Ab Initio Web Services

User's Guide

For use with Co>Operating System Version 2.14

July 28, 2006 > Part Number AB0641

INTELLECTUAL PROPERTY RIGHTS & WARRANTY DISCLAIMER

CONFIDENTIAL & PROPRIETARY

This document is confidential and a trade secret of Ab Initio Software Corporation. This document is furnished under a license and may be used only in accordance with the terms of that license and with the inclusion of the copyright notice set forth below.

COPYRIGHTS

TRADEMARKS

The following are worldwide trademarks or service marks of Ab Initio Software Corporation (those marked ® are registered in the U.S. Trademark Office, and may be registered in other countries):

>®	Continuous Flows®	Enterprise MetaEnvironment	Plan>It
Ab Initio®	Continuous>Flows®	Everything 2 Everything®	Re>Posit®
Ab Initio I>O®	Cooperating Enterprise®	Everything to Everything®	Re>Source®
Abinitio.com®	Cooperating System®	From The Beginning®	Server++®
Applications drawn to scale®	Cooperating®	GDE	Server+Server®
Co>Operating Enterprise	Data>Profiler®	I>O®	Shop for Data®
Co>Operating System®	Dynamic Data Mart®	If You Can Think It, You Can Do It®	The Company Operating System®
Co>Operating®	EME®	init.com	Think, Draw, Run, Scale, Succeed®
Co>Operation®	EME Portal	INIT®	
Co>Operative®	Enterprise Meta>Environment®	Meta Operating System	
Co>OpSys®	Enterprise Metadata Environment	Meta OS	

Certain product, service, or company designations for companies other than Ab Initio Software Corporation are mentioned in this document for identification purposes only. Such designations are often claimed as trademarks or service marks. In instances where Ab Initio Software Corporation is aware of a claim, the designation appears in initial capital or all capital letters. However, readers should contact the appropriate companies for more complete information regarding such designations and their registration status.

RESTRICTED RIGHTS LEGEND

If any Ab Initio software or documentation is acquired by or on behalf of the United States of America, its agencies and/or instrumentalities (the "Government"), the Government agrees that such software or documentation is provided with Restricted Rights, and is "commercial computer software" or "commercial computer software documentation." Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Technical Data and Computer Software provisions at DFARS 252.227-7013(c)(1)(ii) or the Commercial Computer Software – Restricted Rights provisions at 48 CFR 52.227-19, as applicable. Manufacturer is Ab Initio Software Corporation, 201 Spring Street, Lexington, MA 02421.

WARRANTY DISCLAIMER

THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT NOTICE. AB INITIO SOFTWARE CORPORATION MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. AB INITIO SOFTWARE CORPORATION SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGE IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL.

Contents

About this book

SUBJECT

This book describes Ab Initio Web services features included with the Ab Initio Co>Operating System®. These Web services features include graph components, utilities, a plug-in application, samples, reusable subgraphs, and administration tools.

AUDIENCE

This book is for anyone who wants to learn the basic concepts of Ab Initio Web services features — specifically, graph developers who want to create Web services client and provider graphs, and Ab Initio Web services administrators.

PREREQUISITES

To develop Web services graphs, you should be experienced with Ab Initio graph programming, the Ab Initio Graphical Development Environment (GDE™), and the Ab Initio Data Manipulation Language (DML).

Since Web services graphs are continuous flows applications, you should also be familiar with Ab Initio Continuous Flows® programming.

You should have a general knowledge of Web services technology standards such as SOAP, HTTP, XML, and WSDL.

RELATED DOCUMENTATION

You may find the following helpful for understanding the information in this book:

FOR INFORMATION ABOUT	SEE
Continuous Flows	*Guide to Continuous Flows*
Graphical Development Environment (GDE)	Ab Initio Help (online) and *The Tutorial*
Core components	*Component Reference*
Data Manipulation Language (DML)	*Data Manipulation Language Reference* *Data Manipulation Language Core Functions*
Graph programming	*Shell Development Environment User's Guide*
Co>Operating System	*Co>Operating System Graph Developer's Guide* *Co>Operating System Architect's and Administrator's Guide*
Software upgrades	*Release Notes*

AB INITIO DOCUMENTATION

Ab Initio provides documentation in online help, in Adobe PDFs, and in print.

PDFs and printed books

If you do not have the PDFs or printed books you need, please contact your Ab Initio account representative.

Release Notes

- To view *Release Notes* either for the Co>Operating System that is your current run host or for the GDE, select **Help > Release Notes** from the GDE menu bar.
- To view *Release Notes* for the Data Profiler, select **Help > Release Notes** from the Data Profiler menu bar.
- To obtain *Release Notes* for other Ab Initio products, contact your Ab Initio account representative.

Basic book set

The following is the basic set of Ab Initio books:

SUBJECT	BOOK
Co>Operating System	*Co>Operating System Graph Developer's Guide* *Co>Operating System Architect's and Administrator's Guide*
Latest information on the Co>Operating System	*Release Notes*
The core components	*Component Reference*
Connecting to and using databases	*Database Package Guide and Reference*
Data Manipulation Language (DML)	*Data Manipulation Language Reference*
DML built-in functions	*Data Manipulation Language Core Functions*
Graphical Development Environment (GDE)	*The Tutorial*
Plan>It™	*Plan>It Guide and Reference*
Ab Initio Web Services	*Web Services User's Guide*

Other documentation

Other Ab Initio products and their corresponding documentation are the following:

PRODUCT	DOCUMENT
Continuous Flows	*Guide to Continuous Flows*
Data Profiler	*Data Profiler Guide and Reference* *Release Notes*

PRODUCT	DOCUMENT
Enterprise Meta>Environment® (EME®)	*EME Guide for Developers*
	EME Enterprise Metadata Guide
	EME Reference
	Release Notes
Shop for Data® (SFD)	*Shop for Data Administrator's Guide and Reference*
	Shop for Data Developer's Reference
	Shop for Data Developer's Guide
	Release Notes
	Shop for Data User Help

DOCUMENTATION CONVENTIONS

Ab Initio documentation — both online and in print — uses font formatting and symbols to convey special meaning of particular elements in text, examples, and syntax.

Conventions for text

EXAMPLE	DESCRIPTION
mainfile.txt	Bold represents user-entered values, file paths, DML keywords, numerical or character values (such as **abc** or **1**), as well as the names of fields, functions, statements, and parameters.
File > Open	Bold represents text that appears in the user interface. For example: "Choose the **Open** command from the **File** menu."
AB_HOME	Small caps represent names of configuration, environment, and system variables and names of components.
my_file	Italic represents variables as well as emphasis, new terms, and book titles.

Conventions for examples

EXAMPLE	DESCRIPTION
`source code`	A fixed-width font represents code fragments, examples, and computer output.
`user input`	A bold fixed-width font represents entries made by the user in an interactive command-line session.
⇒	An arrow indicates the result of a computation.
␣␣␣␣	Squares indicate blank spaces where the number of spaces is significant.

Conventions for syntax

EXAMPLE	DESCRIPTION
integer	Non-bold text indicates the DML data type of an argument or returned element.
read_xml	Bold text, symbols, and punctuation represent syntactical elements that you must enter exactly as they are shown.
my_file	Italicized text represents values or variables that you must supply.
...	An ellipsis indicates that the preceding item can be repeated one or more times.
a | b | c	The logical **or** symbol separates alternatives.
( )	Bold parentheses are part of the syntax.
[ ]	Non-bold square brackets surround one optional item or a series of optional items. If optional items are separated by the logical **or** symbol, you may choose one or none. If the **or** symbol is not present, you may choose as many as you want or none.
[ ]	Bold square brackets are part of the syntax.
{ }	Non-bold curly braces indicate a series of choices from which you *must* select one.
{ }	Bold curly braces are part of the syntax.

Conventions for standards

EXAMPLE	DESCRIPTION
kilobyte (kB), megabyte (MB), gigabyte (GB)	In Ab Initio software and documentation, the term *kilobyte* (abbreviated kB) represents 1024 (2^{10}) bytes. The term *megabyte* (abbreviated MB) represents 1,048,576 (2^{20}) bytes, and the term *gigabyte* (abbreviated GB) represents 1,073,741,824 (2^{30}) bytes. Similarly, all *-byte* terms refer to powers of 2 rather than powers of 10.

1

About Ab Initio Web services

This chapter introduces the basic concepts of the Ab Initio Web services features and the Web services provider and client graphs. (For a detailed discussion of how to build these graphs, see Chapter 2 "Developing Web services graphs".)

This chapter includes the following topics:

Overview of Ab Initio Web services

With the Ab Initio Web services features, you can create graphs that *provide* Web services and graphs that *call* Web services. These graphs use industry standards — SOAP, HTTP, XML, and WSDL — to communicate over a network with other graphs and applications. The Ab Initio Web services features include graph components, utilities, a plug-in application, samples, reusable subgraphs, and administration tools. This section includes these topics:

WEB SERVICES GRAPH STANDARDS

Transmitting SOAP via HTTP (HyperText Transfer Protocol) is the most fundamental and popular service-oriented standard for Web services provider and client graphs. With the Ab Initio Web services features, you can create client and provider graphs that use this widely used standard.

You can also develop Web services graphs that transmit SOAP via an IBM WebSphere MQ (formerly known as MQSeries) queue or JMS (a Java Message Service queue or topic). These graphs use Ab Initio MQ and JMS connector components to connect with IBM WebSphere MQ application servers and Java EE (Java Platform, Enterprise Edition, formerly known as Java 2 Platform, Enterprise Edition or J2EE) application servers.

When you expose a graph as a Web services provider, it can handle SOAP Web services requests from client applications — including graphs — and respond by providing the requested services. When you develop a client graph, it can request SOAP Web services over a network from Web services provider applications, including Web services provider graphs.

WEB SERVICES SPECIFICATIONS

The following specifications are used for Ab Initio Web services:

SPECIFICATION	WHAT IT DESCRIBES
Simple Object Access Protocol (SOAP)	The format and meaning of SOAP request, response, and fault messages
Web Services Description Language (WSDL)	The XML-based format for defining the external interface to a Web service
Web Services Interoperability (WS-I)	Restrictions to the above two specifications; also includes the *WS-I Basic Profile:* • These interoperability specifications ensure that your Web services provider graphs and clients are interoperable with those implemented using other vendors' infrastructure and tools, assuming they also comply with WS-I standards. • The Web services utilities, **dml-to-wsdl** (page 66) and **wsdl-to-dml** (page 70), process WSDL files that comply with the *WS-I Basic Profile* 1.0 standards

WEB SERVICES SAMPLES

The Ab Initio Web services features include samples — sample Web services graphs, a sample WSDL file, sample Schema DML, and sample business DML. Sample graphs, such as **BankWebService**, can give you an idea of how the Web services graphs work.

You can find Web services samples in subfolders in **$AB_HOME/examples/web-services/simple**:

FOLDER	CONTENTS
components	Sample subgraphs that are parts of typical Web services provider graphs. For example, this folder contains a sample business logic subgraph, **BalanceLogic**, that processes requests for the balance on a specified account ID.
data	Sample data used with the sample Web services graphs. For example, the sample business logic subgraph **TransactionsLogic** uses data from **transactions.dat** in this folder.
dml	Sample DML type definitions and record formats for Schema and business DML used in sample Web services provider graphs.
mp	Sample Web services provider graphs, such as **BankWebService**.
test	**soap_rpc_client.c**, the source file of a nongraphical command-line test client, and a Makefile.
wsdl	A sample WSDL file, **CardTransactions.wsdl**, generated using the **dml-to-wsdl** (page 70) utility.

FOLDER	CONTENTS
xml	Sample SOAP requests for testing Web services provider graphs.
dotnetclient	**BankWebServiceClient.exe**, a test client for testing your Web services provider graphs. This Web services test client runs only with .NET on Windows.
aspdotnetclient	Another Web services test client, **AIBankWebServiceClientSetup.msi**, and related sources. This test client is an IIS application that runs only with ASP.NET on Windows. You can use this test client in a browser.

WEB SERVICES REUSABLE SUBGRAPHS AND SAMPLES

We provide reusable Web services subgraphs, such as **RPC Transport Layer**, so that you can drag them into the GDE and use them as templates as you start to developing your own Web services graphs. You can find reusable subgraphs, Web services components, DML formats, APIs, and other items related to Web services features in the following folders:

FOLDER	CONTENTS
$AB_HOME/connectors/SOAP	Reusable subgraphs, such as **RPC Transport Layer** and **SOAP Layer**, that you can copy (drag and drop) into the GDE as you start developing your own SOAP Web services provider and client graphs (see Chapter 2 "Developing Web services graphs")
$AB_HOME/connectors/RPC	RPC connector components used in Web services graphs
$AB_HOME/connectors/RPC/rpcheader.dml	The DML format of the message header for the RPC connector components
$AB_HOME/connectors/Internet	Batch and continuous versions of the CALL WEB SERVICE component used to make synchronous calls from client graphs for SOAP/HTTP Web services
$AB_HOME/connectors/JMS	JMS connector components that you can use in Web services graphs to connect to Java EE application servers
$AB_HOME/connectors/MQSeries	MQ connector components that you can use in Web services graphs to connect to IBM WebSphere MQ application servers

THE WEB SERVICES PLUG-IN

To run Web services graphs, you need the Ab Initio Web services plug-in, an application that processes requests and responses between Web services clients and providers. You install and run the plug-in on an application server. Separate versions of the plug-in are available for Microsoft IIS and Java EE application servers. To get a copy of the plug-in, contact Ab Initio Support.

The Web services plug-in communicates with a set of several Web services provider graphs. The Web services administrator adds each of these graphs to the plug-in's Web services list. The plug-in uses the URL in a client HTTP request message to direct the request to a particular Web services provider graph.

You use the Ab Initio **Web Services** administration Web page to administer both your plug-in and the Web service provider graphs registered with a plug-in. Using the **Web Services** administration page, you can add, delete, configure, and monitor a plug-in's Web services provider graphs. For more information installing the plug-in and administering Web services, see Chapter 3 "Administering Web services" (page 39).

THE WSDL FILE

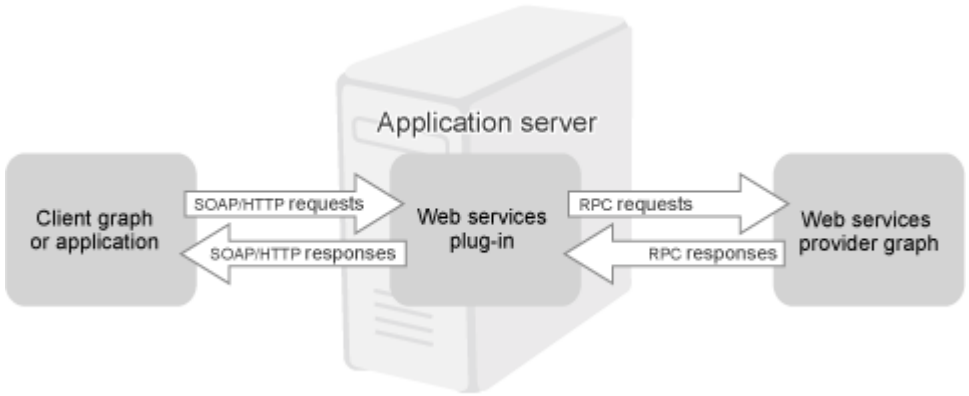
Developing a Web services provider graph requires having a valid WS-I compliant WSDL (Web Services Description Language) file that defines the public or external interface to your graph. A *WSDL file* contains XML-based descriptions of all the messages and operations needed to interact with your Web services provider graph. The *Specification for the Web Services Description Language (WSDL)* and the *WS-I Basic Profile* describe the standards for a valid WS-I compliant WSDL file.

Your organization or industry can define a WSDL file for accessing your Web services, or you can define your own WSDL file, from scratch or based on existing business logic. If you already have defined your business logic, you can use the **dml-to-wsdl** (page 66) utility to generate a WSDL file from the DML record formats and types used in that business logic. For more information, see "Defining the external interface" (page 22).

Regardless of how you got your WSDL file, you must use the **wsdl-to-dml** (page 70) utility to convert the XML-based descriptions in the WSDL file into Schema DML to use in your Web services provider graph. For more information, see "Defining the internal interface" (page 24).

Ab Initio Web services architecture

The following diagram shows the high-level architecture of the Ab Initio Web services features.



MAJOR ELEMENTS

The major elements of the Web services architecture are:

- A client graph or application that calls for Web services

  The client graph or application sends SOAP requests and receives SOAP responses via HTTP. For more information about Web services client graphs, see "Example of a Web services client graph" (page 18).

- An Ab Initio Web services plug-in

  The Ab Initio Web services plug-in is an application you install and run on an application server. It is available for Microsoft IIS and Java EE application servers. For more information, see "The Web services plug-in" (page 5) and Chapter 3 "Administering Web services" (page 39).

  The plug-in processes requests and responses between Web services clients and Web services provider graphs. It communicates with the provider graph via Ab Initio RPCs (Remote Procedure Calls).

- A Web services provider graph

  The Web services provider graph is a continuous nonrecoverable graph that handles client calls for Web services. For more information, see "Example of a Web services provider graph" (page 8) and "Developing Web services graphs" (page 21).

MAJOR PROCESSING STEPS

The major processing steps in the Web services architecture are:

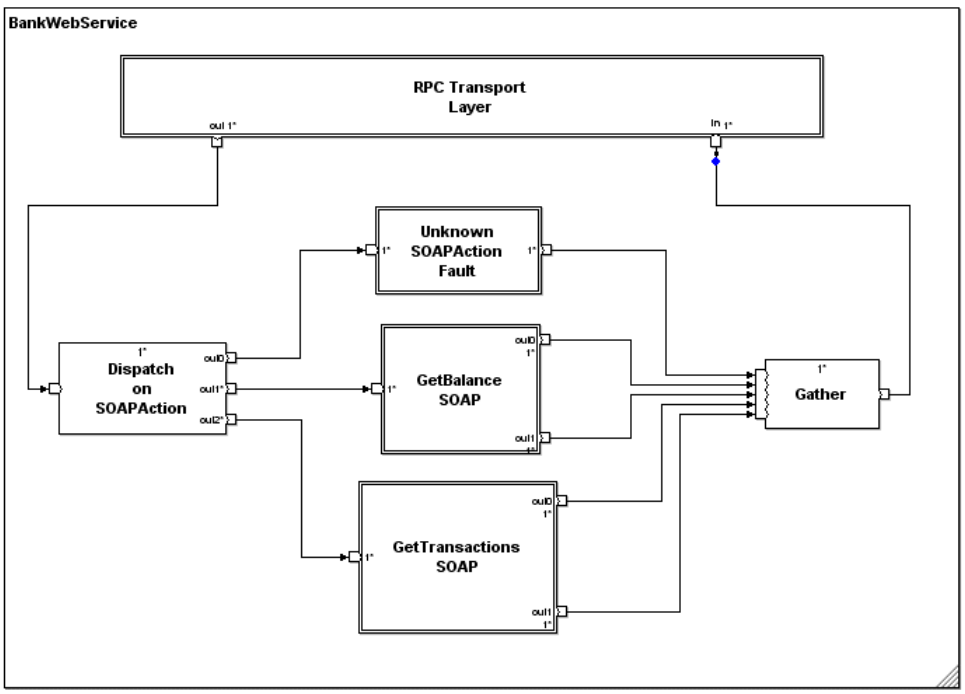
1. The client graph or application calls for Web services by sending a SOAP request via HTTP to an Ab Initio Web services plug-in running on an application server. The URL in the SOAP request indicates the target application host, server, plug-in, and Web services provider graph.
2. The plug-in wraps the SOAP request as an RPC request and sends it to the Web services provider graph.
3. The Web services provider graph performs the requested service and returns a SOAP response or fault wrapped as an RPC response to the plug-in.
4. The plug-in extracts the SOAP response corresponding to the original SOAP request and sends it to the client via HTTP.

Example of a Web services provider graph

This section gives an example of a Web services provider graph, introduces the basic concepts, and provides background information to give you a better understanding of how this graph works. This section includes these topics:

BASICS

The figure below shows the sample SOAP Web services provider graph **BankWebService**:



You can find this sample graph, along with other Web services samples and supporting data, in the folder **$AB_HOME/examples/web-services/simple**. For more information, see "Web services samples" (page 3) and "Web services reusable subgraphs and samples" (page 4).

What does the sample provider graph do?

The sample Web services provider graph handles SOAP requests containing a credit card account ID and a request for either of the following services:

- The balance for the account ID
- A list of all transactions on the account ID for a given time period

This graph also returns faults. For example, the graph returns a fault when a request contains an unrecognized account ID or asks for an unrecognized operation. For more information, see "Error handling" on page 16.

General characteristics

The Web services provider graph is a nonrecoverable continuous graph. It should have an **AB_GRAPH_RUN_MODE** parameter at the graph level set to **continuous-nonrecoverable**. This parameter is not an input parameter. It should be set to **type=string** and **"export to environment"** so that it propagates down to all subgraphs and components in the provider graph.

This type of graph requires one computepoint per input record at the top level. Each request record produces exactly one response record or exactly one fault record, but never both. At the top level, request records and response records must not be reordered.

Subgraph layers of this graph (such as the business logic layer) can decompose a request record into multiple subrecords and recompose these records into a single response record. And components such as NORMALIZE in the subgraph layers can reorder their subrecords during processing, as long as the original record order is retained at the top level.

Most commonly, Web services provider graphs use RPC SUBSCRIBE, RPC PUBLISH, XML READ, and XML WRITE components. In graphs that provide JMS or MQ Web services, you use JMS PUBLISH, JMS SUBSCRIBE, MQ PUBLISH HEADERS, and MQ SUBSCRIBE components. For detailed descriptions of these components, see Ab Initio Help. For information about all the major components used in Web services graphs, see Chapter 6 "Web services components" (page 71).

The sample WSDL file

The sample Web services provider graph, **BankWebService**, uses the WSDL file **CardTransactions.wsdl**. You can find this sample WSDL file in the **wsdl** folder in the **$AB_HOME/examples/web-services/simple** folder.

This WSDL file contains XML-based descriptions of all messages and operations needed to interact with the sample Web services provider graph. For example, the sample WSDL file contains an XML description of the operation for getting the balance on an account ID, along with input and output types and fault types for this operation. For more information about how this WSDL file was generated, see "Defining the external interface" (page 22).

PROVIDER GRAPH ARCHITECTURE

The Web services provider graph is organized in functional layers with a subgraph for each layer. This section gives an overview of this architecture — how these layers are organized and how they work together.

The following sections give more details about how each layer works.

- The RPC transport layer (page 11)
- The operation layer (page 13)
- The SOAP layer (page 14)
- The business logic layer (page 16)
- Error handling (page 16)

What does each layer do?

The major layers of the Web services provider graph and their functions are as follows:

LAYER	FUNCTION
RPC transport	Transports Remote Procedure Call (RPC) messages in and out of the graph, ensuring that RPC headers remain unchanged
Operation subgraph	Contains the SOAP translation layer and the business logic layer and processes requests and responses as they go in and out of the SOAP layer and the business logic layer
SOAP translation	First translates XML-described SOAP requests into DML-described records that can be handled by the business logic layer, then translates DML-described data received from the business logic layer back into XML-described SOAP responses
Business logic	Performs requested business operations

Putting the RPC transport and SOAP translation mechanisms in separate layers allows you to change the transport mechanism without changing the translation mechanism.

Putting the business logic in a separate layer makes it easy to reuse it in other graphs, including nonrecoverable continuous graphs, regular continuous graphs, and batch graphs.

How do the layers work together?

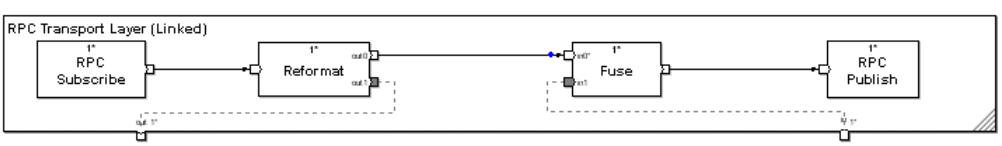
Briefly, here's how the Web services provider graph layers work together:

1. The RPC transport layer:
   a. Receives an RPC from the plug-in.
   b. Removes the RPC header.
   c. Sends the SOAP request to the **Dispatch On SOAPAction** REFORMAT component.
2. The **Dispatch On SOAPAction** REFORMAT component uses the SOAPAction field in the request to determine which operation subgraph should handle the message, as follows:
   - If the request is for the balance on a specified account ID, it goes to **GetBalance SOAP**.
   - If the request is for the transactions on the account ID, it goes to **GetTransactions SOAP**.
   - If the requested SOAPAction is unrecognized, the request goes to the **Unknown SOAPAction Fault** subgraph, which adds a fault string and failure status to the response.
3. The operation subgraph processes Web services requests and responses as they go in and out of the SOAP translation layer and the business logic layer.
4. The SOAP translation layer translates records (including faults) to and from the business logic subgraph.
5. The business logic subgraph performs the requested operation and returns a response record or a fault.
6. This record is converted back into a SOAP response or fault that goes back up to the RPC transport layer and out of the provider graph.

THE RPC TRANSPORT LAYER

The RPC transport layer transports RPCs in and out of the Web services provider graph. The main purpose of this layer is to ensure that RPC headers remain unchanged during the processing of SOAP messages.

This layer is a generic, reusable subgraph that you can copy and adapt to your operations. The figure below shows the **RPC Transport Layer** subgraph:



RPC transport layer processing steps

The major processing steps of the RPC transport layer are:

1. RPC SUBSCRIBE receives the request from the plug-in and adds the RPC header.
2. REFORMAT separates the RPC header from the data payload.
3. The SOAP request header and the data payload pass down to the SOAP layer.
4. A SOAP response header and the data payload of the response or a fault record pass back up from the business logic layer and the SOAP layer.
5. FUSE fuses the RPC header to the response or fault.
6. RPC PUBLISH sends the RPC response or fault out of the graph back to the plug-in.

RPC connectors DML

The RPC SUBSCRIBE **out** port has a DML type definition for the RPC header and the DML record format for the SOAP request record, as follows:

```
include "~$AB_HOME/connectors/RPC/rpcheader.dml";
include "~$AB_HOME/connectors/SOAP/SOAPRequest.dml";

metadata type = record
  rpcheader hdrs;
  SOAPRequest soaphdrs;
  string(big endian integer(4)) body;
end;
```

The RPC PUBLISH **in** port has a DML type definition for the RPC header and the DML record format for the SOAP response record, as follows:

```
include "~$AB_HOME/connectors/RPC/rpcheader.dml";
include "~$AB_HOME/connectors/SOAP/SOAPResponse.dml";

metadata type = record
  rpcheader hdrs;
  SOAPResponse soaphdrs;
  string(big endian integer(4)) body;
end;
```

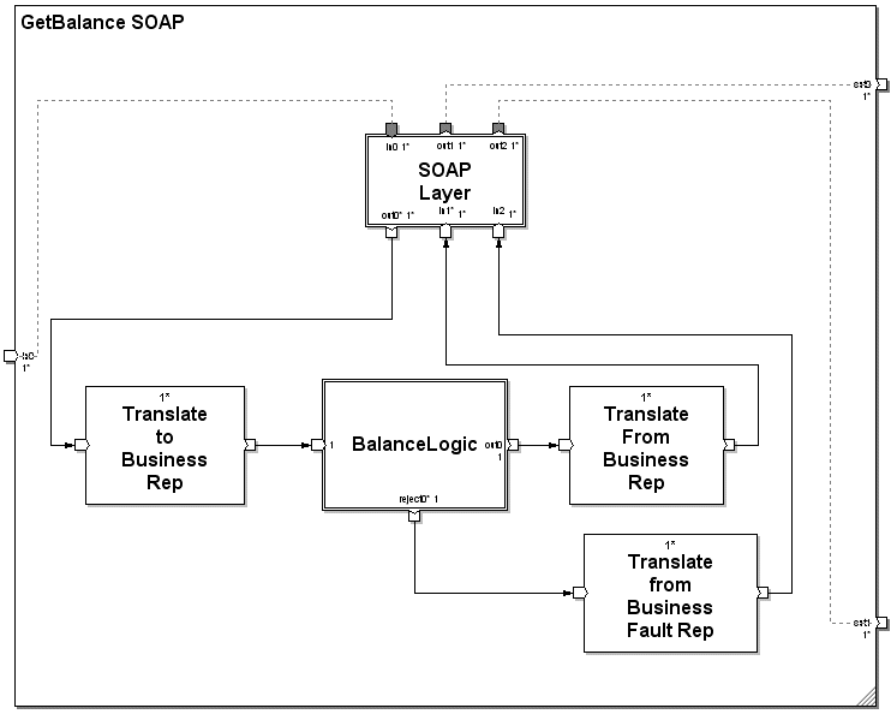
THE JMS TRANSPORT LAYER

The JMS transport layer is a generic, reusable subgraph. You can copy this sample subgraph and use it as a template when you are building a Web services provider graph that communicates with a plug-in running on a Java EE application server.

This subgraph contains JMS connector components — JMS SUBSCRIBE and JMS PUBLISH — that transmit SOAP via JMS (a Java Message Service queue or topic) instead of using Ab Initio RPC. For more information on the JMS transport layer subgraph and the Java EE version of the plug-in, contact Ab Initio Support.

THE OPERATION LAYER

The operation layer performs the requested operation. It contains the SOAP layer and the business logic layer. The figure below shows the sample operation subgraph, **GetBalance SOAP**, that processes Web services requests and responses as they go in and out of the SOAP translation layer and the business logic layer. This operation subgraph contains the SOAP layer subgraph, **SOAP Layer**, and the business logic subgraph, **BalanceLogic**.



XML READ and XML WRITE components in the SOAP layer subgraph (see "The SOAP layer" on page 14) and REFORMAT components (**Translate to Business Rep** and **Translate from Business Rep**) in the operation subgraph use Schema DML to translate between the XML-text data of the SOAP request and response and the DML-described data of the records that the business logic subgraph processes.

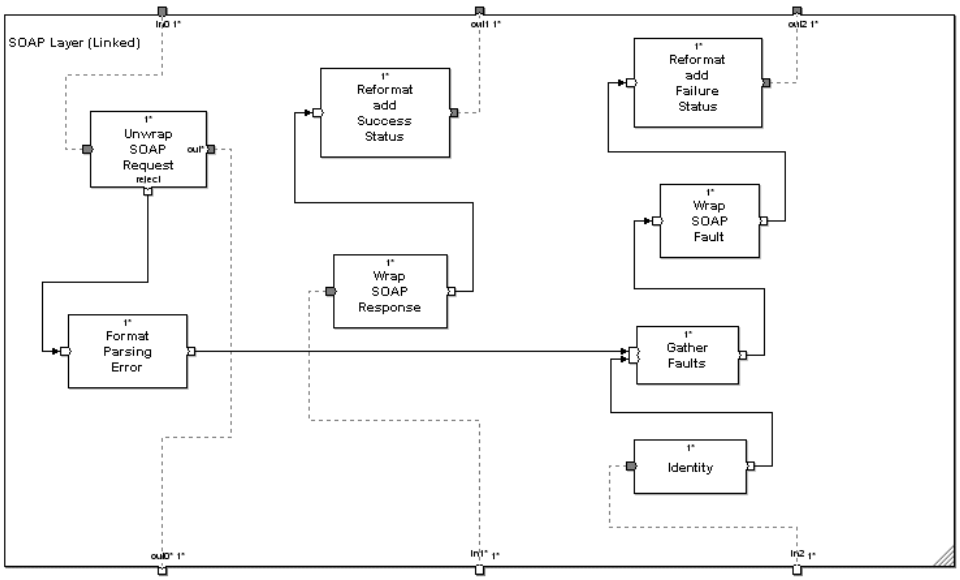
Schema DML represents the XML structure of the input and output types for each operation in a Web services provider graph and thus defines the internal interface of the business logic subgraph. For more information, see "How Schema DML works" (page 24).

The REFORMAT components in the operation layer work as follows:

- **Translate to Business Rep** translates between Schema DML and business DML on the input record received from the SOAP layer and sent to the business logic subgraph, **BalanceLogic**.
- **Translate from Business Rep** translates between business DML and Schema DML on the output record received from **BalanceLogic** and sent to the SOAP layer.
- **Translate from Business Fault Rep** translates fault messages from **BalanceLogic** for handling by the SOAP layer.

THE SOAP LAYER

The SOAP layer is a reusable subgraph that translates records (including faults) to and from the business logic subgraph, **BalanceLogic**. The figure below shows the sample SOAP layer subgraph, **SOAP Layer**.



There are three paths for messages into and out of the SOAP layer. One path is for successful messages, and the two other paths are for faults.

One fault occurs for requests with invalid syntax, SOAP parsing errors. The other fault occurs in the business logic layer because of invalid data or a request for an operation that the business logic layer does not perform.

Successful messages

Successful messages follow this path through the SOAP layer:

1. **Unwrap SOAP Request**, a READ XML component, reads in the SOAP request.
2. If the syntax of the request is valid, **Unwrap SOAP Request** uses Schema DML to translate the XML-described SOAP request into a DML-described record for processing by the business logic subgraph.
3. After the business logic subgraph successfully processes the request, it goes to the WRITE XML component **Wrap SOAP Response**.
4. **Wrap SOAP Response** translates the DML-described data received from the business logic subgraph into an XML-described SOAP response and sends it to the REFORMAT component **Reformat Add Success Status**.
5. **Reformat Add Success Status** adds a success status code to the response and sends it up to the RPC transport layer.

Syntax faults

Faults due to invalid syntax follow this path through the SOAP layer:

- If **Unwrap SOAP Request** cannot translate the request because of invalid syntax, it sends the invalid request via the **reject** port to the **Format Parsing Error** component.
- The invalid syntax fault response then goes to the **Gather Faults** component.
- The three-component chain of **Gather Faults**, **Wrap SOAP Fault**, and **Reformat Add Failure Status** gathers the business logic faults along with the syntax faults, wraps each fault as a SOAP fault message, adds a failure status code to the SOAP response header, and sends the fault response up to the RPC transport layer.

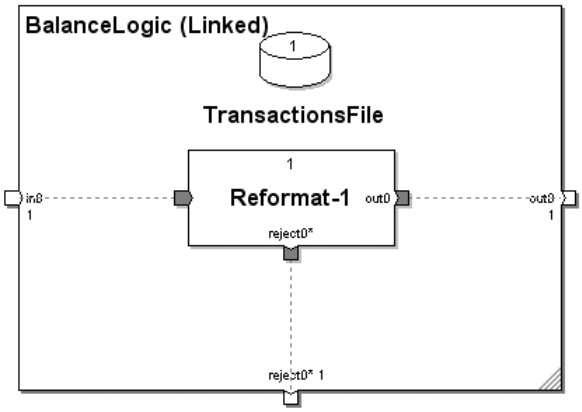
Business logic faults

Faults due to errors in the business logic layer follow this path through the SOAP layer:

- **Unwrap SOAP Request** translates the SOAP request into a DML-described record and sends it on for processing by the business logic layer.
- The business logic layer sends fault responses up to the REFORMAT component **Identity**, which passes them on to the **Gather Faults** component.
- The three-component chain of **Gather Faults**, **Wrap SOAP Fault**, and **Reformat Add Failure Status** gathers the business logic faults along with the syntax faults, wraps each fault as a SOAP fault message, adds a failure status code to the SOAP response header, and sends the fault response up to the RPC transport layer.

THE BUSINESS LOGIC LAYER

The business logic layer performs the operation requested by the Web services client. The figure below shows the business logic subgraph, **BalanceLogic**.



In this simple example, **Reformat-1** in the **BalanceLogic** subgraph computes the balance for a requested credit card account ID, or outputs a fault record to the **reject** port as described in "Error handling" (page 16).

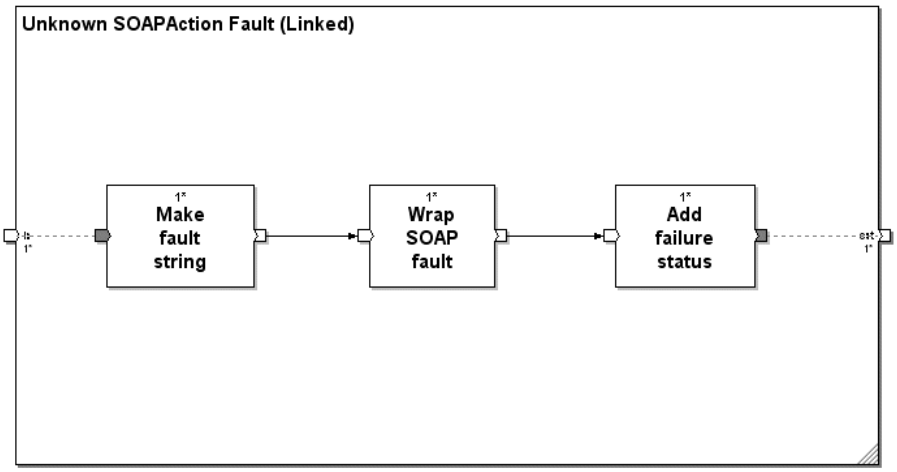
It is possible to wrap very complex business logic into the business logic layer of your graph. But for simplicity, this sample business logic subgraph reads in records from a LOOKUP FILE, **Transactions File**.

ERROR HANDLING

The sample Web services provider graph includes simple error handling. In case of an error, the graph needs to generate and return an appropriate SOAP fault response. All the faults are gathered for output at the top level of the graph. As with nonfault responses, at the top level of the graph it is essential that only one fault record be returned per request.

Unknown SOAPAction faults

The figure below shows the **Unknown SOAPAction Fault** subgraph that handles errors when the request is for anything other than one of two services provided by the operation subgraphs. The **Unknown SOAPAction Fault** subgraph adds a fault string and a failure status to the response.



Syntax error faults

There is also error handling for the case when the request is for one of the recognized operations but when there is an error in the syntax of the request. This error handling occurs in the SOAP layer (see "The SOAP layer" on page 14) where invalid requests are gathered and wrapped as SOAP fault messages, and where a failure status code is added to the SOAP response header.

Invalid data faults

Finally, there is error handling in the business logic layer (see "The business logic layer" on page 16) for invalid data. The **Reformat-1** component sends fault records via the **reject** port to the **Translate from Business Fault Rep** REFORMAT component. This component translates the fault record from a DML-described fault to a SOAP fault.

This fault then goes back up the business logic fault-handling path in the SOAP translation layer subgraph. For details, see the information about fault responses due to errors in the business logic layer in "The SOAP layer" (page 14).

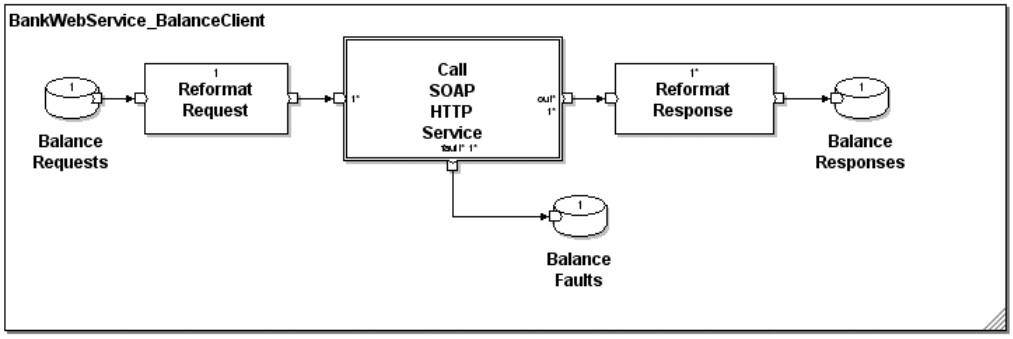
Example of a Web services client graph

This section gives an example of a Web services client graph, introduces the basic concepts, and provides background information to give you a better understanding of how this graph works. This section includes these topics:

- Basics (next)
- The HTTP client subgraph (page 19)
- The RPC client subgraph (page 20)

BASICS

Here's an example of a Web services client graph that calls SOAP Web services via HTTP. This example introduces the basics of how this type of graph works. For an architectural overview of how the client graph interacts with the Web services plug-in and the SOAP Web services provider graph, see "Ab Initio Web services architecture" (page 6).

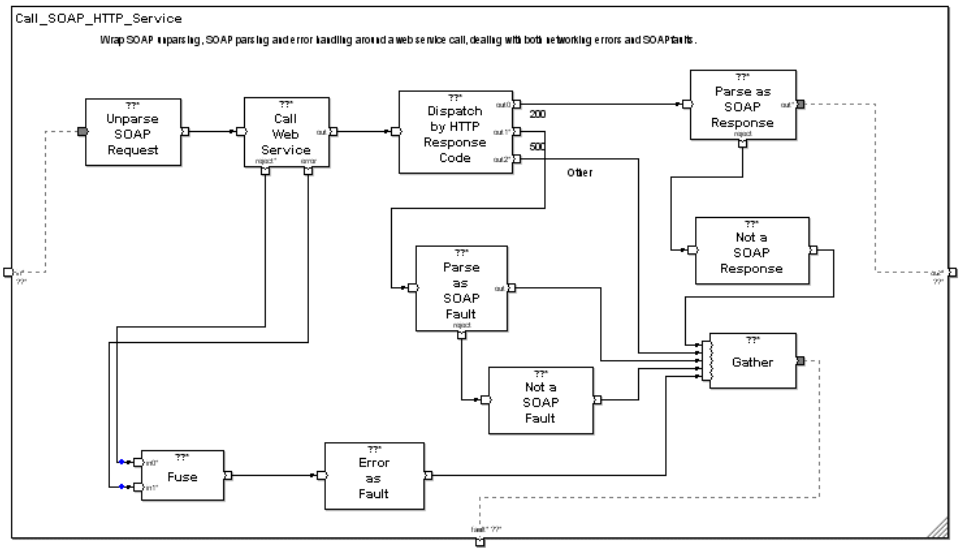


The **Reformat Request** component in this graph receives a DML-described record from **Balance Requests**. This record contains an account ID and requests the balance on that account. The request goes to the **Call SOAP HTTP Service** subgraph (see "The HTTP client subgraph" on page 19).

This client subgraph translates the DML-described request into an XML-described SOAP request and sends it via HTTP to a Web services plug-in running on an application server. The URL in the request indicates which application server, plug-in, and Web services provider graph to send the request to.

THE HTTP CLIENT SUBGRAPH

The heart of the sample Web services client graph is the reusable linked subgraph **Call SOAP HTTP Service**. You can find this subgraph in **$AB_HOME/connectors/SOAP**.



HTTP client subgraph processing

This subgraph processes records as follows:

- On its **in** port, this subgraph receives request records in the format described by the Schema DML.
- This subgraph has two **out** ports. One **out** port is for successful responses, also described by Schema DML. The other **out** port is for failed requests.
- The error output is always formatted as a SOAP fault and is described by the DML in **$AB_HOME/connectors/SOAP/GenericFault.dml**.

The HTTP client subgraph parameters

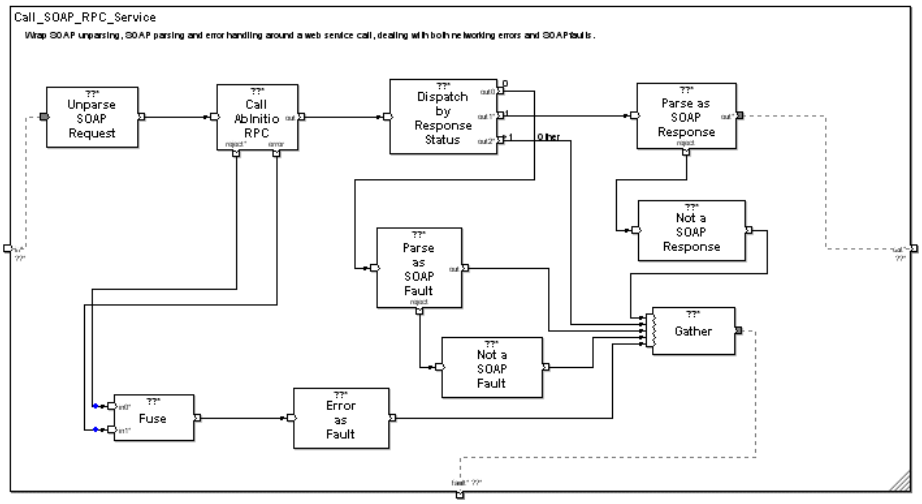
The **Call SOAP HTTP Service** subgraph must have the following parameters set:

- **URL** specifies the URL of the target Web service provider.

  For more information about how clients find out where to send Web services requests, see "Discovering the URL of a Web service" (page 48).
- **SOAPAction** specifies the requested action the client wants the Web service provider to perform.

THE RPC CLIENT SUBGRAPH

$AB_HOME/connectors/SOAP contains the **Call SOAP RPC Service** client subgraph, which uses the CALL AB RPC component and calls Web services via RPCs rather than via HTTP. This client subgraph sends requests directly to an Ab Initio Web services provider graph without connecting to an application server running the plug-in.



RPC client subgraph processing

The processing logic of the **Call SOAP RPC Service** client subgraph is similar to that of the **Call SOAP HTTP Service** subgraph, as follows:

- On its **in** port, this subgraph receives request records in the format described by the Schema DML.
- This subgraph has two **out** ports. One **out** port is for successful responses, also described by Schema DML. The other **out** port is for failed requests.
- The error output is always formatted as a SOAP fault and is described by the DML in **$AB_HOME/connectors/SOAP/GenericFault.dml**.

The RPC client subgraph parameters

The **Call SOAP RPC Service** subgraph must have the following parameters set:

- **Call_AbInitio_RPC_host** specifies the host on which the Web services provider graph's RPC SUBSCRIBE and RPC PUBLISH components are running.
- **Call_AbInitio_RPC_port** specifies the port on which the Web services provider graph's RPC SUBSCRIBE and RPC PUBLISH components are listening.

2

Developing Web services graphs

Though you can develop a Web services provider graph from scratch, you do not need to do so. If you're familiar with graph development, we suggest starting with the reusable subgraphs and other Web services samples in **$AB_HOME/examples/web-services/simple**. To begin developing a Web services provider graph using these samples as templates, all you need to do is drag and drop the reusable subgraphs into the GDE and configure them for the operations of your own graph.

This chapter describes the major tasks for developing a SOAP Web services provider graph using reusable subgraphs. These tasks are described in the following sections:

Defining the external interface

The first requirement for developing a Web services provider graph is a valid WS-I compliant WSDL file that defines the external interface to your graph (how clients communicate with your Web service operations). The WSDL file contains XML descriptions of the protocol bindings and message formats required to interact with your Web service operations.

GETTING A WSDL FILE

You have several options for getting a WSDL file for your Web services provider graph:

- If your organization has created a WSDL file that defines the external interface to your Web services, you can use it.
- You can write a WSDL file from scratch, perhaps using a third-party tool.

  For information about the specifications and standards for writing a valid WS-I compliant WSDL file, see "Web services specifications" (page 3).
- If you have business logic subgraphs for the Web services you provide, you can use the **dml-to-wsdl** utility to generate a WSDL file from the business logic DML record formats (see "Generating WSDL from DML" next.)

GENERATING WSDL FROM DML

If you have existing business logic, you have the option of getting a WSDL file by generating it from the DML record formats and type definitions of end-point components in your business logic subgraph(s). You can use the **dml-to-wsdl** utility to generate your WSDL file.

Syntax

The syntax for the **dml-to-wsdl** utility is as follows:

dml-to-wsdl **-service** *svc_name* **-namespace** *target_ns* **-address** *url_addr*
{ { **-operation** *op_name* **-input** *input_dml* **-output** *output_dml* } **-soap-action** *action* ...}

ARGUMENT	DESCRIPTION
svc_name	Required. Name of the Web service.
target_ns	Required. Namespace where you want the Web service to be defined.
url_addr	Required. URL where the Web service is to be located.
op_name	Required. Name of the operation used to create SOAPAction.
input_dml	Required. Name of the file containing the DML for the operation's input.
output-dml	Required. Name of the file containing the DML for the operation's output.
action	Required. The SOAPAction string for the operation. This string is the same as the binding name in the WSDL file **SOAPAction**=*string*; for example, **SOAPAction**=**Balance**.

Example of generating WSDL from DML

Here's a Web services example of using **dml-to-wsdl** to generate a WSDL file from DML record formats of the end-point components in your business logic subgraphs.

Suppose you have a Web services provider graph called **BankWebService** in the namespace **MyBank** at the Web address **http://mybank.com/BankWebService**. This graph provides two Web services: finding the balance on an account ID and finding all transactions on an account ID for a given time period.

In your Web services provider graph you have business logic subgraphs to implement each of these operations: the **BalanceLogic** and **TransactionsLogic** subgraphs. Each of these subgraphs has a DML input record type and a DML output record type.

Your sandbox has a **dml** directory containing files that define these input and output record types: **GetBalance-in.dml**, **GetBalance-out.dml**, **GetTransactions-in.dml**, and **GetTransactions-out.dml**. The same sandbox also has a **wsdl** directory for the WSDL file (**CardTransactions.wsdl**) that you will generate from these DML files.

Using the **dml-to-wsdl** (page 66) utility to generate the WSDL file, you run the following command from the root of the sandbox:

```
dml-to-wsdl  -svc BankWebService -namespace ns:MyBank
-address http://mybank.com/BankWebService -operation BalanceLogic
-input GetBalance-in.dml -output GetBalance-out.dml
-soap-action Balance
-operation TransactionsLogic -input GetTransactions-in.dml
-output GetTransactions-out.dml -soap-action Transactions
```

This WSDL file contains XML descriptions of protocol bindings and message formats that define the external interface to your Web services graph.

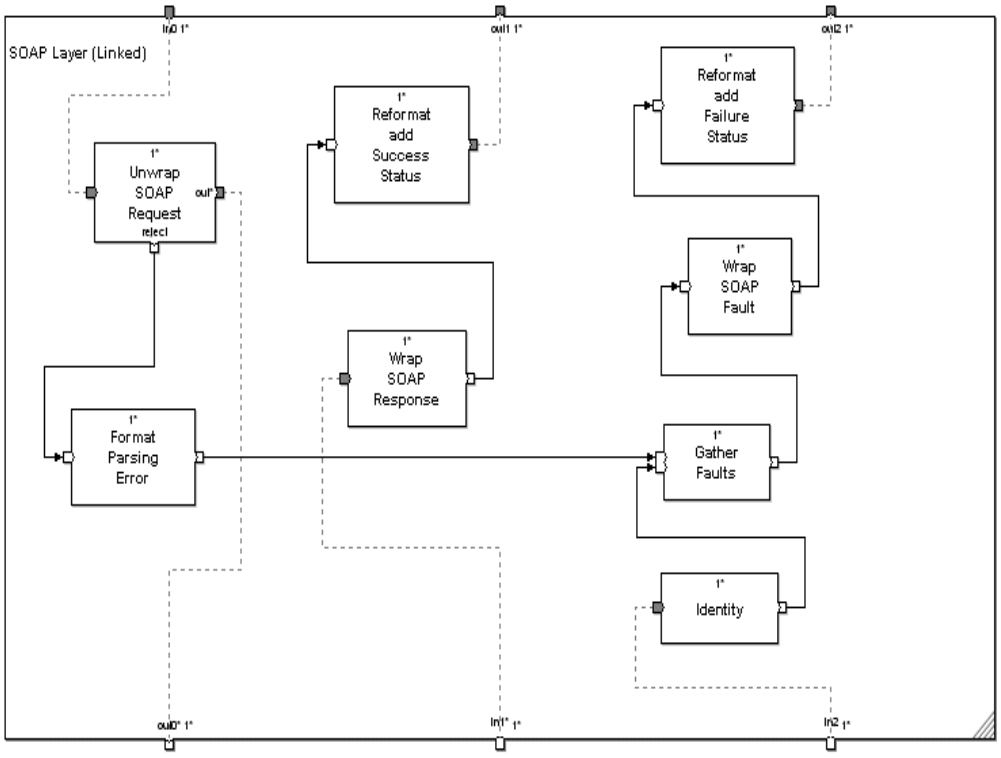
Defining the internal interface

Once you have a valid WSDL file (see "Defining the external interface" on page 22), the next step is to generate Schema DML as an internal interface around your Web services.

HOW SCHEMA DML WORKS

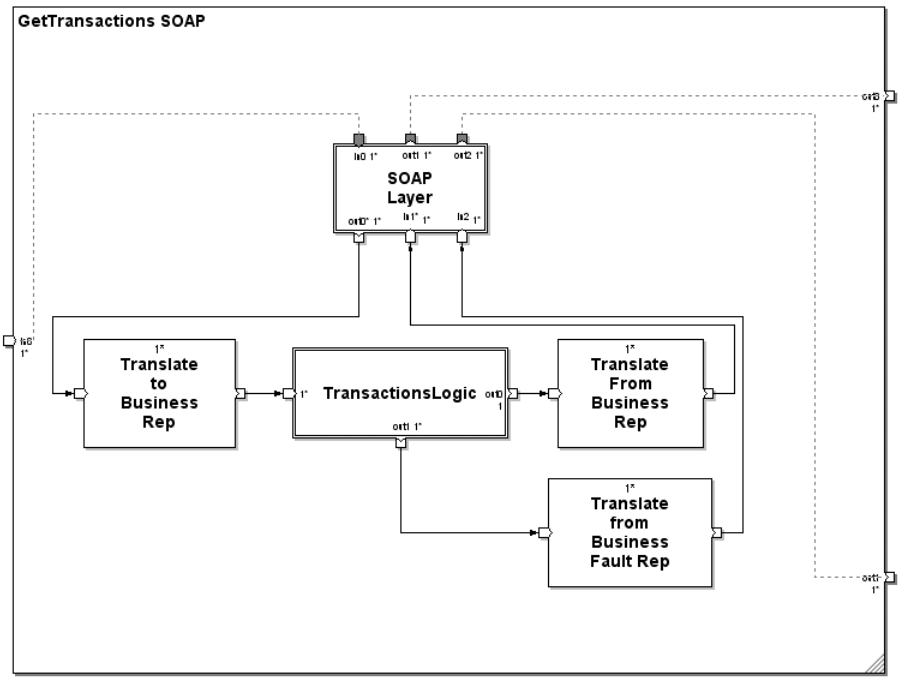
The key function of *Schema DML* is to unparse SOAP messages to and from the business logic subgraph. You use the **wsdl-to-dml** (page 70) utility to generate Schema DML from your WSDL file. The WSDL file contains XML descriptions of the input and output types for each operation. Schema DML represents these operations as DML input and output record formats and type definitions.

The figure below shows the sample SOAP layer subgraph in the operation subgraph, **GetTransactions SOAP**. **Unwrap SOAP Request** and **Wrap SOAP Response**, READ XML and WRITE XML components in this layer, use Schema DML to parse and unparse SOAP messages to and from the business logic subgraph.



Unwrap SOAP Request parses the XML structure of the SOAP request into a Schema DML-described record. This record then goes out of the SOAP layer subgraph to the operation subgraph.

The figure below shows the operation subgraph, **GetTransactions SOAP**. In this subgraph the output record from the SOAP layer goes to a REFORMAT component, **Translate to Business Rep**, and then to the business logic subgraph, **TransactionsLogic**, as shown below.



The REFORMAT components, **Translate to Business Rep** and **Translate from Business Rep**, translate between Schema DML and business DML. These REFORMAT components act as a "gasket" or internal interface around the business logic subgraph, **TransactionsLogic**.

The **in** port of **Translate to Business Rep** and the **out** port of **Translate from Business Rep** have Schema DML defined on them that transforms input records to the business logic subgraph and output records from it. This Schema DML can actually be defined on the bottom three ports (**out0**, **in1**, and **in2**) of the **SOAP Layer** subgraph and then propagated to these two REFORMAT components.

The business logic subgraph, **TransactionsLogic**, processes DML-described request records and sends success or fault records back to the SOAP layer subgraph.

In the SOAP layer subgraph, the WRITE XML component **Wrap SOAP Response** has Schema DML defined on the **in** port so that the component can unparse the DML-described record from the business logic subgraph back into the XML text of a SOAP response.

GENERATING SCHEMA DML

To generate Schema DML, you use your WSDL file as input to the **wsdl-to-dml** (page 70) utility. This utility converts the WSDL protocol bindings and message formats into DML type definitions and record formats.

If you have business logic

▶ **To generate Schema DML if you have business logic subgraph(s) and business DML:**

1. You can use the **dml-to-wsdl** (page 66) utility to generate a WSDL file from your existing business DML (see "Generating WSDL from DML" on page 22). Alternatively, you can create your own WSDL file, or use a WSDL file defined by your company or organization.
2. Use the **wsdl-to-dml** (page 70) utility to generate Schema DML.

   If you generated a WSDL file from existing DML, you can use this generated WSDL file as input to the **wsdl-to-dml** utility. You can also use your organization's WSDL file or one you wrote from scratch (see "Getting a WSDL file" on page 22).
3. Configure the REFORMAT components in your existing operation subgraph to transform between Schema DML and business DML.

   In this case, Schema DML and the business DML in the REFORMAT components are the same.

If you do not have business logic

If you have not created your business logic subgraphs, you have several options for using Schema DML depending on whether you use simple or complex business logic. In either case, use the **wsdl-to-dml** (page 70) utility to generate your Schema DML.

▶ **To generate Schema DML if your graph uses simple business logic:**

- Write the business logic subgraph(s) to use the generated Schema DML as the business DML.

  In this case you don't need REFORMAT components between the SOAP layer and the business logic subgraph.

▶ **To generate Schema DML if your graph uses complex business logic:**

1. Define the business DML.
2. Write your business logic subgraph(s) in terms of this DML.
3. Configure the REFORMAT components to transform between Schema DML and business DML.

Syntax of wsdl-to-dml

The syntax for the **wsdl-to-dml** (page 70) utility is as follows:

wsdl-to-dml *wsdl_name output_dir* [*op_name*] [**-use-envelope**] [**-no-envelope**]

ARGUMENT	DESCRIPTION
wsdl_name	Required. Name of the WSDL file.
output_dir	Required. Directory where the generated DML files will be placed.
op_name	Optional. Specifies a single operation so that **wsdl-to-dml** translates only that one operation.
-use-envelope	Optional. Specifies a SOAP envelope and specifies that **wsdl-to-dml** produces complete DML describing that entire SOAP envelope. This behavior is the default.
-no-envelope	Optional. Specifies that **wsdl-to-dml** does not produce complete DML describing a SOAP envelope.

Example of using wsdl-to-dml

Here's a Web services example of using **wsdl-to-dml** (page 70) to generate Schema DML. The examples folder (**$AB_HOME/examples/web-services/simple**) contains a sample WSDL file (**CardTransactions.wsdl**) in the **wsdl** folder. This example uses that sample WSDL file.

Suppose you have a sandbox that has a **schema_dml** folder for your Schema DML and a **wsdl** folder containing the WSDL file **CardTransactions.wsdl**.

To generate Schema DML from the **CardTransactions.wsdl** WSDL file, run the following command from the root of the sandbox:

```
wsdl-to-dml wsdl/CardTransaction.wsdl schema_dml
```

This command generates the following Schema DML files in the **schema_dml** folder:

- **GetBalance-in.dml**
- **GetBalance-out.dml**
- **GetTransactions-in.dml**
- **GetTransactions-out.dml**

For the content of these files, see "Examples of generated Schema DML and business DML" (next).

Examples of generated Schema DML and business DML

The Schema DML generated by **wsdl-to-dml** (page 70) contains basically the same type definitions as the business DML in the **GetBalance SOAP** and **GetTransactions SOAP** operation subgraphs. The Schema DML must be able to be converted into business DML, and vice versa.

Generally, however, the Schema DML differs from the business DML in two ways:

- Schema DML contains extra information that encodes the relevant XML schema.
- The individual Schema DML field types may differ slightly from those in the business DML; for example, there may be different data type variations and different names.

To accommodate these two differences, you need to put REFORMAT components before and after the business logic subgraph. A REFORMAT preceding the business logic subgraph transforms the *input* Schema DML into the *input* business DML. A REFORMAT after the business logic subgraph transforms the *output* business DML into *output* Schema DML.

For example, the **in** port of the **Translate to Business Rep** REFORMAT component uses the following input Schema DML (the key DML record formats and types are shown in **bold** and recognizable as normal DML formats; the non-bold lines are schema information that you can ignore here):

```
type string_t = string('\0');
type date_t = date('YYYY-MM-DD');
type GetTransactionsRequest_type_t = record
 string_t m_acct_id;
 date_t m_start_date;
 date_t m_end_date;
 string('\0') XML_namespace_mappings() = 'm,=urn:CardTransactions;';
 string('\0') XML_default_namespace() = 'urn:CardTransactions';
end;

type Body_type_t = record
 GetTransactionsRequest_type_t m_GetTransactionsRequest;
 string('\0') XML_namespace_mappings() = 'm,=urn:CardTransactions;';
 string('\0') XML_default_namespace() = 'http://schemas.xmlsoap.org/
soap/envelope/';
end;

metadata type = record
 Body_type_t soap_Body;
 string('\0') XML_namespace_mappings() = 'soap,=http://
schemas.xmlsoap.org/soap/envelope/;';
 string('\0') XML_default_namespace() = 'http://schemas.xmlsoap.org/
soap/envelope/';
 string('\0') XML_base_element() = 'http://schemas.xmlsoap.org/soap/
envelope/:Envelope';
end;
```

The **out** port of the **Translate to Business Rep** component uses the following output business DML:

```
record
 string("") acct_id;
 string("") start_date;
 string("") end_date;
end;
```

The transform function for the **Translate to Business Rep** component picks the relevant data (account ID, start date, and end date) from the Schema DML as follows:

```
out::reformat(in) =
begin
  out.acct_id :: in.soap_Body.m_GetTransactionsRequest.m_acct_id;
  out.start_date ::
in.soap_Body.m_GetTransactionsRequest.m_start_date;
  out.end_date :: in.soap_Body.m_GetTransactionsRequest.m_end_date;
end;
```

The **in** port of the **Translate from Business Rep** component uses the following input business DML:

```
record
  decimal(",") acct_id;
  record
    decimal(",") trans_id;
    date("YYYY-MM-DD") trans_date;
    real(8) amount;
  end [integer(4)] transactions;
end;
```

The **out** port of the **Translate to Business Rep** component uses the following output Schema DML:

```
type string_t = string('\0');
type date_t = date('YYYY-MM-DD');
type decimal_t = decimal('\0');
type dated_trans_type_t = record
 date_t m_trans_date;
 string_t m_trans_id;
 decimal_t m_amount;
 string('\0') XML_namespace_mappings() = 'm,=urn:CardTransactions;';
 string('\0') XML_default_namespace() = 'urn:CardTransactions';
end;

type GetTransactionsResponse_type_t = record
 string_t m_acct_id;
 dated_trans_type_t[big endian integer(4)] m_dated_trans;
 string('\0') XML_namespace_mappings() = 'm,=urn:CardTransactions;';
 string('\0') XML_default_namespace() = 'urn:CardTransactions';
end;

type Body_type_t = record
 GetTransactionsResponse_type_t m_GetTransactionsResponse;
 string('\0') XML_namespace_mappings() = 'm,=urn:CardTransactions;';
 string('\0') XML_default_namespace() = 'http://schemas.xmlsoap.org/
soap/envelope/';
end;

metadata type = record
 Body_type_t soap_Body;
 string('\0') XML_namespace_mappings() = 'soap,=http://
schemas.xmlsoap.org/soap/envelope/;';
 string('\0') XML_default_namespace() = 'http://schemas.xmlsoap.org/
soap/envelope/';
 string('\0') XML_base_element() = 'http://schemas.xmlsoap.org/soap/
envelope/:Envelope';
end;
```

The transform function for the **Translate from Business Rep** component uses the output business DML types to iterate through the vector to pick out the relevant data (account ID, start date, and end date) from each element as follows:

```
out::reformat(in) =
begin
  let integer(4) i;

  out.soap_Body.m_GetTransactionsResponse.m_acct_id :: in.acct_id;
  out.soap_Body.m_GetTransactionsResponse.m_dated_trans :: for (i, i
< length_of(in.transactions)) :
            [record m_trans_date in.transactions[i].trans_date,
                    m_trans_id in.transactions[i].trans_id,
                    m_amount in.transactions[i].amount ];
end;
```

- *NOTE:* The transform functions for both these REFORMAT components transform vectors and are more complex than transform functions used for simpler record formats.

Building the provider graph step by step

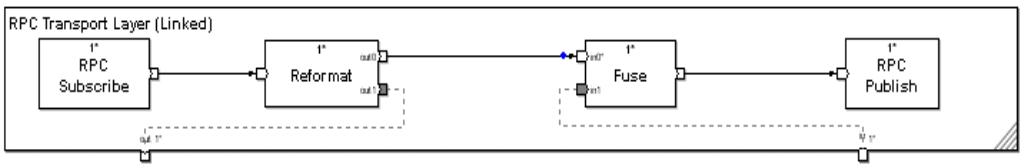
To build your Web services provider graph, you can drag in reusable subgraphs from **$AB_HOME/connectors/SOAP** and configure them for your graph. The following sections use these samples to describe, step by step, how to build and connect each graph layer:

DEFINING THE RPC TRANSPORT LAYER

The first layer you need to define in your Web services provider graph is the RPC transport layer. This layer transports Remote Procedure Calls (RPCs) in and out of your Web services graph. The main purpose of this layer is to preserve the RPC header by detaching it before the request goes to the other layers and reattaching it before the response leaves the graph.

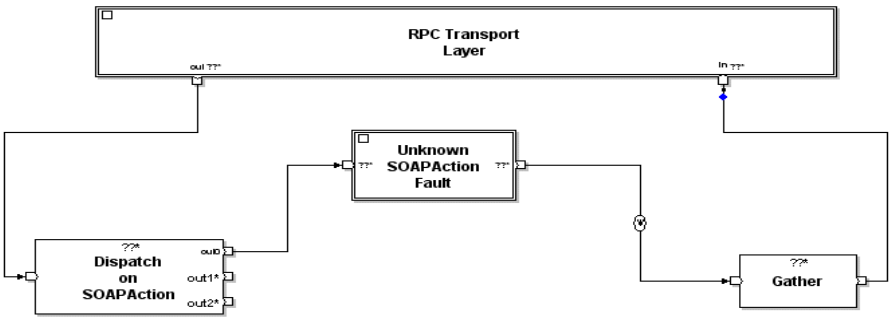
Connecting the RPC transport layer

The figure below shows the reusable sample RPC transport layer subgraph. This layer has RPC connector components at either end, a REFORMAT to detach the RPC header from the request, and a FUSE to reattach the RPC header to the response. For detailed information about the RPC connector components, see "RPC SUBSCRIBE" and "RPC PUBLISH" in Ab Initio Help.



▶ **To connect the RPC transport layer:**

1. Drag the **RPC Transport Layer** sample subgraph from **$AB_HOME/connectors/SOAP** into the workspace.
2. Drag the **Unknown SOAPAction Fault** sample subgraph from **$AB_HOME/connectors/SOAP** into your graph.
3. Connect these two subgraphs with the REFORMAT component **Dispatch on SOAPAction** and with a GATHER component, as shown in the following figure.



Configuring the Dispatch on SOAPAction component

The REFORMAT component **Dispatch on SOAPAction**, connected to **RPC Transport Layer**, drops the SOAP header and outputs only the data payload of a request.

The GATHER component does nothing at this point. It works later after you configure it when you add the operation subgraphs (see "Building an operation subgraph" on page 34).

At this point **Dispatch on SOAPAction** has one **out** port. Configure this port to use the following DML types and record formats to output requests:

```
input type:

    include "~$AB_HOME/connectors/SOAP/SOAPRequest.dml";

    metadata type = record
      SOAPRequest soaphdrs;
      string(big endian integer(4)) body;
    end;

output type:

    string(big endian integer(4));

output_index function:

    out::output_index(in) =
    begin
     out::0;
    end;
```

You add more **out** ports and edit the **output_index** function as you add and define each operation subgraph (see "Building an operation subgraph" on page 34).

BUILDING AN OPERATION SUBGRAPH

Make each operation subgraph contain a SOAP layer subgraph and a business logic subgraph for each operation.

▶ **To build an operation subgraph:**

1. Add an **out** port to the REFORMAT component **Dispatch on SOAPAction**.

   Make sure this port is **out port 1** (the existing port is **out port 0**).

2. Edit the **output_index** function on this **out** port to work with your operation subgraph.

   For example, if you are adding the **BalanceLogic** subgraph, edit the **output_index** function as follows:

   ```
   output_index function:

       out::output_index(in) =
       begin
        out::if (in.soaphdrs.SOAPAction == "Balance") 1 else 0;
       end;
   ```

   - *NOTE:* The SOAPAction string for this operation ("Balance") is defined in the WSDL file.

3. To add an operation subgraph, choose **Insert > Empty Subgraph** from the menu bar.
4. Name this subgraph **GetBalance SOAP**, link it to the main graph, and open it.
5. Write your business logic subgraph, or drag the **BalanceLogic** sample business logic subgraph into the **GetBalance SOAP** operation subgraph.
6. Drag the **SOAP Layer** sample subgraph into the **GetBalance SOAP** subgraph.

DEFINING THE SOAP LAYER

Now you can define the SOAP layer (the sample **SOAP Layer** subgraph) and connect it to work with the sample business logic subgraph, **BalanceLogic**.

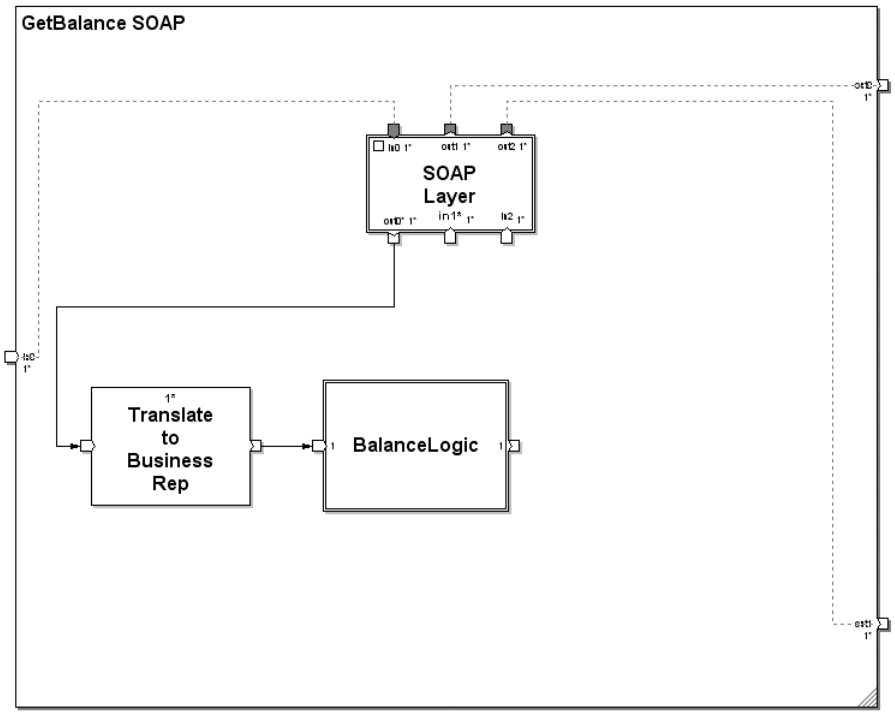
▶ **To define the SOAP layer:**

1. Connect the top ports of the **SOAP Layer** subgraph to the outside of the **GetBalance SOAP** operation subgraph.

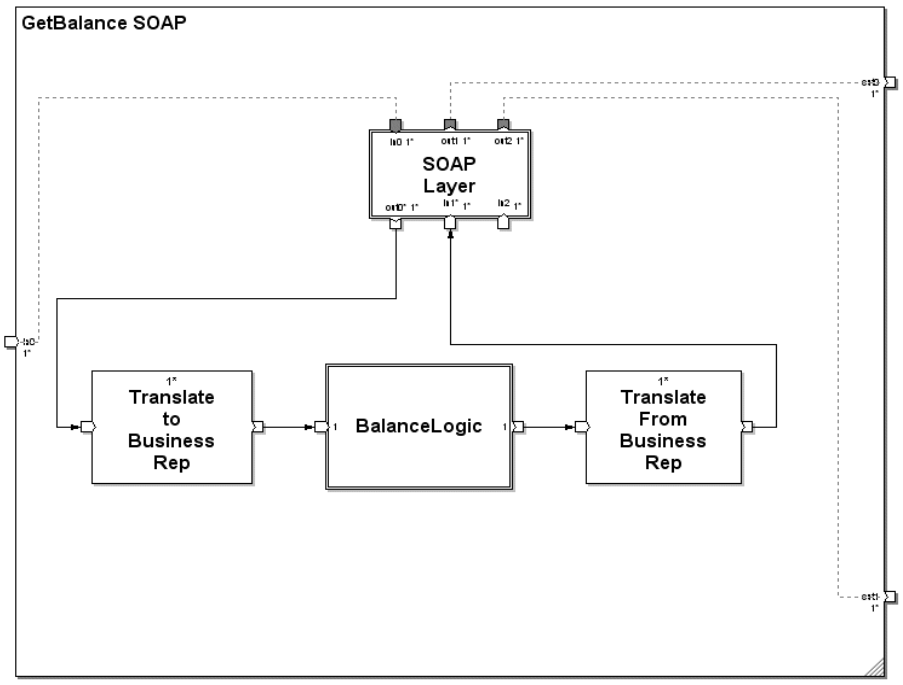
   These three top ports are the **in** port (**input SOAP**), the **out1** port (**output SOAP**), and the **out2** port (**error SOAP**). These three ports should already have had the correct types defined on them when you dragged in the **SOAP Layer** subgraph.

2. Configure the bottom three ports: **out0**, **in1**, and **in2**.
   - To parse the input record to the business logic subgraph, port **out0** should reference **GetBalance-in.dml** from the **schema_dml** folder in **$AB_HOME/examples/web-services/simple**.
   - To unparse the output record from the business logic subgraph, port **in1** should reference **GetBalance-out.dml** from the **schema_dml** folder, and port **in2** should reference the DML file for fault handling.
3. Connect the flows from these ports to the REFORMAT components you add in the next steps, or connect them directly to the business logic subgraph (**BalanceLogic**) if you do not need REFORMAT components.

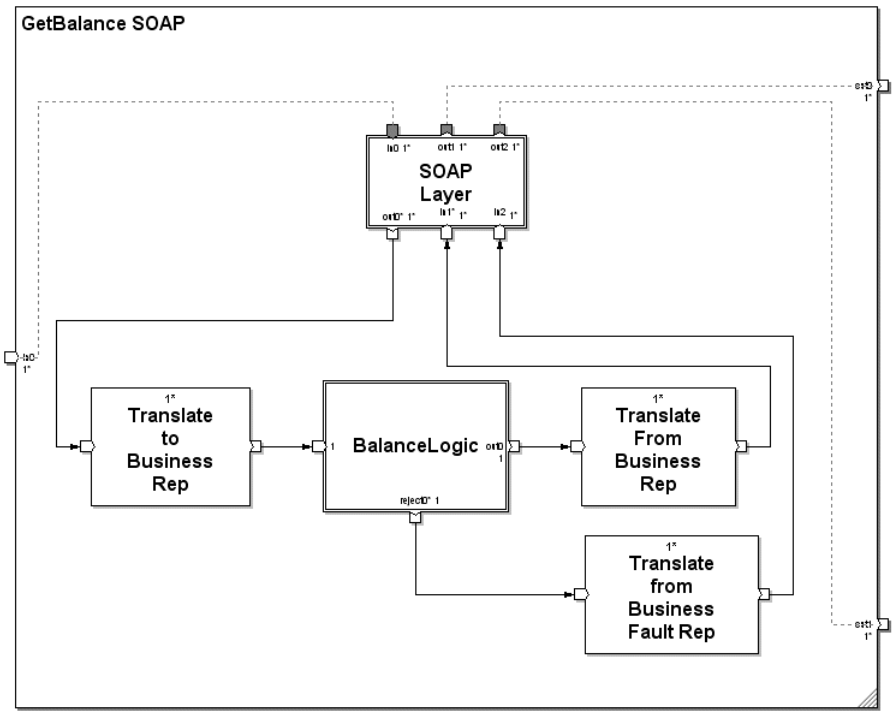
   If you created your business logic subgraph(s) to use the generated Schema DML as the business DML, you don't need REFORMAT components between the SOAP layer and the business logic subgraph.
4. If you need a REFORMAT component (such as **Translate to Business Rep**) to translate the Schema DML into business DML, add it between the **out0** port of the **SOAP Layer** subgraph and the **in** port to **BalanceLogic**.



5. If you need REFORMAT components, add another one (such as **Translate from Business Rep**) to translate the output record business DML from **BalanceLogic** back to Schema DML for output to the **SOAP Layer** subgraph.



6. Add another REFORMAT (such as **Translate from Business Fault Rep**) to the operation subgraph **reject** port to handle the fault responses.



When you drag in these "gasketing" REFORMAT components, they should have the correct record formats and types propagated on the **in** port of **Translate to Business Rep** and the **out** port of **Translate from Business Rep**, respectively.

The business DML types are defined on the **in** port of the business logic subgraph (**BalanceLogic**).

7. Close the **GetBalance SOAP** operation subgraph.

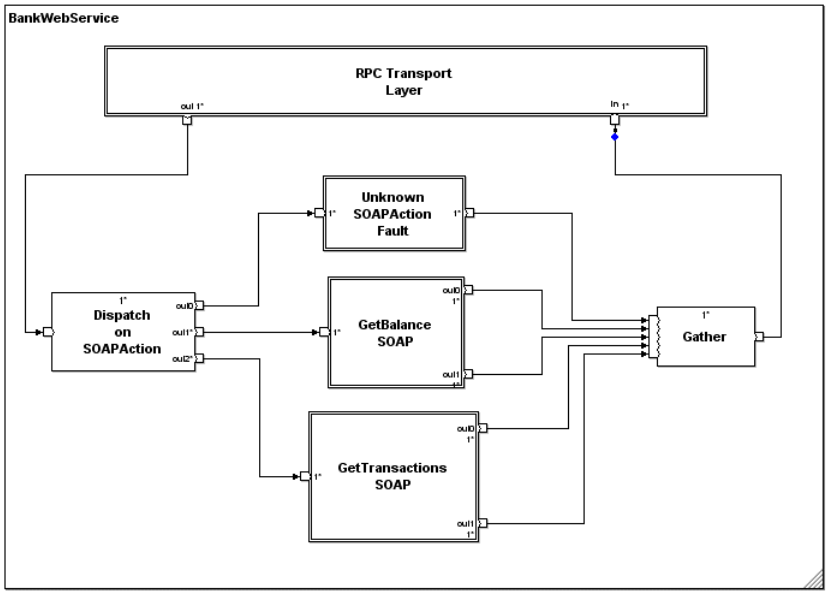
CONNECTING THE LAYERS TOGETHER

Now you can connect the **GetBalance SOAP** operation subgraph to the other layers of the Web services provider graph:

1. Create a new **out** port on the **Dispatch on SOAPAction** REFORMAT component.
2. Connect this **out** port to the **in** port of the **GetBalance SOAP** operation subgraph.

3. Connect both **out** ports of **GetBalance SOAP** to the GATHER component.

   The figure below shows **GetBalance SOAP** connected. It also includes another connected operation subgraph, **GetTransactions SOAP**.



TESTING THE PROVIDER GRAPH

Once you have all the layers of your Web services provider graph connected, you can test it. Make sure you have the plug-in installed on a test application server and have added your Web services provider to the plug-in list, as described with other required administration tasks in Chapter 3 "Administering Web services" (page 39). Then run the graph and test it as described in Chapter 4 "Testing and troubleshooting" (page 51).

3

Administering Web services

To administer Ab Initio Web services features, you must get the Ab Initio Web services plug-in and install it on either a Microsoft IIS or a Java EE application server. The plug-in is an application that handles communication between Web services client applications and Web services provider graphs.

This chapter describes:

Getting and installing a Web services plug-in

Plug-ins are available for Microsoft IIS and Java EE application servers. For IIS application servers, we send the plug-in as a **.MSI** (Microsoft Installer) file. For Java EE application servers, we send the plug-in as a **.WAR** (Web archive) file. To obtain the plug-in, contact Ab Initio Support.

BEFORE INSTALLING THE PLUG-IN

Before installing the Web services plug-in, note the following:

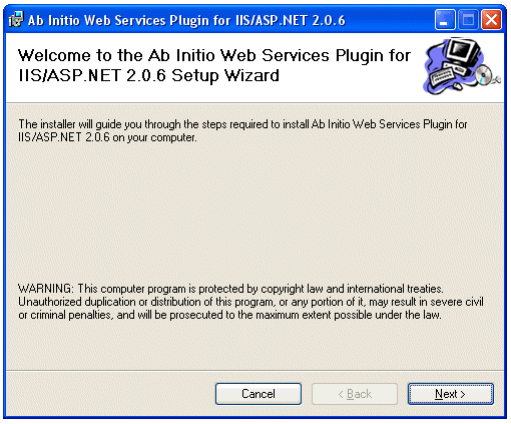
- The application server where you install the plug-in can be on a different host from the one the Co>Operating System is on.
- You cannot install more than one plug-in per application server.
- If there is a firewall between the application server and the Web services provider graph, the port where the graph's RPC SUBSCRIBE component listens must be opened in the firewall.
- To install the Web services plug-in in a clustered application server environment, contact Ab Initio Support.

INSTALLING THE PLUG-IN

▶ **To install the plug-in on an IIS application server:**

1. Load your plug-in onto the application server.
2. Double-click the plug-in icon when it appears.

   The plug-in **Setup Wizard** appears:



3. Click **Next** to display the **Select Installation Address** page.

4. In the **Virtual directory** box, enter the virtual directory where you want to install the plug-in.

   The default virtual directory for the plug-in on an IIS application server is **AIWebServices**; we strongly recommend using this directory.

5. In the **Port** box, enter the number of the port where the application server is listening.

   For example, enter **8080**.

6. Click **Next** to display the **Confirm Installation** page.

7. Click **Next** again to confirm and start the installation.

8. When the **Installation Complete** page appears, click **Close** to exit the **Setup Wizard**.

■ *NOTE:* When installing a new version of the plug-in on an IIS server after uninstalling an earlier version, restart IIS using either the Control Panel or the Windows **NET STOP** and **NET START** commands.

▶ **To install the plug-in on a Java EE application server:**

- Install the **.WAR** file as specified in your Java EE application server documentation.

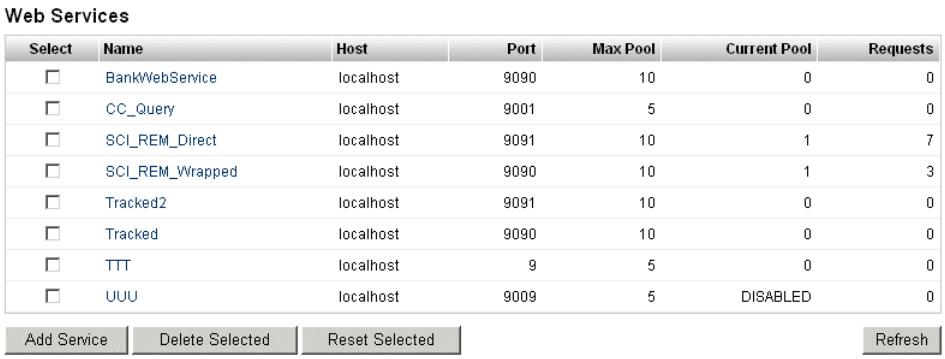
  When you specify the context root, we strongly recommend using **AIWebServices**.

Browsing to the administration Web page

After installing the Web services plug-in, you can browse to the **Web Services** administration page and administer your Web services. For example, you can perform the following tasks:

- Adding Web services (page 44)
- Editing Web services (page 46)
- Deleting Web services (page 47)
- Discovering the URL of a Web service (page 48)
- Resetting Web services (page 49)
- Monitoring Web services (page 50)

The figure below shows a **Web Services** administration page.



THE WEB SERVICES ADMINISTRATION PAGE URL

The URL you use for browsing to your **Web Services** administration page depends on the plug-in installation information. For more information see "The URL for an IIS Web Services administration page" (next) or "The URL for a Java EE Web Services administration page" (page 43), depending on the application server where your plug-in is installed.

The URL for an IIS Web Services administration page

On an IIS application server, the URL for the **Web Services** administration page is:

http://*hostname*:*portnumber*/*VirtualDirectoryName*/**admin/Admin.aspx**

For *hostname*, *port number*, and *VirtualDirectoryName*, use the plug-in installation information (see "Installing the plug-in" on page 40).

Here's an example of the URL for an IIS application server **Web Services** administration page:

```
http://myhost:8080/AIWebServices/admin/Admin.aspx
```

The URL for a Java EE Web Services administration page

On a Java EE application server, the URL for the **Web Services** administration page is:

http://*hostname*:*portnumber*/*ContextRoot*/**admin**/**Admin**

For *hostname*, *port number*, and *ContextRoot*, use the plug-in installation information (see "Installing the plug-in" on page 40).

Here's an example of the URL for a Java EE application server **Web Services** administration page:

```
http://myhost:8080/AIWebServices/admin/Admin
```

Adding Web services

The Web services plug-in can communicate with several Web services provider graphs. To open communication between a plug-in and a Web services provider graph, you add the Web service provider graph to the plug-in's Web services list.

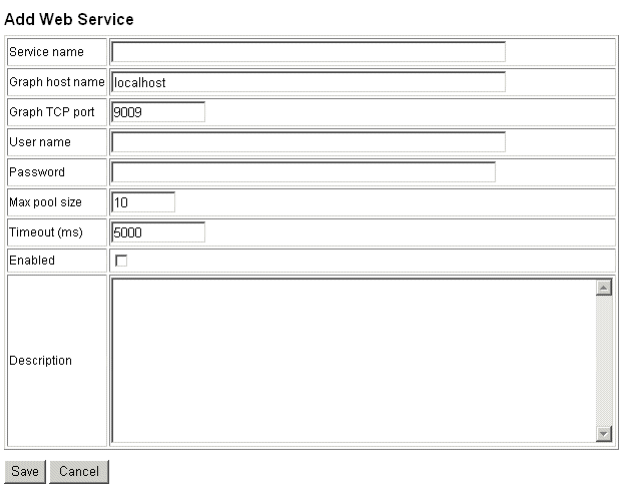
▶ **To add a Web service to a plug-in:**

1. Browse to the **Web Services** administration page.

   See "Browsing to the administration Web page" (page 42).

2. Click the button for adding a Web service.

   For an IIS plug-in, this button is **Add Service.** For a J2EE plug-in, this button is **Add RPC Service.** The figure below shows the **Add Web Service** page for the IIS plug-in:



3. In the **Service name** box, specify a name for the Web service provider graph.

   You can use any combination of alphanumeric characters, hyphens, and underscores in the name.

4. In the **Graph host name** box, enter the name of the host where the Web services provider graph's RPC connector components are running.

   RPC SUBSCRIBE is the entry point into a Web service provider graph in the RPC transport layer. RPC PUBLISH is the exit point of a Web service provider graph. These two components always run in corresponding pairs in the same Web services provider graph. Each corresponding pair runs on the same host, and you specify that host here in the **Graph host name** box.

5. In the **Graph TCP port** box, enter the number of the port where the Web services provider graph's RPC connector components are listening.

   Each corresponding pair of RPC SUBSCRIBE and RPC PUBLISH components uses the same port, and this port is used exclusively by this pair.

6. In the **User name** box, either enter the username specified in the Web services provider graph or accept the default username **AllClients**.

   The username is case-sensitive.

7. Leave the **Password** box blank, or enter the user password (if any) specified for the Web services provider graph.

   The password is case-sensitive.

8. In the **Max pool size** box, enter the maximum number of concurrent connections you want to allow the plug-in to make for client requests to the Web services provider graph.

   - *NOTE:* The plug-in returns a SOAP fault for any requests after the maximum number of connections has been reached.

   Multiple connections increase throughput and help ensure that requests don't back up and exceed the timeout period. However, each connection has a memory cost for both the plug-in and the client's RPC SUBSCRIBE component.

   The application server administrator can set a limit on the maximum number of concurrent connections the server supports. A Web services administrator should not set the **Max pool size** of connections greater than the maximum number of connections set for the application server; the server will not create more connections than its specified maximum.

9. In the **Timeout (ms)** box, enter the maximum amount of time in milliseconds (ms) that you want the plug-in to wait for a response from the Web services provider graph.

   If the graph does not respond within this time limit, the plug-in returns a SOAP fault to the client. The time limit you specify should be greater than the time you expect the graph will take to process the request. Setting a reasonable time limit helps you detect a failed network connection or a hung provider graph.

10. Select **Enabled** to enable processing of requests for this Web service.

    Deselect **Enabled** to disable processing of requests for this Web service. All subsequent requests fail with a SOAP fault.

11. In the **Description** box, you can enter text, such as comments and reminders, about this Web service provider graph.

Editing Web services

You can edit the configuration information for any Web service provider graphs in a plug-in's list of Web services.

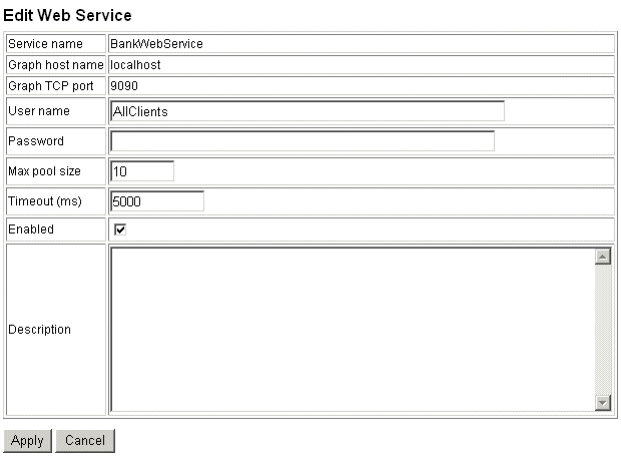
▶ **To edit a Web service:**

1. Browse to the **Web Services** administration page.

   See "Browsing to the administration Web page" (page 42).

2. In the **Name** column, click the name of the Web service you want to edit.

   The **Edit Web Service** page appears.



3. Edit the configuration information as appropriate.
4. Click **Apply**.

Deleting Web services

You can delete a Web service provider graph from a plug-in's list of Web services.

▶ **To delete a Web service:**

1. Browse to the **Web Services** administration page.

   See "Browsing to the administration Web page" (page 42).

2. In the **Select** column, select the checkbox next to the name of the Web service you want to delete.

   To delete multiple Web services, select multiple checkboxes.

3. Click **Delete Selected**.

Discovering the URL of a Web service

Client applications send requests to a particular application server, plug-in, and Web services provider graph. The client uses the Web services provider graph's URL to route HTTP service requests. In client graphs, the CALL WEB SERVICE or CONTINUOUS CALL WEB SERVICE components (see Ab Initio Help) use this URL to route their requests.

The URL indicates where clients send requests, based on hierarchical naming, as follows:

- The host name
- The application server
- The application running on the application server (in this case a Web services plug-in)
- The name of the Web services provider graph where the plug-in sends the request

THE WEB SERVICE URL SYNTAX

For a plug-in on an IIS application server, the syntax for a Web service's URL is:

http://*hostname*:*portnumber*/*VirtualDirectoryName*/**services**/*ServiceName*.**aspx**

For a plug-in on a Java EE application server, the syntax for a Web service's URL is:

http://*hostname*:*portnumber*/*ContextRoot*/**services**/*ServiceName*

HOW DO CLIENTS KNOW WHERE TO SEND REQUESTS?

The URL of a Web service provider graph is derived from information used when the plug-in was installed. You can contact your administrator for this URL: the administrator knows the *hostname*, *portnumber*, and *VirtualDirectoryName* or *ContextRoot* used for installing the plug-in and should know the URL of each Web services provider graph added to a particular plug-in's list of Web services.

- *TIP:* The WSDL file for your graph usually contains the URL in this line:
  soap:address location=*URL*

WEB SERVICE URL EXAMPLE

Suppose your plug-in was installed on an IIS application server in the virtual directory **AIWebServicesForMe**. The host name is **MyHost**, and the port number is **8080**. The Web services provider graph is named **MyService**.

Then the URL that clients use to send requests to this graph is:

```
http://MyHost:8080/AIWebServicesForMe/services/MyService.aspx
```

Resetting Web services

You should reset a Web service whenever a Web services client or provider graph is restarted. Resetting is also a good troubleshooting strategy. When calls to a graph fail, restart the graph and reset the Web service.

Resetting a Web service deletes the service's RPC connections and the statistics associated with the pool of these connections. If a graph is restarted and you do not reset the Web service, the plug-in tries to reopen stale RPC connections and gradually replaces the pool of stale connections with new connections.

- *NOTE:* The Web services in a plug-in's list are independent of one another: resetting one does not affect the others.

▶ **To reset Web services:**

1. Browse to the **Web Services** administration page.

   See "Browsing to the administration Web page" (page 42).

2. In the **Select** column, select the checkbox next to the name of the service you want to reset.

   To reset multiple Web services, select multiple checkboxes.

3. Click **Reset Selected**.

Monitoring Web services

Once clients, the plug-in, and Web services provider graphs are all running together, you can use the **Web Services** administration page to monitor their activity and correct failures.

For example, if a request to a Web services provider graph fails, the entry in the **Current Pool** column for that service changes to **Failed**. Contact the graph's administrator to shut down and restart the graph. When the graph is up again, reset the Web service (see "Resetting Web services" on page 49).

REFRESHING WEB SERVICES

While monitoring Web services, you can refresh the display to view updated statistics, such as the numbers for **Current Pool** and **Requests**, and to show configuration changes that may have been made through a different browser.

▶ **To refresh the display of statistics for Web services:**

1. Browse to the **Web Services** administration page.

   See "Browsing to the administration Web page" (page 42).

2. Click **Refresh**.

4

Testing and troubleshooting

This chapter describes how to test a Web services provider graph and troubleshoot common problems. This chapter includes:

Before testing your Web services graph

Before testing your Web services provider graph, review the pretest checklist and make sure your Web services provider graph has all the correct characteristics.

This section includes these topics:

PRETEST CHECKLIST

Make sure of the following:

FOR THIS ITEM	MAKE SURE THAT	SEE
Plug-in	The Web services plug-in is installed on an application server and the application server is running. For debugging, we recommend installing the plug-in on a test application server rather than your production application server.	"Getting and installing a Web services plug-in" (page 40) for information about how to install the appropriate plug-in on an IIS or Java EE application server
Provider graph	Your Web services provider graph has been added to the correct plug-in's Web services list. The Web services provider graph you want to test is running. You have to start the Web services provider graph manually before sending a query from a test client. The application server isn't necessarily running on the same host as the provider graph and has no way of starting the graph itself. Normally, in production, the Web services provider graph, like any continuous graph, would be started by a scheduler. In a test situation, you have to start the graph manually before sending a request from a test client.	"Adding Web services" (page 44)
Statistics	You can browse to the **Web Services** administration page and see statistics for your Web service.	"Browsing to the administration Web page" (page 42)
URL	Client graphs and applications have the correct URL for your Web services provider graph.	"Discovering the URL of a Web service" (page 48) and "Debugging the Web service's URL" (page 62)

CHECKING GRAPH CHARACTERISTICS

Make sure your Web services provider graph has the following settings and characteristics:

FOR THIS ITEM	MAKE SURE THAT
Parameter	You define an **AB_GRAPH_RUN_MODE** parameter at the graph level and set it to **continuous-nonrecoverable**. Every Web services provider graph must be a nonrecoverable continuous graph. By defining a graph-level **AB_GRAPH_RUN_MODE** parameter, you are setting the AB_GRAPH_RUN_MODE configuration variable to **continuous-nonrecoverable**. This parameter is not an input parameter. You export it to the environment. Set it as **type=string** and **"export to environment"** so it propagates down to all the subgraphs and components in the provider graph.
Computepoint	At the top level of the provider graph, there is one computepoint per input record.
Response or fault	At the top level of the provider graph, each request produces exactly one response or one fault message (never both). Layers of a Web services provider graph (such as the business logic layer) can decompose a single request into multiple subrecords, but must always recompose these records into a single response or fault message.
No reordering	At the top level of the provider graph, requests and responses are not reordered. Components such as NORMALIZE in the subgraph layers can reorder their subrecords during processing, as long as the original order is retained at the top level.

Using the test clients

Ab Initio provides test clients that you can use to send test queries to your Web services provider graph.

Alternatively, you can use a third-party XML tool to send requests, trace queries, and view responses. Some third-party XML tools are XMLSpy Enterprise, WebServiceStudio, and SOAP Scope. If you use a third-party XML tool, see its documentation for how to create a new SOAP request using the appropriate WSDL file, operations, SOAP envelope, and so on.

This section includes:

LOCATION AND DESCRIPTION OF THE TEST CLIENTS

The **$AB_HOME/examples/web-services/simple** folder contains several test clients you can use for sending test queries to your Web services provider graphs.

These test clients are the following:

SUBFOLDER	TEST CLIENT	CONTENTS
test	**soap_rpc_client.c**	The source file of a nongraphical command-line test client. You can use this test client from the command line on any platform. This client uses the sample SOAP requests in the **xml** folder. It sends one of these requests (such as **GetBalanceRequest1.xml**) directly to the Web services provider graph without using a plug-in.
dotnetclient	**BankWebServiceClient.exe** and related sources	This Web services test client runs only with .NET on Windows.
aspdotnetclient	**AIBankWebServiceClientSetup.msi** and related sources	This Web services test client is an IIS application that runs only with ASP.NET on Windows.

USING THE COMMAND-LINE TEST CLIENT

This client does not use the plug-in or run on an application server. You can use this test client to send sample SOAP requests found in the **xml** folder (such as **GetBalanceRequest1.xml**) to the sample Web services provider graphs. You can also create your own requests and send them to the Web services provider graph you created.

▶ **To build and install the command-line test client:**

1. In **$AB_HOME/examples/web-services/simple/test**, use the Makefile (modifying it if necessary) to build this test client for your platform.

   The source file (**soap_rpc_client.c**) is in this directory.

2. Copy the built file (**soap_rpc_client.exe**) to a machine running the Co>Operating System.

▶ **To run the command-line test client:**

1. Navigate to the directory to which you copied **soap_rpc_client.exe**.
2. From the command line, run the following command:

   soap_rpc_client *Host Port SoapAction Principal*

ARGUMENT	DESCRIPTION
Host	Required. The host on which the Web services provider graph's RPC SUBSCRIBE and RPC PUBLISH components are running.
Port	Required. The port on which the Web services provider graph's RPC SUBSCRIBE and RPC PUBLISH components are listening.
SOAPAction	Required. The SOAPAction string for the operation you want the provider graph to perform.
Principal	Required. The username of the person calling the Web service. The username is normally filled in by the application server. It is sent to the provider graph in the **Principal** field of the **SOAPRequest** header. The provider graph could use this argument to check authorization of the incoming request. If a user connects without entering a username, the server substitutes the empty string (" ") for this argument.

If your query is successful, the test client echoes the response at the command line.

If your query fails, there is no echoed response. To debug a failed query, use the methods described in "Tracing a test query" (page 60), "Debugging a test query" (page 61), and "Troubleshooting tips" (page 63).

Example

Here's an example of a command to run the command-line test client:

```
soap_rpc_client myhost 80 Balance UserName < ../xml/GetBalanceRequest1.xml
```

The request is passed in on standard output and redirects from the specified files. So you need to put a **<** (less-than symbol) before the name of the file containing the request. The example uses the sample request **GetBalanceRequest1.xml**. You can also use a file containing your own request.

USING THE .NET TEST CLIENT

This test client has a graphical interface and runs only with .NET on Windows running on an IIS application server. It can call a Web services provider graph running on the same IIS application server or on another application server, either an IIS or a Java EE application server.

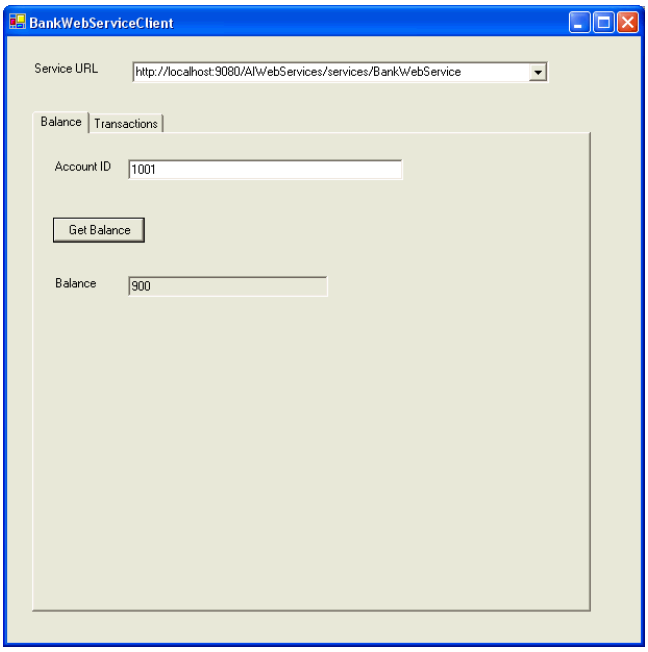
▶ **To use the .NET test client:**

1. Navigate to **$AB_HOME/examples/web-services/simple/dotnetclient**.

   If you are not on a Windows machine, copy **BankWebServiceClient.exe** to a Windows machine and then continue with this procedure.

2. Double-click the icon for **BankWebServiceClient.exe**.

   The **BankWebServiceClient** dialog appears with a tab for each operation: **Balance** and **Transactions**.



3. Click the tab for the operation you are testing, either **Balance** or **Transactions.**
4. In the **Service URL** box, enter the URL of the Web services provider graph; accept the default data in the other boxes.

   For more information about the URL of the Web services provider graph, see "Discovering the URL of a Web service" (page 48).
5. Depending on the operation you are testing, click either **Get Balance** or **Get Transactions**.

If your query is successful, the response data appears. The balance data appears in the **Balance** box, and the transactions data appears under **Trans ID**, **Trans Date**, and **Amount**.

If your query fails, an error box appears. Debug the test query using the methods described in "Tracing a test query" (page 60), "Debugging a test query" (page 61), and "Troubleshooting tips" (page 63).

USING THE .ASPNET TEST CLIENT

You use this test client from a browser. It has a graphical interface and runs only on an IIS application server on the ASP.NET platform running on Windows.

▶ **To use the .ASPNET test client:**

1. Navigate to **$AB_HOME/examples/web-services/simple/aspdotnetclient**.
2. Copy the installer file, **AIBankWebServiceClientSetup.msi**, to the Windows host.
3. To install the test client on your IIS, double-click **AIBankWebServiceClientSetup.msi**.
4. Fill in all fields in the install dialog: **Virtual Directory**, **Host**, **Port**, and so on.

   When you specify virtual directory and port, we recommend using the defaults, **AIBankWebServiceClient** and **80**.
5. Open a browser and browse to the **AIBankWebServiceClient** page.

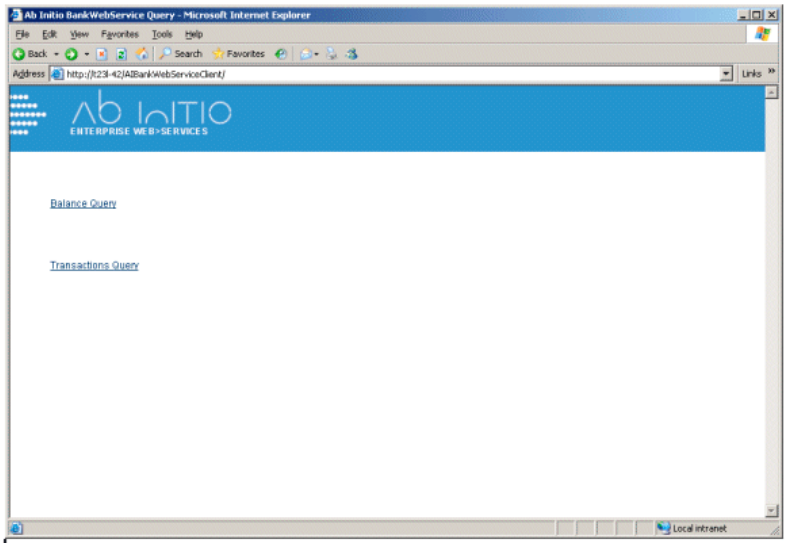
   The URL for the **AIBankWebServiceClient** page depends on the information you entered when you installed the test client, as follows:

   http://*host/port/VirtualDirectory*

   For example:

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

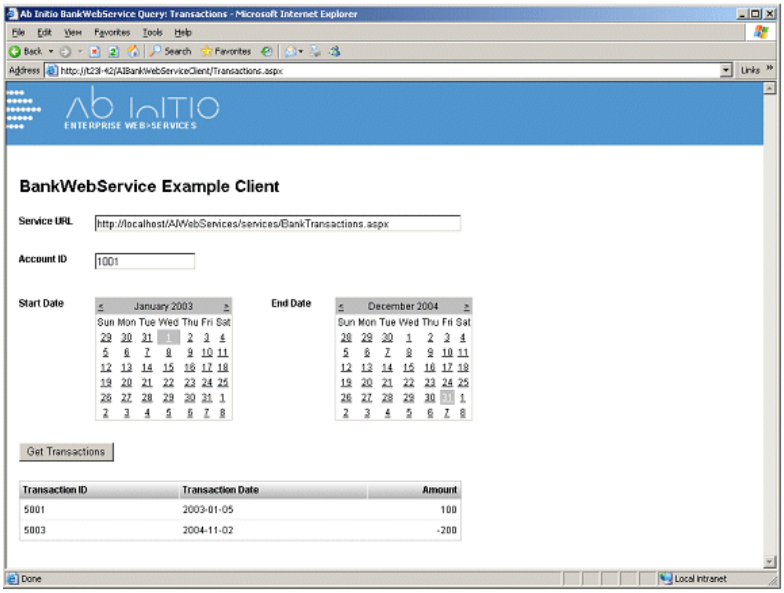
6. Click the link for the operation you want to test.



7. On the Web page for the test client, enter the URL of the Web services provider graph In the **Service URL** box and accept the default data in the other boxes.

   For more information about the URL of the Web services provider graph, see "Discovering the URL of a Web service" (page 48).

The figure below shows the page for testing the transactions operation.



8. Depending on the operation you are testing, click either **Get Balance** or **Get Transactions**.

If your query was successful, the response data appears. The balance data appears in the **Balance** box, and the transactions data appears under **Trans ID**, **Trans Date**, and **Amount**.

If your query failed, an error dialog appears. Debug the test query using the methods described in "Tracing a test query" (page 60), "Debugging a test query" (page 61), and "Troubleshooting tips" (page 63).

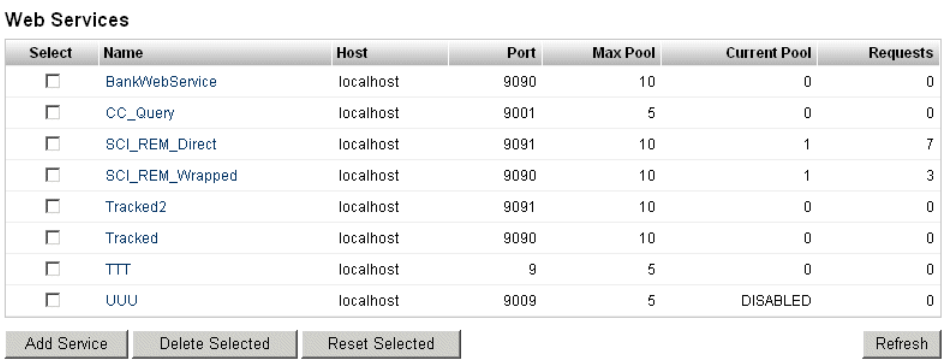
Tracing a test query

You can trace the path of a request and response and find out where it went wrong. The characteristics of Web services provider graphs make tracing a query one record at a time a particularly useful method of debugging.

This section includes the following topics:

BEFORE TRACING A QUERY

Before sending and tracing a test query, browse to the **Web Services** administration page and note the number of requests to your Web service as displayed in the **Requests** column.



THE QUERY ROUTE

The correct route of a single request to and response from a Web services provider graph is as follows:

1. The request goes from the Web services client graph or application to the plug-in.
2. The request goes from the plug-in to the Web services provider graph RPC transport layer.
3. The request goes from the RPC transport layer down through the layers of the provider graph to the business logic subgraph, and a response comes back up to the RPC transport layer.

   There are multiple paths through the layers of a provider graph, but one request going in should result in one response coming out.

4. The response goes back to the plug-in.
5. The response goes back to the client.

DEBUGGING A TEST QUERY

After sending a test query, you can see whether the request made it successfully into the provider graph and whether a response or fault came back to the client. If the test query failed, you can debug its route and find out where the error occurred and what went wrong.

▶ **To trace the route of a test query and debug:**

1. After noting the number of requests before you sent the test query and then sending the query, browse to the **Web Services** administration page.

   See "Browsing to the administration Web page" (page 42).

2. Click **Refresh**.
3. Look in the **Requests** column for the number of requests to your Web service. Did the number of requests increase by one?

   If so, unless the status of the service is marked **Failed**, the request made it to your Web service graph.

   If not, see "Debugging the Web service's URL" (page 62) to figure out whether the record was misdirected because of an incorrect URL to your Web service. And depending on the application server where your plug-in is running, check either the Java EE application server logs or the IIS Windows event log.

4. While your Web services provider graph is running in the GDE, trace its processing:

   a. Did the number of records leaving the RPC SUBSCRIBE component in the RPC transport layer increase by exactly one record?

      If so, your Web services graph received the request.

      If not, the request never made it to your graph. Check to see if the request used the correct URL; see "Debugging the Web service's URL" (page 62).

   b. If the graph received the request, did the number of records entering and leaving the RPC PUBLISH component in the RPC transport layer increase by exactly one record?

      If so, your Web services provider graph sent a reply.

      If not, the reply was lost somewhere in the layers of your graph, and you need to debug your graph logic.

      If the number of records sent increased by more than one, the graph is generating too many reply records for a single request record, and you need to debug your graph logic.

5. At each layer of the graph, check the input and output endpoint components to see whether the number of records processed increased:
   a. Did the record go to the correct layer and subgraph?
   b. If so, did it leave the correct error port or success port?
   c. If not, is the dispatch logic in the subgraph correct, or is the wrong SOAPAction specified in the request?
6. Drill down into the business logic subgraph and look at the test record's progress there.
7. To debug further, run the graph in the GDE and use the graph troubleshooting methods described in "Troubleshooting" and "Debugging your graph" in Ab Initio Help.

   For example:

   - Put watchers on strategically chosen flows.
   - Use features for viewing data and analyzing the record contents.

     For example, while the graph is running, use the **View Data** dialog to view the request data, especially when the record comes out of the RPC transport layer subgraph and again later when it comes out of the SOAP layer subgraph.

DEBUGGING THE WEB SERVICE'S URL

If you have used the wrong URL to your Web services provider graph, requests can be misdirected. See "Discovering the URL of a Web service" (page 48).

If the URL is incorrectly defined or you have used the wrong URL, client applications and CALL WEB SERVICE or CONTINUOUS CALL WEB SERVICE components in client graphs could misdirect requests and responses.

In the URL, check that you have used the correct:

- Host name
- Port number
- Application server
- Web services plug-in
- Web services provider graph

Troubleshooting tips

Here are troubleshooting tips for fixing common errors with your Web services provider graph:

ITEM	WHAT COULD BE WRONG	SEE
SOAPAction	The request did not go to the correct operation subgraph. You could have specified an incorrect SOAPAction in your request. The dispatch logic (**Dispatch on SOAPAction**) in your provider graph could be wrong.	"Error handling" (page 16)
SOAP formats	SOAP formats between the provider graph and the client are mismatched. Your provider graph or your client could be out of date. Update the provider graph to match the client application's SOAP formats.	—
Response	Your provider graph sent a response to the wrong client, or the response did not get back to the plug-in. The RPC header could have been corrupted in the RPC transport layer.	—
Request	The client request was misdirected because the client used the wrong URL.	"Debugging the Web service's URL" (page 62)
Application server	The application server is down or hung blocking communication of the client and the provider via the plug-in. Contact your application server administrator and make sure the application server is restarted and the plug-in is running.	—
Provider graph	The provider graph is down or hung and blocking communication to the plug-in and the client. Restart the provider graph.	—

ITEM	WHAT COULD BE WRONG	SEE
RPC components	The RPC connector components are listening on the wrong port. The RPC SUBSCRIBE and RPC PUBLISH components in the provider graph must be listening on the same port on the same host machine. For example, if the RPC SUBSCRIBE component in the RPC transport layer of your provider graph is listening on port 9090 (specified in the **port** parameter), the corresponding RPC PUBLISH component must also be listening on port 9090.	RPC SUBSCRIBE and RPC PUBLISH in Ab Initio Help
Firewall	There is a firewall between the application server and your Web services provider graph. Make sure the port on which your provider graph's RPC SUBSCRIBE component listens is opened in the firewall.	RPC SUBSCRIBE in Ab Initio Help
Plug-in	The plug-in is not installed or is incorrectly installed on the application server.	"Getting and installing a Web services plug-in" (page 40)
Plug-in's list	Your provider graph was not added to the plug-in's list of Web services.	"Adding Web services" (page 44)
Configuration information	Configuration information for your Web service provider graph is incorrect. For example, the provider graph name is misspelled, or the username or graph host name is incorrect. Edit the configuration page to make the appropriate corrections.	"Adding Web services" (page 44)

5

Web services utilities

This chapter describes the utilities you use when developing Web services provider graphs:

dml-to-wsdl

PURPOSE

Generates a valid WSDL file from DML type definitions.

If you have existing business logic, you can use this utility to generate a WSDL file for your Web services provider graph. As input DML, use the file containing the DML record formats and type definitions of the input to your business logic subgraph(s). As output DML, use the file containing the DML record formats and type definitions of the output from your business logic subgraph(s).

SYNTAX

dml-to-wsdl **-service** *svc_name* **-namespace** *target_ns* **-address** *url_addr*
{ { **-operation** *op_name* **-input** *input_dml* **-output** *output_dml* } **-soap-action** *action* ...}

ARGUMENT	DESCRIPTION
svc_name	Required. Name of the Web Service.
target_ns	Required. Namespace where you want the Web service to be defined.
url_addr	Required. URL where the Web service is to be located.
op_name	Required. Name of the operation used to create SOAPAction.
input_dml	Required. Name of the file containing the DML for the operation's input.
output-dml	Required. Name of the file containing the DML for the operation's output.
action	Required. The SOAPAction string for the operation. This string is the same as the binding name in the WSDL file **SOAPAction**=*string*; for example, **SOAPAction**=**"Average"**.

EXAMPLE

Suppose you want to create a Web service provider graph to provide the average of two numbers, and you have the following two DML files:

AverageIn.dml:

```
record
   decimal('\0') a;
   decimal('\0') b;
end
```

AverageOut.dml:

```
record
   decimal('\0') a;
   decimal('\0') b;
   decimal('\0') avg;
end
```

Suppose also that you want the XML types in the Web service defined in the context of a namespace **urn:namespace** and located at the URL **http://average.com/Average**.

Use this command to generate your WSDL file:

```
dml-to-wsdl -service AverageService -namespace "urn:namespace"
     -address "http://average.com" -operation "Average"
     -input AverageIn.dml -output AverageOut.dml -soap-action Average
```

The command generates the following WSDL file:

```
<definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="urn:namespace" targetNamespace="urn:namespace">

<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="urn:namespace" elementFormDefault="qualified">
      <xsd:element name="averageRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="a" type="xsd:decimal"></xsd:element>
            <xsd:element name="b" type="xsd:decimal"></xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="averageResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="a" type="xsd:decimal"></xsd:element>
            <xsd:element name="b" type="xsd:decimal"></xsd:element>
            <xsd:element name="avg" type="xsd:decimal"></xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
  </xsd:schema>
</types>
  <service name="AverageService">
    <port name="AverageServiceBinding" binding="tns:AverageServiceBinding">
      <soap:address location="http://average.com"></soap:address>
    </port>
  </service>
  <binding name="AverageServiceBinding" type="tns:AverageServicePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"></soap:binding>
    <operation name="Average">
      <soap:operation Soapaction="Average"></soap:operation>
      <input>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"></soap:body>
      </input>
      <output>
         <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"></soap:body>
      </output>
    </operation>
  </binding>
  <message name="averageResponse">
    <part name="parameters" element="tns:averageResponse"></part>
  </message>
  <message name="averageRequest">
    <part name="parameters" element="tns:averageRequest"></part>
  </message>
  <portType name="AverageServicePortType">
    <operation name="Average">
      <input message="tns:averageRequest"></input>
      <output message="tns:averageResponse"></output>
    </operation>
  </portType>
</definitions>
```

You can use this WSDL file as input to the **wsdl-to-dml** utility and generate DML types that the READ XML, READ XML TRANSFORM, WRITE XML, and WRITE XML TRANSFORM components can use. For more information about these components, see Ab Initio Help.

Alternatively, any WSDL-enabled third-party application can use this WSDL file to call the operation using SOAP/XML. Examples of such applications are XMLSpy Enterprise, WebServiceStudio, and SOAP Scope.

- *NOTE:* All DML utilities require AB_HOME to be set to the location of the Co>Operating System you want to use to run the utility. Your PATH must also contain the location of the **/bin** directory of the Co>Operating System.

wsdl-to-dml

PURPOSE

Generates DML bindings from XML definitions in a WSDL file.

SYNTAX

wsdl-to-dml *wsdl_name output_dir* [*op_name*] [**-use-envelope**] [**-no-envelope**]

ARGUMENT	DESCRIPTION
wsdl_name	Required. Name of the WSDL file.
output_dir	Required. Directory where the generated DML files will be placed.
op_name	Optional. Specifies a single operation so that **wsdl-to-dml** translates only that one operation.
-use-envelope	Optional. Specifies that **wsdl-to-dml** should use a SOAP envelope and produce complete DML describing that entire SOAP envelope. This behavior is the default.
-no-envelope	Optional. Specifies that **wsdl-to-dml** should not use a SOAP envelope and not produce complete DML describing the SOAP envelope.

EXAMPLE

For an example and detailed information about using **wsdl-to-dml** to generate Schema DML for a Web services provider graph, see "Generating Schema DML" (page 26).

6

Web services components

This chapter lists the Ab Initio components used in Web services client and provider graphs and refers you to detailed descriptions of these components in Ab Initio Help. It includes the following topics:

- Web services provider graph components (page 72)
- Web services client graph components (page 73)
- Web services subgraph components (page 74)

For information about how to use these components in the Web services client and provider graphs, see:

- "Developing Web services graphs" (page 21)
- "Example of a Web services provider graph" (page 8)
- "Example of a Web services client graph" (page 18)
- **$AB_HOME/ Connectors/SOAP** — Web services example graphs

Web services provider graph components

This section lists the essential components for building a Web services provider graph. You can find detailed descriptions of these components in Ab Initio Help.

KEY COMPONENTS FOR PROVIDER GRAPHS

The essential components for developing Web services provider graphs are:

- READ XML
- READ XML TRANSFORM
- RPC PUBLISH
- RPC SUBSCRIBE
- WRITE XML
- WRITE XML TRANSFORM

KEY CORE COMPONENTS FOR PROVIDER GRAPHS

These widely used core components are also essential in Web services provider graphs:

- FUSE
- GATHER
- REFORMAT

JMS AND MQ COMPONENTS FOR PROVIDER GRAPHS

In graphs that provide JMS or MQ Web services, you also use the following components:

- JMS PUBLISH
- JMS SUBSCRIBE
- MQ PUBLISH HEADERS
- MQ SUBSCRIBE

Web services client graph components

This section lists the essential components for building a Web services client graph. You can find detailed descriptions of these components in Ab Initio Help.

KEY COMPONENTS FOR CLIENT GRAPHS

The essential components for developing Web services client graphs are:

- CALL AB INITIO RPC
- CALL AB INITIO RPC TRANSACTION
- CALL WEB SERVICE
- CONTINUOUS CALL WEB SERVICE
- READ XML
- READ XML TRANSFORM
- WRITE XML
- WRITE XML TRANSFORM

KEY CORE COMPONENTS FOR CLIENT GRAPHS

These widely used core components are also essential in Web services client graphs:

- FUSE
- GATHER
- REFORMAT

MQ COMPONENTS FOR CLIENT GRAPHS

For accessing MQ queues from Web services client graphs, you use these components:

- CALL MQ-SERIES SERVICE
- CALL MQ-SERIES SERVICE TRANSACTION

Web services subgraph components

In the **Component Organizer**, the **$AB_HOME/connectors/SOAP** folder contains reusable subgraphs that you can copy (drag and drop) into the GDE and use as templates as you build your own SOAP Web services provider and client graphs (see "Developing Web services graphs" on page 21).

These subgraph components in the **SOAP** folder are:

SUBGRAPH COMPONENT	SEE
RPC Transport Layer	The RPC transport layer (page 11)
JMS Transport Layer	The JMS transport layer (page 12)
SOAP Layer	The SOAP layer (page 14)
Call SOAP HTTP Service	The HTTP client subgraph (page 19)
Call SOAP RPC Service	The RPC client subgraph (page 20)
Unknown SOAPAction Fault	Unknown SOAPAction faults (page 17)

Index

Symbols

A

B

C

D

E

G

I

J

M

O

P

R

S

T

U

W