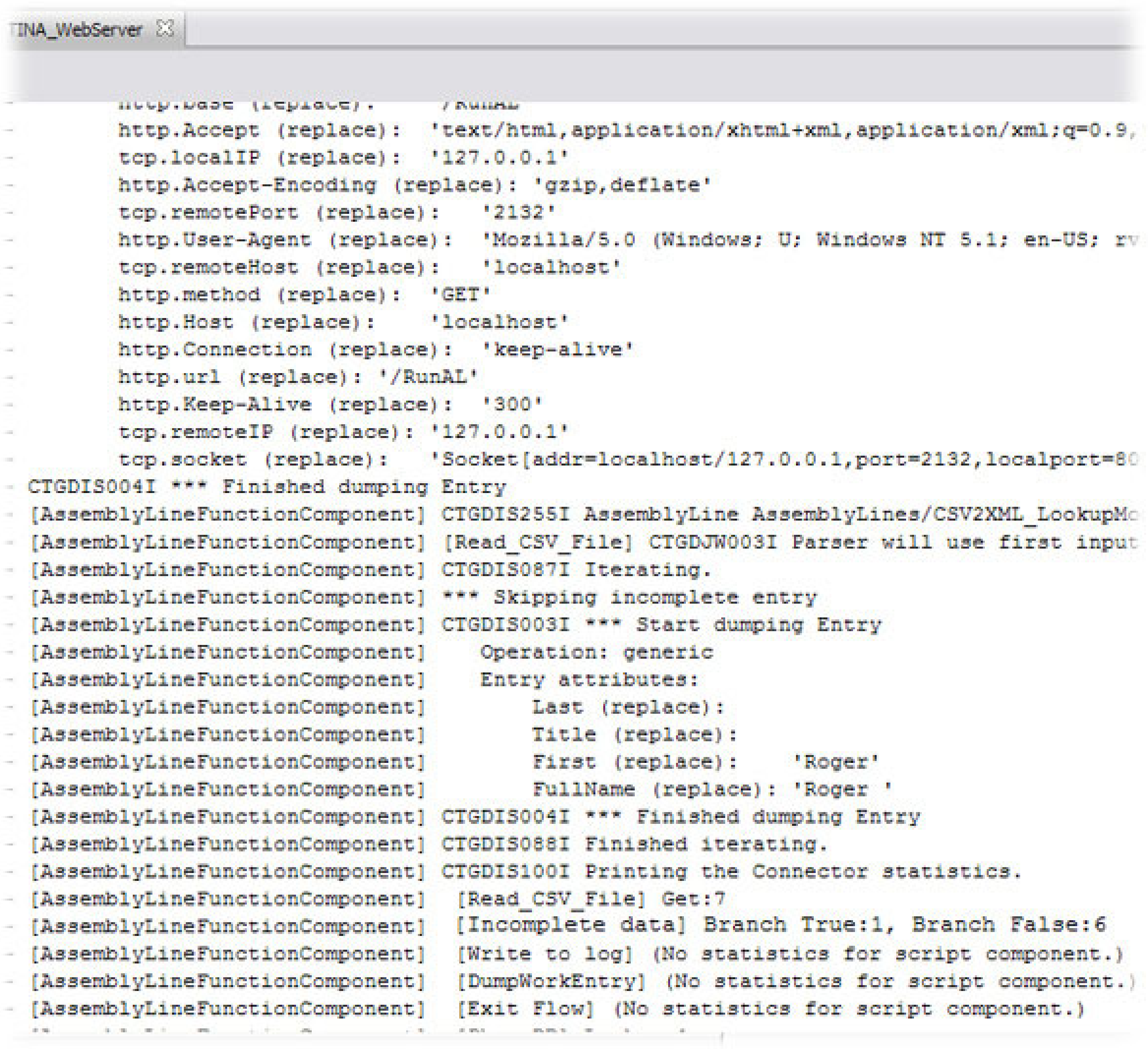


Figure 83. Drag in the AssemblyLine Function component (AL FC)

Now re-run your AssemblyLine and enter this text into the address field of your browser:

```
http://localhost/RunAL
```

You will now see the Work Entry dump in the log output, followed by the statistics from the called AssemblyLine.

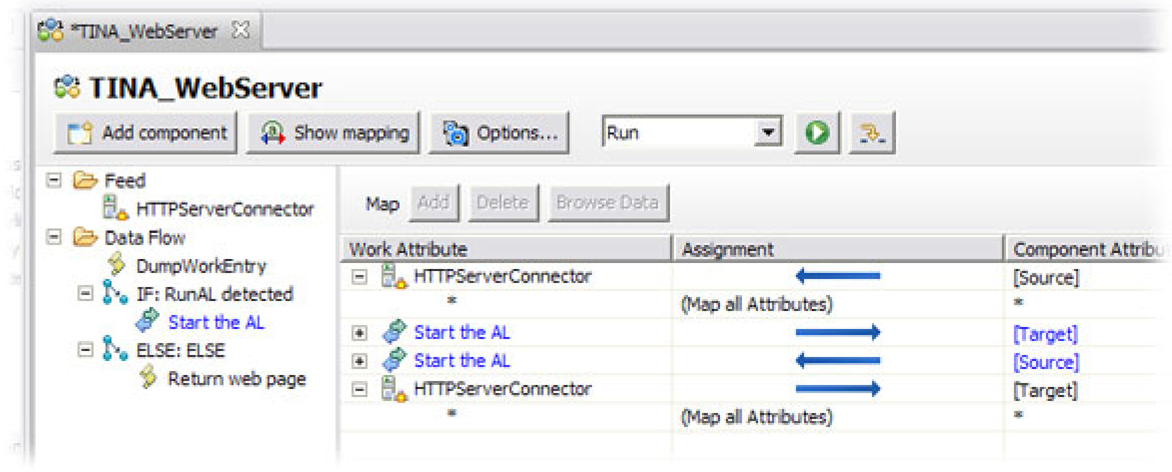


```
-        http.Accept (replace):  'text/html,application/xhtml+xml,application/xml;q=0.9,
-        tcp.localIP (replace):  '127.0.0.1'
-        http.Accept-Encoding (replace): 'gzip,deflate'
-        tcp.remotePort (replace):   '2132'
-        http.User-Agent (replace):  'Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv
-        tcp.remoteHost (replace):   'localhost'
-        http.method (replace):  'GET'
-        http.Host (replace):    'localhost'
-        http.Connection (replace):  'keep-alive'
-        http.url (replace): '/RunAL'
-        http.Keep-Alive (replace):  '300'
-        tcp.remoteIP (replace): '127.0.0.1'
-        tcp.socket (replace):   'Socket[addr=localhost/127.0.0.1,port=2132,localport=80
- CTGDIS004I *** Finished dumping Entry
- [AssemblyLineFunctionComponent] CTGDIS255I AssemblyLine AssemblyLines/CSV2XML_LookupMo
- [AssemblyLineFunctionComponent] [Read_CSV_File] CTGDJW003I Parser will use first input
- [AssemblyLineFunctionComponent] CTGDIS087I Iterating.
- [AssemblyLineFunctionComponent] *** Skipping incomplete entry
- [AssemblyLineFunctionComponent] CTGDIS003I *** Start dumping Entry
- [AssemblyLineFunctionComponent]    Operation: generic
- [AssemblyLineFunctionComponent]    Entry attributes:
- [AssemblyLineFunctionComponent]        Last (replace):
- [AssemblyLineFunctionComponent]        Title (replace):
- [AssemblyLineFunctionComponent]        First (replace):    'Roger'
- [AssemblyLineFunctionComponent]        FullName (replace): 'Roger '
- [AssemblyLineFunctionComponent] CTGDIS004I *** Finished dumping Entry
- [AssemblyLineFunctionComponent] CTGDIS088I Finished iterating.
- [AssemblyLineFunctionComponent] CTGDIS100I Printing the Connector statistics.
- [AssemblyLineFunctionComponent]  [Read_CSV_File] Get:7
- [AssemblyLineFunctionComponent]  [Incomplete data] Branch True:1, Branch False:6
- [AssemblyLineFunctionComponent]  [Write to log] (No statistics for script component.)
- [AssemblyLineFunctionComponent]  [DumpWorkEntry] (No statistics for script component.)
- [AssemblyLineFunctionComponent]  [Exit Flow] (No statistics for script component.)
```

Figure 84. Work Entry dump followed by AL statistics

Your service is working, but the exercise does not end here. First you will make it a little prettier and a lot easier to use by having your web server AssemblyLine return some HTML pages. This would normally require a fair bit of scripting. Fortunately you have been given a few more tutorial files that make this a simple case of drag-and-drop.

Before you begin, disable the 'Dump Work Entry' Script to minimize the log output. Then add an ELSE branch immediately after 'IF RunAL detected'. Name it 'Return web page'. Now use a file browser to locate the script file named `Return web page.script` in the Tutorial directory and drag it into **Resources** > **Scripts**. From there you can pull it into your AL, dropping it on top of the ELSE branch. Your AL should now look like this:

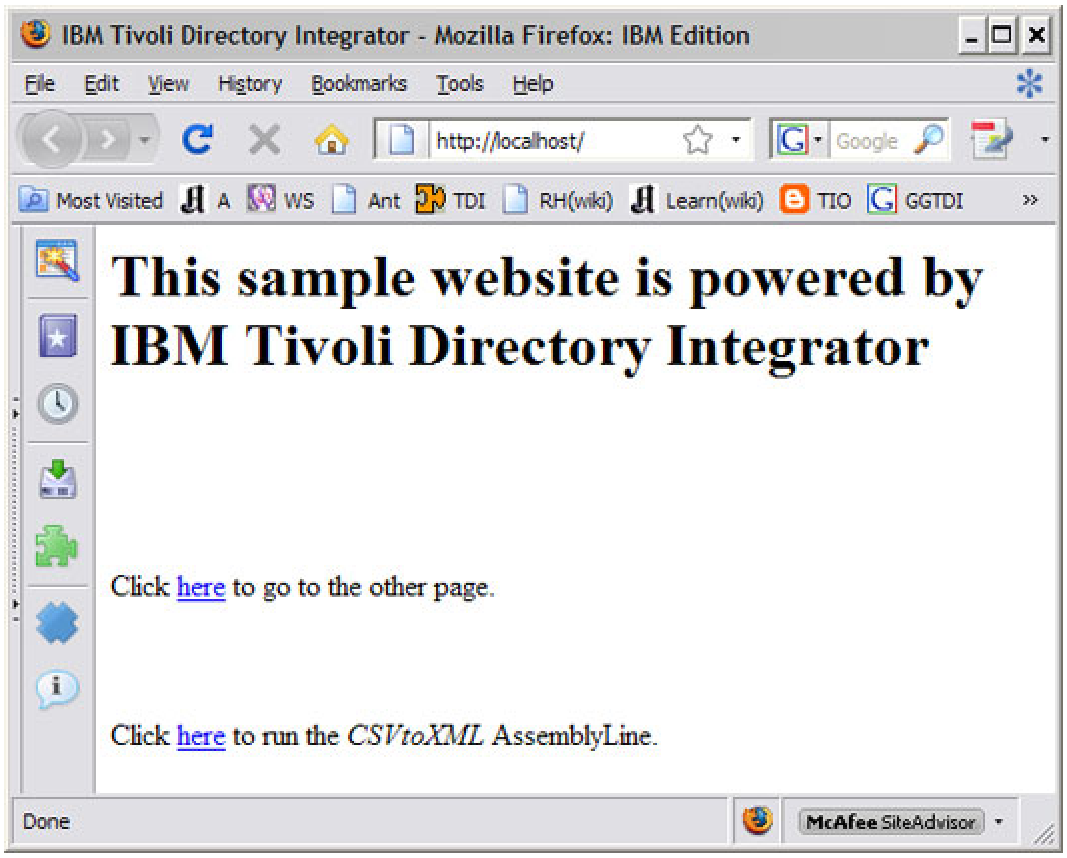


- Feed
  - HTTPServerConnector
- Data Flow
  - DumpWorkEntry
  - IF: RunAL detected
    - Start the AL
  - ELSE: ELSE
    - Return web page

Work Attribute	Assignment	Component Attribu
HTTPServerConnector	←	[Source]
*	(Map all Attributes)	*
Start the AL	→	[Target]
Start the AL	←	[Source]
HTTPServerConnector	→	[Target]
*	(Map all Attributes)	*

Figure 85. Completed TINA_WebServer AssemblyLine

Run the AssemblyLine again and when you dial up http://localhost you should see this web page:



This sample website is powered by IBM Tivoli Directory Integrator

Click here to go to the other page.

Click here to run the *CSVtoXML* AssemblyLine.

Figure 86. Simple Web Interface to your solution

The topmost link will navigate to `OtherPage.html`, while the link at the bottom should send the required 'RunAL' text in the http.base in order to launch your AssemblyLine.

And that concludes the hands-on part of this guide.

Chapter 4. Hardening your Integration Solutions

As you've seen, creating AssemblyLines can be a pretty quick business. However, a completed AL run does not mean that your solution is ready for 'prime time'. Even simple integration tasks warrant a minimum of both forethought and outcome analysis.

Some pertinent questions to reflect on are:
- Is all source data being processed as expected? How can you confirm this?
- Are anomalies in data content and/or quality detected? Are they handled?
- Does processing affect other data sets, systems or ALs? How?
- Does the integration carry an audit burden?
- Who will be deploying your solution? Who will be using it, and who will administrate it, as well as how?

For long-running AssemblyLines, like those used for synchronizations and to power services, you can tack on additional considerations like availability and fault-tolerance, performance, scalability and security.

The goal of this last chapter is making you aware of these issues, plus a number of Tivoli Directory Integrator features and techniques that you can use to address them.

Note: If you are new to Tivoli Directory Integrator then don't worry if this chapter seems complex and difficult to follow. Instead, come back and re-read these sections as your experience and comfort with the system grows.

Legibility, re-use and configurability

All development work requires troubleshooting, maintenance and extension. Tivoli Directory Integrator solutions are no exception.

You can facilitate this by following a few basic guidelines.
1. Write your AssemblyLines, keeping in mind that others need to understand, use and maintain them. That means to keep ALs as short as possible and naming components clearly and descriptively. Logic implemented in the AL flow through Branches and Loops will be simpler for non-programmers to read and debug than Script 'hidden' in Hooks or packed into Script components.
2. A corollary to the *Short AL* rule is to keep script snippets short as well. Instead of writing monolithic blocks of code, divide these into smaller units, even putting these into separate Script components to enhance legibility and debug-ability. It is possible then to disable an SC in order to skip code.

   Another way to improve legibility and avoid code duplication is by using Script component inheritance (from the 'Scripts' folder in the Navigator tree-view) and by defining functions for common tasks. An AssemblyLine executes in the context of its own Script engine, so all variables and functions declared in one place are available throughout. A common place to define these is in the AL Prolog Hooks, or in Scripts that have been selected as "Additional Prologs"[40].
3. Choose legibility over elegance when it comes to your algorithms, keeping in mind that when you pick up your own work six months from now, it will probably feel like somebody else's. Consideration for colleagues will be kindness to self as well.

40. Additional Prologs are executed before any of the AssemblyLine's own Prolog Hooks are invoked. These are selected in the AssemblyLine Settings panel which can be accessed by right-clicking on an AL and selecting **AssemblyLine Settings...**

4. Be aware that people with no Configuration Editor skills may need to modify settings and run your AssemblyLines. ALs can be easily started from the command line:

   ```
   ibmdisrv –c myConfig.xml –r myAssemblyLine
   ```

   This means that you can prepare scripts or batch-files to facilitate this.
5. Make your ALs simpler to reconfigure by externalizing parameter settings by using Properties. Properties are key-value pairs that can be stored in files or databases, and will allow your solution to be reconfigured from outside the Config Editor. Properties are tied to component parameters by clicking on the parameter label and pressing **Add property**.

   Properties can also be queried and modified from your scripts with the `system.getTDIProperty()` and `system.setTDIProperty()` calls, allowing you to make custom logic easily switchable through external property settings as well.

   Properties can furthermore be changed in a running Server by using the command-line utility, `bin/tdisrvctl`, which also lets you start and stop AssemblyLines, query status and (re)load Configs – all without stopping the Server.
6. As mentioned before, using relative paths for files makes it easier to move your solution to a new installation. It is recommended that you make your paths relative to the directory where the Config XML file is loaded from. This is accessible via the `{config.$directory}` property, which can then be used to specific path parameters using the **Text w/substitution** option; for example:

   ```
      {config.$directory}/html
   ```
7. As mentioned before, but especially when building solutions that will be deployed and run by others, do not anticipate that these users have TDI skills. Provide batch-files/scripts to start your AssemblyLines, including test and validation ALs. These could, for example, simply connect to data sources and report back success or failure. Note that in order to print messages to the console commandline so that your batch-files/scripts return status info, use the Server method, `main.logmsg()`, instead of the AL version that you used in the tutorial exercises: `task.logmsg()`. This latter call will only send your message to the log.

These are just a few pointers. More can be found in other Tivoli Directory Integrator literature and in the newsgroups.

Logging and auditing

Tivoli Directory Integrator uses *log4j* to provide flexible log management. You can choose between a number of standard *Appenders* including for Unix syslog, Windows eventlog, daily files and rolling logs. New Appenders can be created or downloaded and used as well.

By default, only minimal Server logging is enabled. At the very minimum, you should define logging for your AssemblyLines using the FileRoller Appender, writing to the `'logs'` sub-folder of your Solution Directory and naming the log-file the same as the AssemblyLine. So for your 'CSV2XML' AL you would define a set of rolling log-files based on this filepath: `logs/CSV2XML.log`.

Note that the `logmsg()` method lets you optionally define the log level for your message by passing one of the following keywords as the first argument just prior to your log message: DEBUG, INFO, WARN, ERROR, FATAL. Log levels are inclusive, so WARN will include ERROR and FATAL, and DEBUG means that messages of all levels are logged. For example, a message like:

```
task.logmsg("DEBUG", "Updated: " + conn);
```

will only be issued by Appenders set for DEBUG level logging.

You can add audit messages to your solution that can be turned on and off from outside a running Server by prefixing calls to `task.logmsg()` calls with an IF-statement that checks the value of a property. For example, this script snippet might appear in a 'DataFlow - Update Successful' Hook:

```
if (system.getTDIProperty("MyProps","audit").equalsIgnoreCase("true"))
     task.logmsg("DEBUG", "Updated the following data: " + conn);
```

By using the `tdisrvctl` command-line utility to change the value of the "audit" property in the "MyProps" Property Store, you can dynamically turn this type of audit message on or off for a running Server.

In general, it’s better to log too much information than too little. Although you ought not to flood the log output either. It can be difficult to locate messages of interest in cluttered log output.

Connectivity problems

Connectivity problems can generally be divided into two categories: initialization errors and lost connections.

By default, all components fire up connections at the very start of AL operation during its initialization phase. If for some reason a connection fails at this point then the 'Prolog – On Connection Error' Hook is invoked, giving you a script container from which to send alerts or even change parameter settings and attempt to connect again. Similarly, if connection errors occur during AL cycling then you use the 'DataFlow – On Connection Lost' Hook to handle this situation.

In addition to this custom handling, Connectors and Function components also provide built-in *reconnect* functionality via a **Connection Errors** tab. Here you can instruct the component to attempt to re-establish a lost connection, or in the case of an initialization problem, to continue trying to set up the connection.

In the case of initialization errors, it is not common practice to enable reconnect unless you are experiencing recurring issues like sporadic timeouts during SSL negotiation, or similar sporadic connection problems.

It is however recommended to enable **Auto Reconnect on Connection Loss**, allowing your component to re-establish its connection and then continue as though nothing had happened. Be aware that in the case of *Iterator* mode Connectors, reconnecting will also mean that the iteration 'cursor' may also be reset, such that cycling begins again at the first entry in the result set. Of course, for state-aware Iterators, like Change Detection Connectors, this is not a problem since these components automatically use state information to resume where they left off.

AssemblyLine availability

Improving AL availability means two things: 1) doing what you can to ensure that your AssemblyLines do not stop, and 2) restarting ALs that have stopped as quickly as possible. For scenarios like long-running migrations and synchronizations, restarted AssemblyLines may also need to continue where they left off at the point of failure.

An AssemblyLine will stop if an unhandled exception occurs. You can safeguard against this eventuality by handling all errors, which in Tivoli Directory Integrator terms amounts to at least enabling the 'Default On Error' Hook of each Connector and Function component. By enabling Error Hooks you are instructing the Server to continue in spite of an exception.

As you saw during the tutorial exercises, if an AssemblyLine stops due to an error then you get a stack trace which is preceded with info on where the error occurred and why. If you prevent the AL from stopping by enabling Error Hooks then it is your responsibility to report error status by using special objects available to your scripts.

```
task.logmsg("ERROR", "[" + thisConnector.getName() + "] - " +
            error);
```

The above example script makes use of two such objects: thisConnector which always references the component to which the executing script is tied, and the pre-defined variable called error. The error object is an Entry, just like work and conn, and it holds Attributes like 'status', 'connectorname'[41] and 'message', plus other relevant details about any recent error situation. Since it's an Entry object, you can use task.dumpEntry(error) to display its contents, as well as direct references to Attribute names – for example: work.message – or just simply appending error to a string message like in the above example since all Entry objects can convert themselves to string representations as required.

To handle exceptions occurring in scripts, like in Attribute Map assignment, Hooks and Script components, wrap your code in try-catch blocks. This allows you to catch exceptions and deal with them yourself:

```
try {
 res = myLib.callToSomeFunctionThatMightFail();
} catch (excptn) {
 task.logmsg("Call failed with error: " + excptn);
}
```

In addition to handling errors, you will want to use the Auto Reconnect feature described in the previous section. This will prevent your AssemblyLine from failing due to transient connectivity problems, like being timed out by a data source or firewall.

If for some reason your AssemblyLine still stops prematurely, your next step is to get it restarted. One approach is to use the Web Admin tool to define failure/response behaviors.

Another, even complimentary approach is to make a 'Launcher' AL and leverage the AssemblyLine Function Component that you used in the tutorial exercises of the last chapter. By placing this AL FC in a ConditionalLoop which never stops – in other words, with a Condition that is always *true* – and then configuring the AL FC to call the desired service AL and wait for it to complete, you ensure that whenever control returns to the 'LauncherAL' then the never-ending Loop will simply restart your data flow again.

If you apply the restart-loop technique outlined above for a synchronization AL based on one of the Change Detection Connectors (or the Delta Engine, both described elsewhere) then the AssemblyLine will automatically continue from the point where it failed. If not, then the burden of state handling rests on you. A common technique is to use the System Store to persist state information, like timestamps or other key values for sorted result sets. Then, whenever the AssemblyLine initializes, this state information is applied to the Iterator Connector in order to resume processing immediately after the previously handled entry.

If the iteration data source for the AL does not support sorted returns, then it might be necessary to start iteration from the very beginning again. In this case, note that the Connector Update mode offers a **Compute Changes** feature which compares Output Mapped Attributes with those currently found in the target system, skipping the modify operation if no differences are detected.

You can avoid a single point of failure by introducing a secure transport between multiple Tivoli Directory Integrator Servers, like IBM MQ. In this way, any number of Servers (and AssemblyLines) can be used to initiate processing by placing data and even processing instructions into the queue. At the receiving end, multiple Servers/ALs pick these up on a first-come-first-serve basis and carry out the requested work. This not only results in a more robust solution, it allows for easy scaling through adding sender and receiver AssemblyLines.

41. You may have noticed that the 'connector' word is sometime synonymous with 'component', such that variables like thisConnector can also refer to FC's, SC's and AttributeMap components. The same applies to the 'connectorname' Attribute of the error object, which can also hold the name of any type of component.

This has been a very brief discussion of a much larger subject. However, the goal is more for inspiration than the prescription of a particular approach. It is advised that you look to community websites and discussion groups for more specific recommendations and examples.

Scaling and performance

Again, this is a topic that merits a lengthier discussion than you will find here. However, a few points are worth mentioning, even if in a general fashion.

The primary gating factor to the speed of your data flows will be I/O; the time it takes to retrieve data from sources, or to push changes out to targets greatly overshadows processing time inside your AL. Furthermore, negotiating connections can also be very costly, especially when security handshaking is involved. Keeping this in mind when you design and build your AssemblyLines will directly impact their performance.

For example, imagine a Server Mode-based AL that receives incoming protocol requests from clients, like the HTTP Server solution you built in the last exercise. Each request received causes a service AssemblyLine to be launched in order to carry out the actual work. If this called AL has to initialize components, then the turnaround for each request will be at least as long as the sum of all connection times.

Three ways to alleviate this situation are:
- Have the main Server Mode AL do the actual processing instead of dispatching it, therefore not requiring connections to be set up and broken down for each request;
- Design the service AssemblyLine(s) so that it can be invoked in *Manual/Cycle Mode*. Manual/Cycle mode causes the service AL to initialize when the AL FC initializes. Furthermore, the AssemblyLine Function component drives the service AL only a single cycle for each call. In this way, the service AL acts just like a component of the calling AssemblyLine and must be built to accommodate this behavior;
- Use Global Connector Pooling. This feature is described in the *IBM Tivoli Directory Integrator V7.0 Reference Guide* and allows you to define a pool of Connectors that are initialized at Server start-up and shared between AssemblyLines as needed.

Another way to improve performance is to divide heaving processing tasks across multiple simultaneous AssemblyLines. Take for instance a migration task where instead of having one AssemblyLine work with the entire source data set, you launch multiple ALs that each handles a subset. Since you can pass initialization parameters into an AssemblyLine when you call it, a single AL can be developed that is started multiple times with a filter parameter for controlling the range of data that instance should process.

Yet another technique is to incorporate a message bus in your solution, as described in the previous section. This approach has been used with great success by some of IBM’s largest clients.

In cases where processing speed is hampered by unstable network links or systems with low availability, you can deploy additional ALs as background tasks to synchronize hard-to-reach data to local high-speed stores. Particularly when implementing real-time services, this technique can help ensure satisfactory response time for client requests.

Monitoring

Out-of-the-box your Tivoli Directory Integrator solutions are quickly and simply administrated using the Web Admin tool. This Integrated Service Console (ISC) plug-in is an AppServer application that can monitor any number of solutions running on any number of Servers in your infrastructure. In addition to out-of-the box features for viewing AL statistics, logs and start/stop times, the Web Admin tool allows you to customize the health and incident console, as well as defining schedules and failure/response behaviors to keep your AssemblyLines in flight.

You can also configure your Tivoli Directory Integrator solutions to send status events, for example as SNMP traps or using the system's custom event format. The system is readily configured for JMX administration and monitoring, for example using ITM.

The AssemblyLine Debugger

Although mentioned before, this warrants repeating: The time you spend time getting skilled with the AssemblyLine Debugger will be rewarded tenfold in accelerated solution development, troubleshooting and deployment.

'Nuff said. Get started TDI'ing :)

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 grant 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:IBM World Trade Asia Corporation Licensing 2-31 Roppongi 3-chome, Minato-ku Tokyo 106-0032, 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 may 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 _Department number/Building number_ _Site mailing address_ _City, State; Zip Code_ _U.S.A. (or appropriate country)

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.

Programming interface information

Trademarks

__________ is a registered trademark of Alphablox Corporation in the United States, other countries, or both.

Adobe, Acrobat, Portable Document Format (PDF), PostScript, and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom

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.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

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

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Electronic emission notices

Federal Communications Commission (FCC) statement

Index

Printed in USA

GI11-8185-00