

Directory Integrator  
Version 7.1.1

## *Getting Started Guide*





Directory Integrator  
Version 7.1.1

## *Getting Started Guide*



**Notices**

Before using this information and the product it supports, read the general information under “Notices” on page 101.

**Edition notice**

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

© **Copyright IBM Corporation 2003, 2012.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

|                |          |
|----------------|----------|
| <b>Figures</b> | <b>v</b> |
|----------------|----------|

|                |            |
|----------------|------------|
| <b>Preface</b> | <b>vii</b> |
|----------------|------------|

|                                    |      |
|------------------------------------|------|
| Who should read this book?.        | vii  |
| Publications                       | vii  |
| Accessing publications online      | viii |
| Accessibility                      | ix   |
| Accessibility features             | ix   |
| Keyboard navigation                | ix   |
| Interface information              | ix   |
| Vendor software                    | x    |
| Related accessibility information. | x    |

|                                |          |
|--------------------------------|----------|
| <b>Chapter 1. Introduction</b> | <b>1</b> |
|--------------------------------|----------|

|                                  |   |
|----------------------------------|---|
| Simplify and solve               | 3 |
| Kernel/Component Architecture.   | 3 |
| Entry-Attribute-value data model | 4 |
| Data flows = AssemblyLines       | 5 |
| Getting Started                  | 6 |

|  |  |
|--|--|
| <b>Chapter 2. Introducing IBM Tivoli</b> |  |
|--|--|

|                              |          |
|------------------------------|----------|
| <b>Directory Integrator.</b> | <b>9</b> |
|------------------------------|----------|

|  |    |
|--|----|
| Creating your first AssemblyLine         | 13 |
| Running your Assemblyline                | 26 |
| Null Behavior: Dealing with missing      |    |
| Attributes/values                        | 29 |
| Debugging your AssemblyLine.             | 36 |
| Looking up data from a sequential source | 43 |

|                                     |    |
|-------------------------------------|----|
| Using Lookup Mode                   | 54 |
| Inheritance                         | 59 |
| Lookup search rules = Link Criteria | 61 |
| Deciphering Run errors              | 62 |

|  |           |
|--|-----------|
| <b>Chapter 3. Event-driven integration</b> | <b>65</b> |
|--|-----------|

|                               |    |
|-------------------------------|----|
| Scheduling AssemblyLines      | 66 |
| Service request AssemblyLines | 67 |

|  |  |
|--|--|
| <b>Chapter 4. Hardening your Integration</b> |  |
|--|--|

|                   |           |
|-------------------|-----------|
| <b>Solutions.</b> | <b>75</b> |
|-------------------|-----------|

|  |    |
|--|----|
| Legibility, re-use and configurability | 75 |
| Logging and auditing                   | 76 |
| Connectivity problems.                 | 77 |
| AssemblyLine availability              | 77 |
| Scaling and performance                | 79 |
| Monitoring                             | 80 |
| The AssemblyLine Debugger              | 80 |

|                                |           |
|--------------------------------|-----------|
| <b>Appendix. EasyETL Guide</b> | <b>81</b> |
|--------------------------------|-----------|

|                    |    |
|--------------------|----|
| Creating a Project | 83 |
| Detecting Changes  | 95 |

|                |            |
|----------------|------------|
| <b>Notices</b> | <b>101</b> |
|----------------|------------|

|              |            |
|--------------|------------|
| <b>Index</b> | <b>105</b> |
|--------------|------------|



---

## Figures

|   |    |  |    |
|---|----|--|----|
| 1. The Entry-Attribute-value data model . . . . .                       | 5  | 49. JavaScript Evaluation commandline . . . . .                                  | 43 |
| 2. Data flowing down an AssemblyLine . . . . .                          | 6  | 50. The scenario flow diagram . . . . .  | 44 |
| 3. Tutorial scenario . . . . .  | 7  | 51. Dragging 'FullName' to the Input Map of your<br>Iterator Connector . . . . . | 44 |
| 4. Starting the Configuration Editor. . . . .                           | 9  | 52. Editing the assignment for 'FullName' . . . . .                              | 45 |
| 5. Selecting your workspace . . . . .                                   | 9  | 53. Drag the ConnectorLoop . . . . .   | 46 |
| 6. Configuration Editor Welcome Screen. . . . .                         | 10 | 54. ConnectorLoop Configuration . . . . .  | 46 |
| 7. Naming your new project. . . . .                                     | 11 | 55. ConnectorLoop Advanced Settings. . . . .                                     | 47 |
| 8. Config Editor main screen . . . . .                                  | 12 | 56. Hierarchical Attributes . . . . .  | 47 |
| 9. Simplified scenario diagram with just two data<br>sources . . . . .  | 13 | 57. Dragging from Schema to Attribute Map<br>AssemblyLine. . . . .               | 48 |
| 10. New AssemblyLine dialog box . . . . .                               | 14 | 58. Condition editor for IF branch . . . . .                                     | 48 |
| 11. Empty AssemblyLine editor . . . . .                                 | 15 | 59. Scripting the End of Data Hook. . . . .                                      | 49 |
| 12. Inserting a new component . . . . .                                 | 16 | 60. Component list in the AssemblyLine Data<br>Flow section . . . . .            | 50 |
| 13. Choosing the component . . . . .                                    | 17 | 61. Scripting a Condition for the IF branch . . . . .                            | 51 |
| 14. Renaming the Connector and changing its<br>mode . . . . .           | 18 | 62. AssemblyLine complete with FOR-EACH Loop . . . . .                           | 52 |
| 15. File Connector Configuration panel . . . . .                        | 19 | 63. Log Output with IF branch statistics . . . . .                               | 53 |
| 16. Selecting Parser during Insert new object . . . . .                 | 20 | 64. XML output with 'telephoneNo' Attribute . . . . .                            | 54 |
| 17. Browse Data context menu selection . . . . .                        | 21 | 65. AssemblyLine Copy function. . . . .  | 55 |
| 18. The Data Browser . . . . .  | 21 | 66. Copying an AssemblyLine into your project . . . . .                          | 56 |
| 19. Interactively discovering schema by browsing<br>live data . . . . . | 22 | 67. Run the CreatePhoneDB AL . . . . .   | 57 |
| 20. AL with Iterator Connector in place . . . . .                       | 22 | 68. Log output from the 'CreatePhoneDB'<br>AssemblyLine. . . . .                 | 58 |
| 21. Add component button . . . . .                                      | 23 | 69. Drag a Connector to Resources . . . . .                                      | 58 |
| 22. AL with two Connectors in place . . . . .                           | 24 | 70. Drag the new resource into your AssemblyLine . . . . .                       | 59 |
| 23. Dragging Attributes to the Output Map . . . . .                     | 24 | 71. Setting Inheritance for the Hooks tab . . . . .                              | 60 |
| 24. Renaming an Attribute Map rule . . . . .                            | 25 | 72. Restoring inheritance for a mapping rule . . . . .                           | 60 |
| 25. Adding the 'FullName' Attribute to the Output<br>Map . . . . .      | 25 | 73. Changing mode, discovering and mapping<br>Attributes . . . . .               | 61 |
| 26. Editing the assignment. . . . .                                     | 26 | 74. A simple Link Criteria . . . . .   | 62 |
| 27. The Run button . . . . .  | 26 | 75. Error message in log output . . . . .  | 62 |
| 28. Log Output from the AssemblyLine run . . . . .                      | 27 | 76. Partial Flow Diagram for Lookup mode . . . . .                               | 63 |
| 29. Button bar for the Log Output window . . . . .                      | 27 | 77. First tutorial exercise completed. . . . .                                   | 64 |
| 30. Browsing Data created by an Output<br>Connector . . . . .           | 28 | 78. TDI Scheduler. . . . .   | 67 |
| 31. Browsing the resulting XML . . . . .                                | 28 | 79. HTTP Server Connector Attribute Map panel . . . . .                          | 68 |
| 32. Null Behavior button for AL-level<br>configuration . . . . .        | 29 | 80. Add Input Attribute Map item . . . . .                                       | 69 |
| 33. Null Behavior configuration dialog. . . . .                         | 30 | 81. Wildcard map item . . . . .  | 70 |
| 34. Result in the XML output of Null Behavior<br>settings . . . . .     | 31 | 82. TCP and HTTP header properties returned as<br>Attributes . . . . .           | 71 |
| 35. Selecting the IF branch component. . . . .                          | 32 | 83. Drag in the AssemblyLine Function component<br>(AL FC). . . . .              | 72 |
| 36. Editing conditions for the IF branch . . . . .                      | 33 | 84. Work Entry dump followed by AL statistics . . . . .                          | 73 |
| 37. Adding simple Conditions to the IF branch . . . . .                 | 33 | 85. Completed TINA_WebServer AssemblyLine . . . . .                              | 74 |
| 38. Your first complete AssemblyLine . . . . .                          | 34 | 86. Simple Web Interface to your solution. . . . .                               | 74 |
| 39. Resetting Null Behavior for the AssemblyLine . . . . .              | 35 | 87. Welcome screen . . . . .   | 82 |
| 40. Log output with your messages and Work<br>Entry dump . . . . .      | 36 | 88. Figure 2. EasyETL Workbench . . . . .  | 83 |
| 41. Debugging your AssemblyLine . . . . .                               | 36 | 89. New Project button . . . . .   | 84 |
| 42. The AssemblyLine Data Stepper . . . . .                             | 37 | 90. Simple AssemblyLine editor . . . . .   | 84 |
| 43. Stepping into the AL run . . . . .                                  | 38 | 91. Selecting Source information . . . . .                                       | 85 |
| 44. Stepping to the Write_XML_File Connector . . . . .                  | 39 | 92. Setting the File Path parameter . . . . .                                    | 86 |
| 45. Advanced Debugger mode . . . . .                                    | 40 | 93. Testing the connection and discovering schema . . . . .                      | 87 |
| 46. Debugger buttons . . . . .  | 41 | 94. Input Source configured . . . . .  | 88 |
| 47. Setting a breakpoint. . . . .                                       | 41 | 95. Renaming an Output Attribute . . . . .                                       | 89 |
| 48. Setting a Breakpoint in script. . . . .                             | 42 | 96. One record read and collected . . . . .                                      | 90 |
|   |    | 97. EasyETL AssemblyLine completed . . . . .                                     | 90 |
|   |    | 98. Enabling Transformation . . . . .  | 91 |

|  |    |   |     |
|--|----|---|-----|
| 99. Show Transformation script . . . . .                             | 92 | 104. All entries unchanged and skipped . . . . .                  | 96  |
| 100. Evaluate Expression. . . . .                                    | 92 | 105. Selecting your link criteria . . . . .                       | 97  |
| 101. Output collection with computed FullName<br>Attribute . . . . . | 93 | 106. Creating command line assets to run the ETL<br>job . . . . . | 98  |
| 102. XML Output . . . . .  | 94 | 107. Running your ETL job at full speed . . . . .                 | 99  |
| 103. Delta configuration . . . . .                                   | 95 | 108. Defining Link Criteria for an Input Connector                | 100 |



---

## 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 the Tivoli 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 Oracle 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 V7.1.1 library and the related publications to determine which publications you might find helpful. After you determine the publications you need, refer to the instructions for accessing publications online.

### IBM Tivoli Directory Integrator library

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

#### *IBM Tivoli Directory Integrator V7.1.1 Getting Started*

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

#### *IBM Tivoli Directory Integrator V7.1.1 Installation and Administrator Guide*

Includes complete information about installing, migrating from a previous version, configuring

the logging functionality, and the security model underlying the Remote Server API of Tivoli Directory Integrator. Contains information on how to deploy and manage solutions.

*IBM Tivoli Directory Integrator V7.1.1 Users Guide*

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

*IBM Tivoli Directory Integrator V7.1.1 Reference Guide*

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

*IBM Tivoli Directory Integrator V7.1.1 Problem Determination Guide*

Provides information about Tivoli Directory Integrator tools, resources, and techniques that can aid in the identification and resolution of problems.

*IBM Tivoli Directory Integrator V7.1.1 Messages Guide*

Provides a list of all informational, warning and error messages associated with the Tivoli Directory Integrator.

*IBM Tivoli Directory Integrator V7.1.1 Password Synchronization Plug-ins Guide*

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

*IBM Tivoli Directory Integrator V7.1.1 Release Notes*

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

## Related Publications

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

- IBM Tivoli Directory Integrator V7.1.1 uses the JNDI client from Oracle. For information about the JNDI client, refer to the *Java Naming and Directory Interface™ Specification* at <http://download.oracle.com/javase/6/docs/technotes/guides/jndi/index.html>.
- The Tivoli Software Library provides a variety of Tivoli publications such as white papers, datasheets, demonstrations, redbooks, and announcement letters. The Tivoli Software Library is available on the Web at: <http://www.ibm.com/software/tivoli/library/>
- The *Tivoli Software Glossary* includes definitions for many of the technical terms related to Tivoli software. The Tivoli Software Glossary is available on the Web, in English only, at <http://publib.boulder.ibm.com/tividd/glossary/tivoliglossarymst.htm>
- A list of most requested documents as well as those identified as valuable in helping answer your questions related to IBM Tivoli Directory Integrator can be found at <http://www.ibm.com/support/docview.wss?rs=697&context=SSCQGF&uid=swg27010509>.

---

## Accessing publications online

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

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

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

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

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

---

## Accessibility

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

---

### Accessibility features

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

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

---

### Keyboard navigation

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

---

### Interface information

The accessibility features of the user interface and documentation include:

- Steps for changing fonts, colors, and contrast settings in the Configuration Editor:
  1. Type **Alt-W** to access the Configuration Editor **Window** menu. Using the downward arrow, select **Preferences...** and press Enter.
  2. Under the **Appearance** tab, select **Colors and Fonts** settings to change the fonts for any of the functional areas in the Configuration Editor.
  3. Under **View and Editor Folders**, select the colors for the Configuration Editor, and by selecting colors, you can also change the contrast.
- Steps for customizing keyboard shortcuts, specific to IBM Tivoli Directory Integrator:
  1. Type **Alt-W** to access the Configuration Editor **Window** menu. Using the downward arrow, select **Preferences...**
  2. Using the downward arrow, select the General category; right arrow to open this, and type downward arrow until you reach the entry **Keys**.  
Underneath the **Scheme** selector, there is a field, the contents of which say "type filter text." Type **tivoli directory integrator** in the filter text field. All specific Tivoli Directory Integrator shortcuts are now shown.
  3. Assign a keybinding to any Tivoli Directory Integrator command of your choosing.

4. Click **Apply** to make the change permanent.

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

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

---

## Vendor software

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

---

## Related accessibility information

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

---

## Chapter 1. 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.

If you are impatient to quickly start extracting information from files, directories, databases or Lotus Notes and transferring this data someplace else then you may want to skip directly to the appendix, “EasyETL Guide,” on page 81. This feature lets you harness the power of IBM Tivoli Directory Integrator without having to first learn core concepts. Instead you choose your source and target, and then press Run and watch your data flow. If on the other hand you want more control over how data is read, filtered, enriched, transformed and moved then continue reading here; “EasyETL Guide,” on page 81 will still be there for your future reading pleasure.

IBM Tivoli Directory Integrator<sup>1</sup> 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

---

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.

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.

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 newsgroups and websites for recommendations and links.

For more information about scripting in IBM Tivoli Directory Integrator, see the *IBM Tivoli Directory Integrator V7.1.1 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<sup>2</sup> 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 V7.1.1 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\V7.1.1\examples\Tutorial

The 'Tutorial' directory should contain the following files:

- CreatePhoneDB.assemblyline
- index.html
- OtherPage.html
- People.csv
- PhoneNumbers.xml

---

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

- `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.

---

## 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<sup>3</sup>;
- 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 be 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

---

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



Attributes, or none at all.

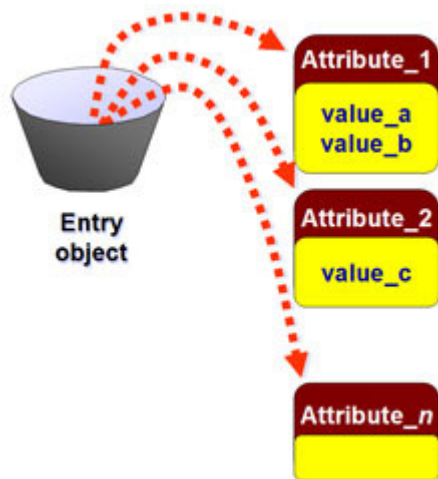


Figure 1. The Entry-Attribute-value data model

Each data flow has a primary Entry 'bucket' called its Work Entry. Whenever a Connector 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.

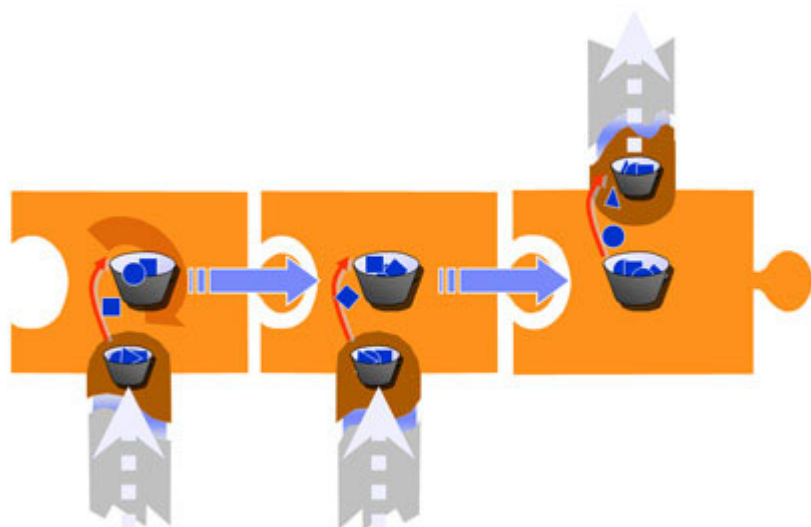


Figure 2. Data flowing down an AssemblyLine

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<sup>4</sup>.

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 Work Entry to the Conn 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<sup>5</sup>, 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.

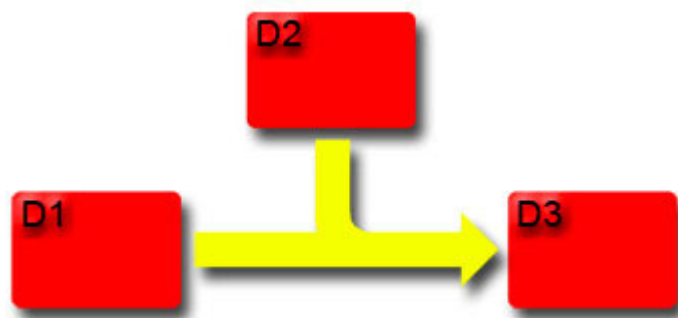


Figure 3. Tutorial scenario

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.



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.

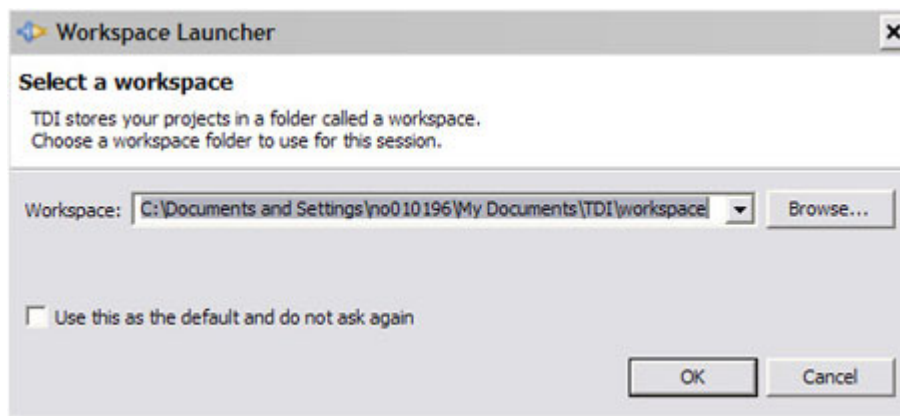


Figure 5. Selecting your workspace

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.

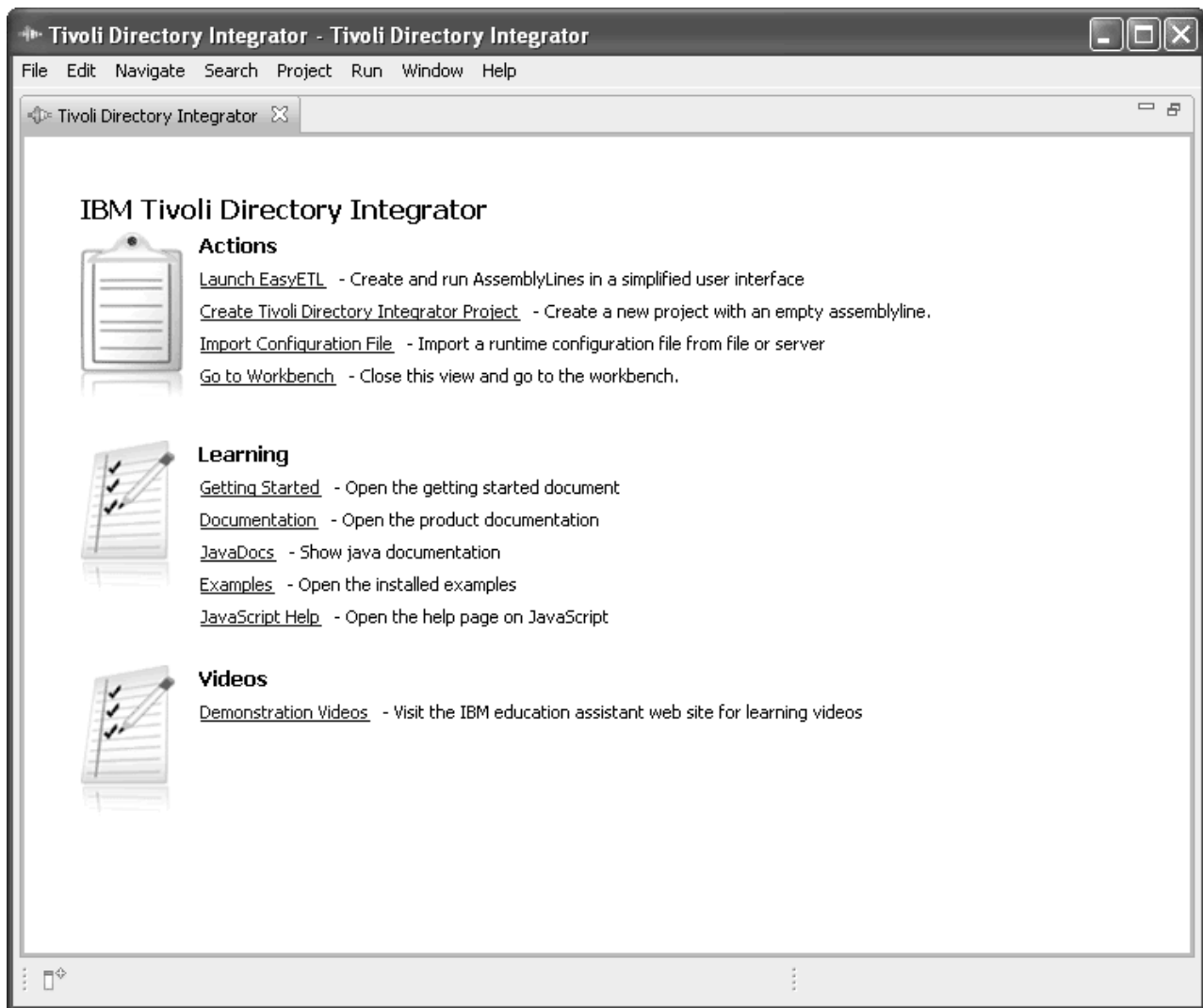


Figure 6. Configuration Editor Welcome Screen

The Welcome screen offers a number of quick-start links<sup>6</sup>.

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 second link from the top of this page<sup>7</sup> (*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.

7. The topmost link, Launch EasyETL, opens a simplified workbench and is covered in the appendix, "EasyETL Guide," on page 81.

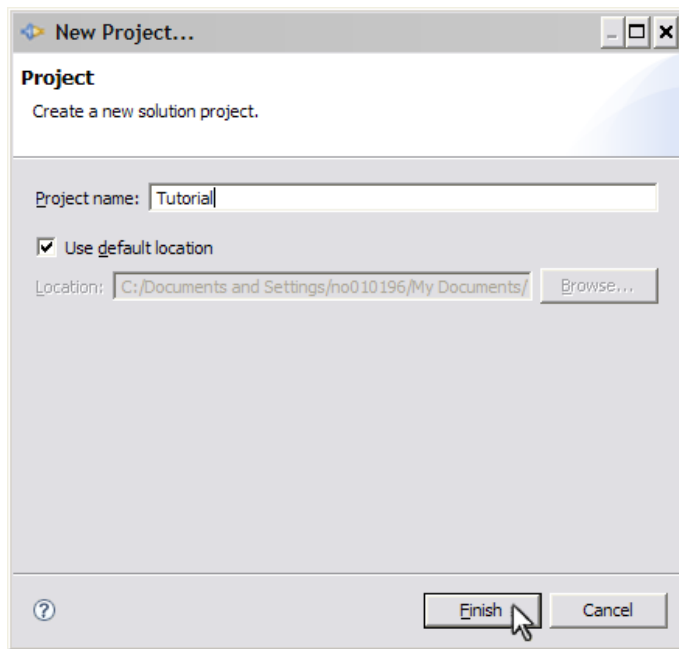


Figure 7. Naming your new project

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<sup>8</sup>.

---

8. 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.

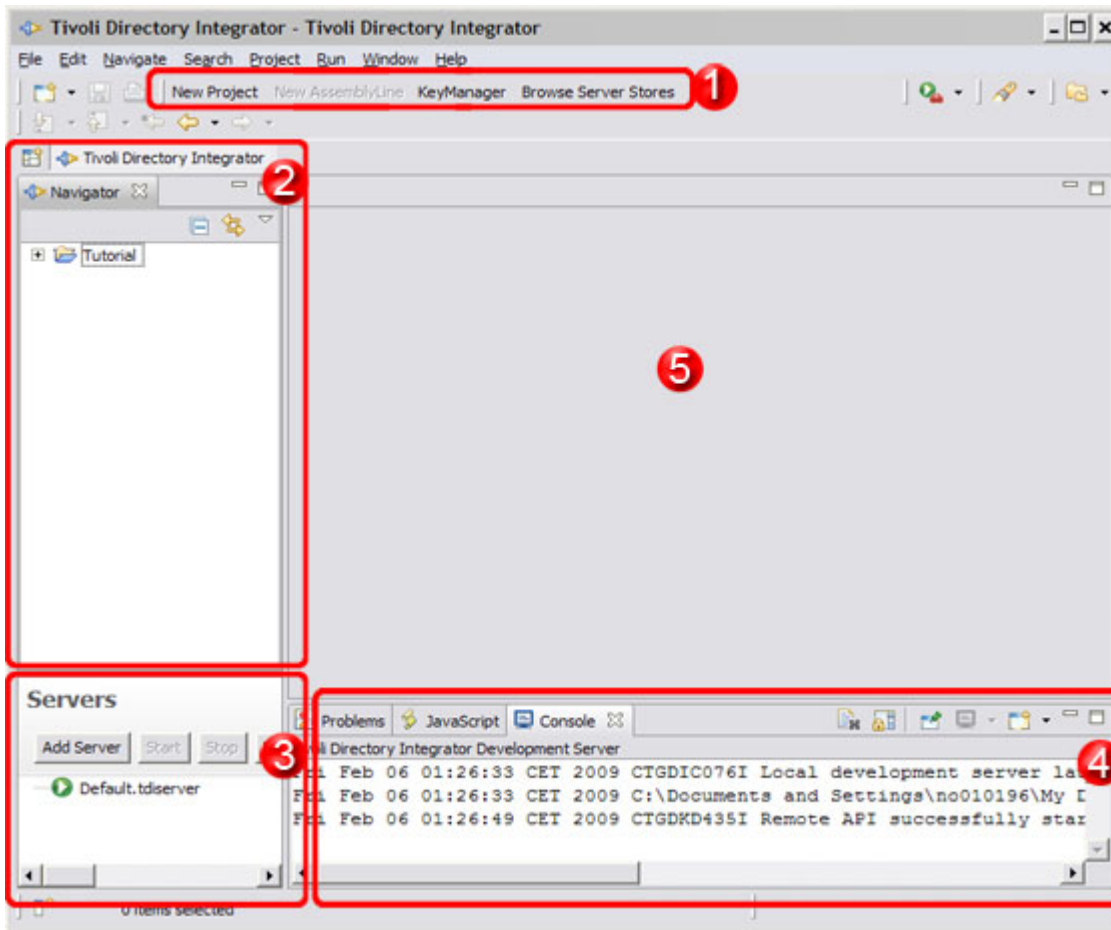


Figure 8. Config Editor main screen

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<sup>9</sup>, 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<sup>10</sup>.

9. As with most Eclipse-based applications, there will be a number of ways to perform the same operation. The *IBM Tivoli Directory Integrator V7.1.1 Users Guide* describes all the various options and panels available.

10. 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<sup>11</sup> 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'.

---

11. 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.

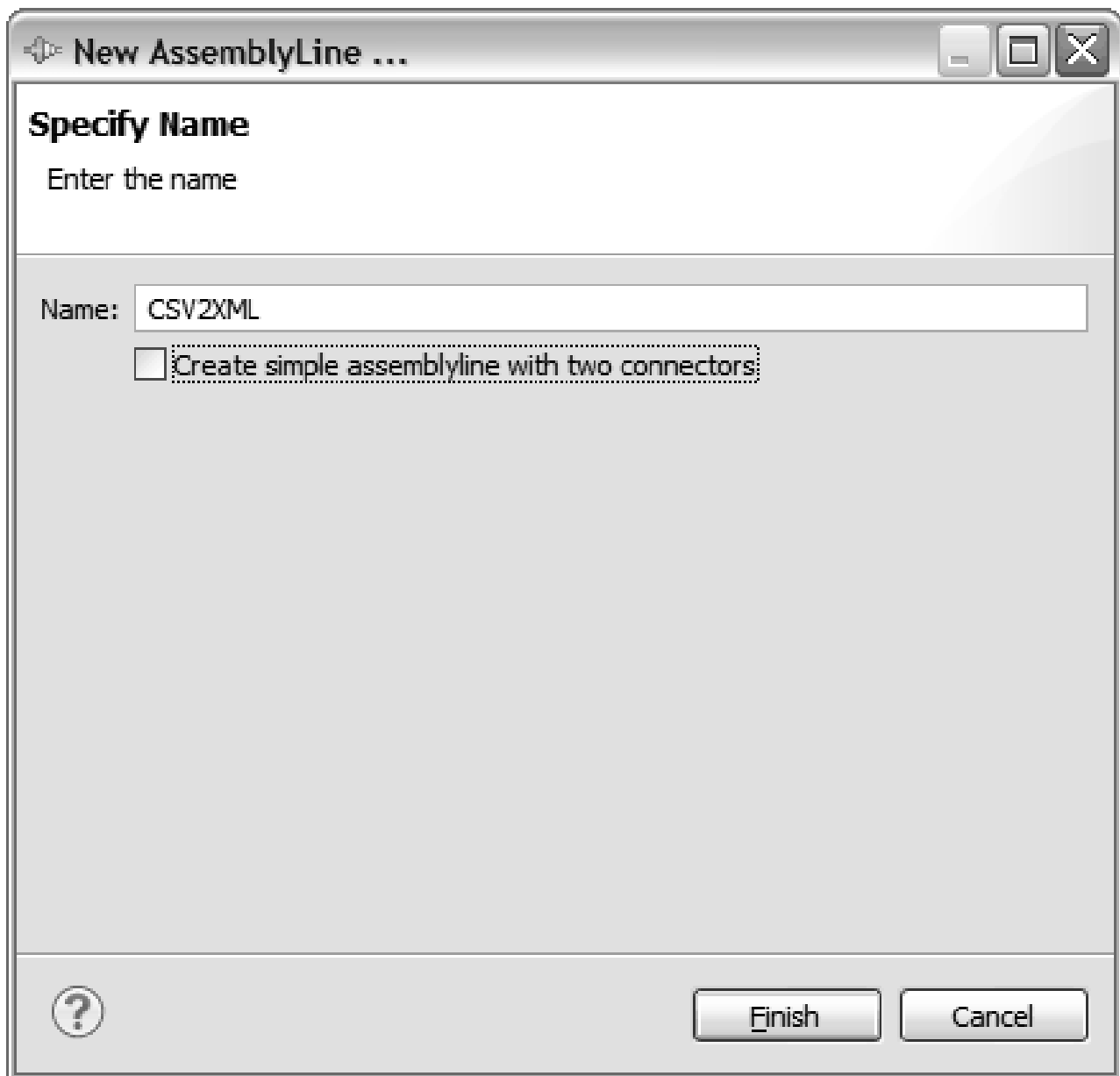


Figure 10. New AssemblyLine dialog box

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

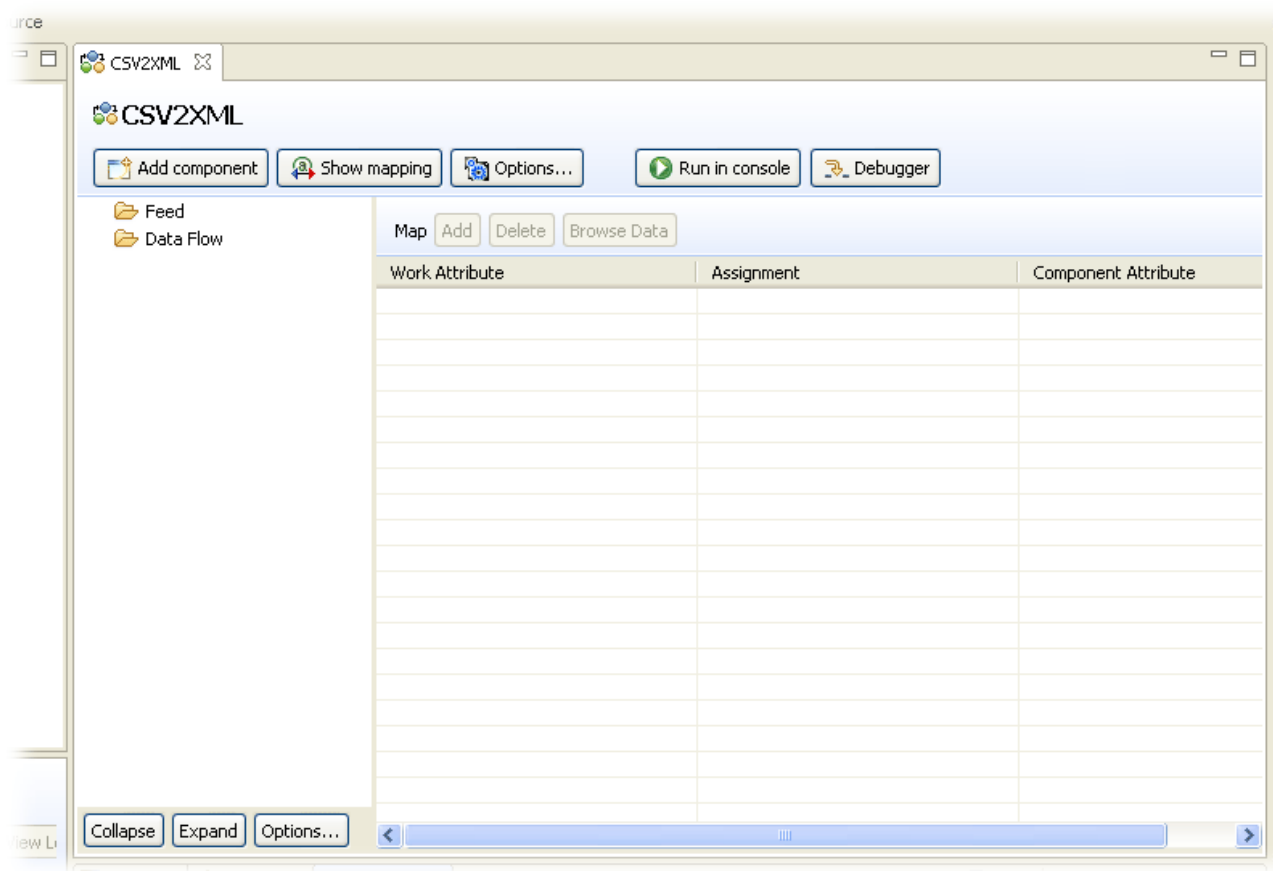


Figure 11. Empty AssemblyLine editor

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<sup>12</sup>.

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 **Add Component...**

12. 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.

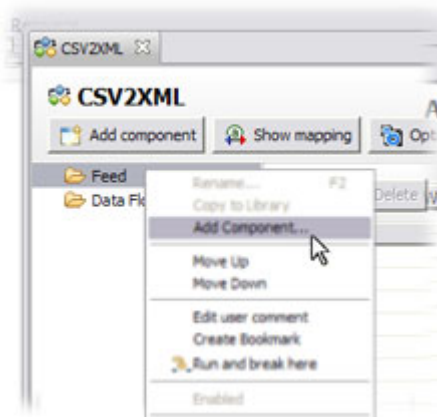


Figure 12. Inserting a new component

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

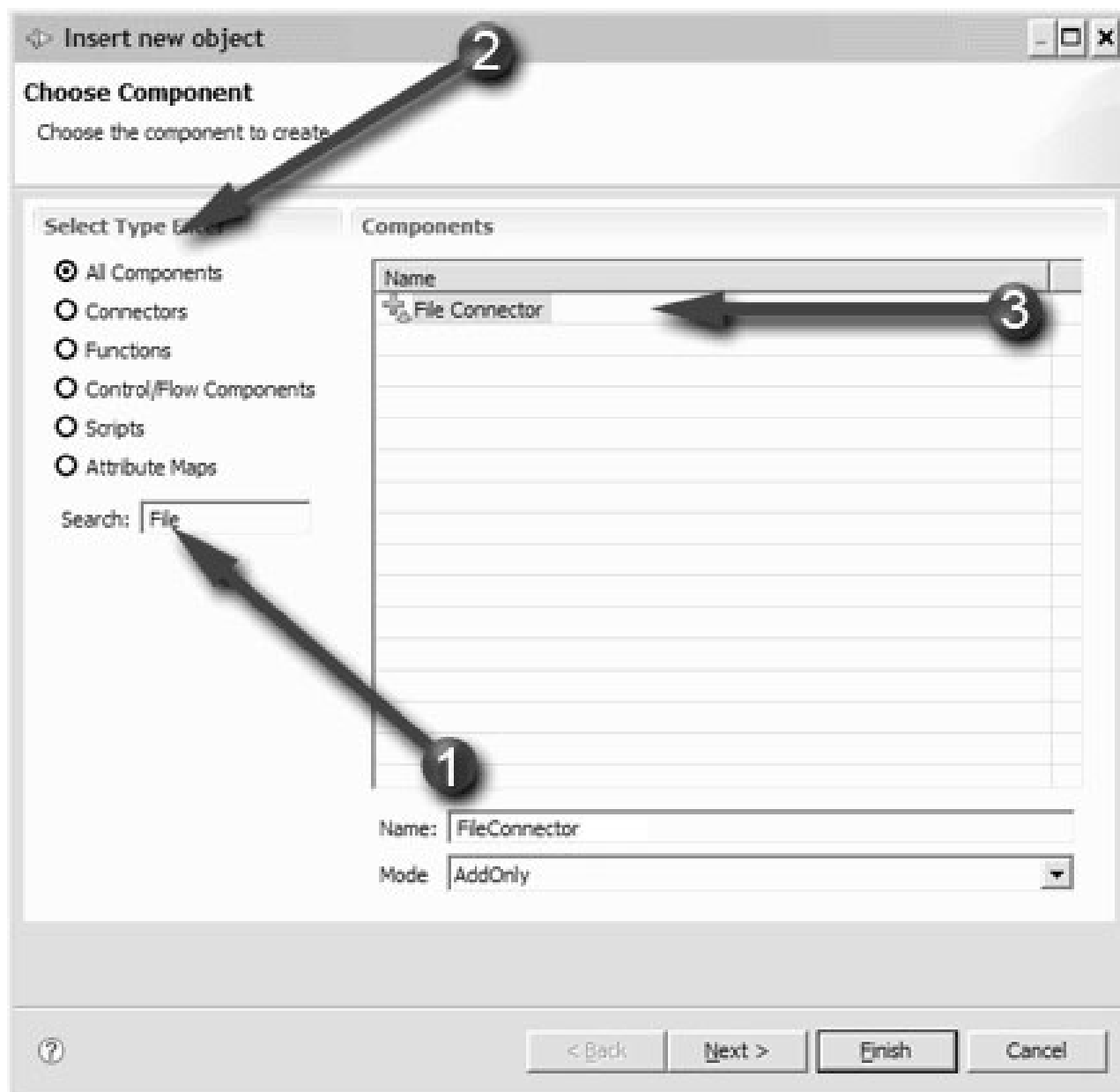


Figure 13. Choosing the component

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 component you want from this list. In our example this will be 'File 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<sup>13</sup> and then select **Iterator** from the **Mode** drop-down.

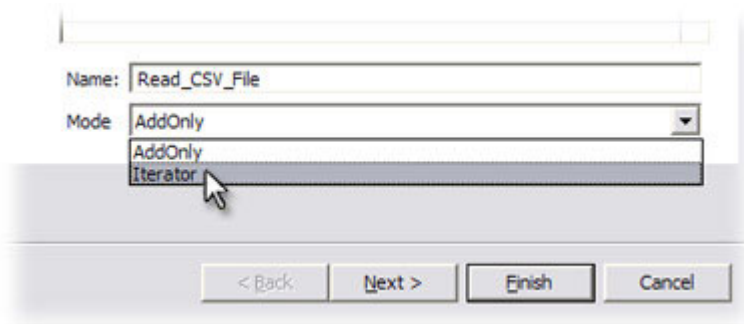


Figure 14. Renaming the Connector and changing its mode

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.

---

13. 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.

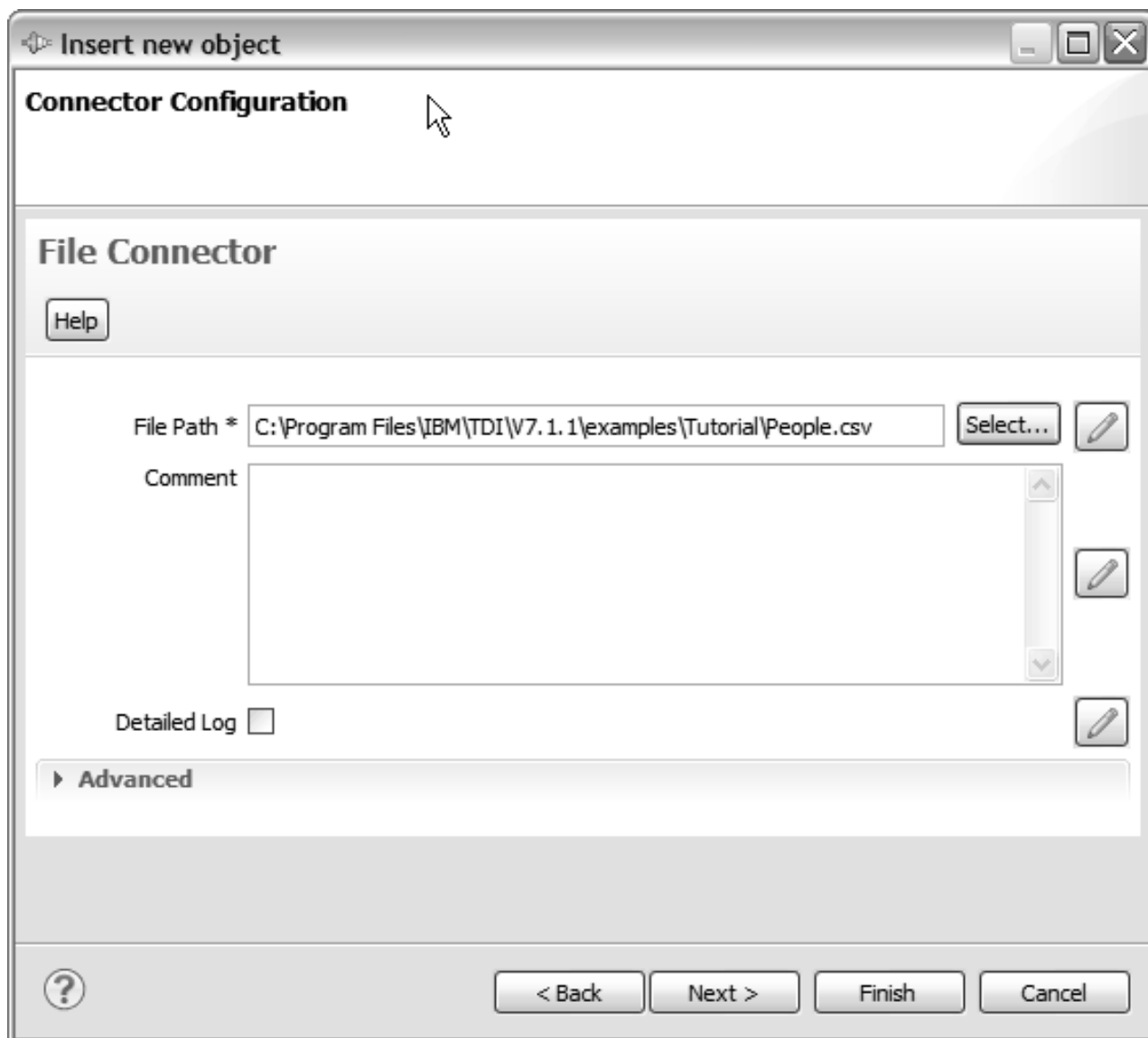


Figure 15. File Connector Configuration panel

Each component provides its own set of configuration parameters. The ones shown onscreen now are for the File 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<sup>14</sup> – 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 **CSV Parser** to select it.

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

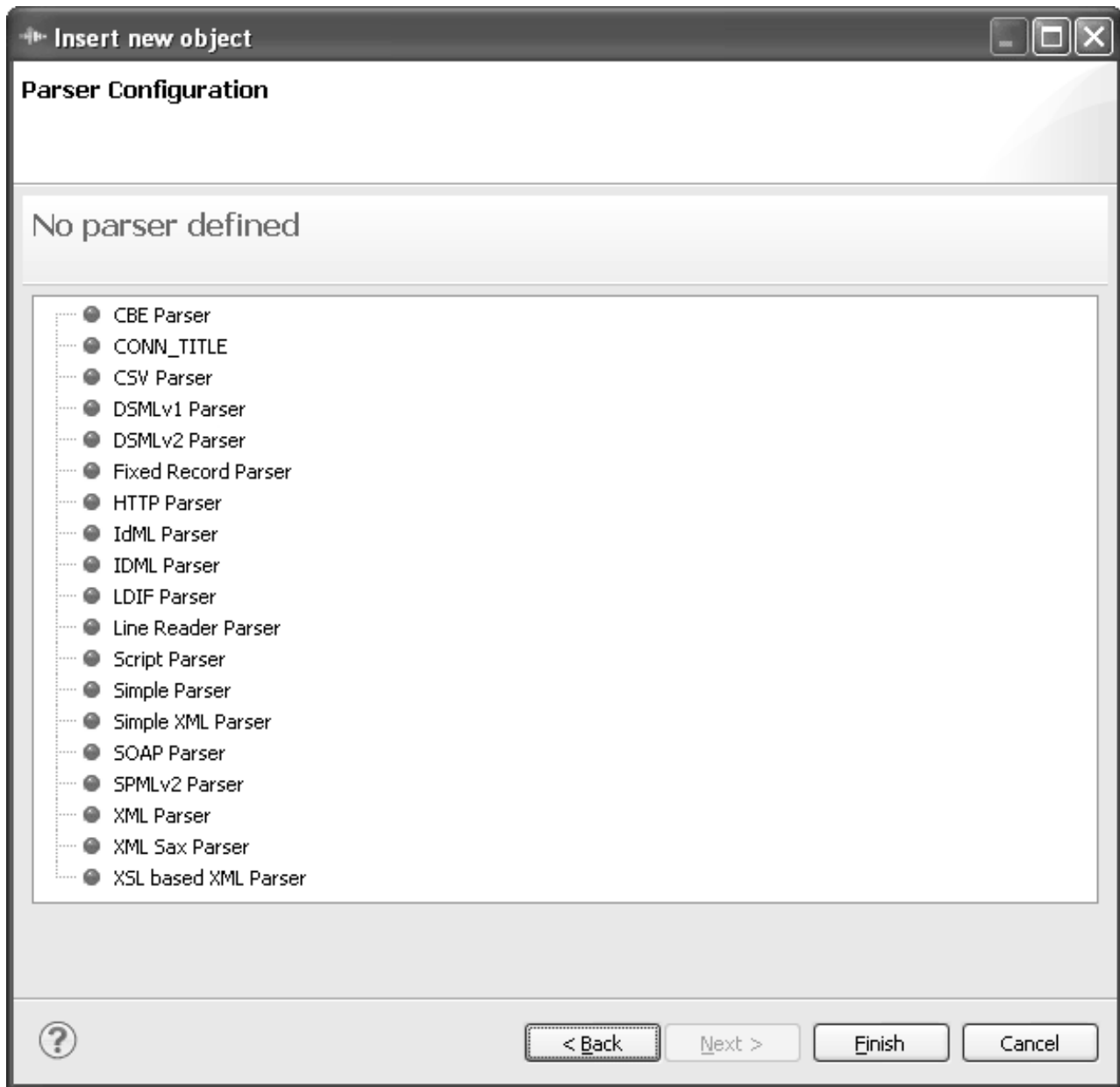


Figure 16. Selecting Parser during Insert new object

Once you have selected the CSV Parser then press **Finish** to close the wizard. 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<sup>15</sup>. 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.

15. 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 you want. The Data Browser is useful when you are unsure of the format. But I still thought you ought to try it :)



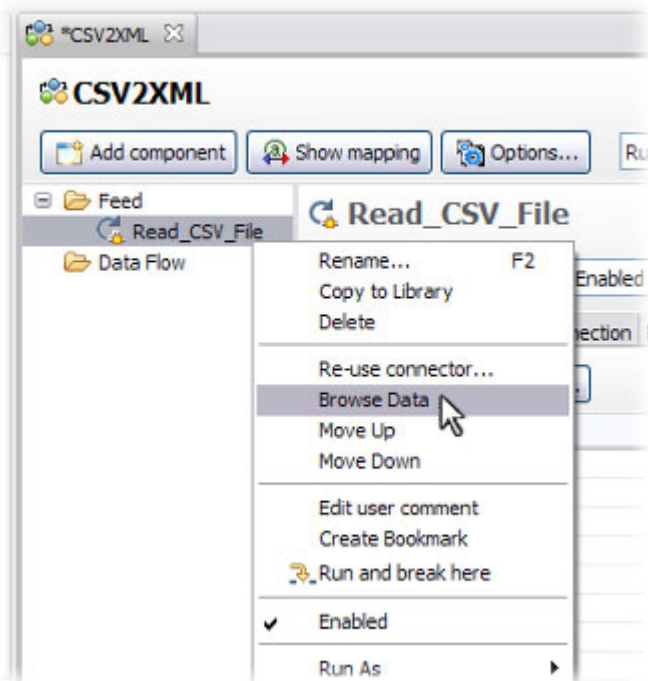


Figure 17. Browse Data context menu selection

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

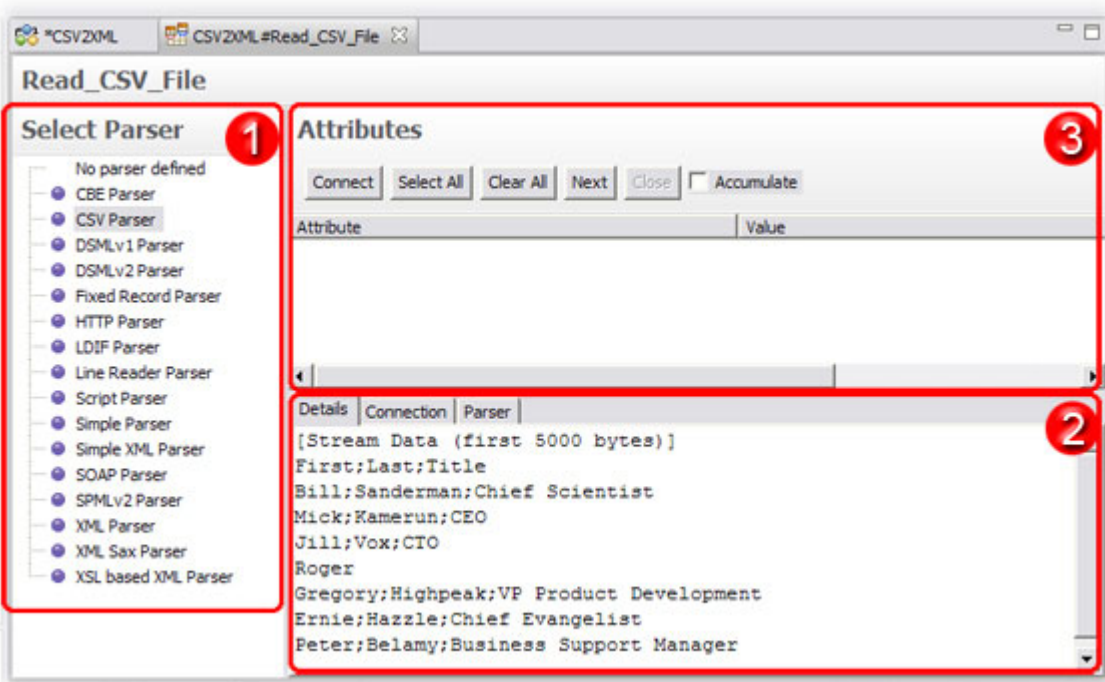


Figure 18. The Data Browser

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.

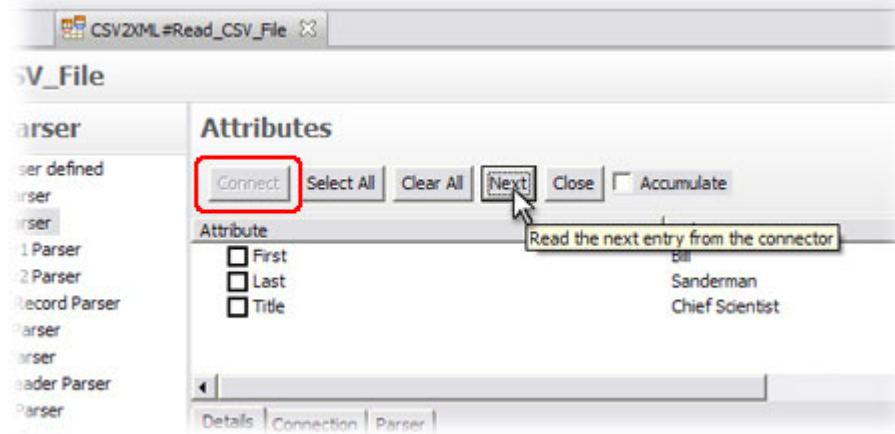


Figure 19. Interactively discovering schema by browsing live data

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<sup>16</sup>.

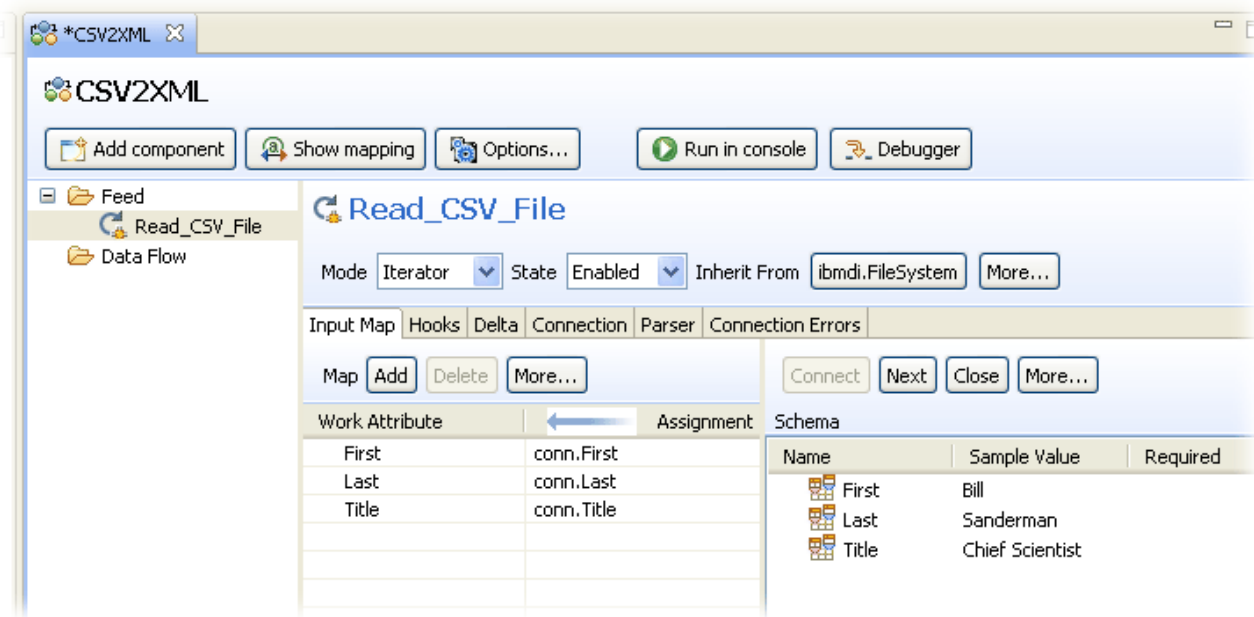


Figure 20. AL with Iterator Connector in place

16. 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**.

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 *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`<sup>17</sup>. 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");18
```

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.

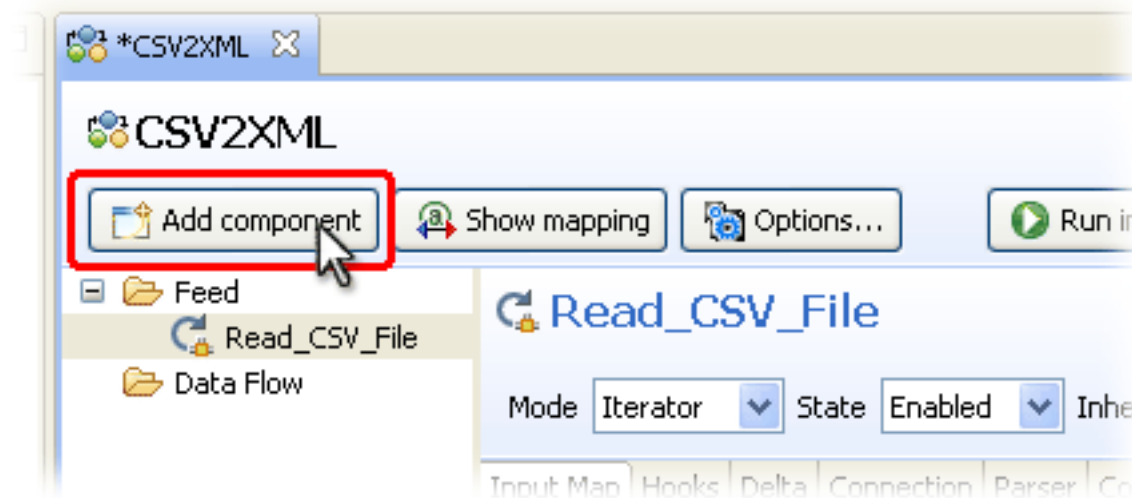


Figure 21. Add component button

Again choose the File 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,

17. 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.

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

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

you can't do Schema Discovery since there is no Output.xml file to discover from.

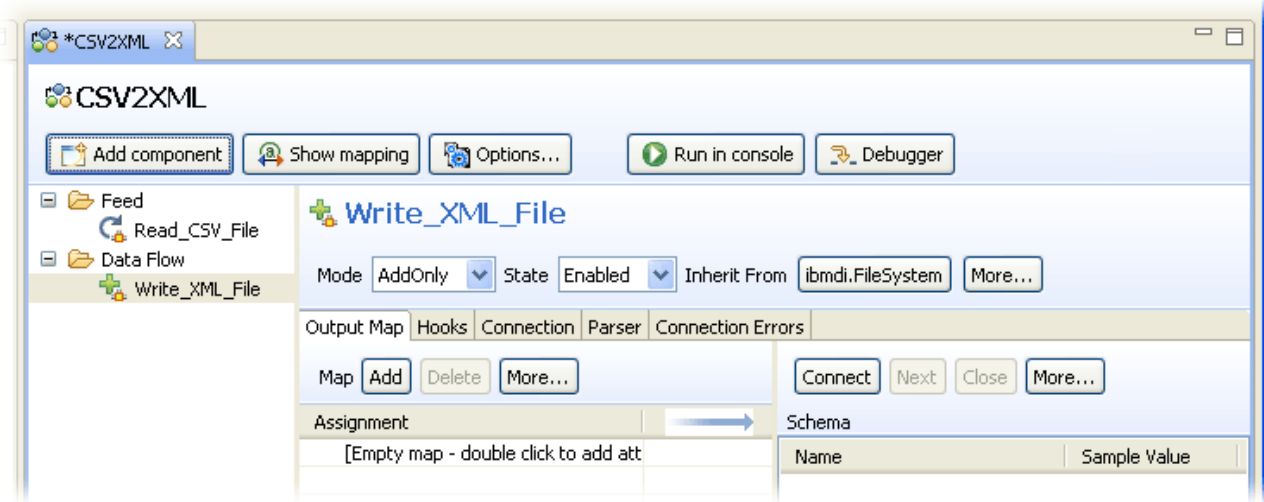


Figure 22. AL with two Connectors in place

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<sup>19</sup> and drag them down to the Output Map of the 'Write\_XML\_File' Connector, completing the data flow.

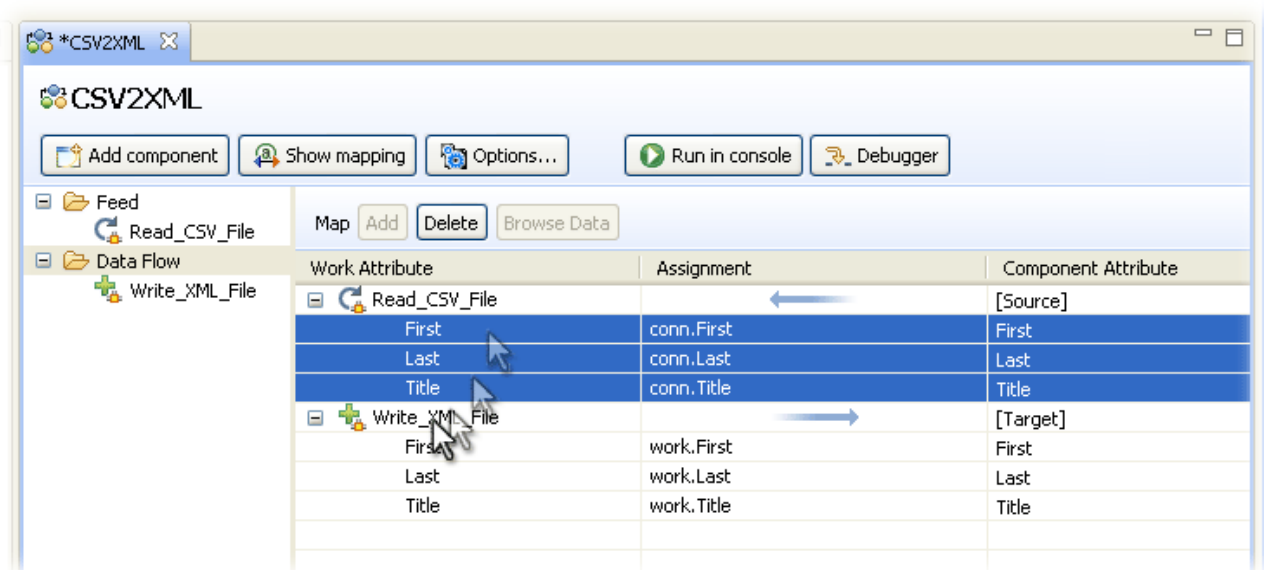


Figure 23. Dragging Attributes to the Output Map

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

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 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.

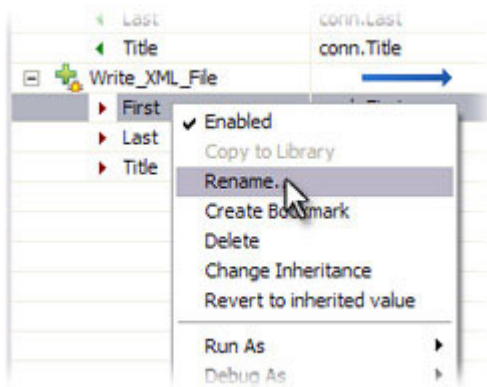


Figure 24. Renaming an Attribute Map rule

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**.

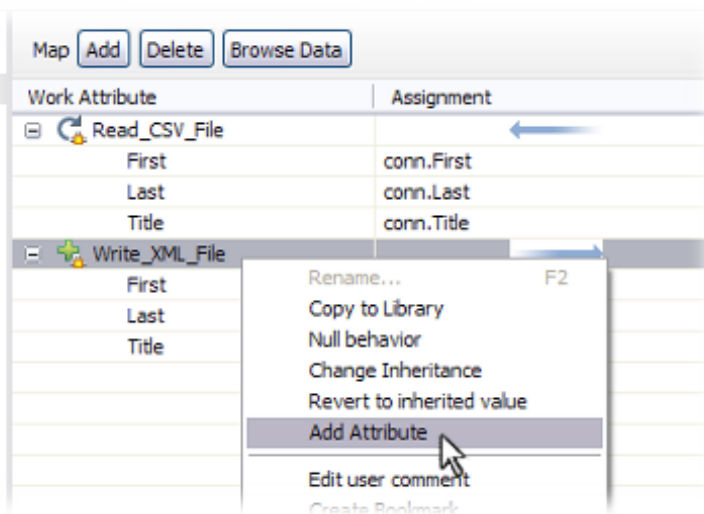


Figure 25. Adding the 'FullName' Attribute to the Output Map

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:

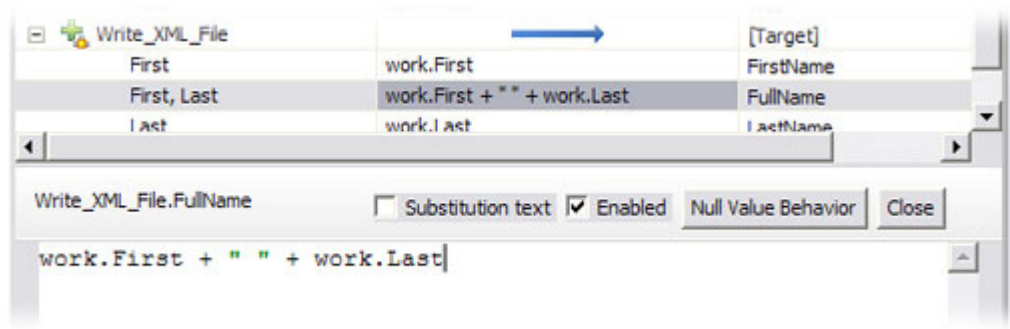


Figure 26. Editing the assignment

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<sup>20</sup>. 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.

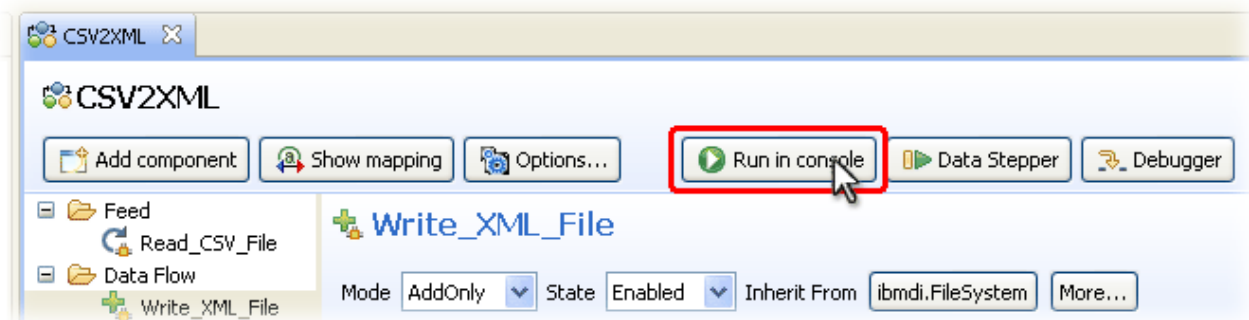


Figure 27. The Run button

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

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

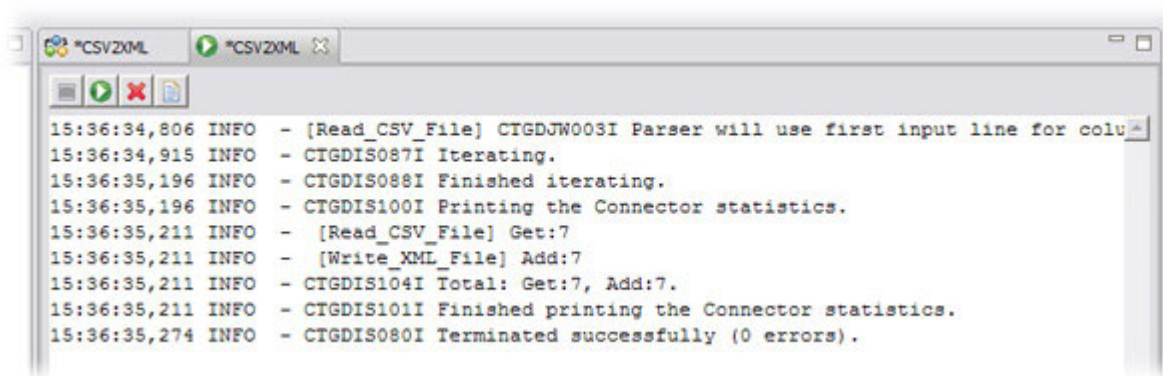


Figure 28. Log Output from the AssemblyLine run

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.

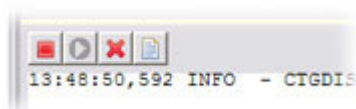


Figure 29. Button bar for the Log Output window

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**.



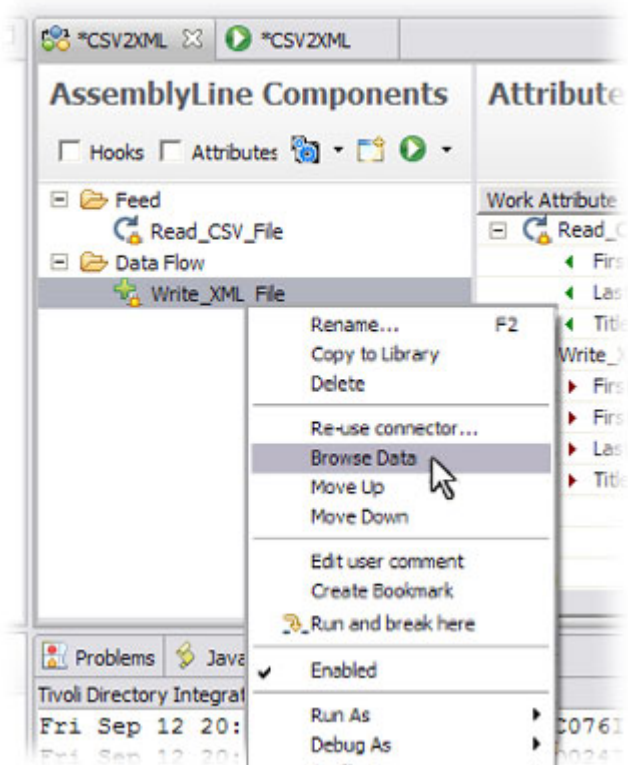


Figure 30. Browsing Data created by an Output Connector

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.

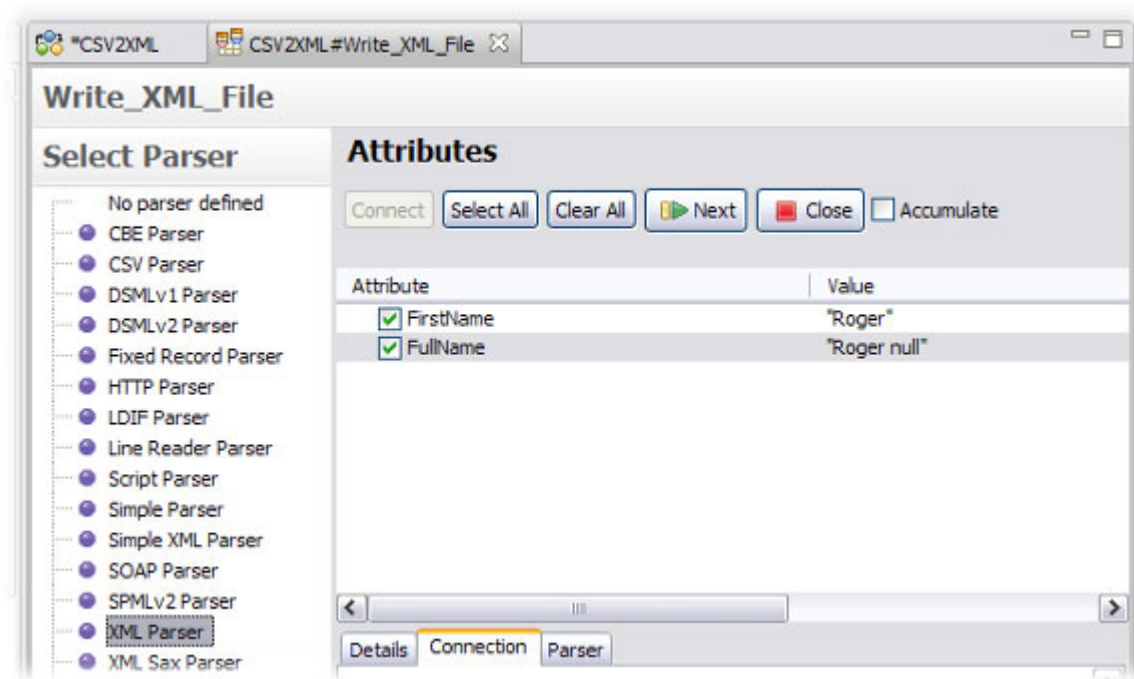


Figure 31. Browsing the resulting XML



The output XML should be an accurate representation of the input values, plus your mapping logic; everything looks good apart from the fourth 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<sup>21</sup>.

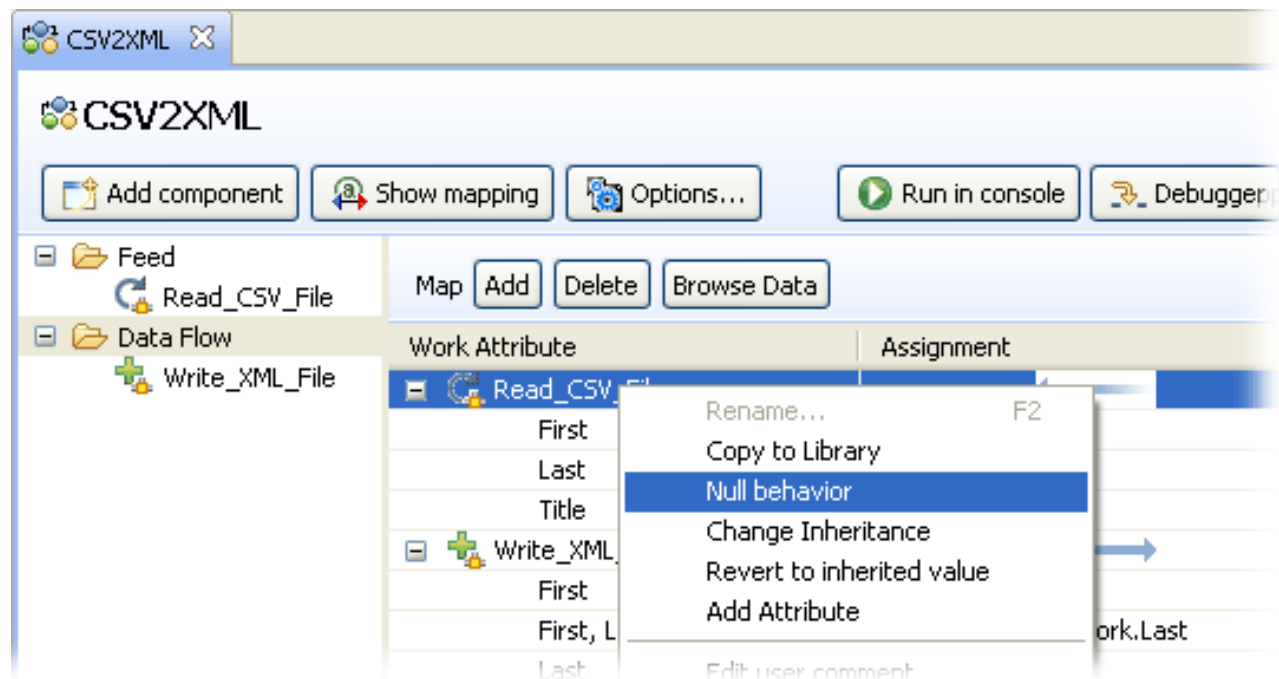


Figure 32. Null Behavior button for AL-level configuration

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

21. 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.

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.

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.

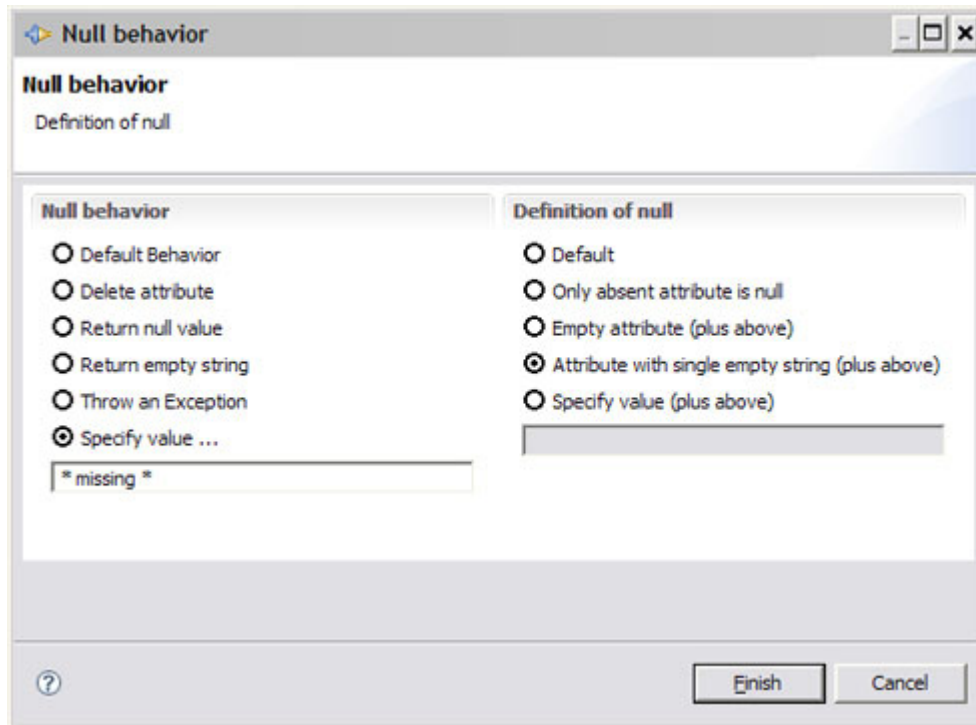


Figure 33. Null Behavior configuration dialog

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'.

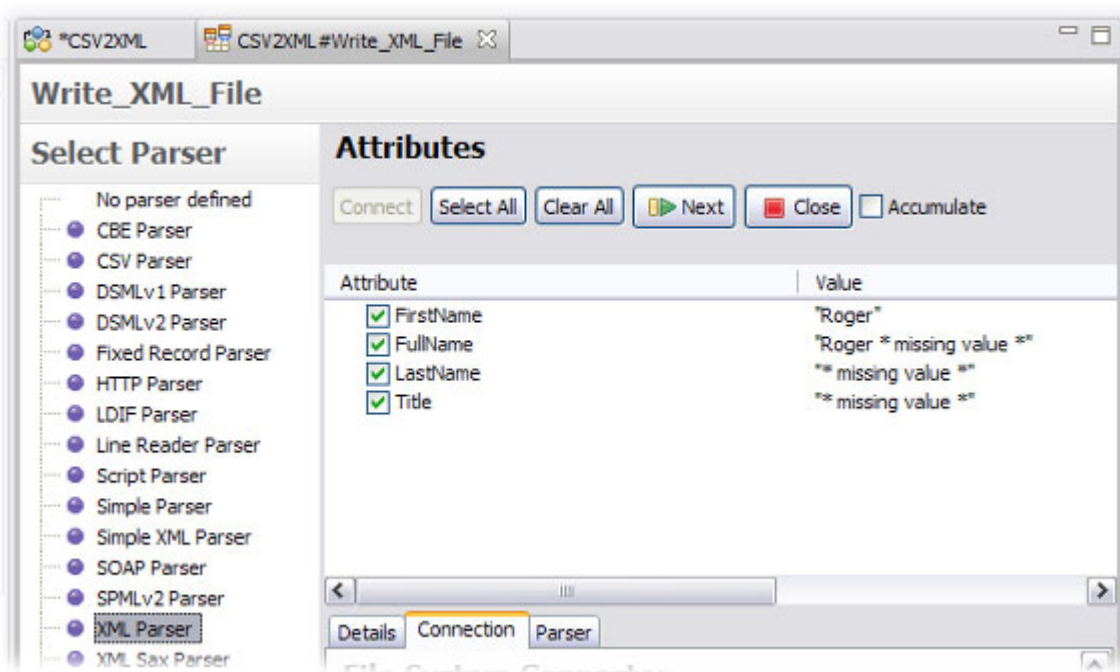


Figure 34. Result in the XML output of Null Behavior settings

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'<sup>22</sup> and press **Finish**.

22. 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.

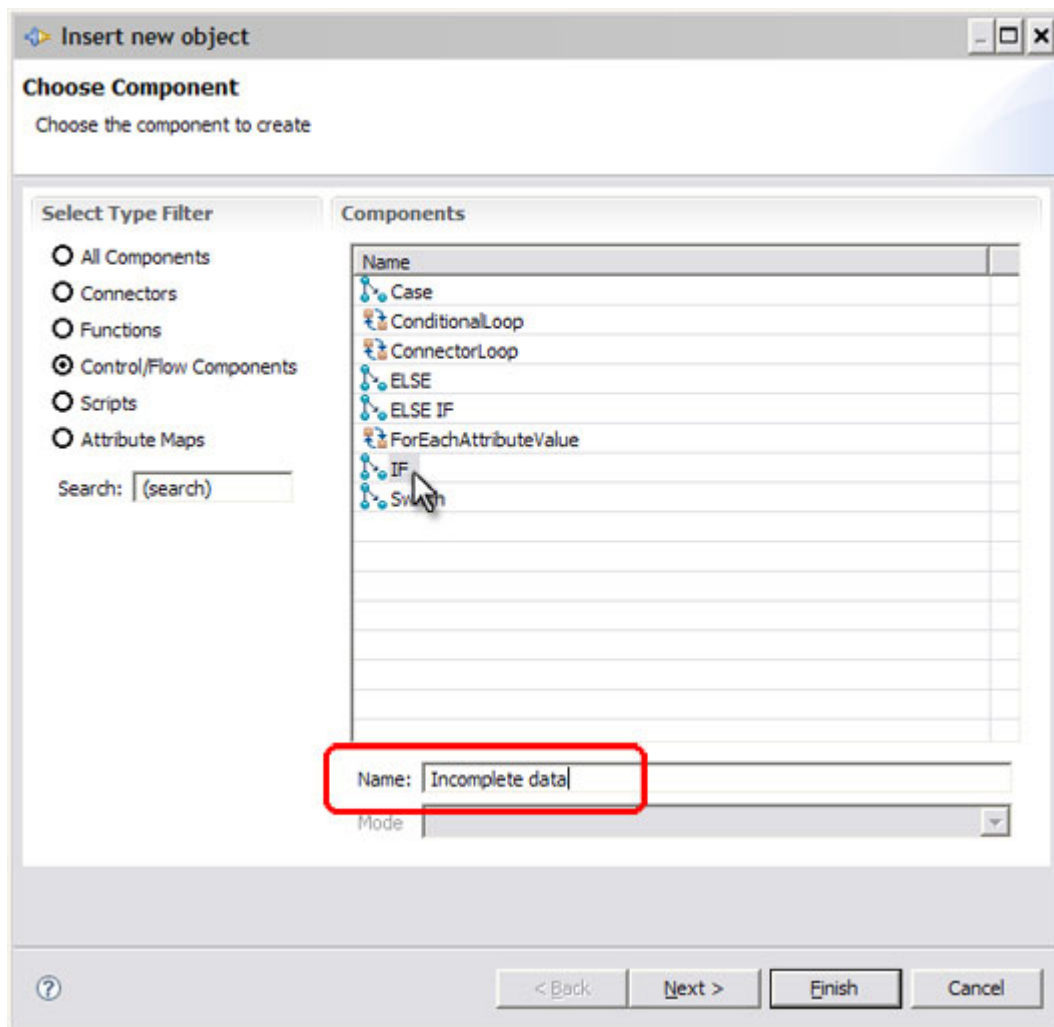


Figure 35. Selecting the IF branch component

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

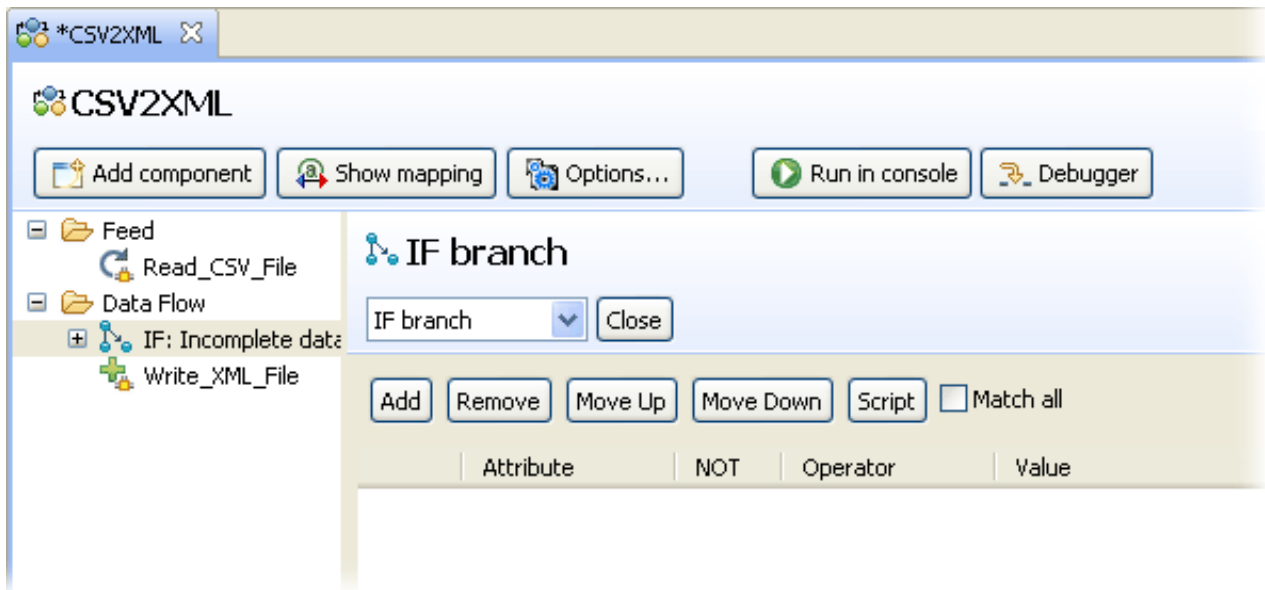


Figure 36. Editing conditions for the IF branch

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.

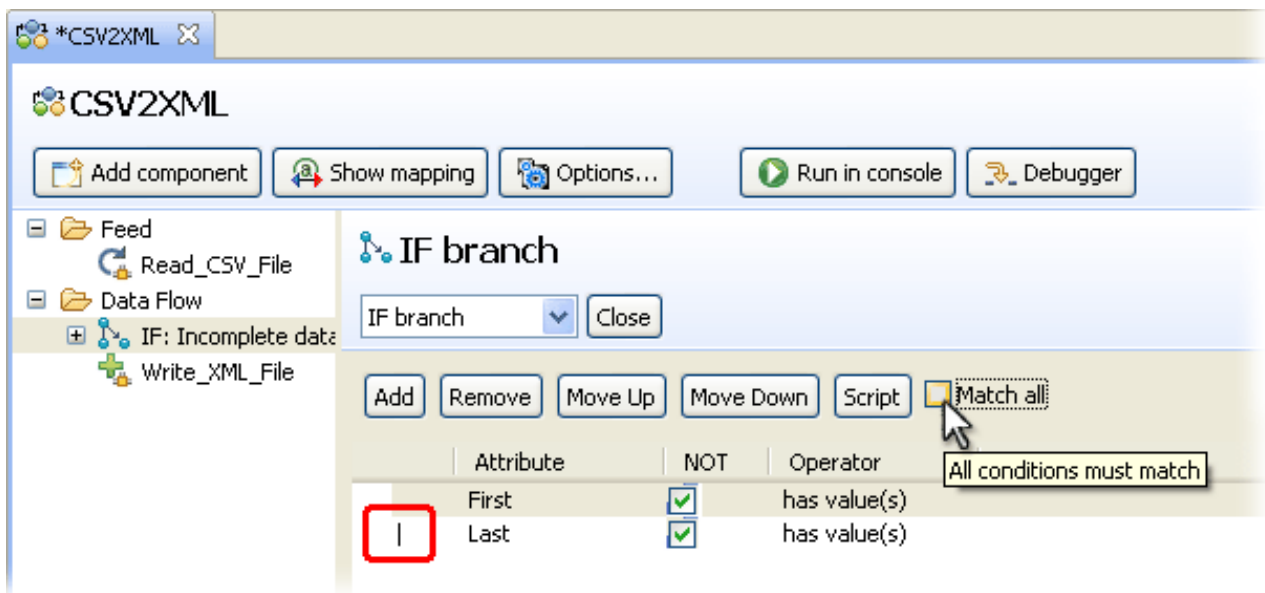


Figure 37. Adding simple Conditions to the IF branch

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<sup>23</sup>:

```
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 **Add 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 **Add Component...** Choose **Scripts** and then select the SC called 'Exit Flow'. Now your AL should look like this:

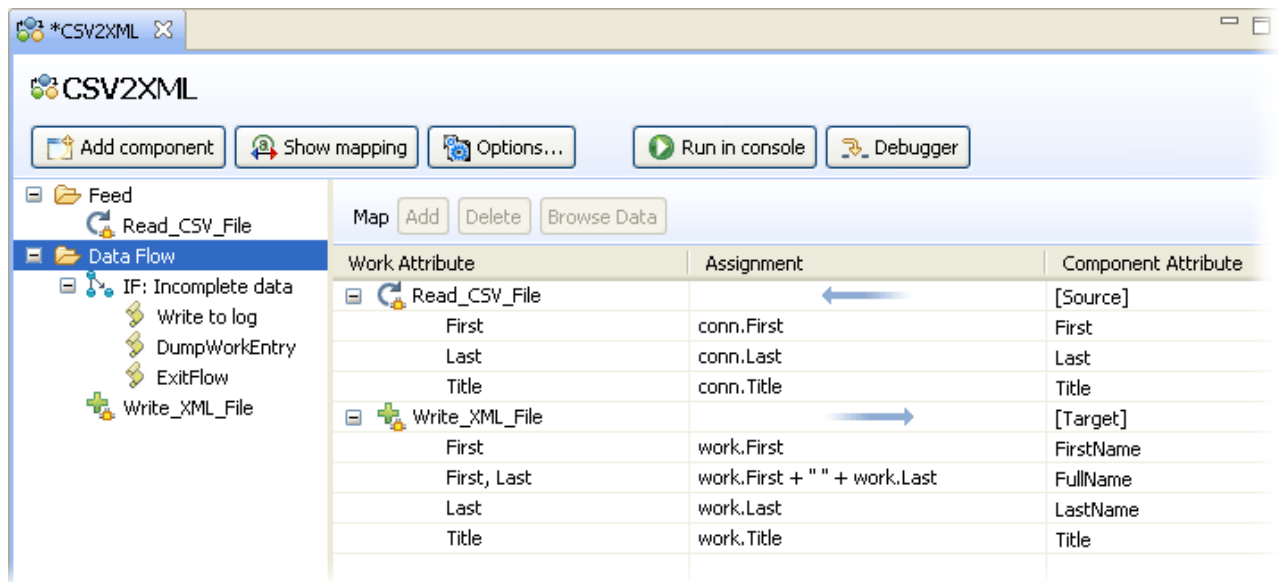


Figure 38. Your first complete AssemblyLine

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

23. 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.

never evaluate to *true*.

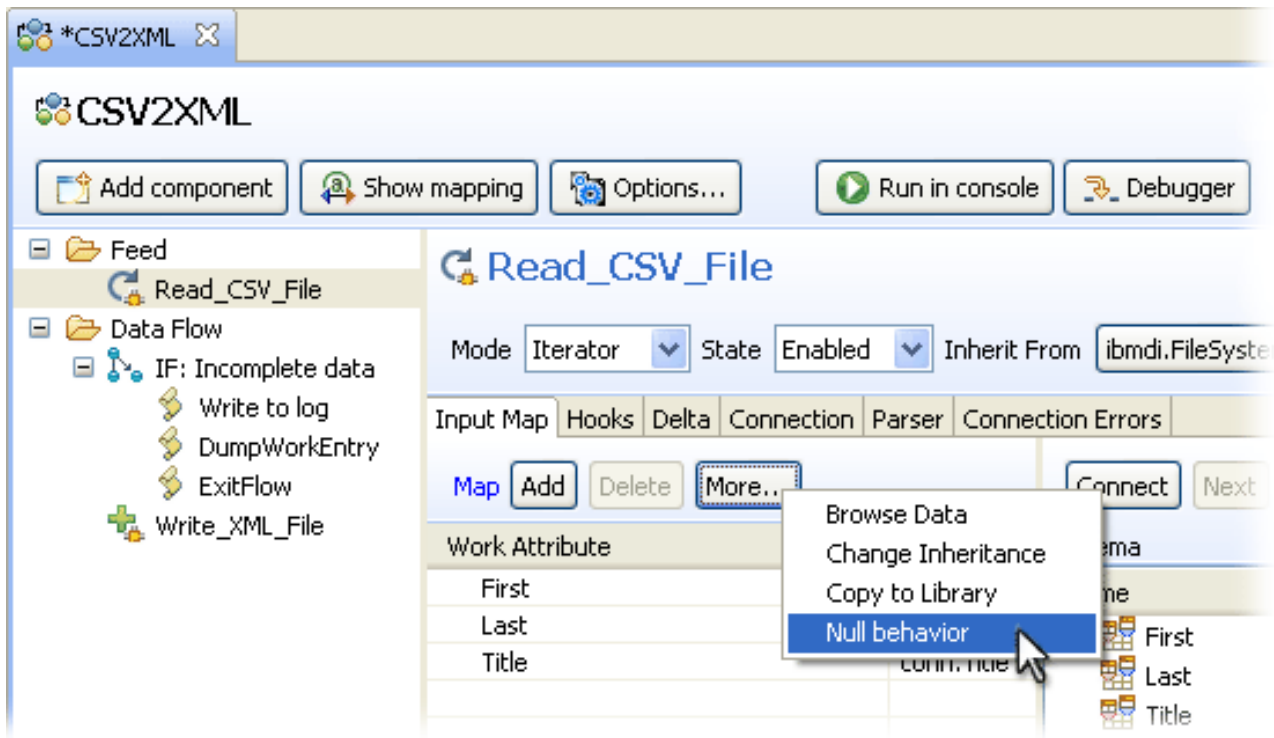


Figure 39. Resetting Null Behavior for the AssemblyLine

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 11
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 40. 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.

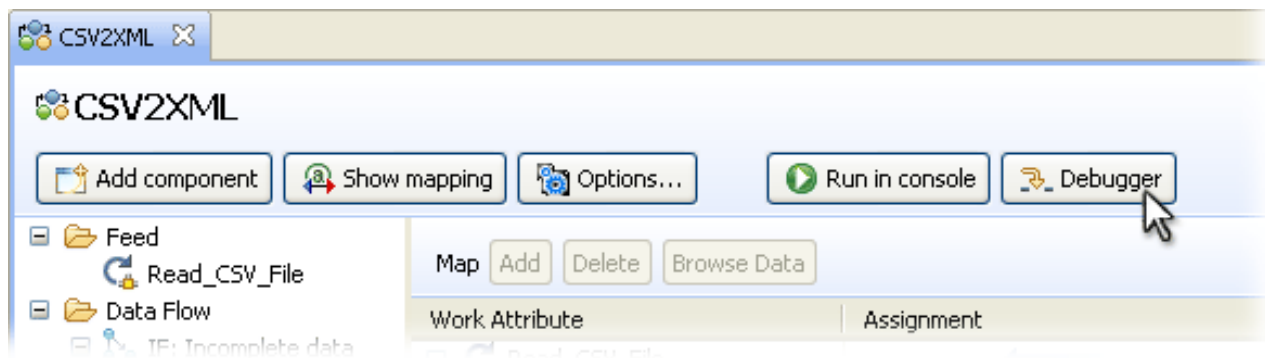


Figure 41. Debugging your AssemblyLine

Instead of the standard Run Console window, you will find yourself in the Debugger.



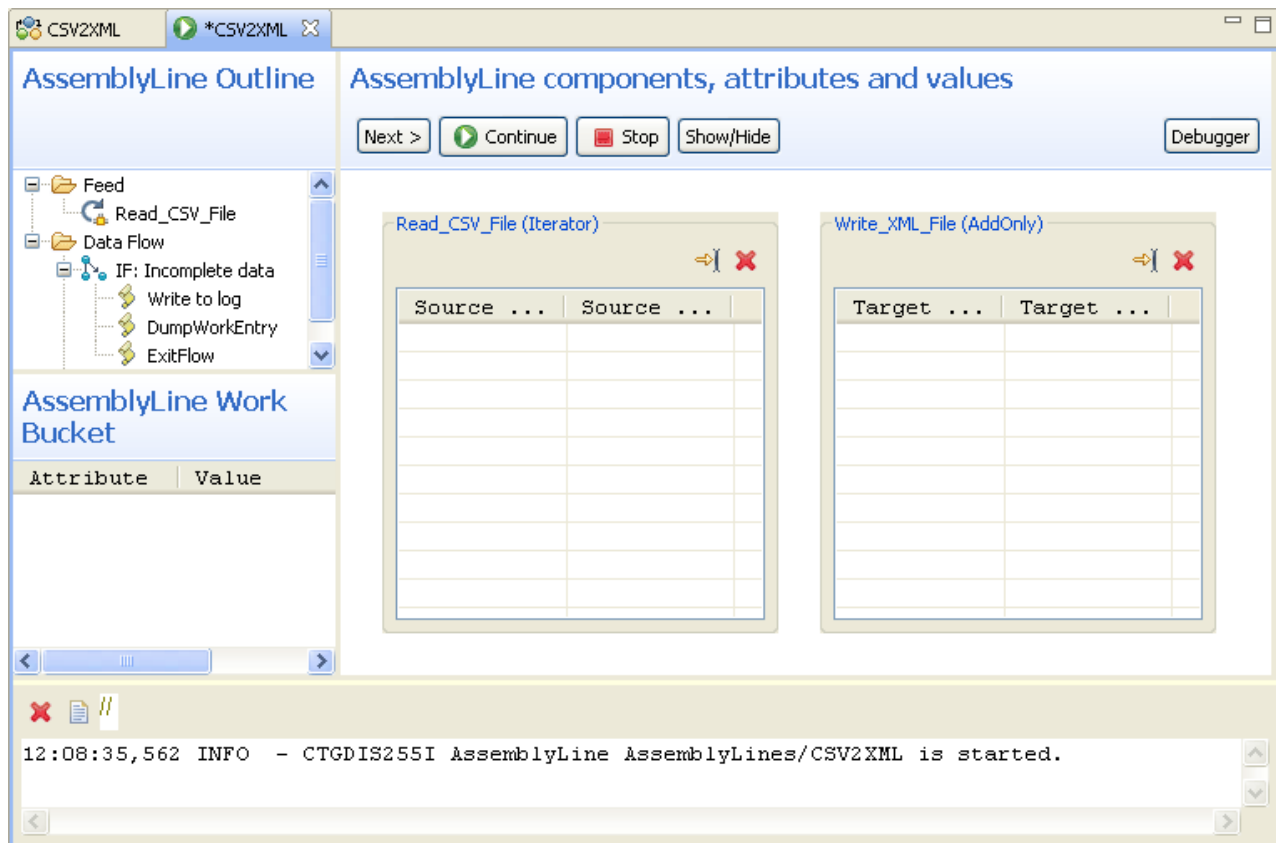


Figure 42. The AssemblyLine Data Stepper

The AssemblyLine Debugger offers two modes: the Data Stepper, which provides simple, straightforward testing features, and the advanced AL Debugger where you can dig deeper – like stepping through scripts, interactively working with Java libraries and modifying data on-the-file.

The Data Stepper is a useful tool for stepping through the execution of your AssemblyLine Connectors and viewing the data read, written and transformed. This screen is divided into three main areas:

- The **AssemblyLine Outline** shows your AL, highlighting where execution is paused;
- The **AssemblyLine Work Bucket** which displays all Attributes mapped into the Work Entry – that is, those found in Input Maps or Attribute Map components.
- **AssemblyLine Components, Attributes and Values** where you have a button row for controlling your debug session and a set of data display grids for all Connectors in the AL.

Along the bottom you can see a Console output window that shows the same information that you got when you ran your AssemblyLine using the Run button.

At this point your AssemblyLine has been dispatched to the test Server and is ready to start running at your command. Press the Next button to begin stepping.

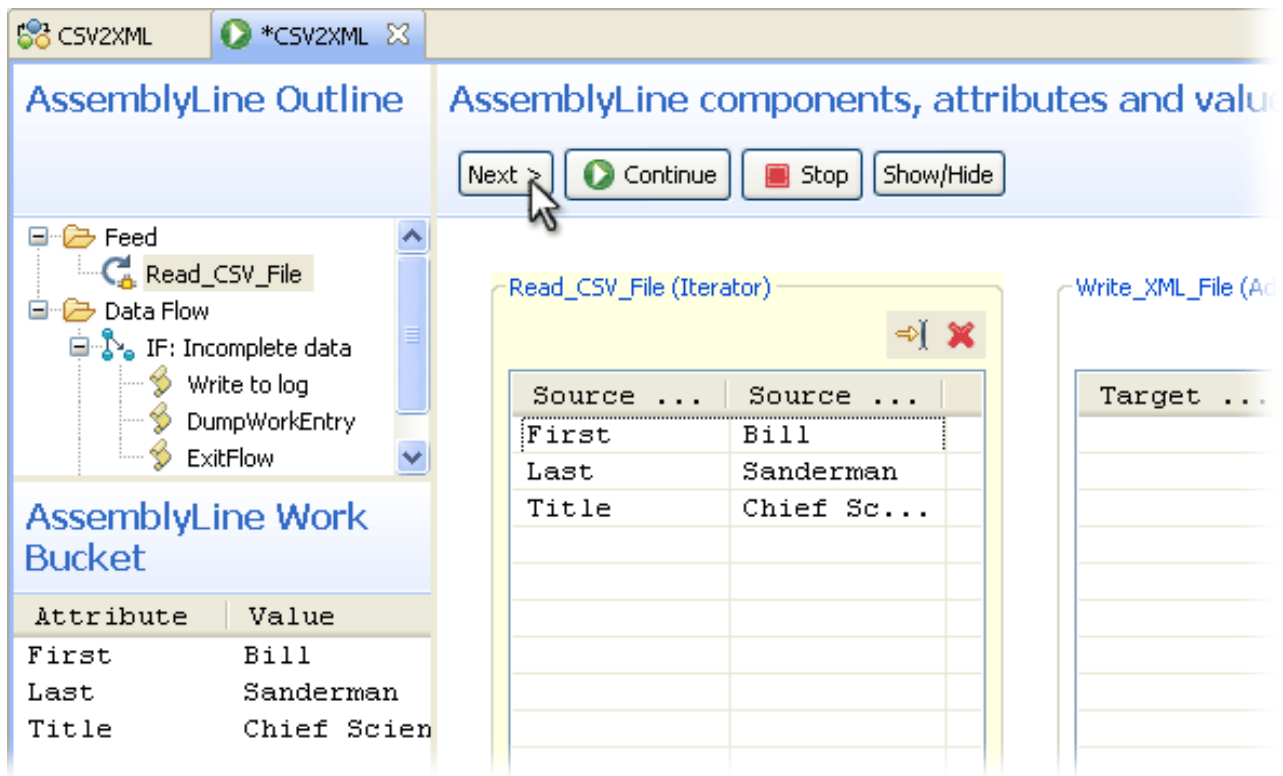


Figure 43. Stepping into the AL run

Notice how three things happen onscreen: the **AssemblyLine Outline** shows that the "Read\_CSV\_File" Connector is currently active; the **AssemblyLine Work Bucket** displays the Attributes just read by this Connector; and the data display grid for this Connector is also populated with these Attributes. Each time you press the **Next** button execution continues to the step and the information displays are refreshed.

You can also use the **Run To Here** button at the top of the data grid for a Connector to jump to this point. Do this now for the "Write\_XML\_File" Connector.

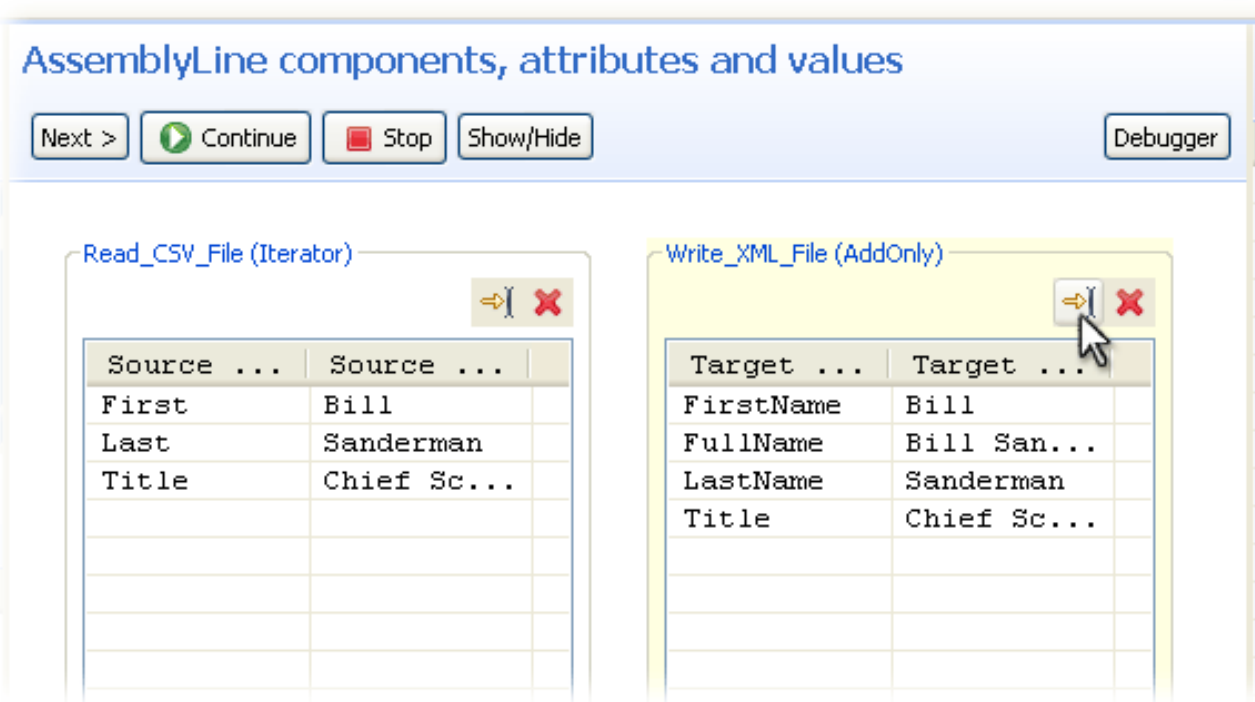


Figure 44. Stepping to the Write\_XML\_File Connector

This data grid then displays the Attributes in the Output Map of this Connector along with their values – including the computed value for "FullName".

Now let's look at the Data Stepper toolbar buttons to see what options you have:

- **Next >** moves processing to the next component and updates all data display areas;
- **Continue** will cause your AL to continue execution until it is finished;
- **Stop** terminates the run immediately;
- **Show/Hide** lets you decide which Connector data grids are displayed;
- **Debugger** switches you to Full Debugger mode where you can step through script code, set breakpoints, view and modify any Attributes and script variables, and interactively execute JavaScript commands in the context of your running AssemblyLine.

Although the Data Stepper provides a wealth of information about how your AssemblyLine will perform, sometimes you need the added power of the advanced Debugger. Note that you can switch between Stepper and Advanced modes as often as you want during a debugging session. Try it now by pressing the **Debugger** button located at the far right of the Data Stepper button row.

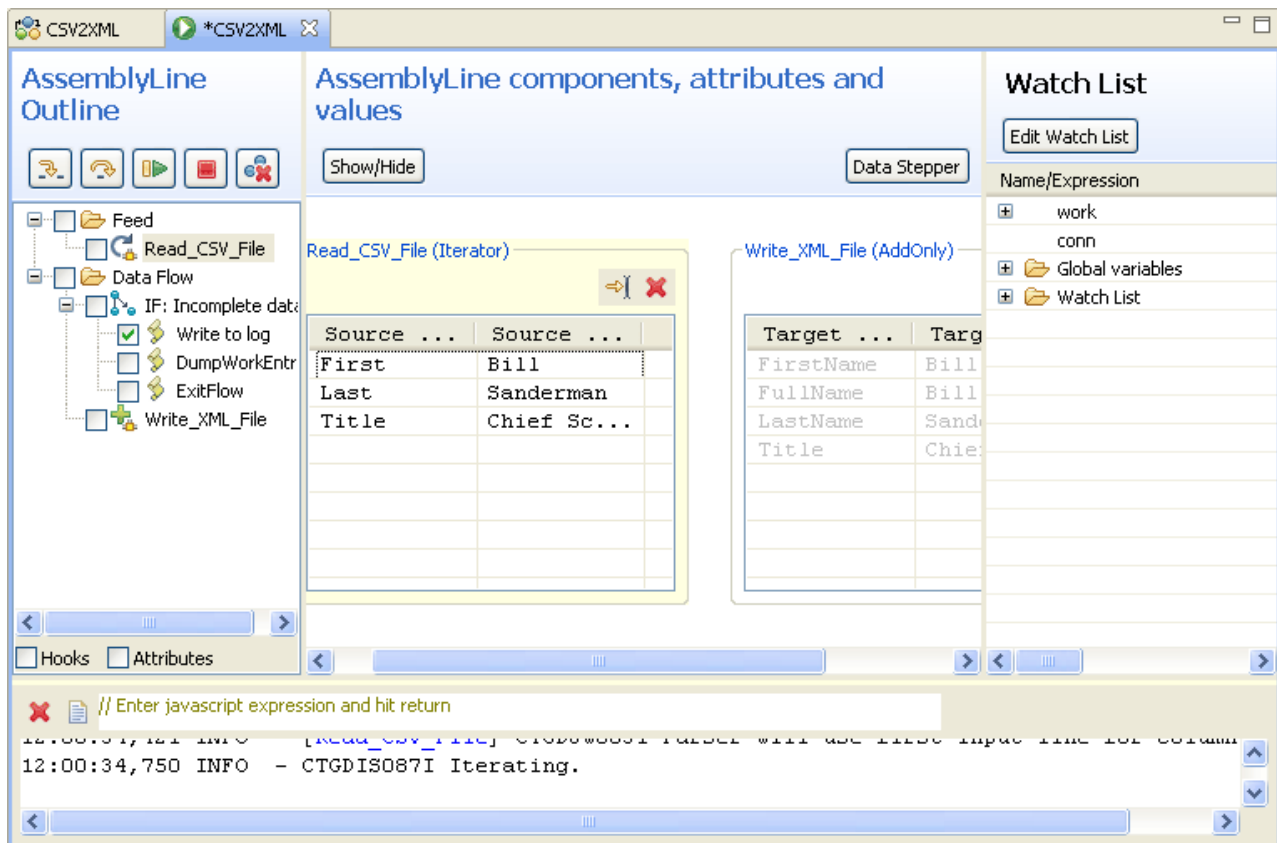


Figure 45. Advanced Debugger mode

When you switch modes the screen is redrawn to provide new controls for the **AssemblyLine Outline** and the **AssemblyLine Work Bucket** is replaced with the **Watch List** on the right side of the window. The **Watch List** shows the standard Attribute *buckets*: **work** and **conn**. There is also a folder called "Global variables" that if opened displays all variables defined for your AssemblyLine: both the built-in ones like **work** and **system**, plus any that you define in your script code. The last Watch folder is for your own use and you can add variables or entire JavaScript expressions that you want to watch by using the **Edit Watch List** button at the top of this panel.

Turning our attention to the **AssemblyLine Outline**, the boxes next to components in this tree-view are called Breakpoints and you can tell TDI to pause at any component during execution by clicking on one of these. You can also right-click on any node in and select **Run and break here** to bring AL execution to this point. The toolbar above the outline gives you some of the same controls that you had in the Data Stepper, plus a couple of new ones:

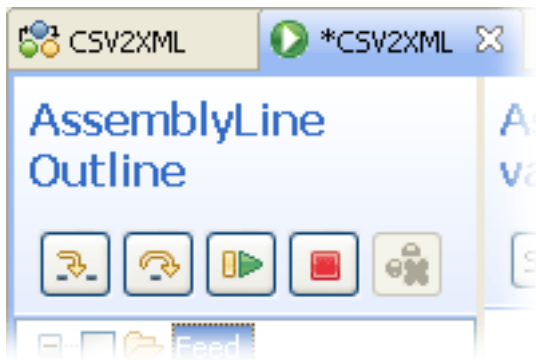


Figure 46. Debugger buttons

Starting from the left these buttons are:

- **Step Into** which allows you to step into Attribute Maps, scripts and even into the underlying workflows of the AssemblyLine and its components. These waypoints in the built-in flows are called "Hooks" are covered in a later exercise;
- **Step Over** is the same as the **Next >** button you saw in the Data Stepper. In the Debugger it also lets you stop over script function calls instead of into them;
- **Continue** causes the AL to run until completion (just like in the Data Stepper) or until a Breakpoint is reached;
- **Stop** halts your AssemblyLine, as it does in the Data Stepper;
- **Clear All Breakpoints** removes any Breakpoints that you have set for your AL.

To get a feel for how Breakpoints work try setting one for the "Write to log" script by clicking in the box next to this component.

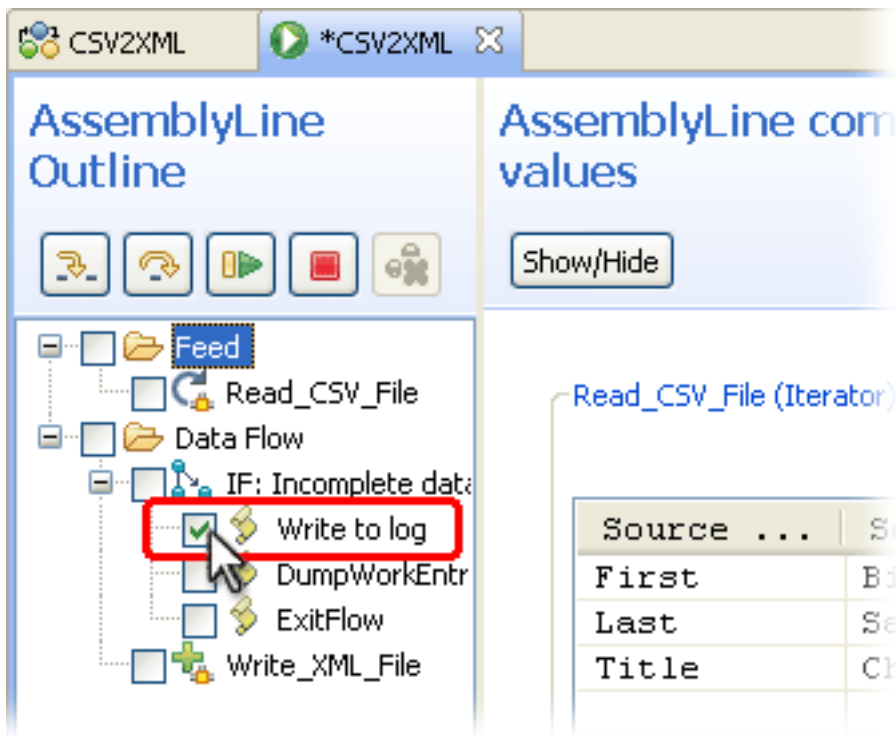


Figure 47. Setting a breakpoint

Now press the **Continue** button and your AL will run until the IF-Branch is *true* and you find control at the "Write to log" Script. TDI also opens up a Script area allowing you to step through the code here. You can even set Breakpoints at any script line by double-clicking in the margin to the left of that line.

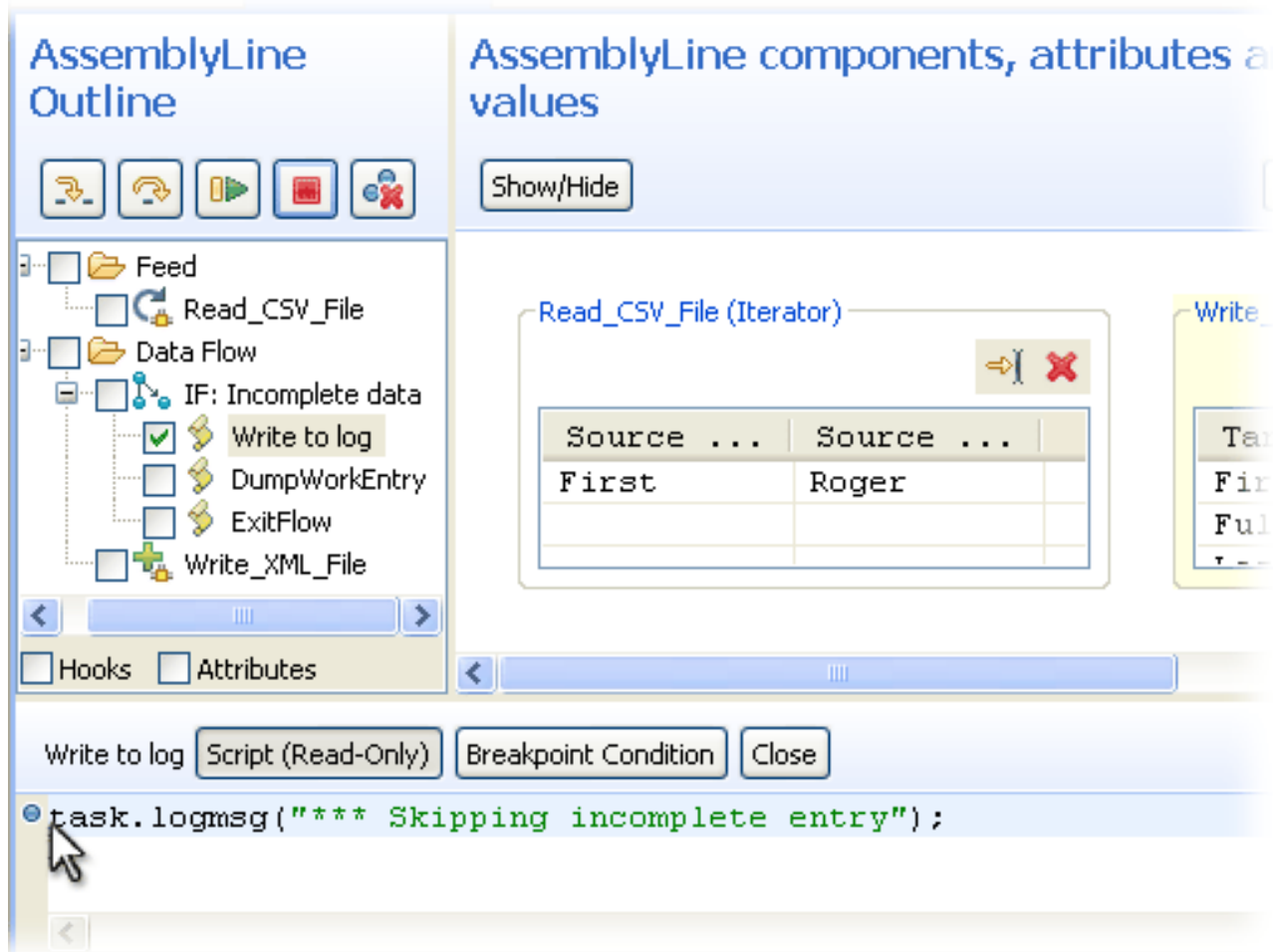


Figure 48. Setting a Breakpoint in script

Furthermore, you can double-click on any node in the component list to bring up the Debug display. As you can see in the figure above, there is a button titled **Breakpoint Condition**. You can use this to set a JavaScript expression that must evaluate to either *true* or *false* and which will determine if a Breakpoint is active or not. For example, the Breakpoint shown above could be set to be true if:

```
work.First.startsWith("R")
```

or

```
mycounter > 1000
```

This is very handy for debugging issues that only occur deep in some input data set.

And if for some reason you need to go back to a previous step then simply stop and restart your debug session. You can also switch back to the Data Stepper by pressing the Data Stepper button.

But before you leave the advanced Debugger there is one more feature worth noting: the **JavaScript Evaluation commandline**.

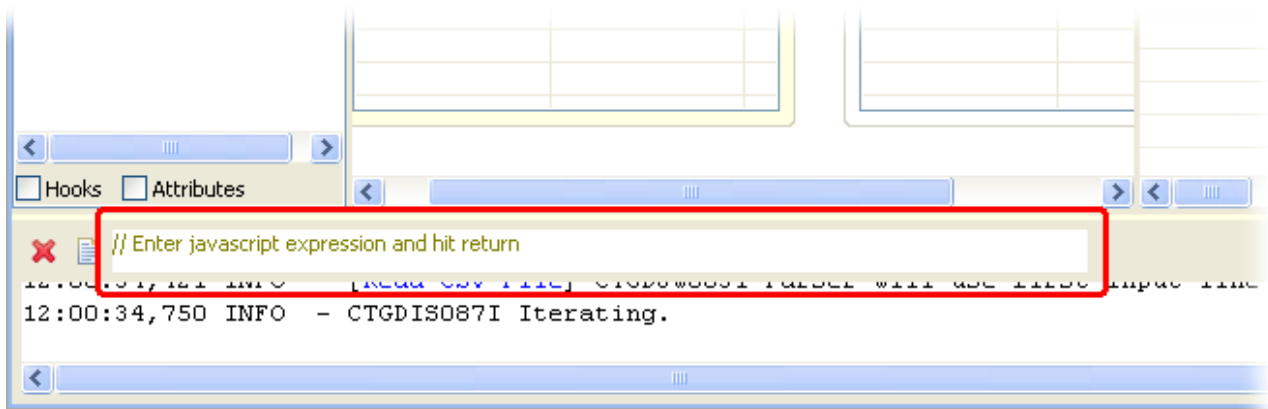


Figure 49. JavaScript Evaluation commandline

This innocent looking input field above the log output area allows you to execute any snippet of script in the context of your running AssemblyLine. Try it now by typing this command and pressing Enter:

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

This will display the contents of the Work Entry in the log output window. Now try this:

```
i = 42
```

You will see the following message appear in the log: `i=42 >> 42.0`

This tells you that you have defined a new variable ('i') with the value of 42. The expression itself evaluates (as all script statements do) to the value of the assignment. You can also change the values of variable and Attributes already defined in your AL, for example:

```
work.First="Rudy"
```

After executing this line then the value of the "First" Attribute will be "Rudy". The ability to modify data in-flight means that you can make sure your AssemblyLine steps into all branch logic, allowing you to thoroughly test your solution.

It is highly recommended that you spend some time to familiarize yourself with the AssemblyLine Data Stepper and Debugger. Not only does it provide unique insight into how your AL operates, including all the built-in workflows provided by the TDI Server kernel, but it will also 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.

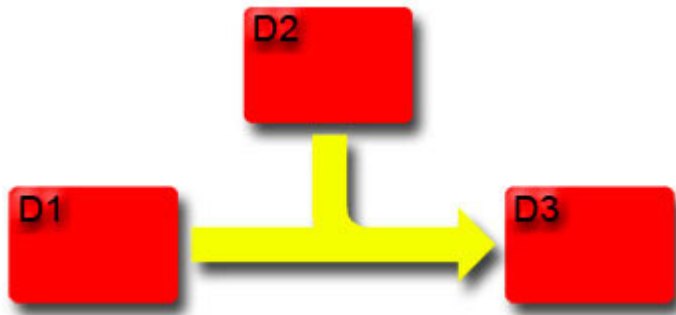


Figure 50. The scenario flow diagram

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.

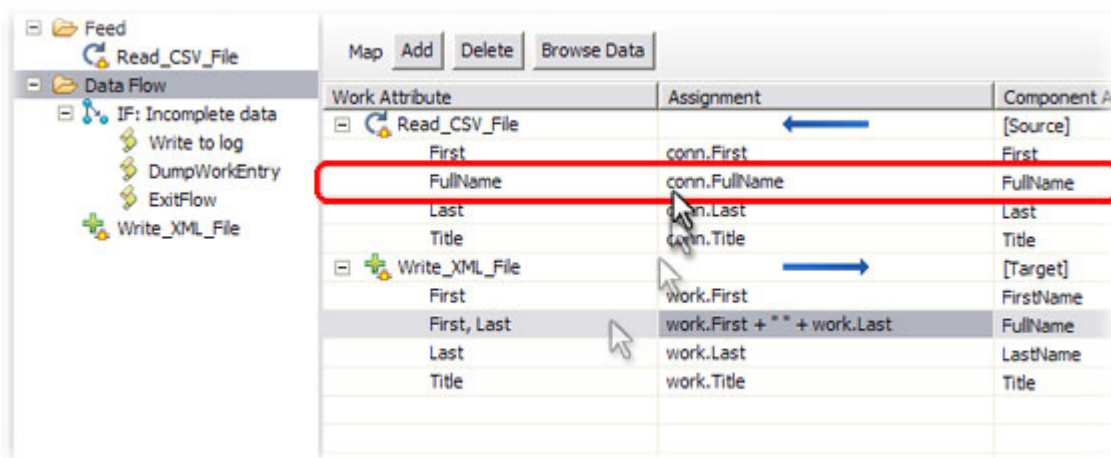


Figure 51. Dragging 'FullName' to the Input Map of your Iterator Connector

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
```



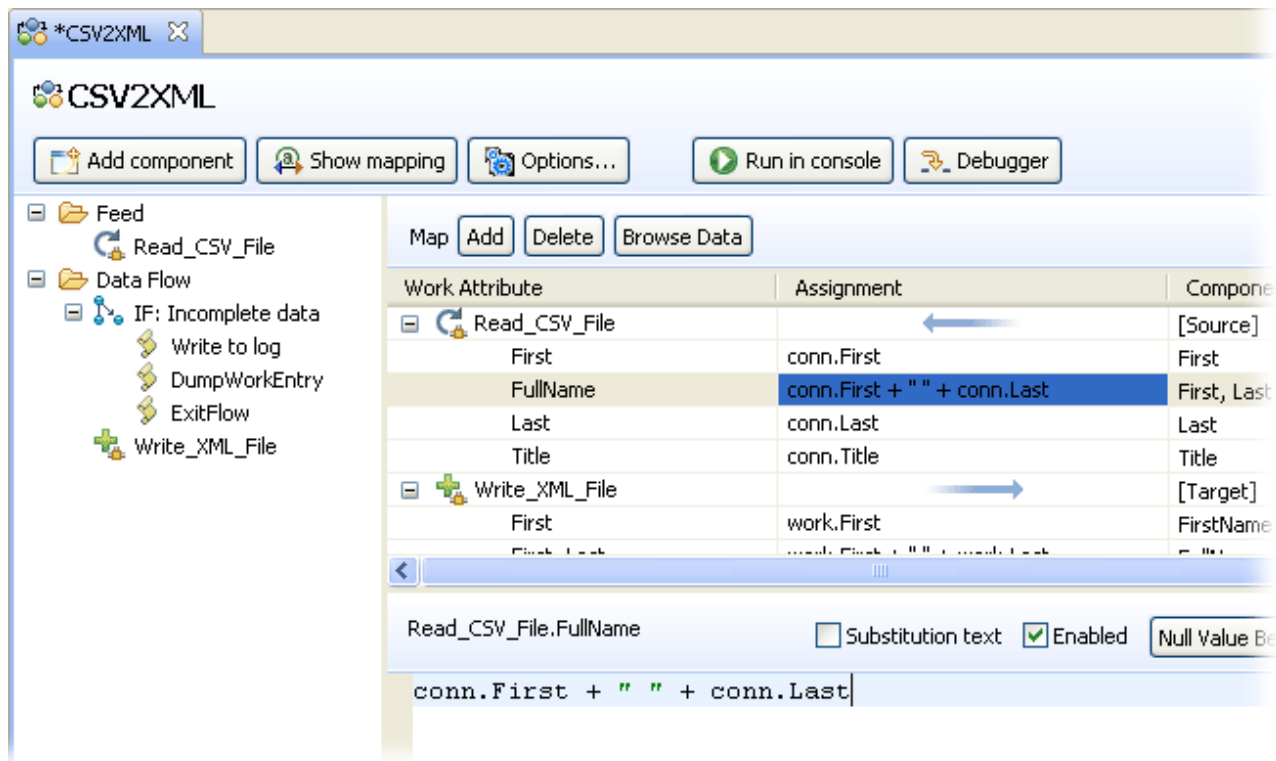


Figure 52. Editing the assignment for 'FullName'

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<sup>24</sup>.

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.

24. 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.

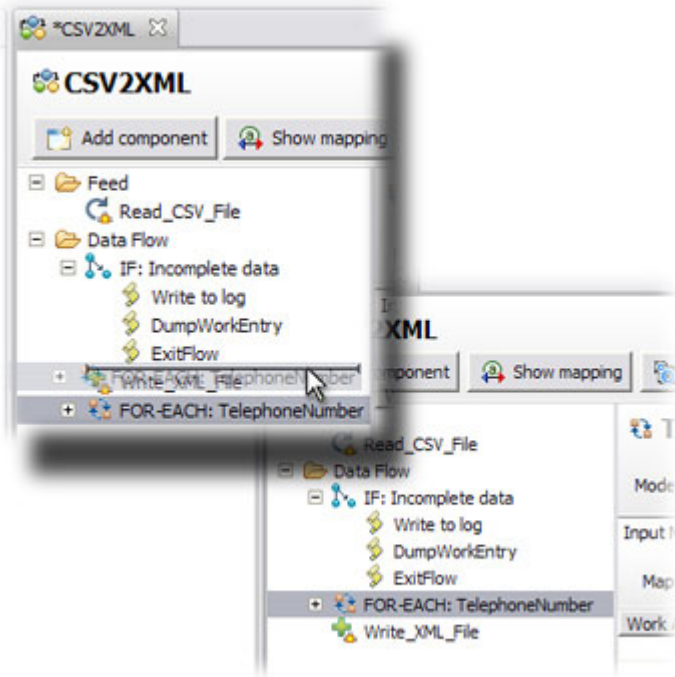


Figure 53. Drag the ConnectorLoop

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

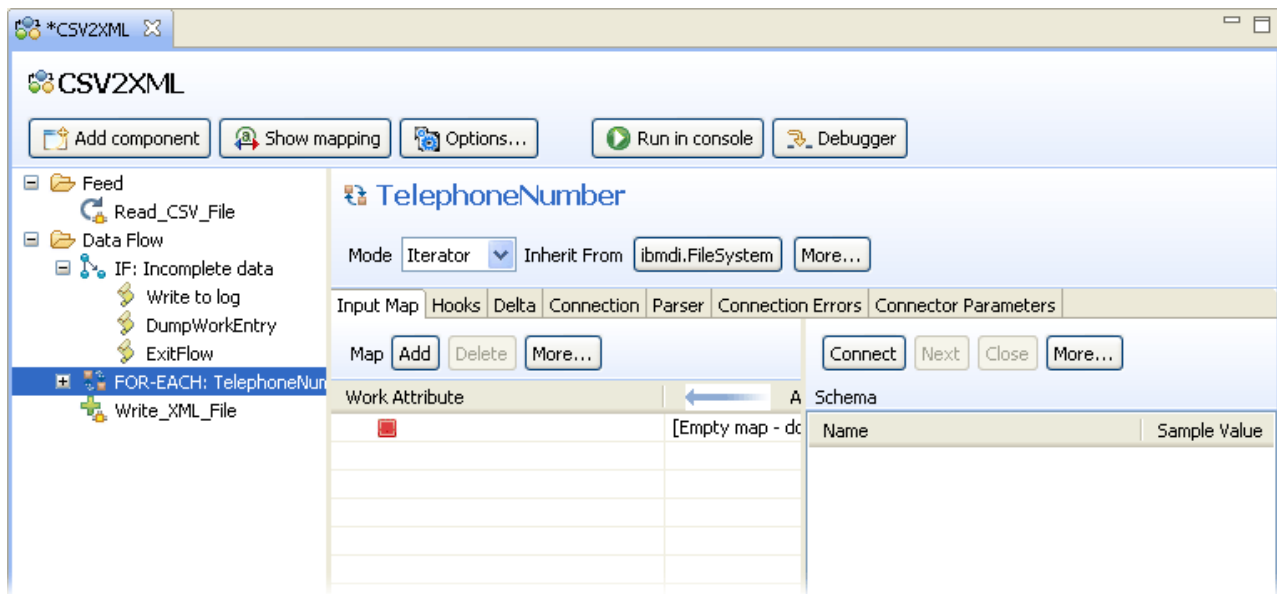


Figure 54. ConnectorLoop Configuration

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:

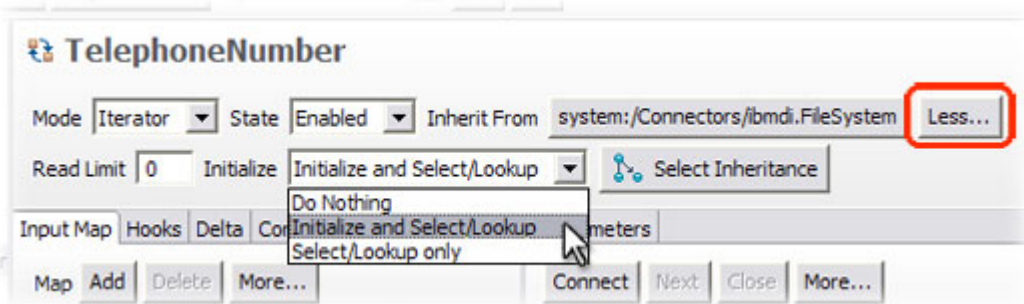


Figure 55. ConnectorLoop Advanced Settings

- **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 whenever the Loop starts to cycle. Use this option since your ConnectorLoop 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.

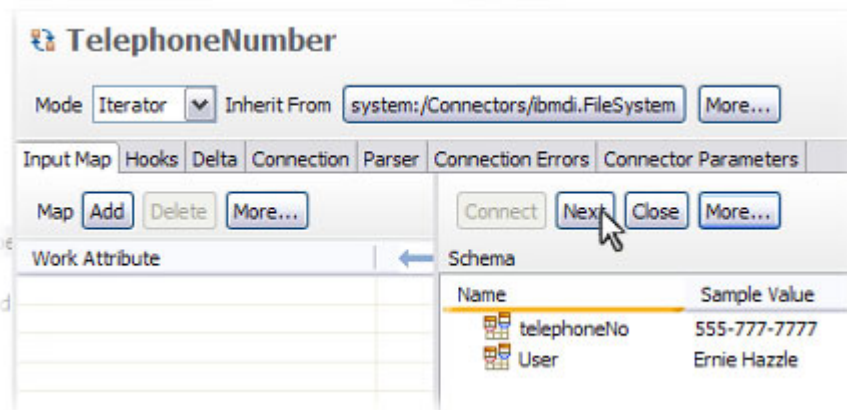


Figure 56. Hierarchical Attributes

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.

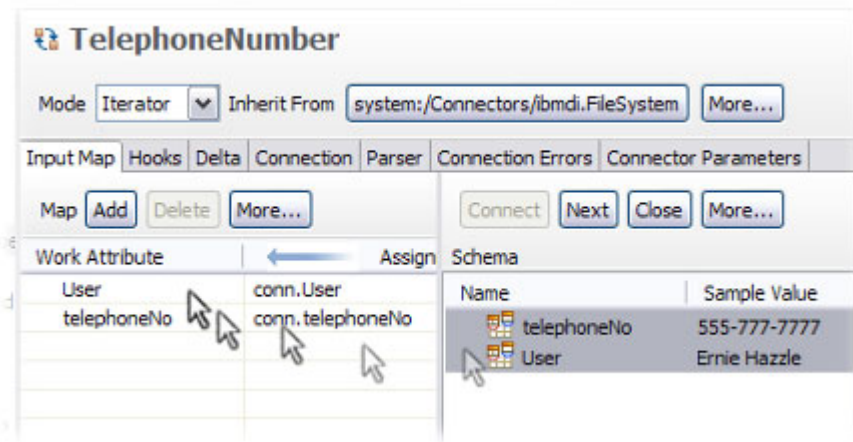


Figure 57. Dragging from Schema to Attribute Map

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 **Add Component...**. Call this IF branch 'Matching name found'. Now add a simple condition that checks if 'User' equals '\$FullName'<sup>25</sup>.

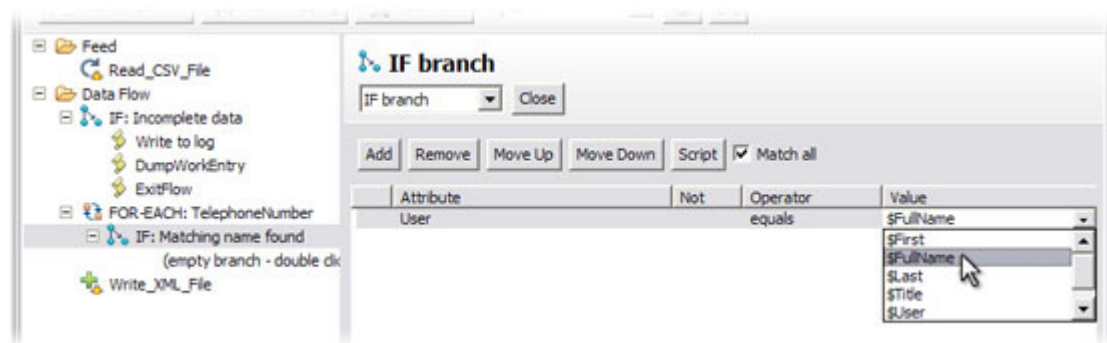


Figure 58. Condition editor for IF branch

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;
```

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

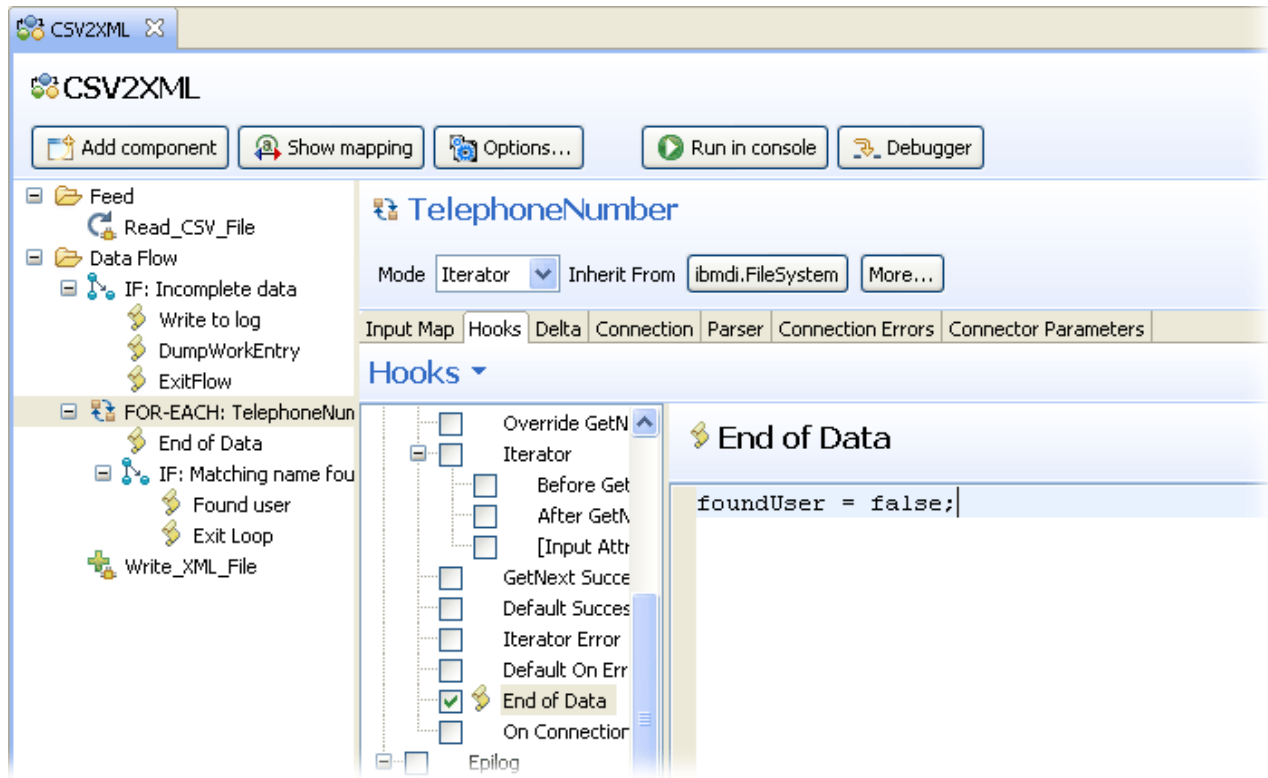


Figure 59. Scripting the End of Data Hook

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:

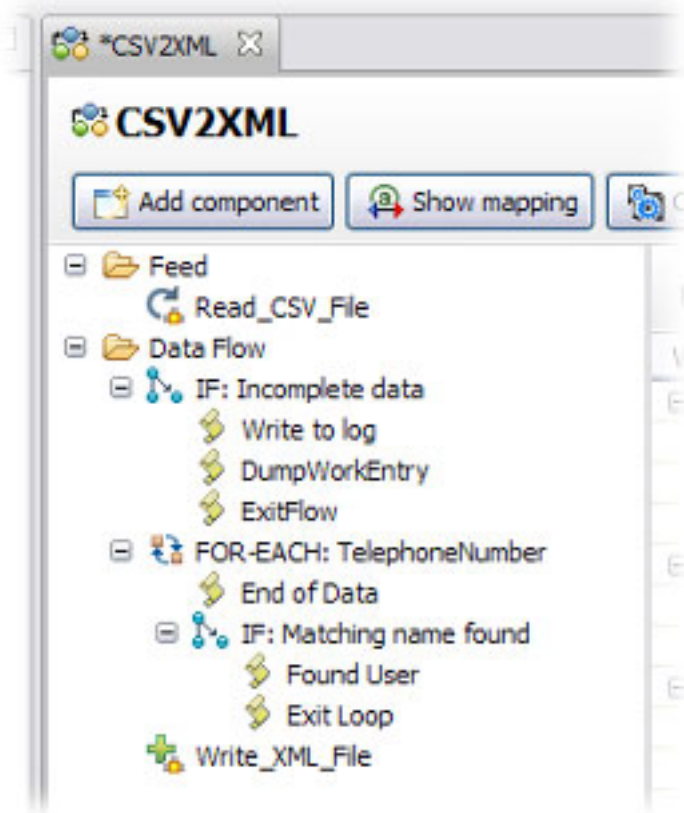


Figure 60. Component list in the AssemblyLine Data Flow section

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*.

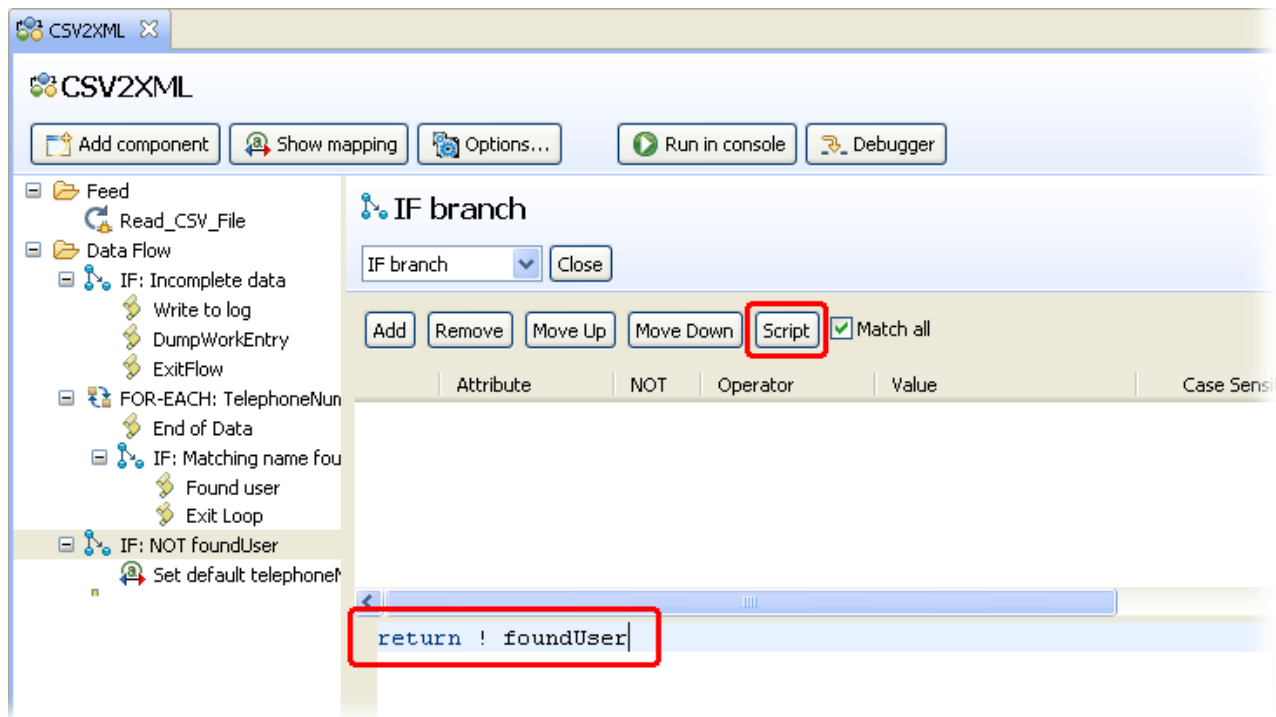


Figure 61. Scripting a Condition for the IF branch

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"<sup>26</sup>.

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.

26. 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.

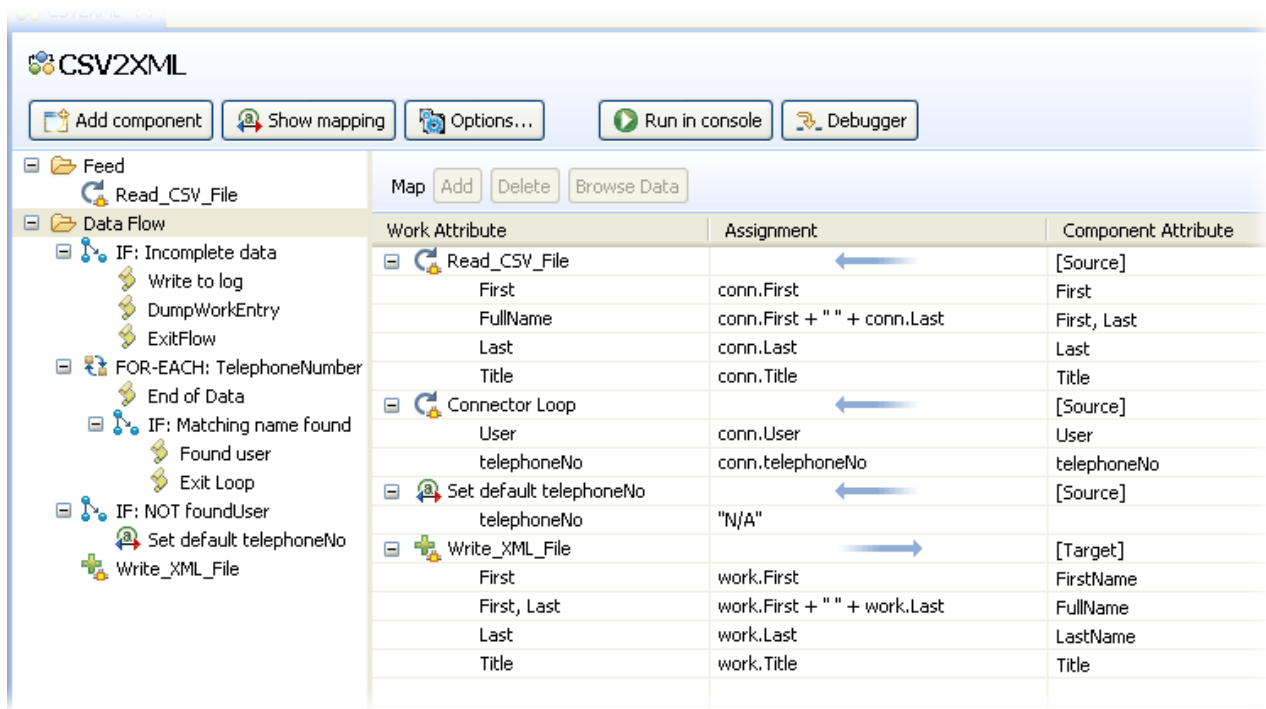
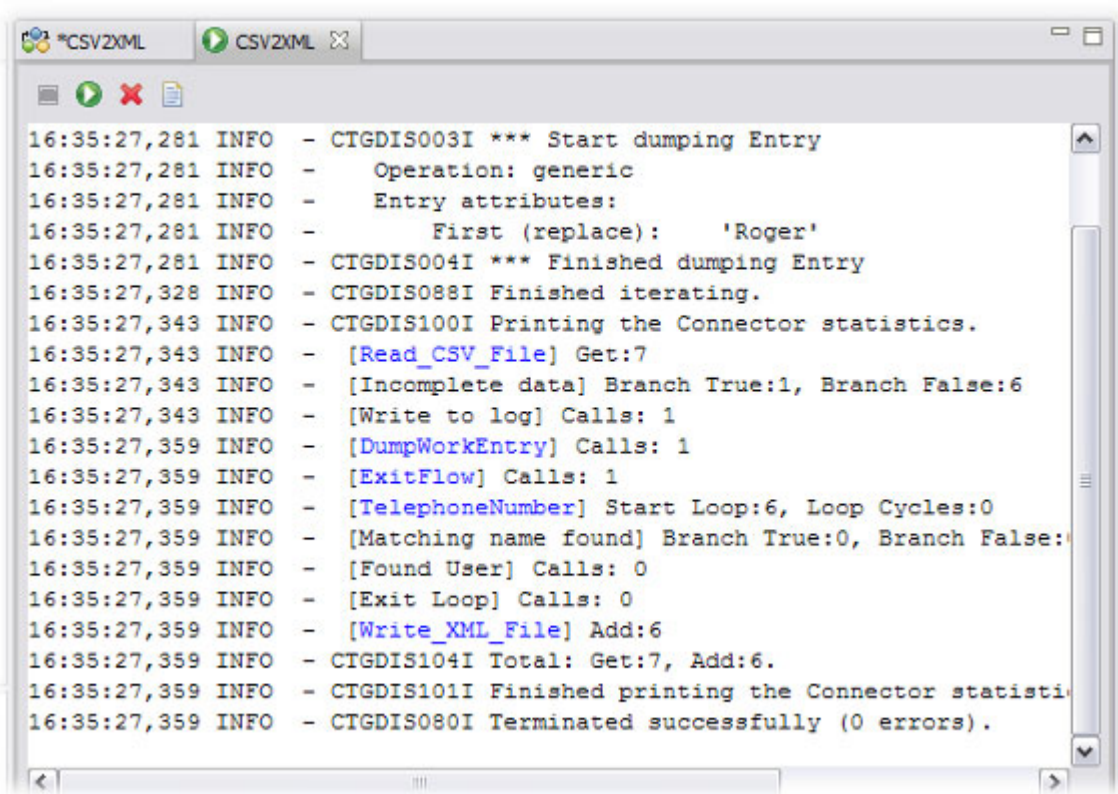


Figure 62. AssemblyLine complete with FOR-EACH Loop

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.



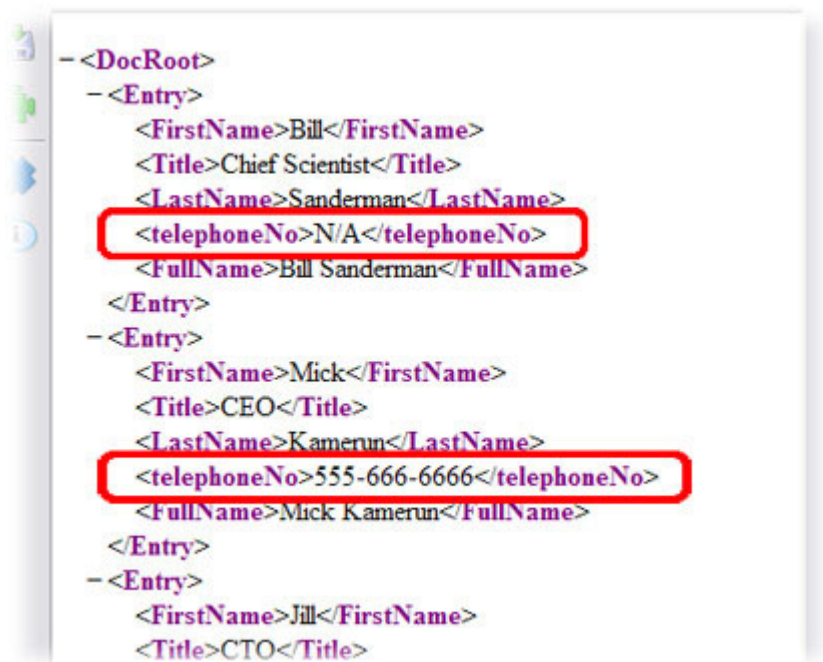


```
16:35:27,281 INFO - CTGDIS003I *** Start dumping Entry
16:35:27,281 INFO - Operation: generic
16:35:27,281 INFO - Entry attributes:
16:35:27,281 INFO - First (replace): 'Roger'
16:35:27,281 INFO - CTGDIS004I *** Finished dumping Entry
16:35:27,328 INFO - CTGDIS088I Finished iterating.
16:35:27,343 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:0
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 statistics.
16:35:27,359 INFO - CTGDIS080I Terminated successfully (0 errors).
```

Figure 63. 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 64. 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**.

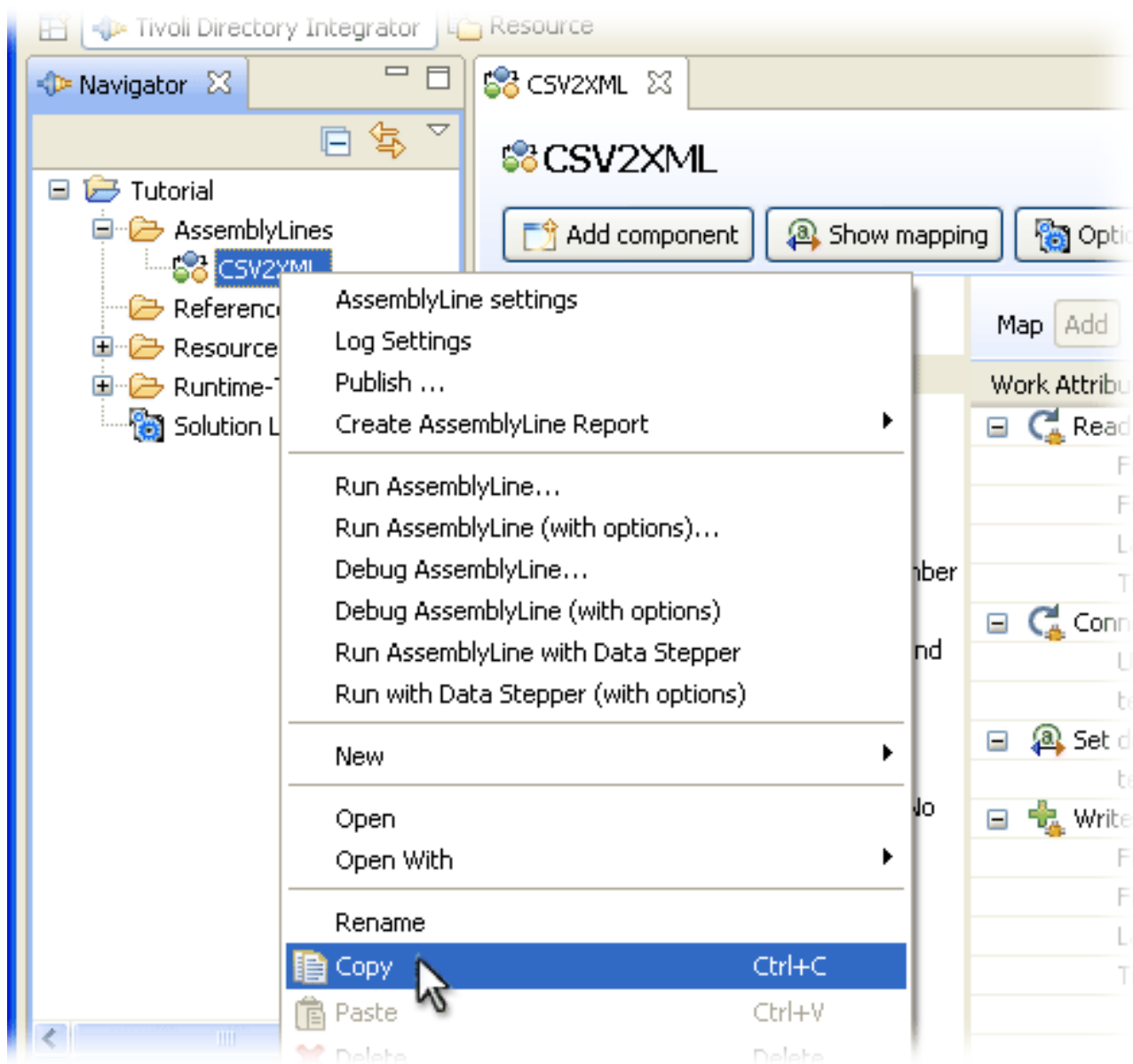


Figure 65. AssemblyLine Copy function

Now right-click on the 'AssemblyLines' folder in the Navigator and select the **Paste** option. Call this new AssemblyLine 'CSV2XML\_LookupMode' 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.

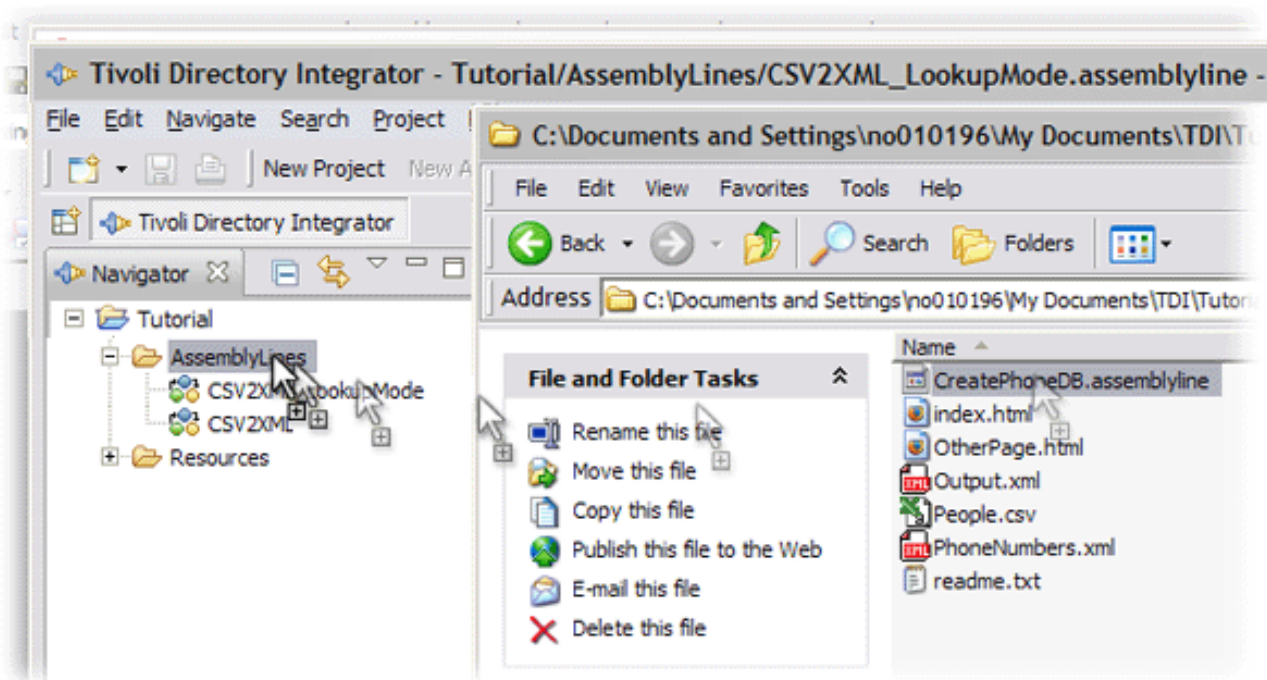


Figure 66. Copying an AssemblyLine into your project

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

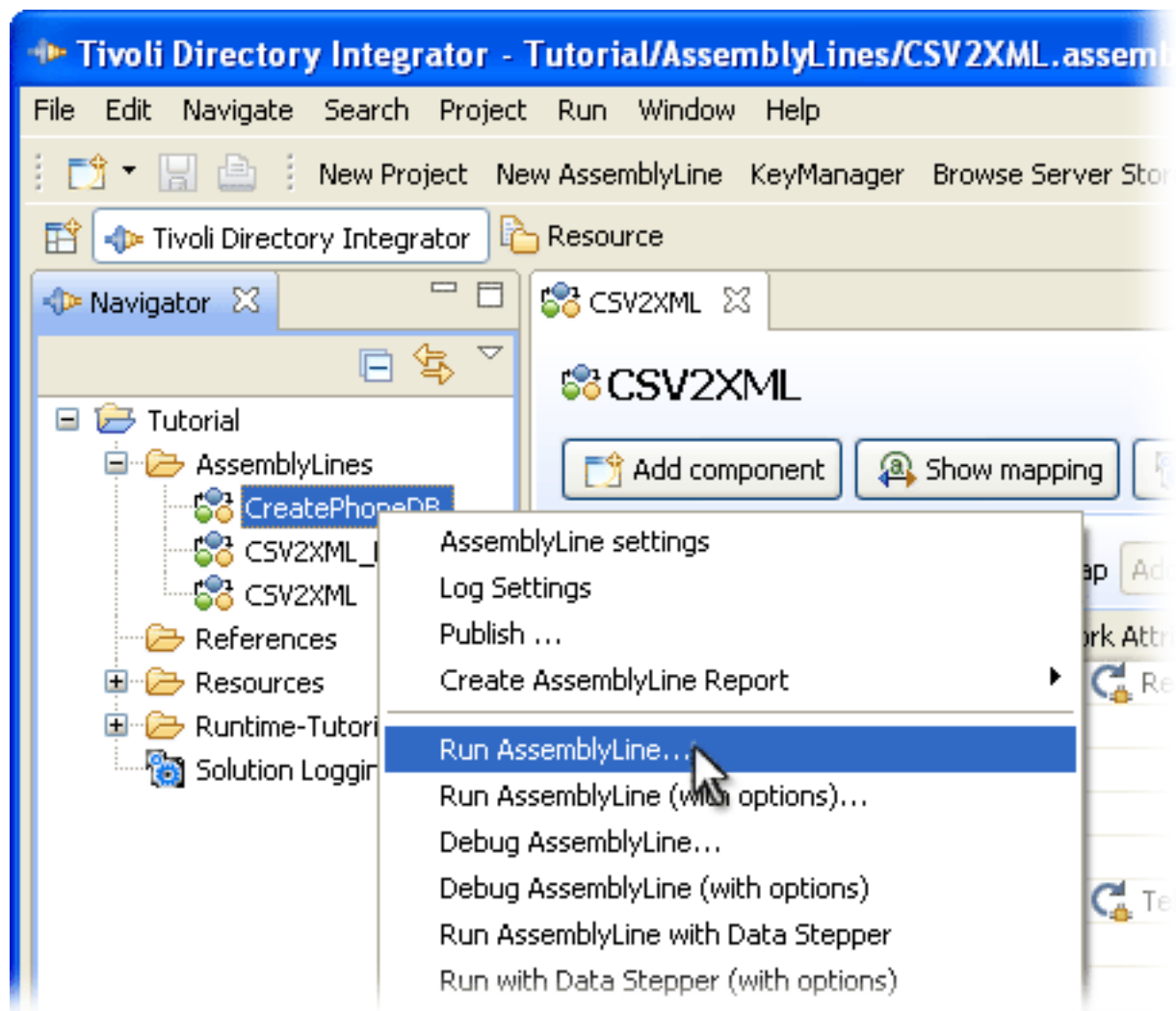


Figure 67. Run the CreatePhoneDB AL

This AssemblyLine will first create a Derby<sup>27</sup> 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<sup>28</sup>.

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

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

28. 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.

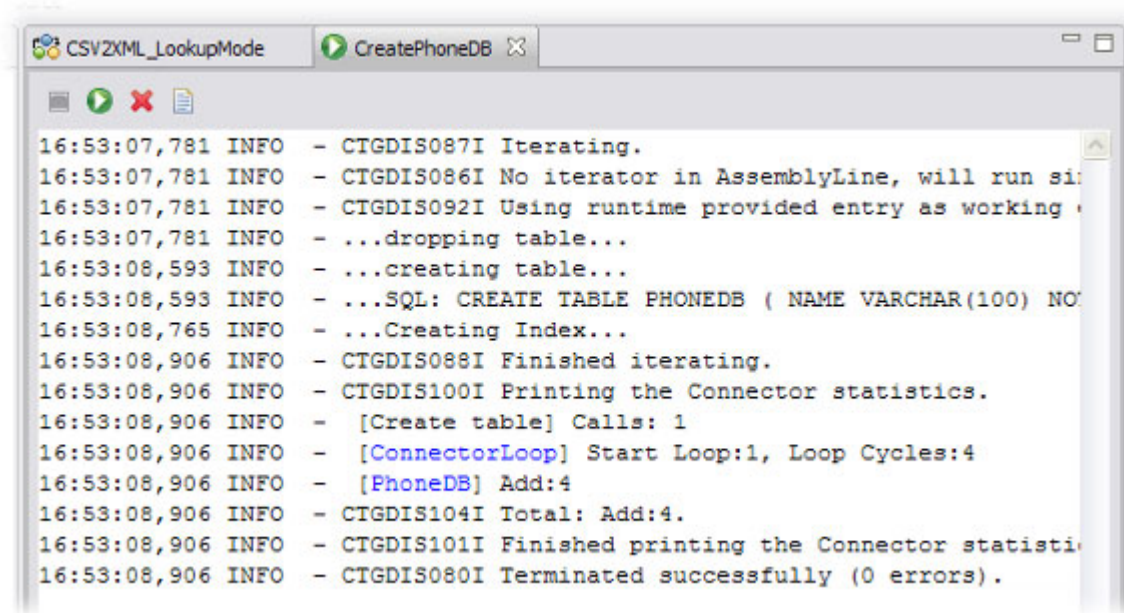


Figure 68. Log output from the 'CreatePhoneDB' AssemblyLine

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.

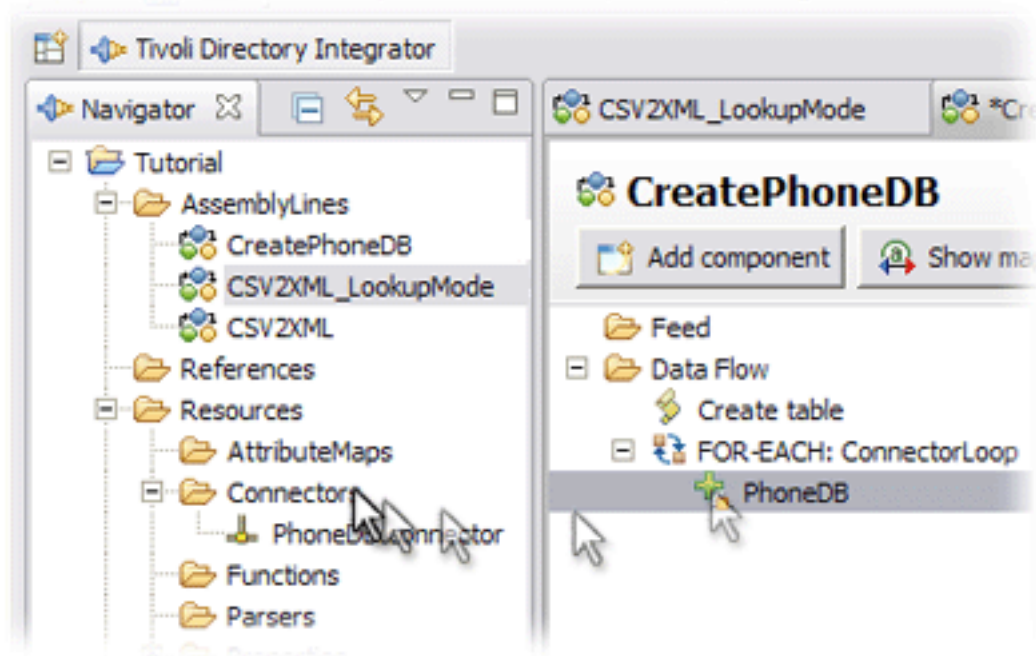


Figure 69. Drag a Connector to Resources



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.

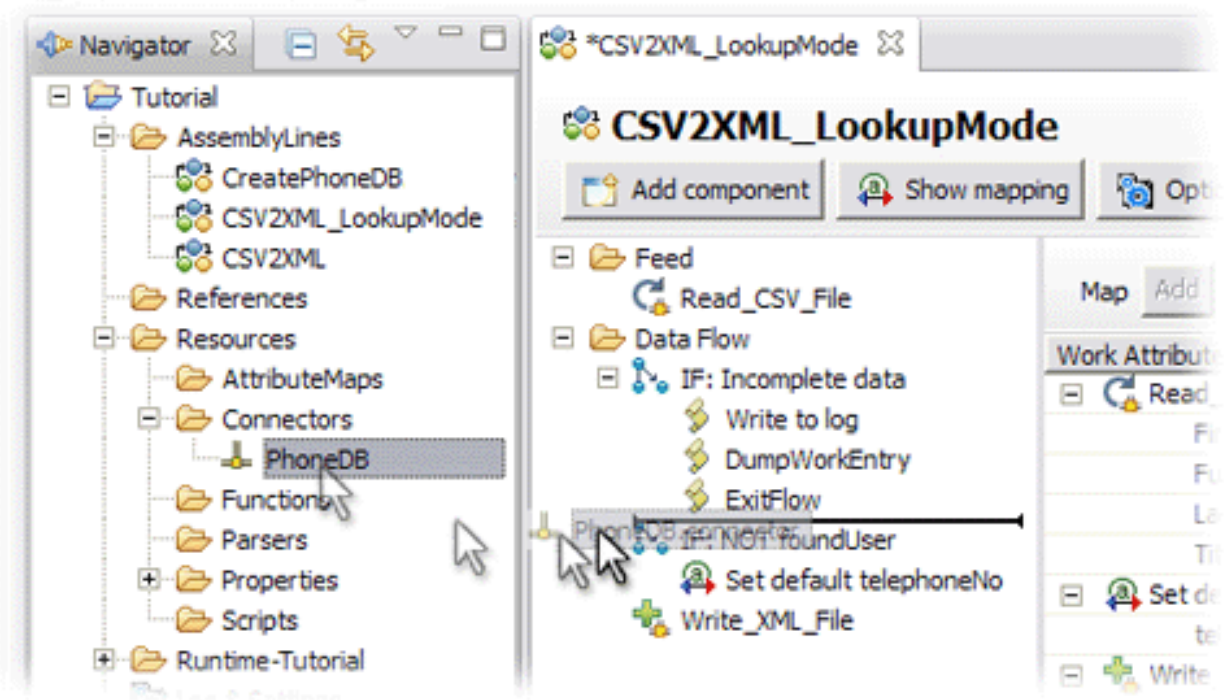


Figure 70. Drag the new resource into your AssemblyLine

## 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.

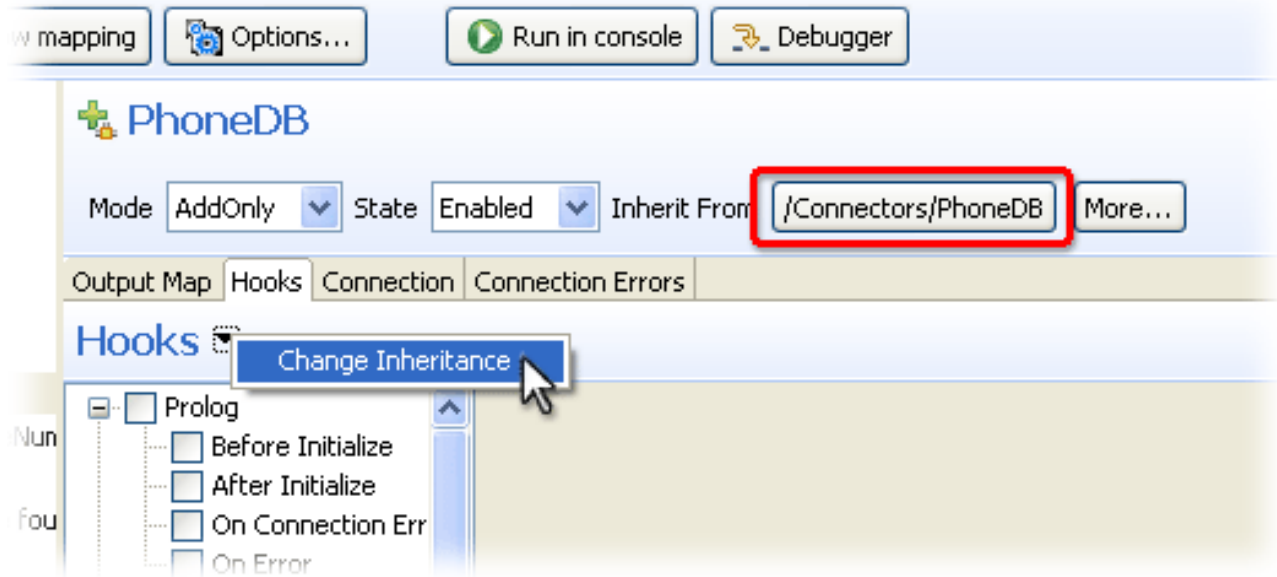


Figure 71. Setting Inheritance for the Hooks tab

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.

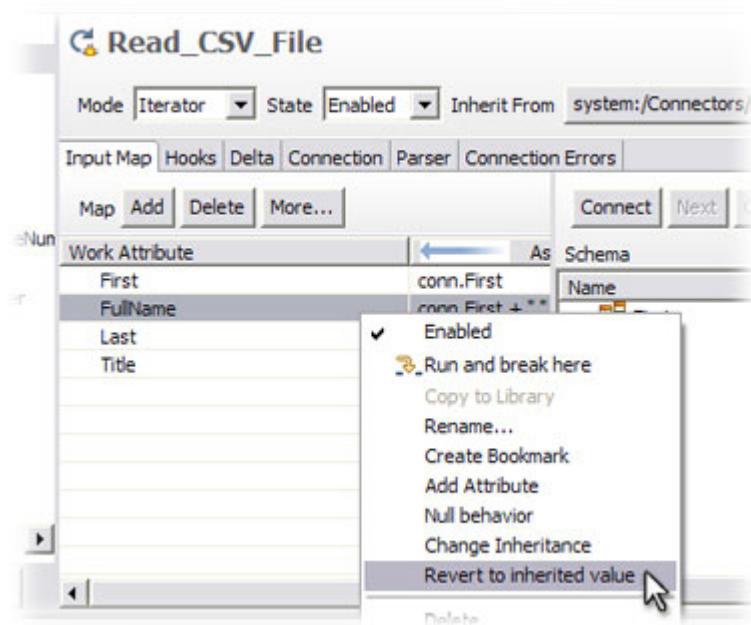


Figure 72. Restoring inheritance for a mapping rule

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.

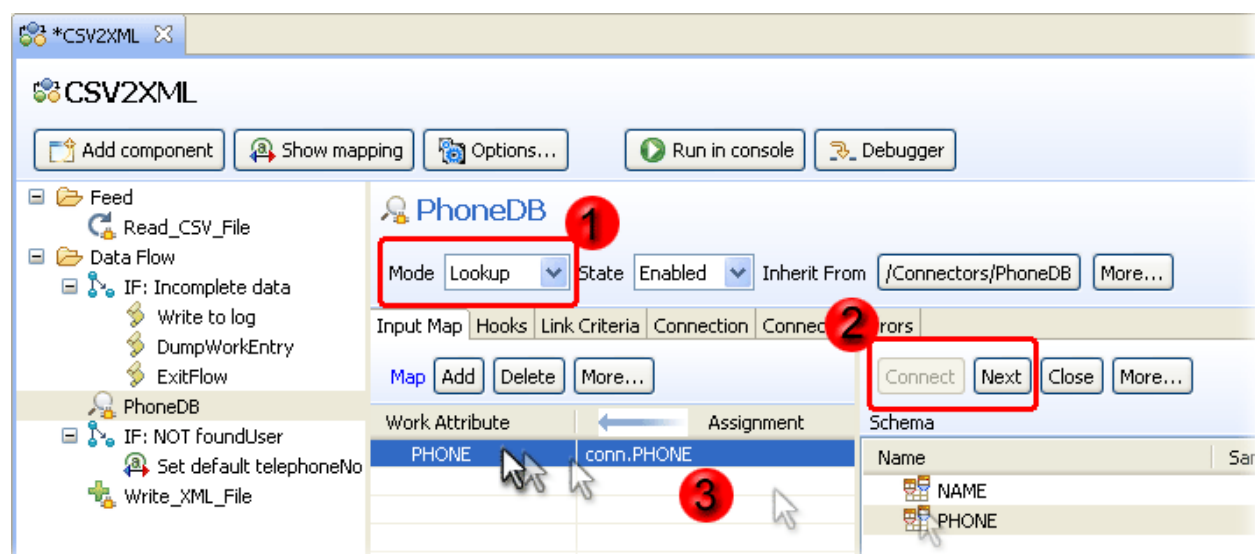


Figure 73. Changing mode, discovering and mapping Attributes

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.

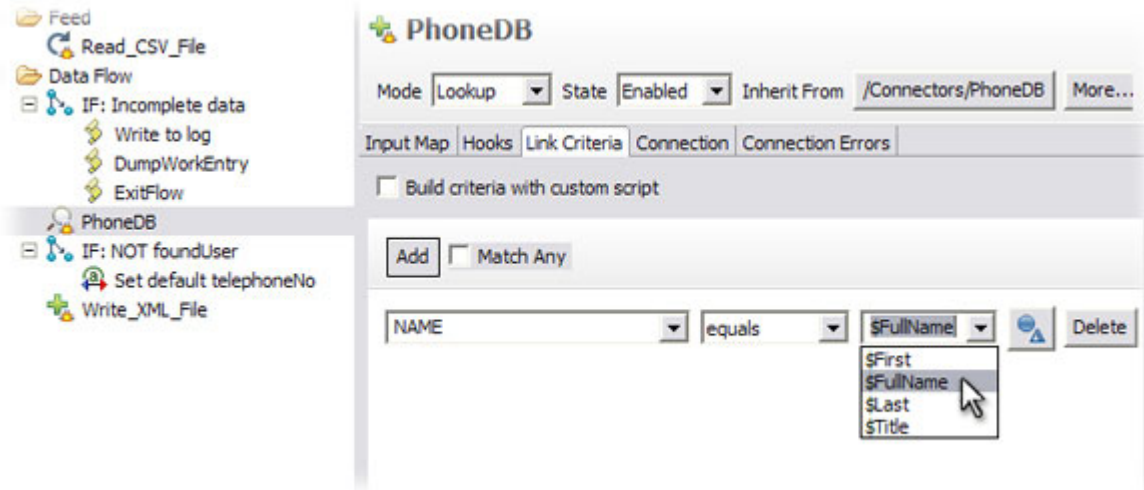


Figure 74. A simple Link Criteria

Remember to save your work, for example by pressing CTRL-S<sup>29</sup>. You will want to do this regularly so that you don't lose any work.

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

## 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. Here you will see information on both *where* the problem occurred as well as *what* caused it.

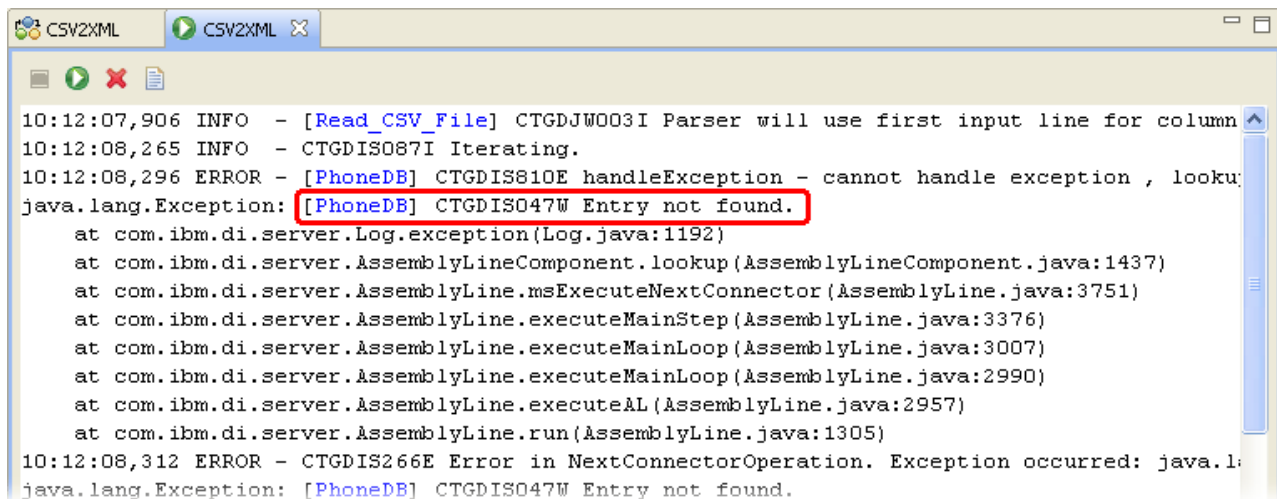


Figure 75. Error message in log output

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.

29. 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.

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 V7.1.1 Reference Guide*. Here is an excerpt from the page detailing *Lookup* mode:

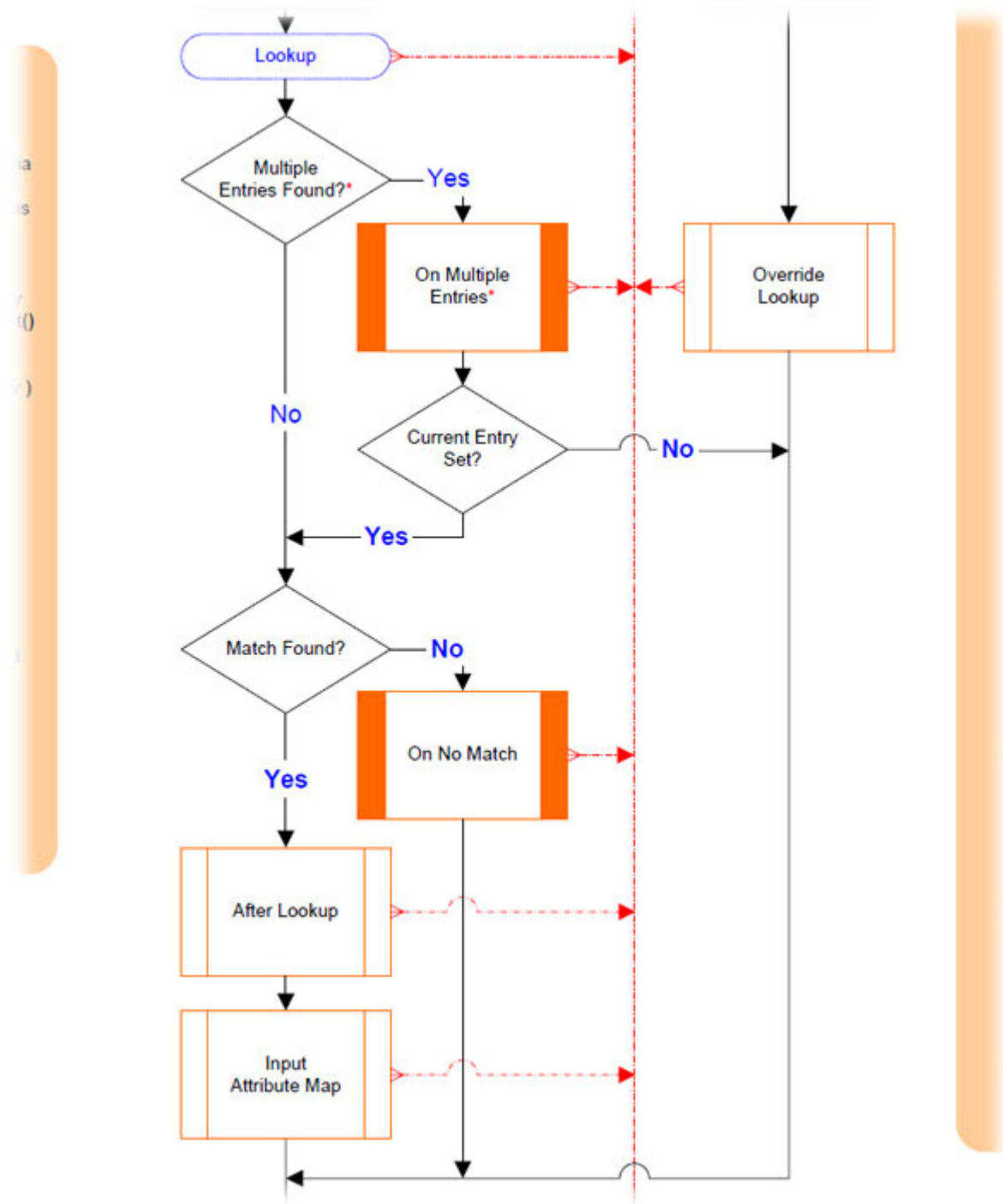


Figure 76. Partial Flow Diagram for Lookup mode

You can take advantage of this behavior to set your foundUser 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<sup>30</sup>:

```
foundUser = true;
```

Your AssemblyLine should now resemble the one in this screenshot:

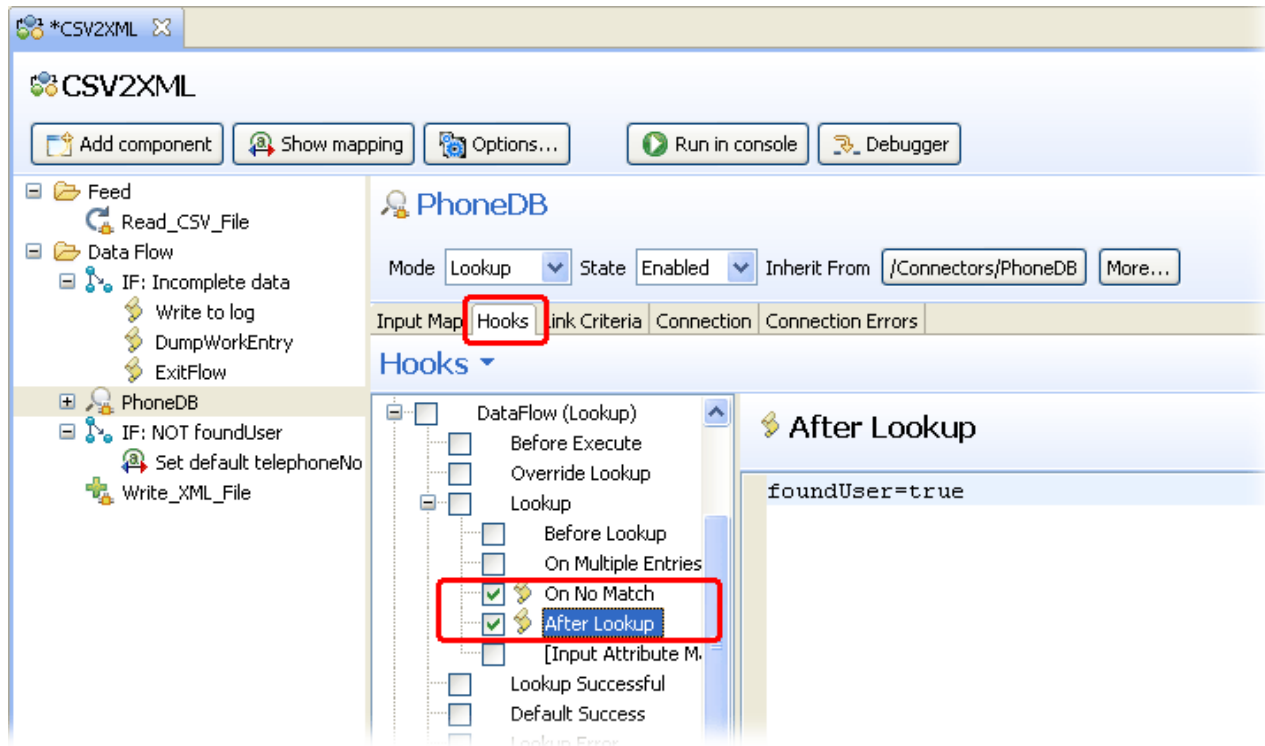


Figure 77. First tutorial exercise completed

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.

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

---

## 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<sup>31</sup>:

```
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.

---

31. 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

Using the user interface in the Configuration Editor, you can create a Scheduler to run an AssemblyLine at specified times. For example, you can create a Scheduler to run every day at 3:05 AM, every Saturday at 7AM, or even for more complex schedules.

To create a Scheduler, in the Configuration Editor, click **File > New > Scheduler**. Alternatively, right-click on the AssemblyLine and select **Create Schedule**.

When the TDI Server loads a configuration file, the enabled Schedulers contained in the configuration file are automatically started. The command to load a configuration file is `ibmdisrv -c myconfig.xml -d`, where `myconfig.xml` is the exported configuration file. Stopping the Config Instance stops all the associated Schedulers.

Figure 78. TDI Scheduler

The TDI Scheduler details are detailed in the “TDI Scheduler” chapter of *IBM Tivoli Directory Integrator V7.1.1 Reference Guide*.

## 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'<sup>32</sup>. Then insert a new component, choosing the 'HTTP Server Connector' and pressing **Finish** to end the wizard. Now open its **Input Map** tab.

<sup>32</sup>. TINA here stands for "TINA Is Not Apache" :-)

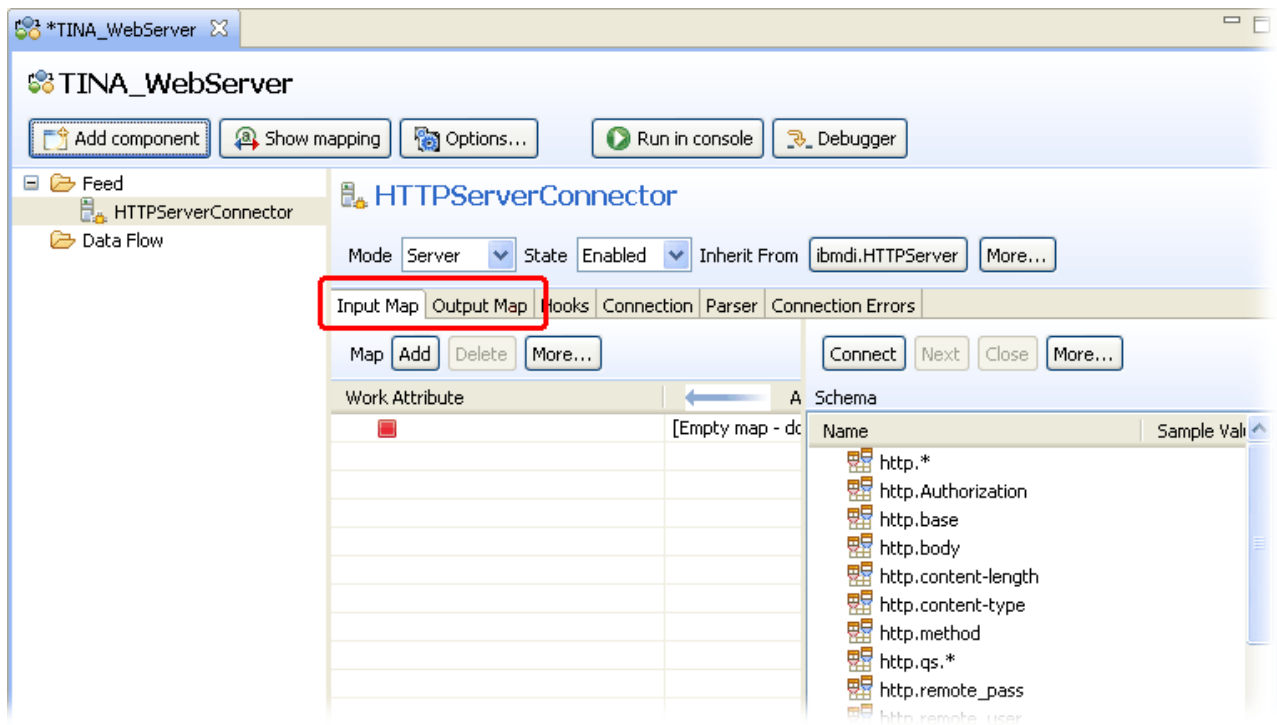


Figure 79. HTTP Server Connector Attribute Map panel

Server Connectors are complementary 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'.<sup>33</sup>

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;
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.

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



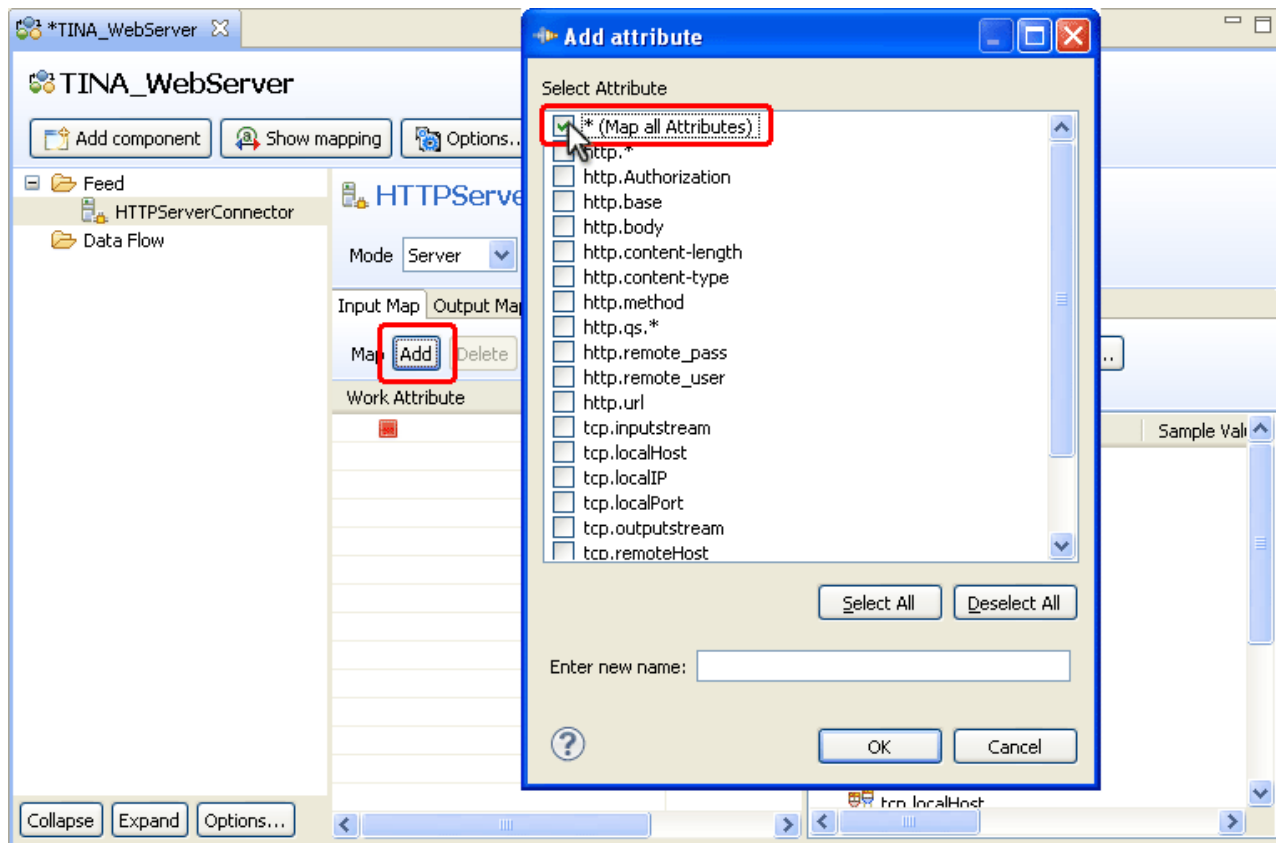


Figure 80. Add Input Attribute Map item

Simply select the topmost option in the dialog presented. Note that you could also have typed the asterisk character (\*) in the Entry new name field instead. This is the special wildcard map rule that tells TDI to map in all Attributes read by the Connector.

Your Input Map will look like this:

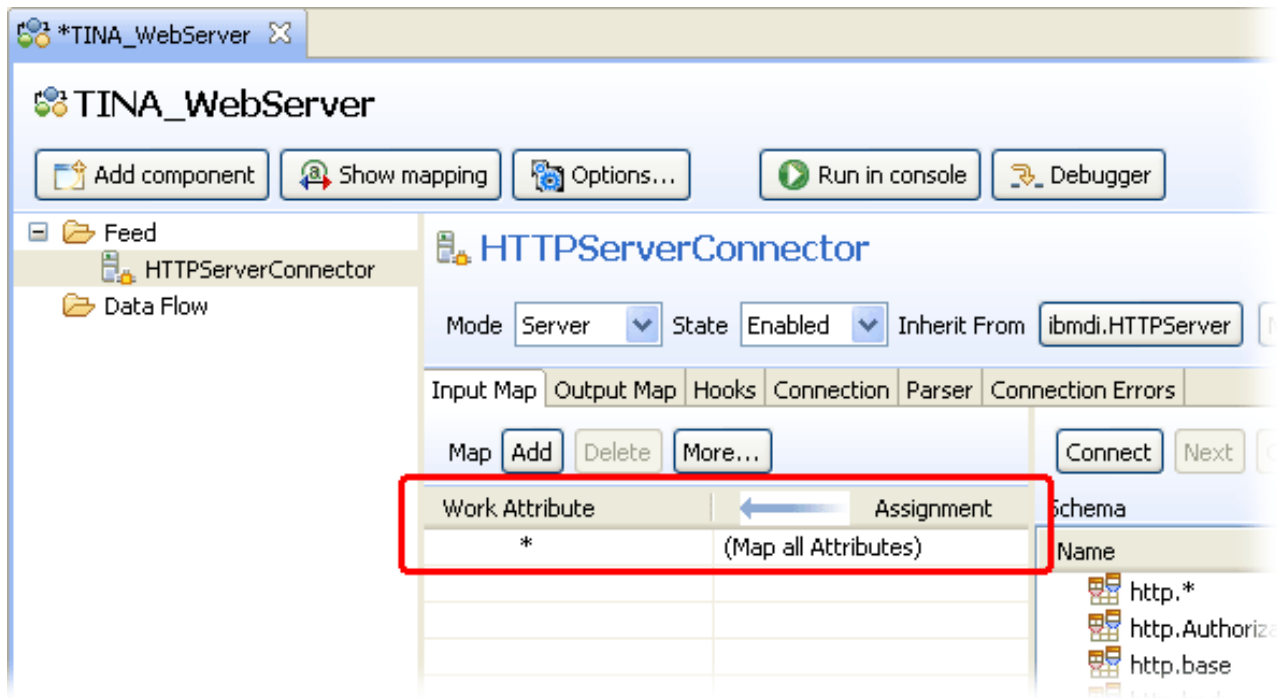


Figure 81. Wildcard map item

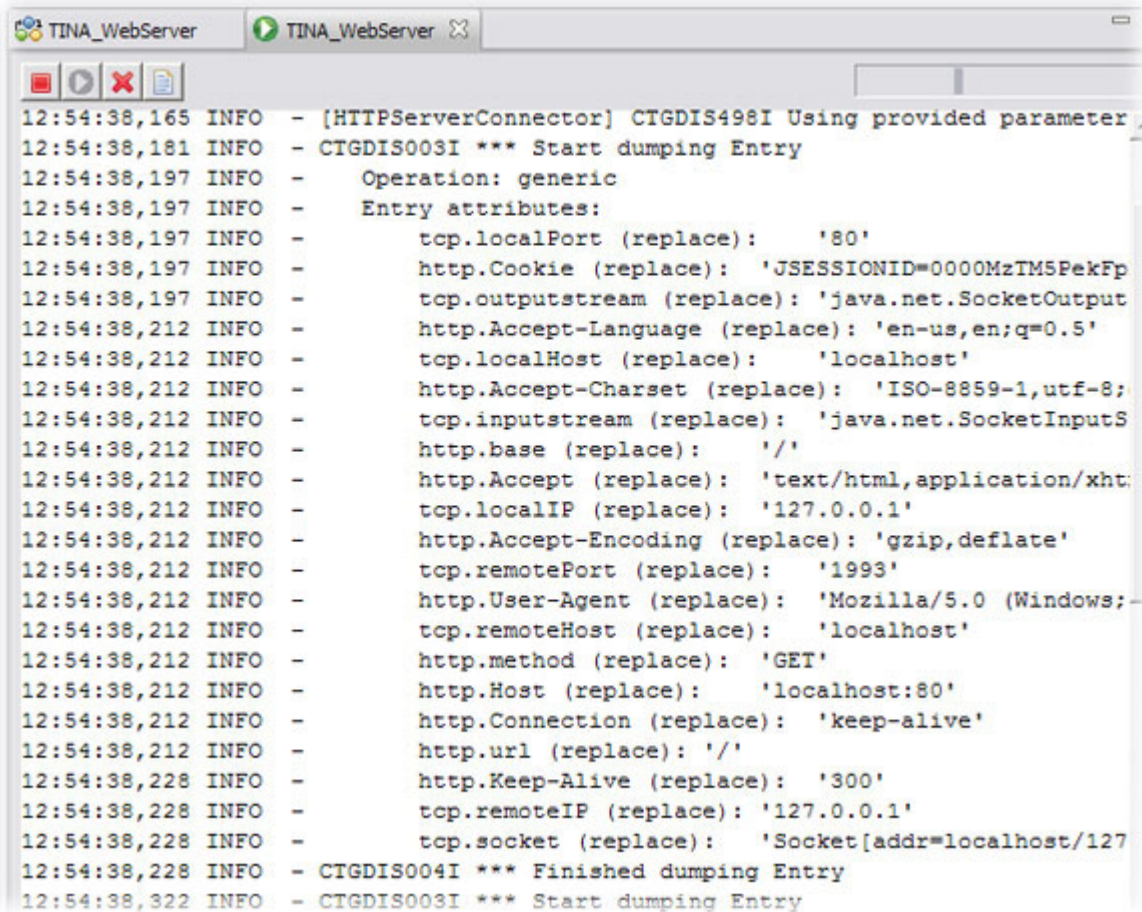
Add a wildcard map item to the Output map as well to ensure that any Attributes set up in the Work Entry for the response message will get mapped back to the client.

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<sup>34</sup>. 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.

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



```
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 top of the new IF branch.

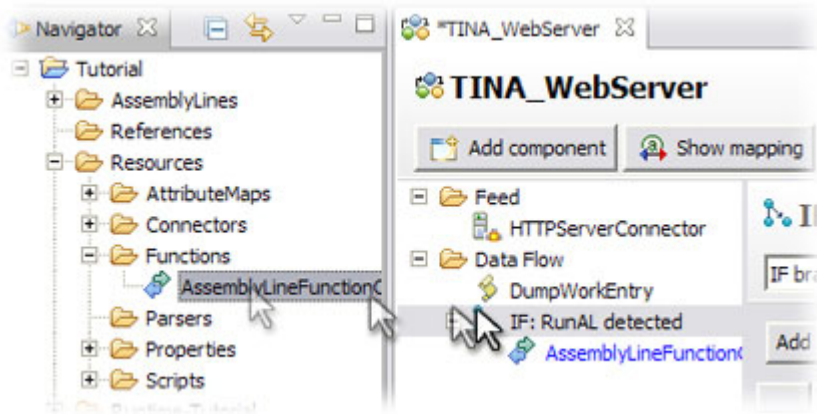


Figure 83. Drag in the AssemblyLine Function component (AL FC)

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.

```

TINA_WebServer
- http.Base (replace): /RunAL
- 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:



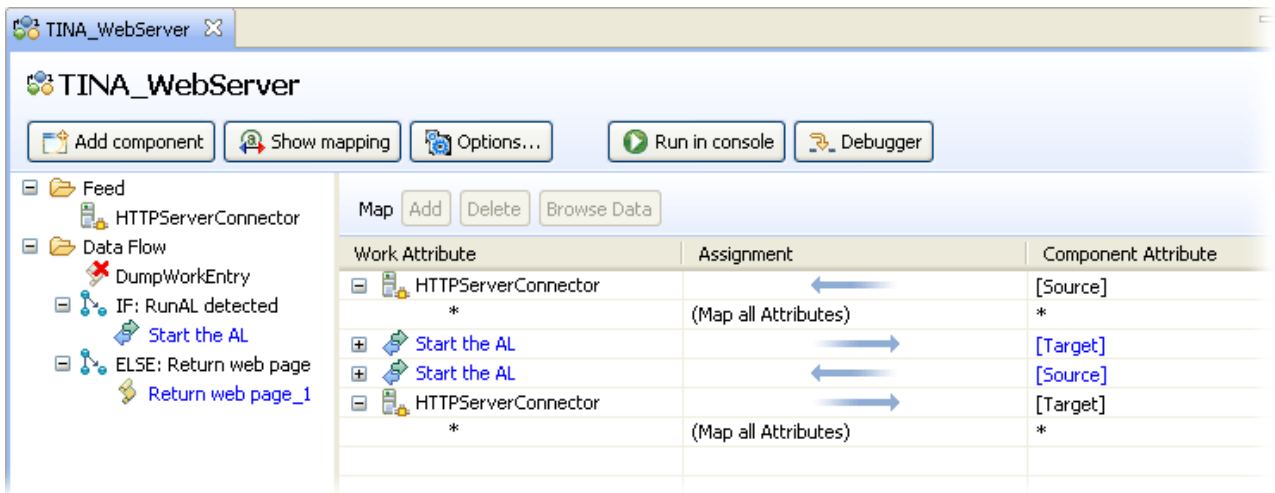


Figure 85. Completed TINA\_WebServer AssemblyLine

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

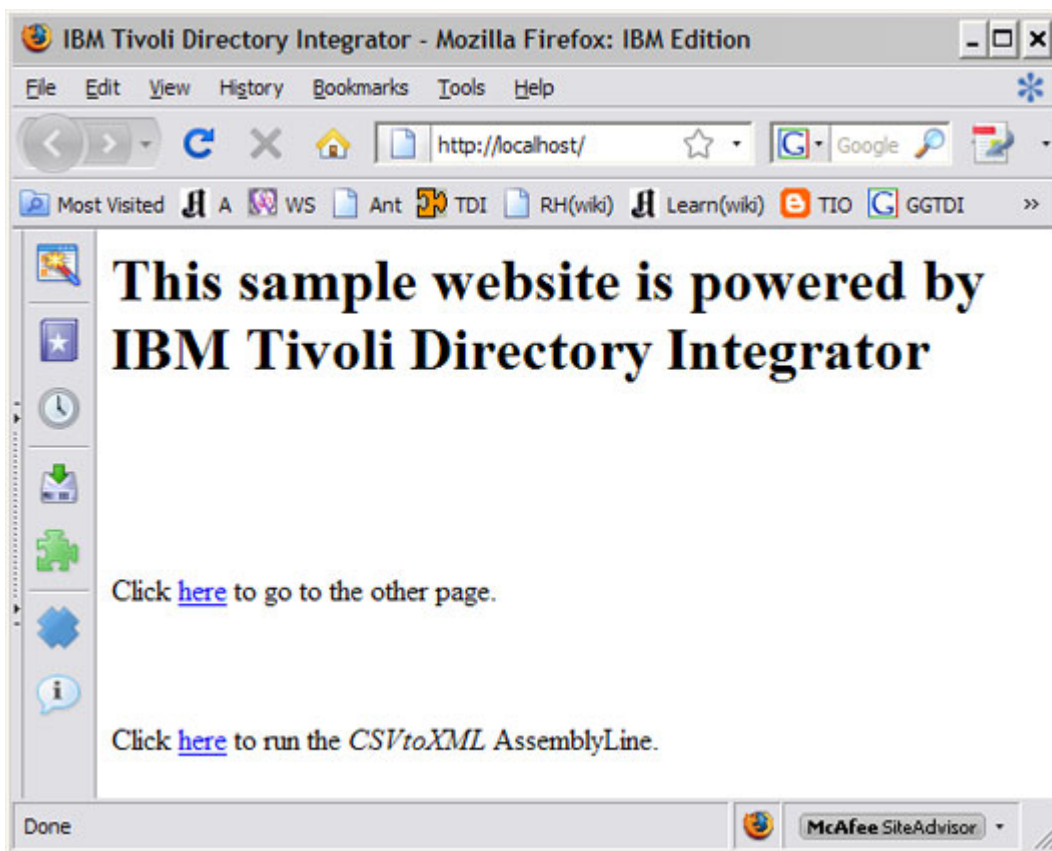


Figure 86. Simple Web Interface to your solution

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"<sup>35</sup>.
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.

---

35. 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.getTDIPProperty("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'<sup>36</sup> 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 (excpn) {
    task.logmsg("Call failed with error: " + expn);
}
```

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.

---

36. 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 V7.1.1 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 :)

---

## Appendix. EasyETL Guide

'ETL' stands for Extract, Transform and Load, and boils down to getting data from one place, changing it as needed and then putting it someplace else. 'EasyETL' is a feature of Tivoli Directory Integrator (TDI) that lets you do this quickly and interactively in just a few keystrokes.

Common ETL examples are:

- Exporting database records, Notes Documents, directory entries or even incoming mail or MQ messages to a file;
- Loading data from file into a system or data store;
- Migrating data directly from one system to another, or between versions of the same system in the case of software/schema updates.

TDI EasyETL lets you solve these and other scenarios in a few intuitive steps, resulting in solutions suitable for both one-off data movement needs, as well as for mission critical data flows in your infrastructure.

The first step in creating a new EasyETL task is to choose your Input Source and select the attributes that you want to transfer. Already at this point, TDI lets you run your EasyETL job and collect the data read into your copy buffer for pasting. If the data needs to be transformed or computed then EasyETL lets you add transformation, re-run the ETL job and copy/paste the transformed data. And you can also choose an Output Target and have your EasyETL job write the data directly there.

Once your EasyETL solution is working as desired, TDI can generate the command line assets (batch-files or scripts) for launching and scheduling the integration task. Finally, EasyETL leverages the change detection features of TDI to quickly turn your ETL job into a data synchronization task.

**Note:** The good news for TDI users is that each EasyETL solution is a TDI Project with a single AssemblyLine, and it can be opened in the full-featured development environment. However, once you change it there it's no longer available as EasyETL.

### Using EasyETL

Launch the TDI Config Editor and select the workspace to use. When TDI starts the first time it opens in the Welcome page<sup>37</sup>.

---

37. You can return to the Welcome page at any time by selecting **Help > Welcome** from the main menu.

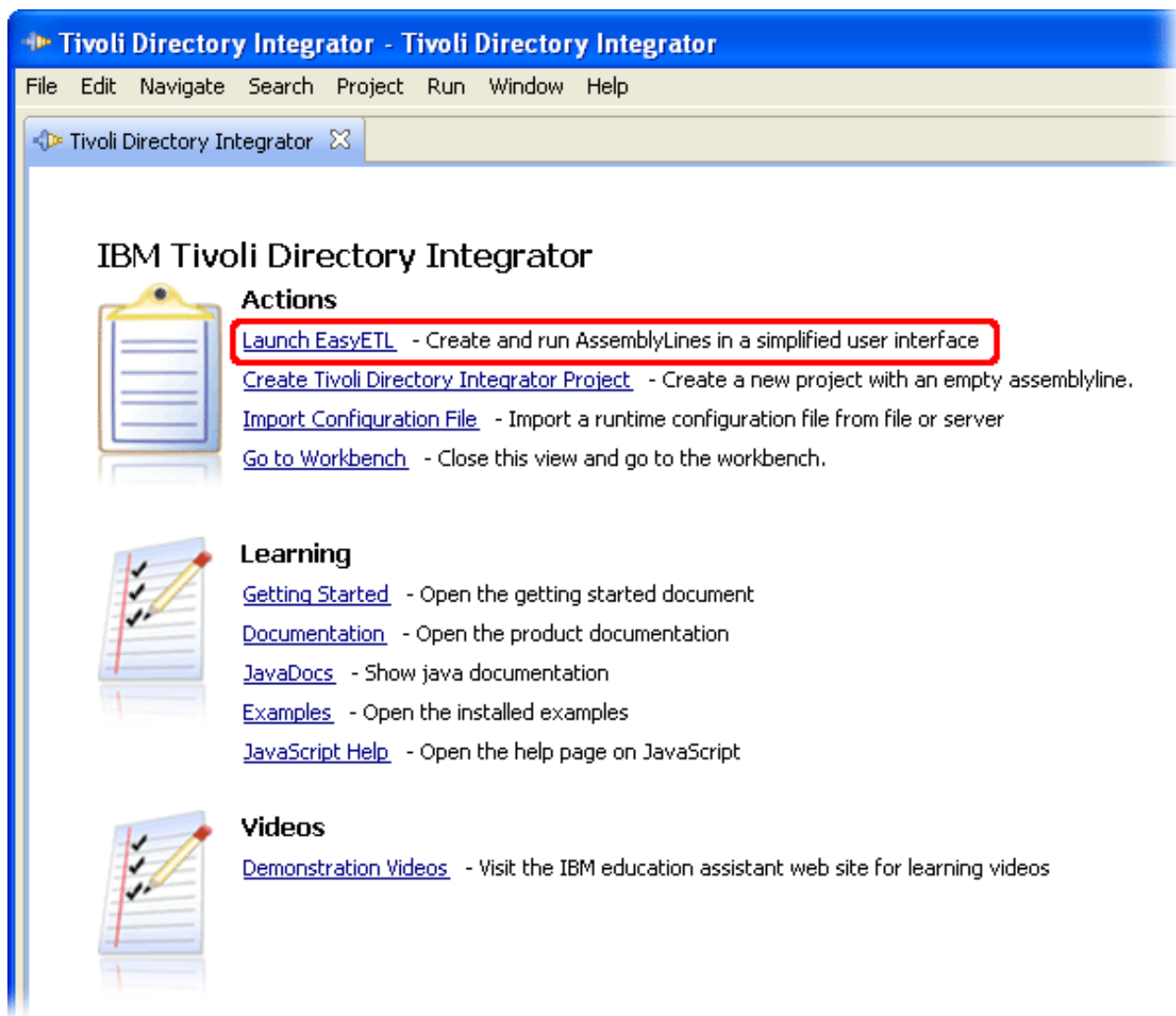


Figure 87. Welcome screen

As shown in the screenshot above, the topmost link here opens up the EasyETL<sup>38</sup> workbench. Click this link now to open the TDI EasyETL workbench.

38. EasyETL is a TDI *perspective*, and you can switch between perspectives using the menu selection: **Windows > Open Perspective > Other...** If you have made changes to a perspective and would like to reset it back to default, simply select **Windows > Reset Perspective**.

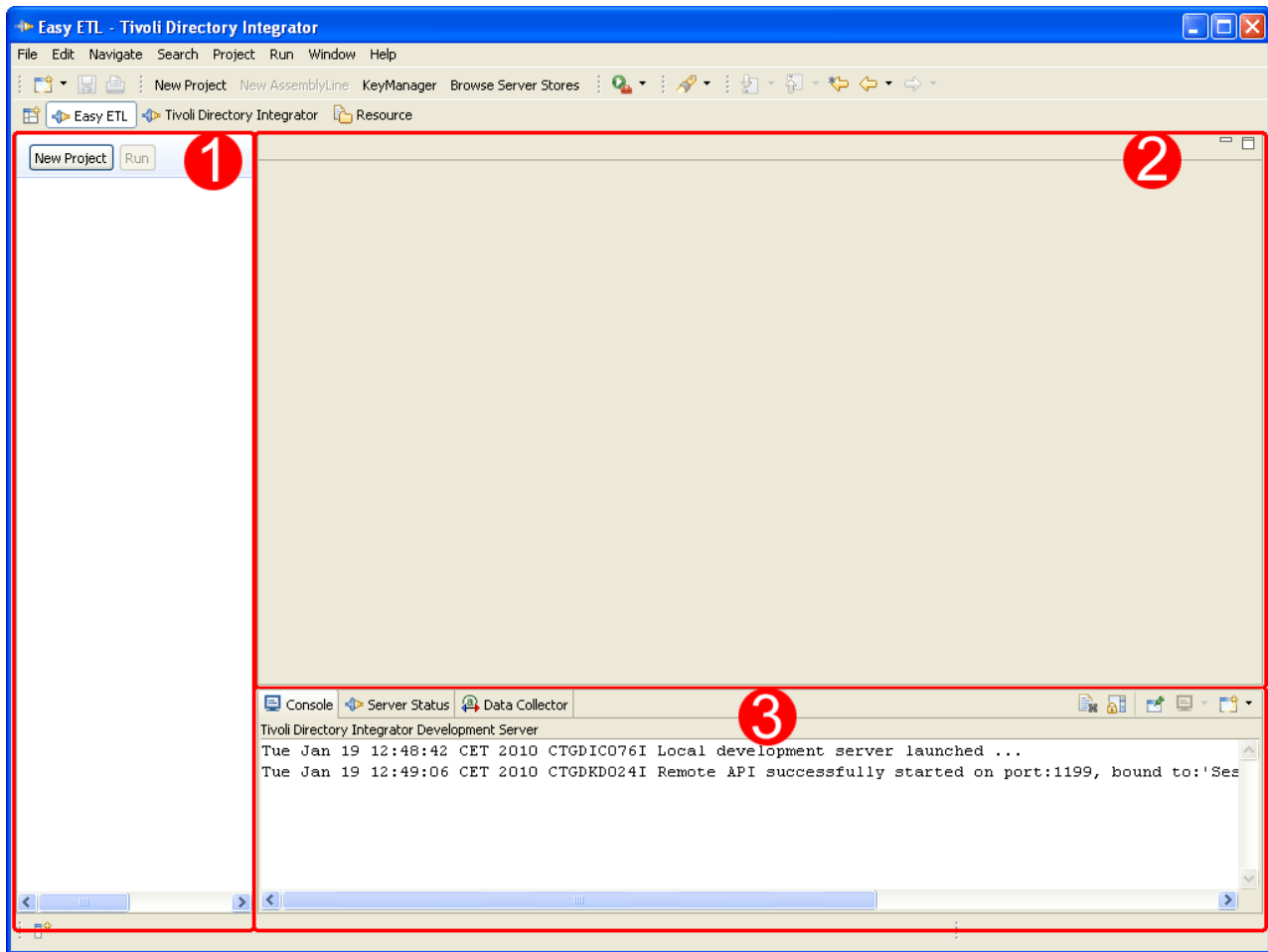


Figure 88. Figure 2. EasyETL Workbench

The EasyETL workbench shows you three things:

- The Project Navigator that lists your ETL jobs. You can right-click on any Project to do things like running it or creating command line assets to launch it;
- A Simple AssemblyLine<sup>39</sup> editor for each open project;
- Various Views as tabs along the bottom part of the screen. By default you get three Views:
  - the Console output from the test TDI Server;
  - the Server status view where you can monitor both the Server and any running EasyETL Projects;
  - and the Data Collector where the resulting data for each cycle is displayed in a tabular list.

You will find more information on each section as you work through this document.

## Creating a Project

Create a new EasyETL Project by pressing the **New Project** button at the top of the Project Navigator.

39. An 'AssemblyLine' is the implementation of a data flow in TDI, so when you create or open an EasyETL Project then the underlying AssemblyLine is presented in the editor.

Note also that the term 'AssemblyLine' is abbreviated as 'AL' in this and other TDI literature.

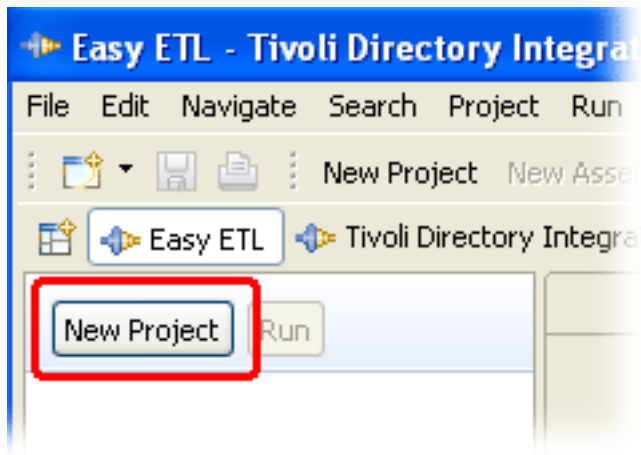


Figure 89. New Project button

Name this Project 'CSVtoXML' and press **Finish**. This opens your new Project in the Simple AL editor.

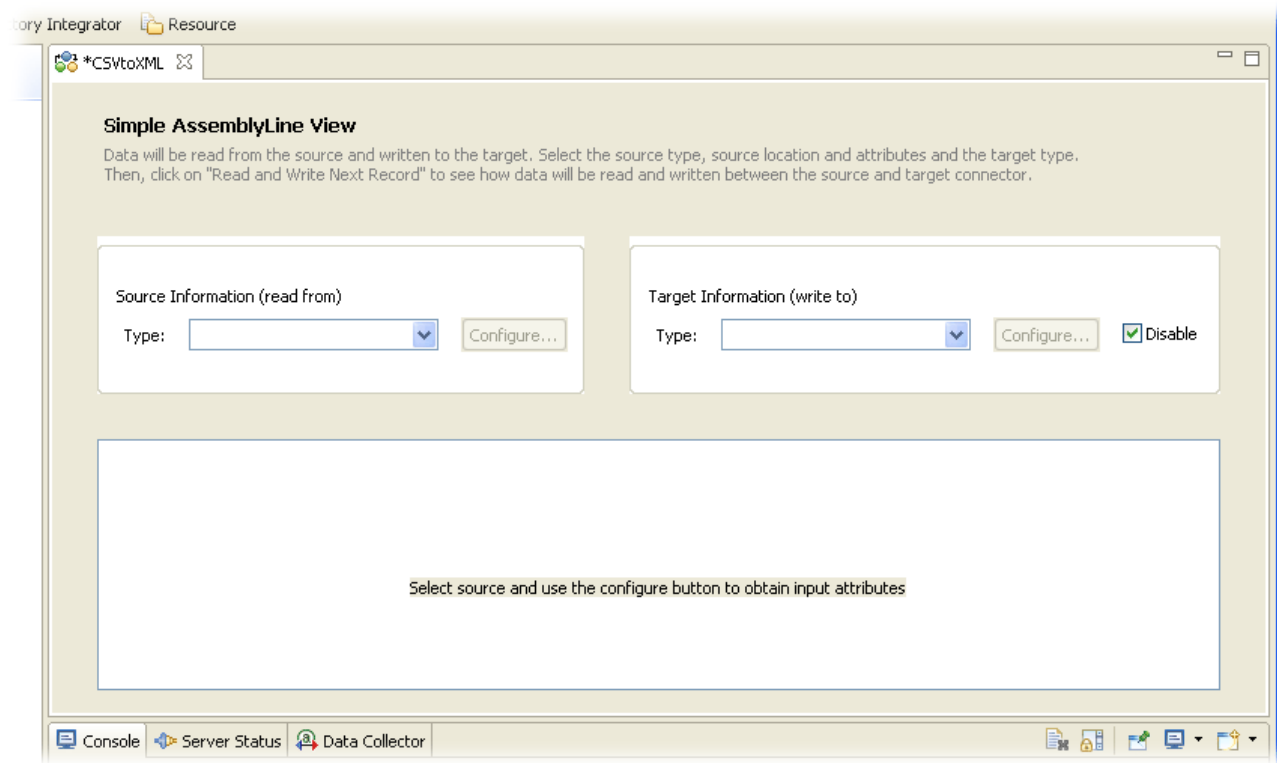


Figure 90. Simple AssemblyLine editor

The editor provides two drop-downs: one for selecting the Input source and one for the target. The area below is empty (apart from the assistance text) until you have chosen your source.

## Setting up input for your ETL AL

Configure input for your EasyETL AssemblyLine by clicking on the drop-down for Source Information and selecting the **File Connector**.



## Simple AssemblyLine View

Data will be read from the source and written to the target. Select the source. Then, click on "Read and Write Next Record" to see how data will be read and

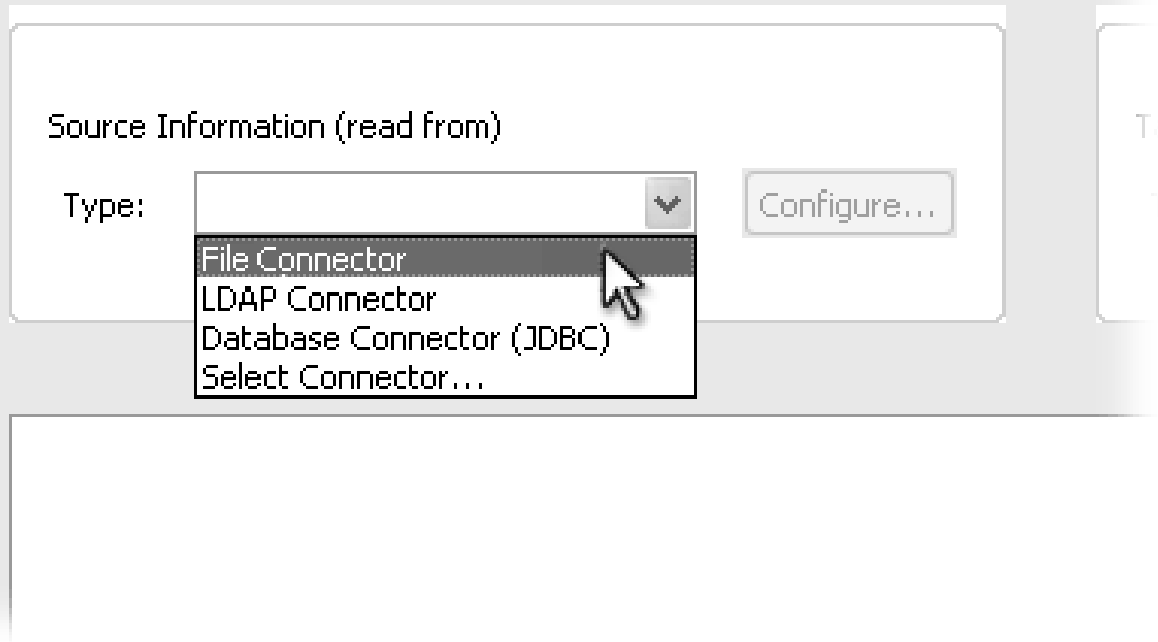


Figure 91. Selecting Source information

You will then be presented with the configuration dialog for this Connector.

Point the File Path parameter at the 'People.csv' file found here:

`TDI_HOME/examples/Tutorial`

where `TDI_HOME` is replaced with the TDI installation directory on your machine<sup>40</sup>.

40. Note that TDI supports both forward slash and backslash as the path separator when running on Windows. Your TDI solutions will be more portable between Windows and all the other platforms TDI runs on if you use forward slash.

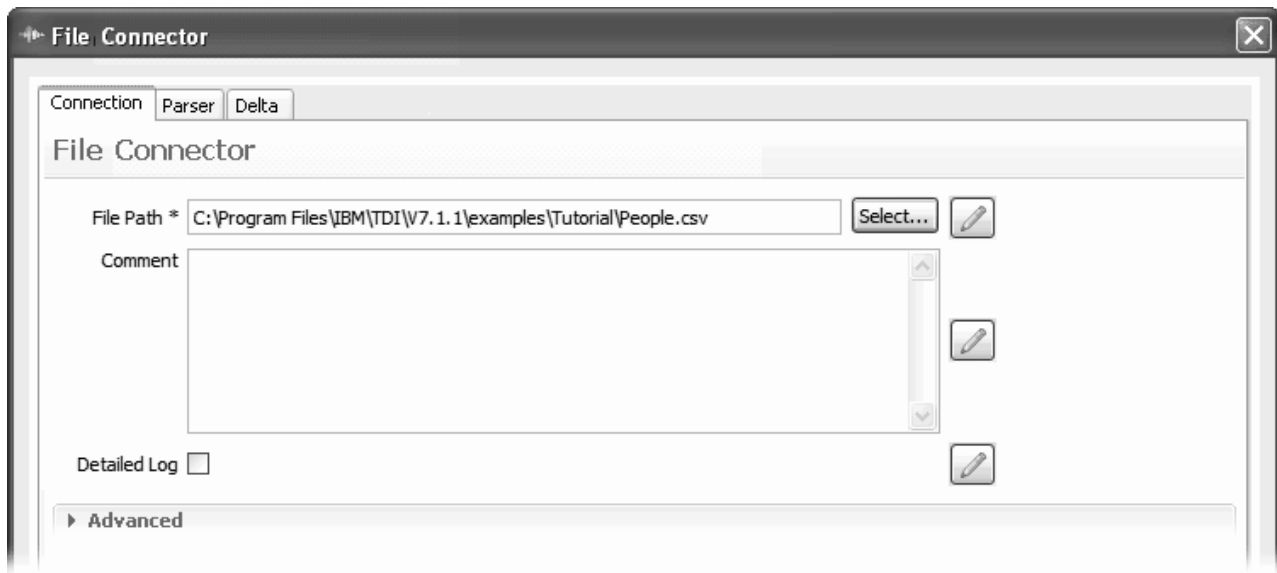


Figure 92. Setting the File Path parameter

Now click on the tab labeled **Parser** and select the **CSV Parser**, keeping the default configuration parameters as-is. Finally, press the **Connect** button at the bottom of the dialog box to test the connection and discover available Attributes.

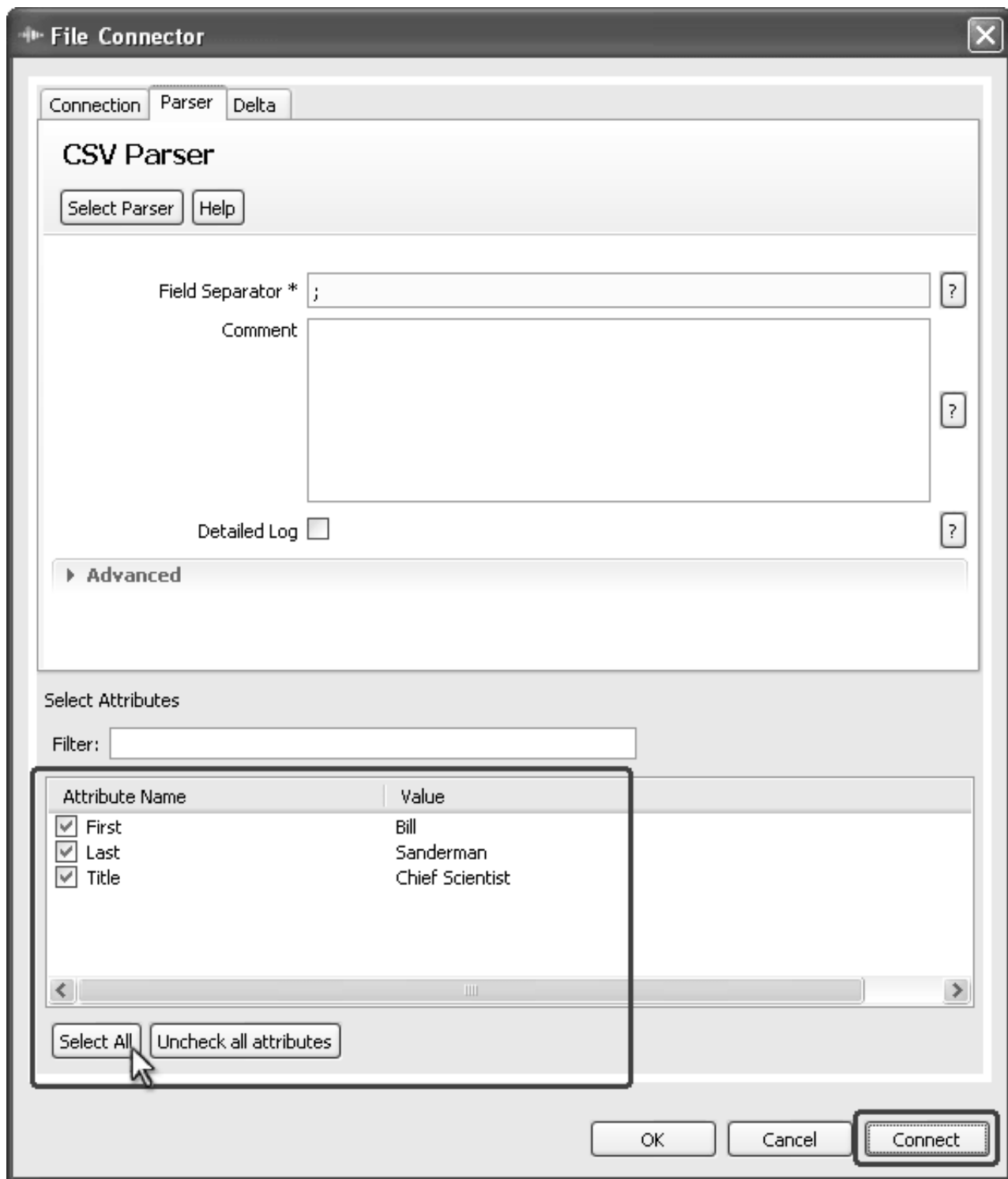


Figure 93. Testing the connection and discovering schema

The schema for the connected system is displayed and from here you can choose which of these to use in your data flow. For this example, use the **Select All** button and then press **OK** to close the configuration dialog.

Back in the EasyETL workbench you will see that the bottom half of the Simple AL editor has changed to reflect that you now have an Input Source configured. TDI now presents you with a button for stepping

through this information one record at a time, as well as buttons to run the ETL task to completion and to stop it.

The screenshot shows the 'Data Collector (CSV/XML)' configuration window. It is divided into several sections:

- Source Information (read from):** A dropdown menu is set to 'File Connector' with a 'Configure...' button next to it.
- Target Information (write to):** A dropdown menu is empty, with 'Configure...' and 'Disable' (checked) buttons next to it.
- Control Buttons:** A row of buttons: 'Read and Write next record', 'Run' (with a play icon), and 'Stop' (with a square icon).
- Data Viewers:** Two tables side-by-side.
  - Source Attribute Viewer:**

| Source Attribute | Source Value |
|------------------|--------------|
| <b>First</b>     |              |
| <b>Last</b>      |              |
| <b>Title</b>     |              |
  - Target Attribute Viewer:**

| Target Value | Target Attribute |
|--------------|------------------|
|              | <b>First</b>     |
|              | <b>Last</b>      |
|              | <b>Title</b>     |
- Show data transformations:** A checkbox that is currently unchecked.

At the bottom, there is a status bar with tabs for 'Console', 'Server Status', and 'Data Collector (CSV/XML)'. The 'Data Collector (CSV/XML)' tab is active.

Figure 94. Input Source configured

Below these buttons are two grid boxes called *Data Viewers* that list the Attributes handled by your data flow. The Data Viewer on the left shows your input Attributes. Those that you selected for reading in the previous step are displayed in **bold** at the top and are called the *Input Map*. Below will be any unselected Attributes displayed in gray, and you can include these for input mapping by double-clicking on them. Similarly, you remove Attributes from the map by either double-clicking or deleting them<sup>41</sup>.

The box to the right is the Output Viewer and it shows the set of Attributes to be written. In TDI terms, this is your *Output Map* and by default it is identical to the list you selected for your Input Map. Note that you can change the name of any Output Attribute by clicking in the right-hand column and then editing the value. Use this technique to rename the Attributed called 'Title' to 'JobTitle'.

41. You can also right-click and choose **Add Attribute** or **Remove Attribute**.

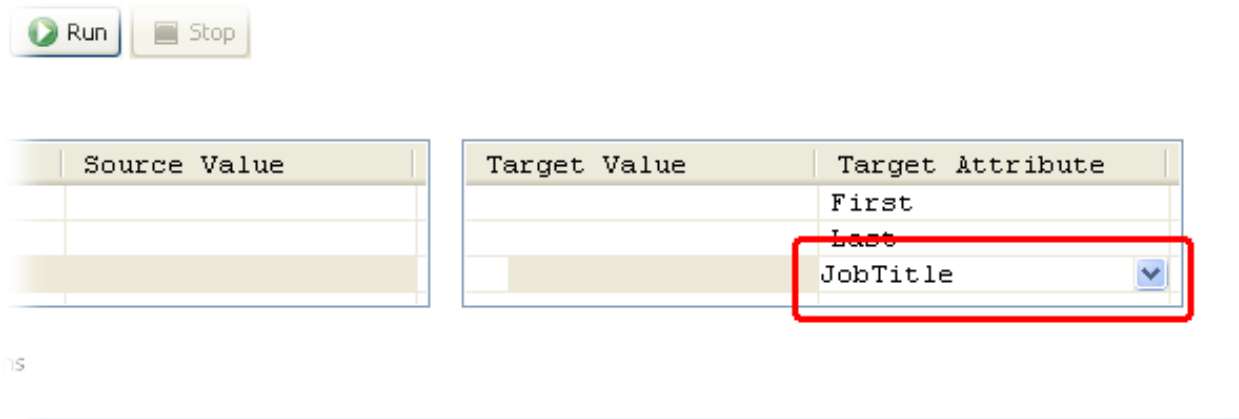


Figure 95. Renaming an Output Attribute

Your EasyETL project is now ready for its first test.

## Running your EasyETL AssemblyLine

Select the **Data Collector** view at the bottom of the screen and then press the **Read and Write next record** button. This causes the following to happen:

1. There is a delay as your EasyETL AssemblyLine is transferred to the running Server and started;
2. The first record is read and parsed from your CSV input source and the data is displayed in both the input and output grid displays;
3. The Attributes you selected for output are written and collected in the Data Collector view<sup>42</sup>.

So even though you have not selected an Output target yet, you can still run and test your ETL project, viewing the data as it flows down the AssemblyLine.

42. The leftmost button at the top of the **Data Collector** view opens a configuration dialog where you can increase the **Data Collector Buffer** size. Note that if you plan to collect large amounts of data then you may need to increase the memory available to TDI. Do this by locating the 'ibmditk' batch-file/script in the TDI installation folder and opening it in an editor. Near the bottom is the line that launches 'miadmin' and you could insert the following text after the -vmargs option: -Xmx512M This example will allow TDI data memory to grow to 512 Mbytes.

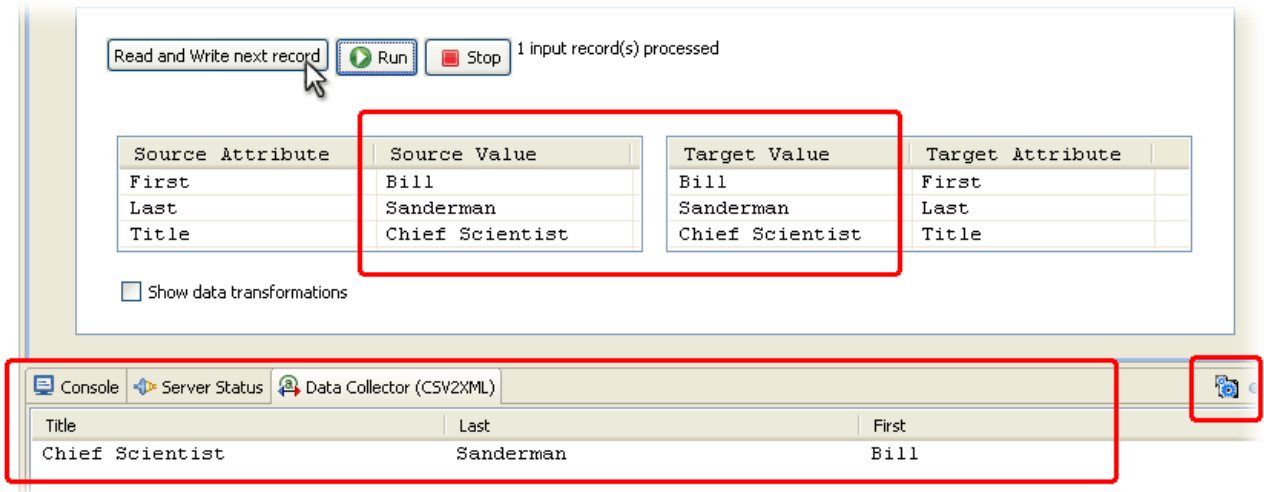


Figure 96. One record read and collected

Each time you press the **Read and Write...** button, another record is read, displayed and collected. If you now press the Run button then your ETL job runs to completion and you will see this AL report.

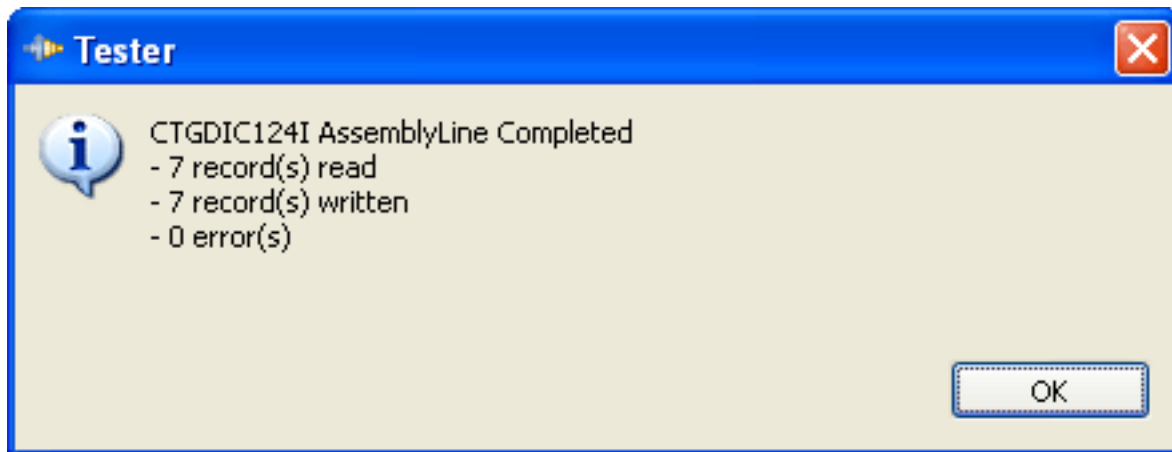


Figure 97. EasyETL AssemblyLine completed

As shown in the dialog above, no records were actually written anywhere. However, the **Data Collector** still provides handy visual feedback on the information being extracted and transferred.

Furthermore, you can select rows in the collected data and copy/paste this information to a file or other target<sup>43</sup>.

## Transformations

Up to now your Output values have been identical to your input. However, there will be situations where you want to manipulate or even compute these based on the data read. You do this in TDI EasyETL by writing Transformations in JavaScript.

43. Data is copied in CSV format to simplify importing values into spreadsheets and report tables. 'CSV' stands for Character Separated Value, and the separator character used by TDI is the semi-colon (;).

In order to work with Transformations you first have to enable them by selecting the **Show data transformations** check box.

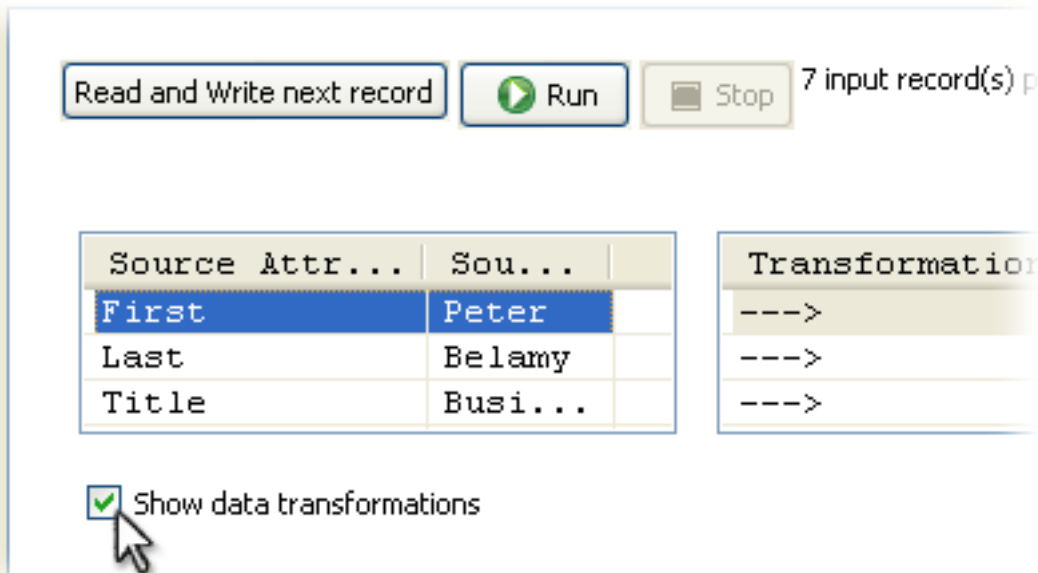


Figure 98. Enabling Transformation

This causes a new grid box to appear between the Data Viewers: the Transformation Viewer. Here you see arrows indicating that all three Output Attributes get their values directly from Input Attributes – in other words, no transformations. You are going to define a new Output Attribute and then add the Transformation script to compute its value. Do this now by right-clicking in the Output Map, choosing Add Attribute and naming it 'FullName'. Now double-click on the Transformation to the left of this new Output Attribute and then enter this script snippet:<sup>44</sup> `return First + " " + Last`

44. Note that you can press Ctrl + Space to get up a list of suggestions to what you can type. This list includes some special objects like 'task' and 'main' and lists the Attributes being read in from your Input Source at the bottom.

Note also that the notation shown here for accessing Input Attributes only works in EasyETL. In the full TDI Workbench you must prefix an Attribute name with the Entry object carrying it - for example the Work Entry (work). As a result, the above example would look like this:

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

The Work Entry and other TDI concepts are discussed in the first chapter of this document.

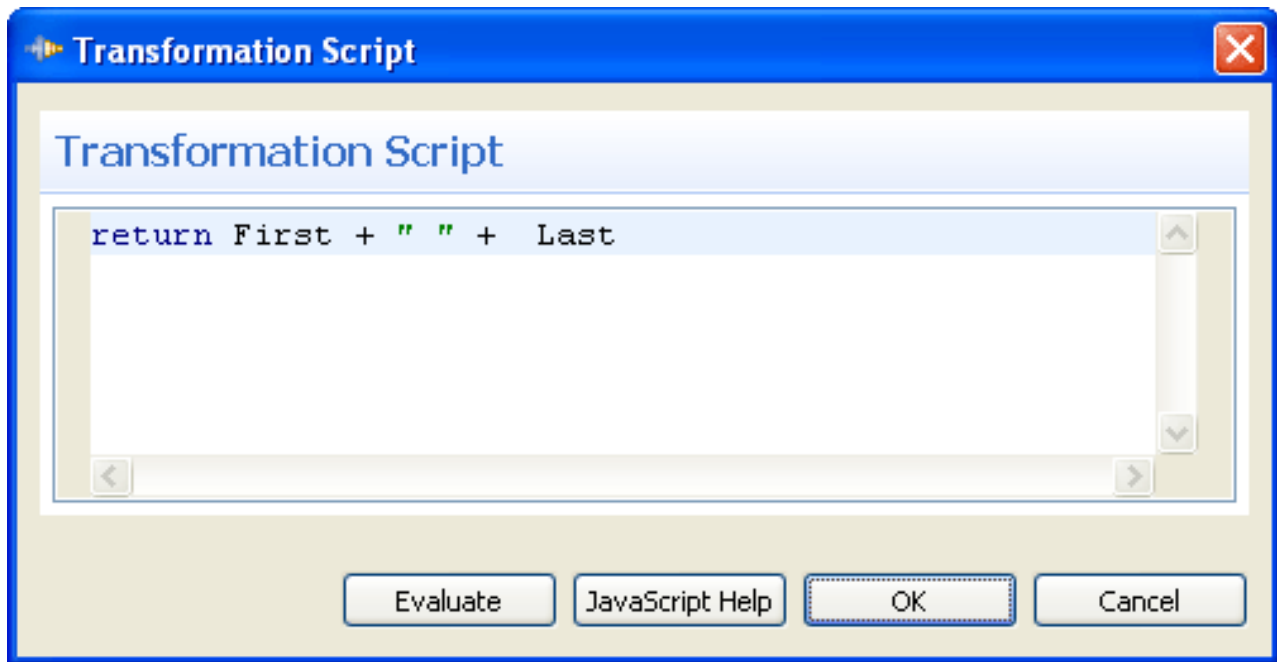


Figure 99. Show Transformation script

There is an **Evaluate** button in the Transformation Script editor dialog for testing the script, along with one for bringing up some JavaScript tips and examples.

Press **Evaluate** now to get an idea how your Transformation works.

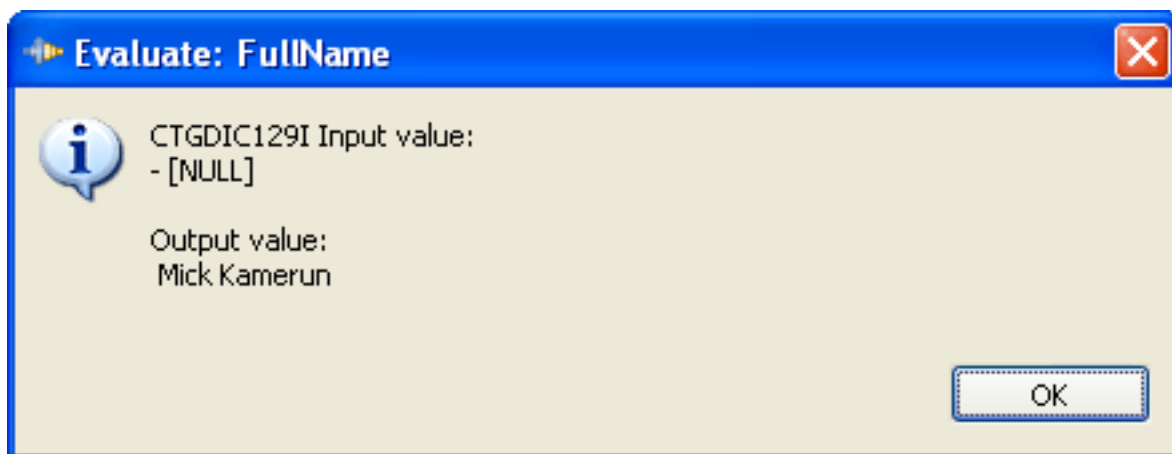


Figure 100. Evaluate Expression

The Output value shown was computed using data that you collected when you ran your AssemblyLine. Close the evaluation results dialog by pressing **OK**.

Now press **OK** now to accept this script and close the Transformation Script editor and re-run your EasyETL AssemblyLine by pressing **Run** and then viewing collected entries. Notice how the **Data Collector** now gives you two Component Collections to choose from: *Output* and *Input*. Choose **Output** and see how your Transformation script generated a 'FullName' value for each entry.



☒ Show data transformations

| Component | Last     | JobTitle             | FullName         | First   |
|-----------|----------|----------------------|------------------|---------|
| Output    | Highpeak | VP Product Develo... | Gregory Highpeak | Gregory |
| Input     | Hazzle   | Chief Evangelist     | Ernie Hazzle     | Ernie   |
|           | Belamy   | Business Support ... | Peter Belamy     | Peter   |

Figure 101. Output collection with computed FullName Attribute

So now you know how to set up an Input Source and select the Attributes to extract, as well as how to transform this data to fit your output needs. The next step is selecting an Output target and driving the data there.

## Selecting an output target

Select the **File Connector** for your Output target using the drop-down above the Output View and have it write to a file called 'Output.xml'<sup>45</sup>. Choose the **XML Parser** and press **OK**. Note that you could use the **Connect** button to ensure that the file path you entered is valid. However, there will be no data to discover – unless of course you point your Connector at an existing XML file.

Once our Output is configured, run your ETL AssemblyLine again. Once completed you can open the output file and verify the results.

45. If you do not enter a full path, or a relative one, then TDI will base this from the Solution Directory that you specified during installation.

```

- <DocRoot>
  - <Entry>
    <First>Bill</First>
    <Last>Sanderman</Last>
    <Title>Chief Scientist</Title>
  </Entry>
  - <Entry>
    <First>Mick</First>
    <Last>Kamerun</Last>
    <Title>CEO</Title>
  </Entry>
  - <Entry>
    <First>Jill</First>
    <Last>Vox</Last>
    <Title>CTO</Title>
  </Entry>
  - <Entry>
    <First>Roger</First>
  </Entry>
  - <Entry>
    <First>Gregory</First>
    <Last>Highpeak</Last>
    <Title>VP Product Development</Title>
  </Entry>
  - <Entry>
    <First>Ernie</First>
    <Last>Hazzle</Last>
    <Title>Chief Evangelist</Title>
  </Entry>
  - <Entry>
    <First>Peter</First>
    <Last>Belamy</Last>
    <Title>Business Support Manager</Title>
  </Entry>
</DocRoot>

```

Figure 102. XML Output

Your Output target could as easily have been a database table, just as your input could be coming from a Lotus Notes® application or LDAP directory. The steps you took to create this example ETL job are the same, regardless of the systems or data stores you are working with.

## Detecting Changes

TDI provides a number of features for detecting changes in input data. In addition to offering a set of Change Detection Connectors<sup>46</sup>, you also have the option of enabling the *Delta Engine* for your Input source.

The Delta Engine takes snapshots of data as it's read and then compares these with snapshots taken during the previous run to determine what has changed. Those entries that are unchanged are skipped, and only modified entries are retrieved for processing in your EasyETL AssemblyLine.

Press the **Configure** button for your Input source and then select the **Delta** tab.

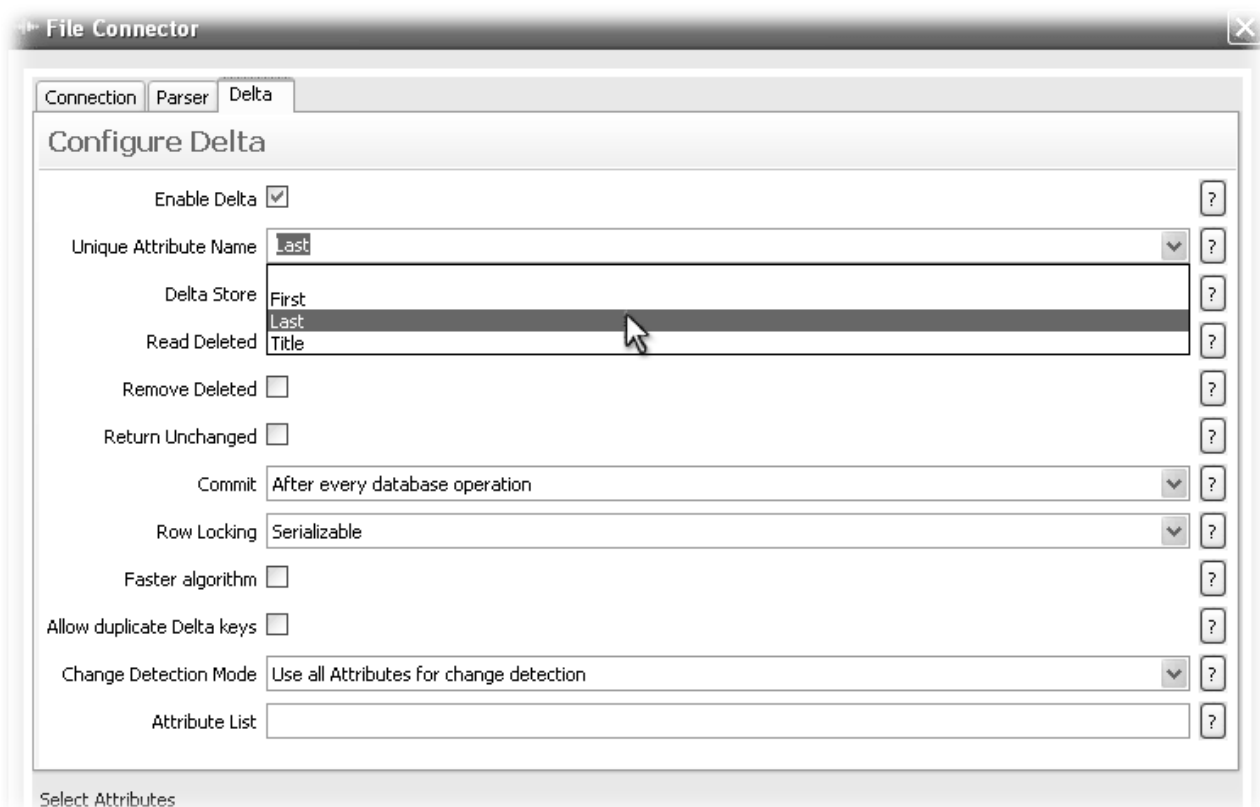


Figure 103. Delta configuration

You must first enable the Delta Engine by selecting the check box at the top of the con-figuration panel. Then use the drop-down to select 'First' as the **Unique Attribute Name**<sup>47</sup>.

There are several other parameters available here, some of which make more sense when working in the standard TDI Workbench and not in EasyETL. For example, al-though an EasyETL AL can detect and transfer new and modified entries, it will not handle deleting a row from a database or entry in a directory. However, it will write this infor-mation to an Output target like a File Connector with the LDIF Parser. LDIF files can contain change operation tags, and some systems support LDIF import.

46. Note that the General Purpose Edition of TDI only offers the Change Detection Connector for databases (RDBMS's), while the Identity Edition includes those for a number of other systems, like Tivoli Directory Server, Domino and Sun One.

47. As you may have deduced, the Delta Engine uses one of your input attributes to uniquely identify snap-shots. If there is there is no unique value available in the input data then you can specify multiple attributes that will be concatenated together to from the snapshot id. You do this by typing in the names of multiple attributes separated by a plus symbol (+). For example: First + Last

You can learn more about the full Delta Handling features of TDI here:

[http://www.tdi-users.org/twiki/pub/Integrator/HowTo/HowTo\\_SyncData\\_6.1.1070523.pdf](http://www.tdi-users.org/twiki/pub/Integrator/HowTo/HowTo_SyncData_6.1.1070523.pdf)

One change that you may wish to make is to the **Commit** parameter. This controls when new and changed snapshots are committed to the TDI System Store database. By default this is set to 'After every database operation' and so occurs during the read phase.

However, if you wish to ensure that a change has been successfully transferred before committing the snapshot, set this drop-down to 'On end of AL cycle' instead so that it happens after the Output target has been updated.

In order for the Delta Engine to do its work it needs a baseline snapshot set. You create this by running your ETL job the first time after Delta has been enabled. Once it has completed you will notice that the popup reports twice as many writes occurring. This is because TDI also counts the snapshots being written to the System Store, so you get two writes for every entry processed.

Try running your EasyETL AssemblyLine again and you will see that no entries were written this time. The Delta Engine detected that input records were all unchanged and skipped them.

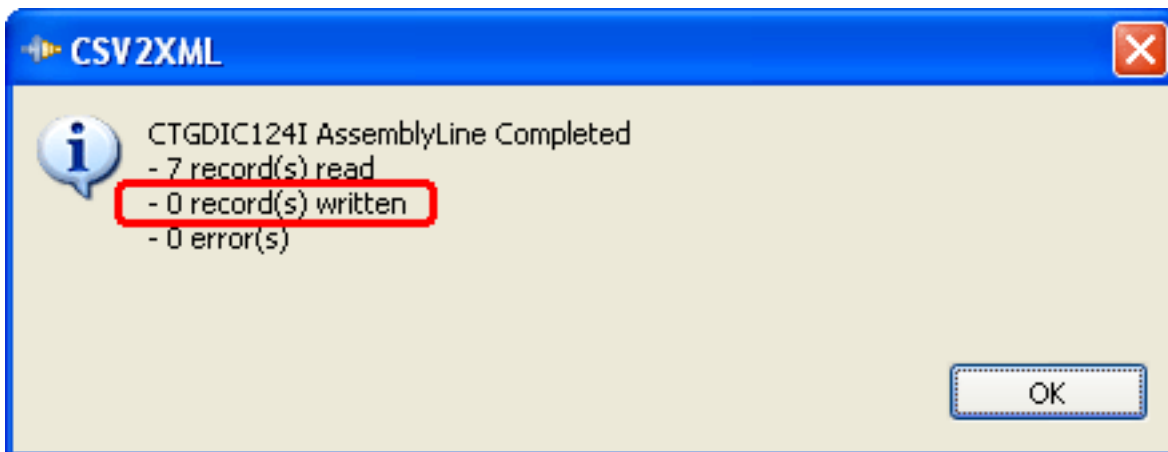


Figure 104. All entries unchanged and skipped

As a final test, bring up the input CSV file and change any of the field values – except for 'Last'<sup>48</sup>. Save the change and then re-run your ETL job and you will see that only modified entries are processed.

## Configuring the output target for Updates

The current setup works fine for output to a file. However, if you were driving these changes to a directory, RDBMS or similar data store then you will want to add new data as well as updating existing records. In order for your EasyETL job to do this you must first select which Output Attribute to use as the criteria for locating the record to modify.

This is done by right-clicking on the Output Attribute you want and selecting the **Use as link criteria** option.

---

<sup>48</sup>. Since this is the attribute used to identify snapshots, any change to its value for an entry will cause it to appear as a *new* record to the Delta Engine.

record(s) processed

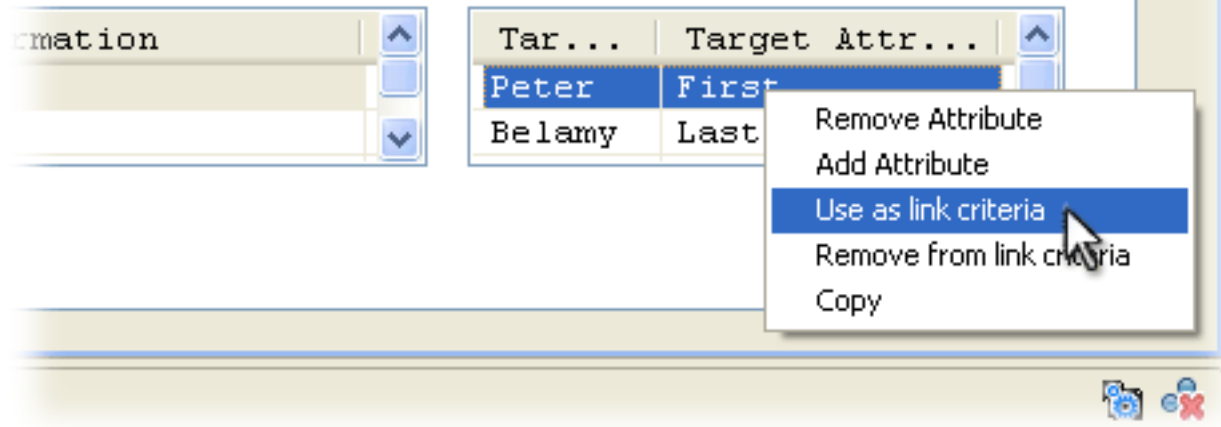


Figure 105. Selecting your link criteria

Now when the Output Connector writes to the target, it first searches for a record using the Link Criteria attribute specified. If no match is found then a new entry is added. If the match was successful then this record is updated.

It's as simple as that: your ETL job has now been configured to provide ongoing synchronization between your input source and output target.

## Command line assets for running and scheduling your ETL job

Once your ETL AssemblyLine is ready for deployment you can right-click on the Project in the Navigator and choose the **Create files needed...** option.

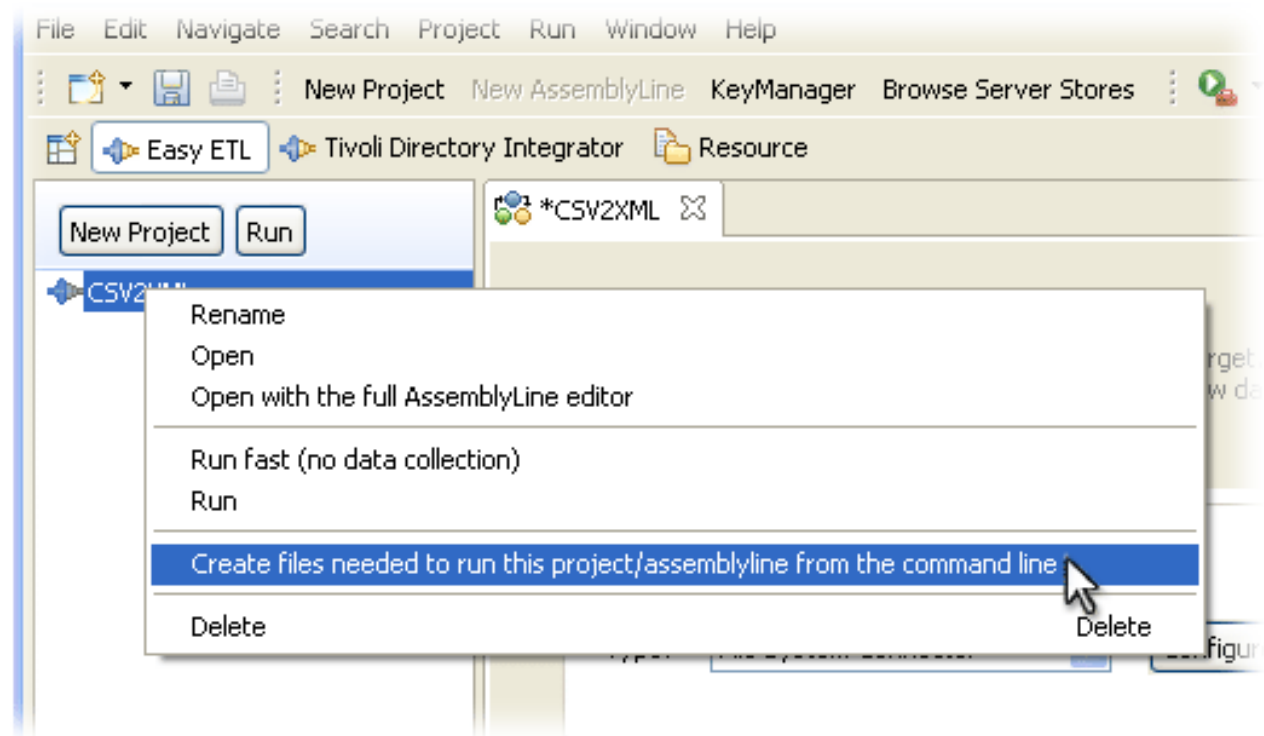


Figure 106. Creating command line assets to run the ETL job

This brings up an Export Files dialog where to write this script/batch-file. Note that it will be given the same name as the Project, so in the case of this tutorial exercise running on Windows it will be called 'CSV2XML.bat'. Executing your EasyETL Project from the command line provides maximum performance for your solution.

You will also get an XML file created in the same location. This is called a TDI *Config* file and contains the details of your EasyETL AssemblyLine that the TDI Server needs to run it. If you open the generated script in a text editor you will see the one-liner needed to start a TDI Server, point it at a Config and then specify the AssemblyLine to run. All you need to do now is set up a scheduled task or cronjob to periodically invoke this script and your synchronization/migration service will be in place.

## Additional options

### High Speed ETL

Although the Data Collector is a powerful tool, your ETL AssemblyLine runs slower due to data collection and presentation on screen. If instead you want your EasyETL AL to process as quickly as possible then you can either select the Project and press the Run button at the top of the Navigator, or right-click the Project and select the **Run fast...** option.

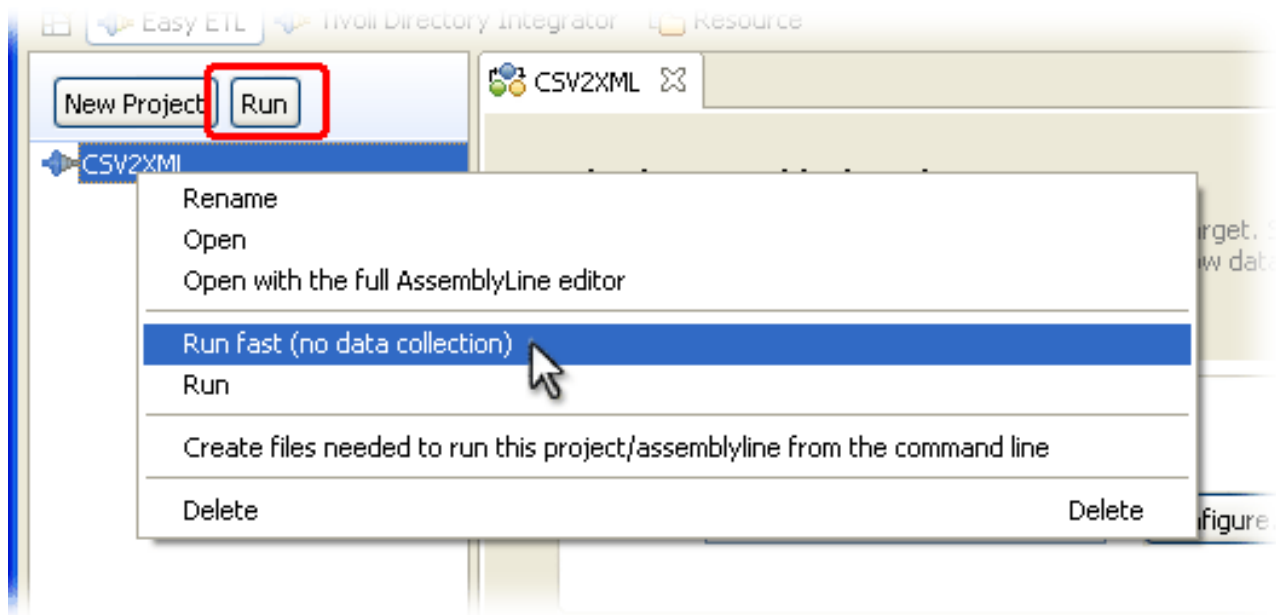


Figure 107. Running your ETL job at full speed

Either option will open a console display where log messages from your AssemblyLine will appear as your AL executes at top speed.

Note that the **Run** option in the Project context menu runs the ETL job with data collection.

### Filtering the input data set

Another powerful feature is the ability to control the contents of your Input data set. This is available whenever your Input source is a database or directory.

For example, select the 'LDAP Connector' for input and take a look at the configuration dialog for this component. Next to the **Search Filter** parameter is a button labeled with three dots (...). This opens up the Link Criteria editor where you can define search rules that will be applied to build the result set for this Connector to read.

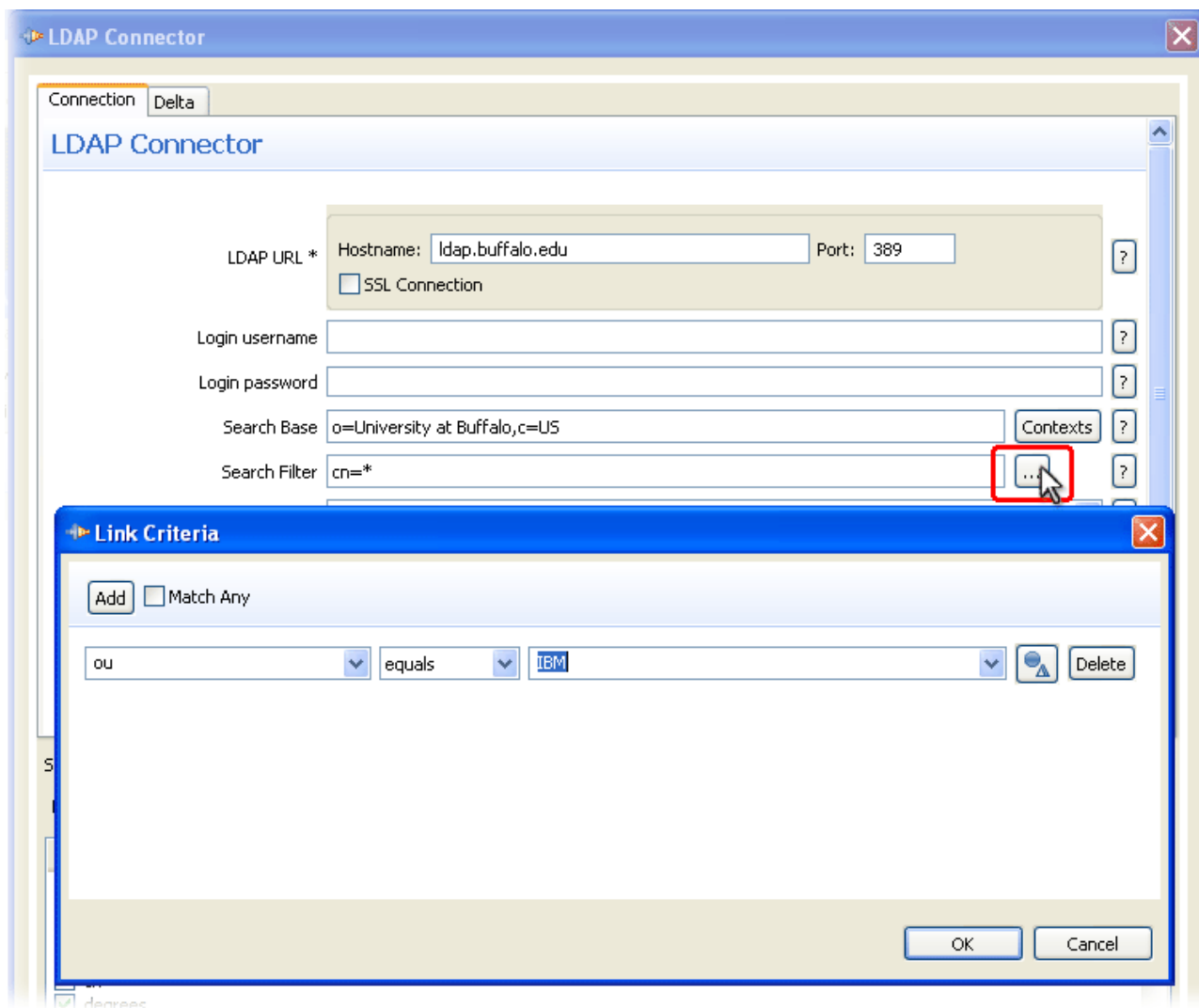


Figure 108. Defining Link Criteria for an Input Connector

This same feature is available for the Database and JDBC Connectors, where you'll find the (...) button next to the **Select** parameter.

Although you can enter the LDAP search syntax yourself directly in the search parameter, this requires you to know the syntax for LDAP search filters or JDBC Select statements. It is often simpler to express the selection you want by using Link Criteria and letting the Connector deal with the underlying syntax.

### Taking your EasyETL AssemblyLine to the next level

Opening your ETL Project in the full-featured TDI AssemblyLine editor lets you to add custom logging and auditing, error handling, failover logic, auto-reconnect, data augmentation (joins) and much more to your migration or synchronization solution. You do this by right-clicking a Project and choosing the **Open with full AssemblyLine editor** option. You'll still be working in the EasyETL Workbench, but you will be able to reach additional functionality available to your AssemblyLine.

If you find this to your liking and are ready to take the plunge then switch to the Tivoli Directory Integrator perspective (**Windows > Open Perspective > Tivoli Directory Integrator**) and starting working in the full TDI Workbench. Better yet - now that you've mastered EasyETL, go back to Chapter 1 and start digging into the full power of Tivoli Directory Integrator.



---

## Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing  
Legal and Intellectual Property Law  
IBM Japan, Ltd.  
1623-14, Shimotsuruma, Yamato-shi  
Kanagawa 242-8502 Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law :**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
2Z4A/101  
11400 Burnet Road  
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. \_enter the year or years\_. All rights reserved.

If you are viewing this information in softcopy form, the photographs and color illustrations might not be displayed.

## Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

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

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

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.

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



Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

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



---

# Index

## A

- accessibility
  - keyboard ix
  - shortcut keys ix
- Accessibility ix, x
- add 7
- AMC 65
- appender 76
- AssemblyLine 5, 13, 26
- assistive technology ix
- Attribute 4
- Attribute Map 5
- Attribute mapping 13
- availability 75, 77

## B

- beginning very basic solutions 3

## C

- commandline 65
- Conn 5
- connectivity 77
- console 66

## D

- Data 4
- data flow 5
- debugger 80
- Debugging 36
- designing solutions 1, 3
- disability ix

## E

- Entry 4
- Event 65
- Event handling 67
- Events 65
- exceptions 77

## F

- fault-tolerance 75

## G

- Getting Started 7

## H

- hardening 75

## I

- IBM Tivoli Directory Integrator
  - projects 1
- inheritance 59
- initialization error 77
- Input Map 5
- installer x
- iterate 7

## J

- JavaScript 1
- join 43

## K

- Keyboard shortcuts ix

## L

- legibility 75
- Link Criteria 61
- Log output 26
- log4j 76
- logging 76
- lookup 7
- Lookup data 43
- Lookup mode 61
- Lookup Mode 54
- lost connection 77

## M

- maintenance 75
- match 61
- missing data 29

## N

- Null behavior 29
- Null values 29

## O

- Output Map 5

## P

- performance 79

## R

- re-use 75
- respond 65
- Run 26
- Run errors 62

## S

- scaling 79
- scheduler 66
- scheduling 66
- screen reader ix
- search 61
- security 75
- shortcut keys
  - keyboard ix
- solutions using IBM Tivoli Directory Integrator 3
- stepper 36

## T

- TINA 67
- trigger 65
- tutorial files 1

## V

- Value 4

## W

- Web server 67







Printed in USA

GI11-9325-01

