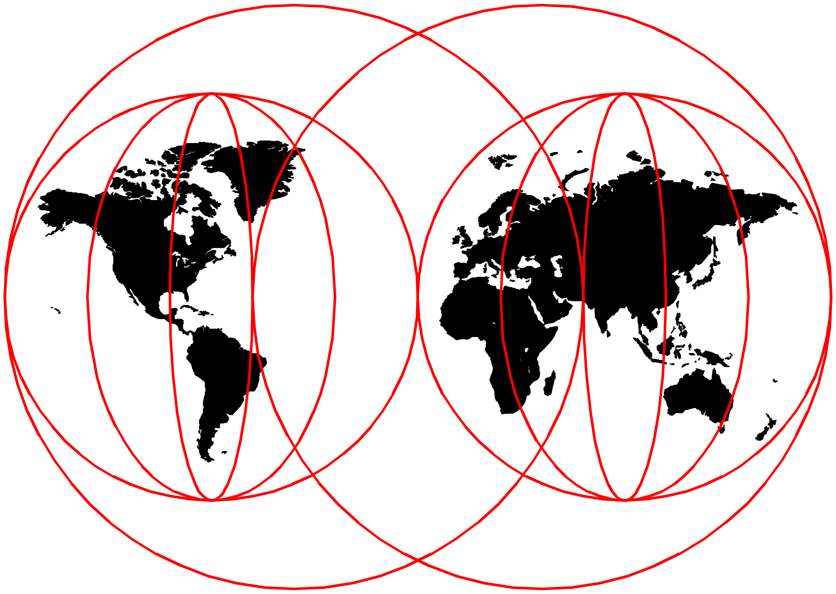
Developing e-business Applications Using Lotus Enterprise Solution Builder R3.0

Seiji Hamada, Masaya Higuchi, Isao Kadowaki, Makoto Katayama,
Shuhichi Murai, Kaori Nanba, Takashi Saitoh, Naomi Zenge



International Technical Support Organization

www.redbooks.ibm.com

SG24-5405-00

SG24-5405-00

International Technical Support Organization

Developing e-business Applications Using Lotus Enterprise Solution Builder R3.0

March 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special notices” on page 269.

First Edition (March 2000)

This edition applies to Lotus Enterprise Solution Builder for Domino Release 3.0.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

Learn how to develop applications using Lotus Enterprise Solution Builder (ESB) R3.0! This redbook was written for developers who intend to integrate the solutions of the e-business view from an actual implementation by showing a sample source program. This redbook is especially designed for users who plan to access enterprise backend systems from ESB. It includes sample programs that were tested error free and run on the ESB R3.0 environment.

ESB was previously called *LSCube*, which was available only in Japan. It was released in September 1997 as version 1.0. Please note that if you see the word LSCube in a running ESB product, it is actually referring to ESB. The difference is only in the name, not in the functionality.

Prior to reading this redbook, you must understand what ESB is and its functionality. You must also have read the *ESB R3.0 User's Guide*.

The team that wrote this redbook

This redbook was produced by the ESB development team at the Yamato Software Lab at IBM Japan.

Seiji Hamada is a Technical Master at the Solution Technology Development, Business Intelligence Solution Development, Yamato Software Lab, IBM Japan. He is a technical lead of ESB development team, and coordinated the writing of this redbook from a technical point of view. He wrote Chapter 1 and Chapter 2 of this redbook.

Masaya Higuchi works in Quality Evaluation for the ESB development team. His experience with in-depth connectivity testing on backend systems proved as a solid basis for him to write Chapter 8 on connecting to a relational database.

Isao Kadowaki is a staff programmer on the ESB development team. He is responsible for IDE and connectivity with the backend system. His three years of experience in accessing transaction systems and involvement in implementing actual customer solutions provided the foundation he needed to write Chapter 9.

Makoto Katayama is a staff programmer on ESB development team. He leads the Quality Evaluation group of the ESB development team. He wrote the frequently asked questions (FAQs) in Appendix A from a user's point of view.

Shuhichi Murai works in Quality Evaluation for the ESB development team. He specializes in System Verification Testing. He wrote on the complex topic of WebSphere in Chapter 6.

Kaori Nanba writes manuals and develops ESB demonstration programs for the ESB development team. She created most of the demonstration programs that were presented here and shown at Solution ’99 in Las Vegas, Nevada and at DevCon99 in San Francisco, California. She contributed her expertise in writing Chapter 5 and Chapter 7.

Takashi Saitoh is a staff programmer on the ESB development team. His expertise in debugging ESB application programs and experience with the ESB System Manager helped him to write Chapter 10 on deploying ESB applications.

Naomi Zenge is a programmer on the ESB development team, and is responsible for the Client Link component. His vast knowledge of ESB was an asset in writing Chapter 3 and Chapter 4.

Thanks to the following people for their invaluable contributions to this project:

Takashi Ogura
First line manager of ESB development team

Masahiro Ohkawa
Tomoko Mito
Michio Kikuchi
Tadaaki Kawamura
Iwao Inagaki
Osamu Furusawa
ESB development team — Yamato Software Lab, IBM Japan

Mark Field
Martha Hoyt
Bart Lautenbach
Lance Young-Ribeiro
Lotus Cambridge, Lotus Corporation

A special thank you goes to the following people who dedicated special efforts to oversee the publication of this redbook:

Takeshi Sakai (Project coordinator)
ESB development team – Yamato Software Lab, IBM Japan

Yasuhiro Kozuru (Japanese to English translator)
ITAS (International Translation and Services) Corp.

Jenifer Servais (Editor)
ITSO Rochester

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks review” on page 285 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Chapter 1. Introduction

Lotus Enterprise Solution Builder (ESB) provides an Integrated Development Environment (IDE) and a runtime environment, which allows LotusScript to be used in developing and deploying sophisticated high-volume enterprise applications. ESB solutions cover a full range of business activities and can be used to access any enterprise system or application. By using the enhanced functions provided by the IDE, you can develop three-tier applications that integrate a Lotus Domino application, existing relational databases, and such application systems as the Enterprise Resource Planning (ERP) and host enterprise system. The development cycle for integrating the ESB solution entails only a relatively short period of time. You'll benefit from the efficient performance that ESB applications offer in the enhanced ESB runtime environment.

This chapter describes the characteristics and positioning of ESB.

1.1 Business environment

The typical office-working environment has dramatically changed with the introduction of groupware. Groupware installations help to improve work efficiency and to manage an enormous amount of data that accumulates in office environments. Lotus Notes and Lotus Domino are the leading groupware solutions used in office environments, which enable you to send and receive electronic mail, manage schedules, share information through a document database, and track project progress.

However, in addition to these office-related tasks that Notes excels at, an office environment also typically includes such enterprise systems as an order processing system and a management information system. Integration of these systems and a Lotus Notes or Domino system in your current business environment is urged. Taking advantage of ESB, you can easily develop and operate such applications.

1.2 Features of ESB

ESB offers the following features:

- **An enhanced runtime environment**

  ESB provides an enhanced runtime environment that supports multi-thread processing to handle requests from multiple clients simultaneously. You can develop and run any business application, such as order entry, stock control, reservations, or accounting, that accesses external data sources. This environment also supports thread pooling functionality, which enables you to quickly respond to client requests. The ESB Runtime environment can directly manage client requests by using CORBA/IIOP, which are the basic technologies for handling distributed objects, and Microsoft DCOM (for the Windows NT version) instead of routing them through Domino. This means that Notes-based and Web-based clients can run applications in real time without knowing what is happening in the background.

- **Integrated Development Environment (IDE) of ESB**

  In addition to features like compound document interface (Multiple Document Interface (MDI)) support, a project browser, and class creation tools, ESB provides an extended environment for developing server applications in the IDE provided in the Notes environment. It also enables the development of work applications composed from multiple source files. As a result, you can develop deployable full-scale working applications efficiently in a very short period of time. In addition to these programming tools, we have added and support new, original ESB classes.

- **Support for various clients**

  ESB supports a variety of clients. Since Web clients and Notes clients are treated in the same way, you can develop and run business applications on the server without worrying about differences between client types. ESB also works with clients that support object linking and embedding (OLE) automation (such as Visual Basic). Since the business logic of application program is executed on the ESB server, there is no need to install any programs on client machines to gain access to ESB external data sources.

- **Support for LotusScript**

  The LotusScript supported by ESB is an object-oriented interpreter language that meets cross-platform requirements. Since it is already provided in Lotus Notes and other Lotus applications, you do not have to learn a new programming language. Anyone who has used LotusScript or who has script-like skills can quickly and easily develop programs with ESB.

- **Easy backend access**

  Lotus offers an entire family of connectors for accessing and integrating enterprise data and e-business applications. The Lotus Domino Standard Connectors are included with Domino and with Lotus Enterprise Integrator (LEI) for access to DB2, Oracle, Sybase, OLEDB, ODBC, EDA/SQL, text, and flat file systems. The Lotus Domino Standard Connectors are also included with ESB for access to DB2, Oracle, Sybase, and ODBC. The Lotus Domino Premium Connectors are sold separately and enable access to SAP, R/3, PeopleSoft V7.0 and V7.5, Oracle applications, J.D. Edwards with connectors for MQSeries, CICS, and XML in development. You can also use other Lotus Software eXtension (LSX) groups where the LSX is limited to be multi-thread-safe designed.

- **Load balancing with the Domino server**

  When ESB is used in conjunction with Lotus Domino, you may offload LotusScript server agents where those agents work in collaboration with external resources to ESB. In doing so, you significantly reduce the load on the Domino server.

1.3 Positioning ESB

Lotus has developed tools and techniques that work with enterprise systems. The Lotus Domino family of enterprise integration tools offers a comprehensive range of functionality to progressively deliver increasingly sophisticated and interactive e-business solutions to meet the dynamic and changing needs of the Web-enabled, extended enterprise. It is important that you understand the characteristics and the value proposition of each product. It is equally important that you use these products together as your first step in obtaining the best overall solution.

The following sections offer a brief explanation of the Lotus Domino family of enterprise integration tools that enable collaboration with enterprise backend systems.

1.3.1 Domino Enterprise Connection Services: Data virtualization

Domino Enterprise Connection Services (DECS) is included and supported as a standard component from Domino R4.6.3 and Domino R5 or later. DECS is ideal for rapid application development. It provides real-time, forms-based connectivity between Domino and any enterprise system or application, with no programming required. DECS lets you access and update mission-critical enterprise information as if it were native to Domino, creating “virtual” views of RDBMS, ERP, and transactional data without moving, duplicating, or compromising the integrity of the data.

1.3.2 Lotus Enterprise Integrator: Data movement

Lotus Enterprise Integrator (LEI) is a server-based tool that supports high-volume, bi-directional data transfer. It also maintains the synchronization of data between any enterprise source or target that is supported by a Domino Connector, including Domino applications, without programming. LEI provides high-volume and high-performance data transfer and synchronization, which can be a pre-selected schedule or event.

1.3.3 Lotus Enterprise Solution Builder: Complete customization

Lotus Enterprise Solution Builder (ESB) is an integrated development and runtime environment specifically designed for high-performance access to backend systems. ESB provides Domino solutions with a multi-tiered environment, while offering application developer enhanced flexibility to deliver sophisticated and interactive solutions. Applications that run on ESB can execute remotely from Web clients, Notes clients, or 32-bit Windows systems.

1.3.4 Lotus Domino Connectors

Lotus Domino Connectors are dynamic library links that handle the connection, authentication, and data translation between Domino and any Domino Connector support system. The Lotus Domino Connectors are used with ESB, LEI, and DECS. The data sources that are supported include RDBMS, ERP, and Transaction Processing Systems and Applications. The Lotus Domino Connectors, which include RDBMS support, are bundled with each of the Enterprise Integration products as a standard feature. The Lotus Domino Premium Connectors, with access to ERP and Transaction Processing Monitors, are provided as individual products and are available for purchase from Lotus Development Corporation.

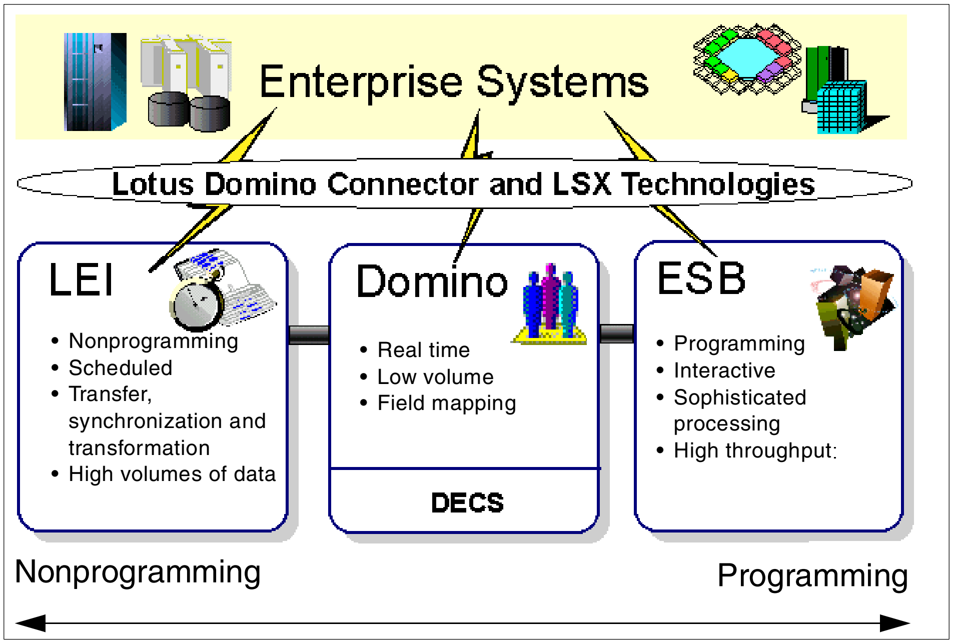


Figure 1. Positioning Lotus Enterprise family connectivity

1.4 ESB functions

ESB functions (Figure 2 on page 6) are divided into a runtime environment and a development environment. The functions are described in the following sections.

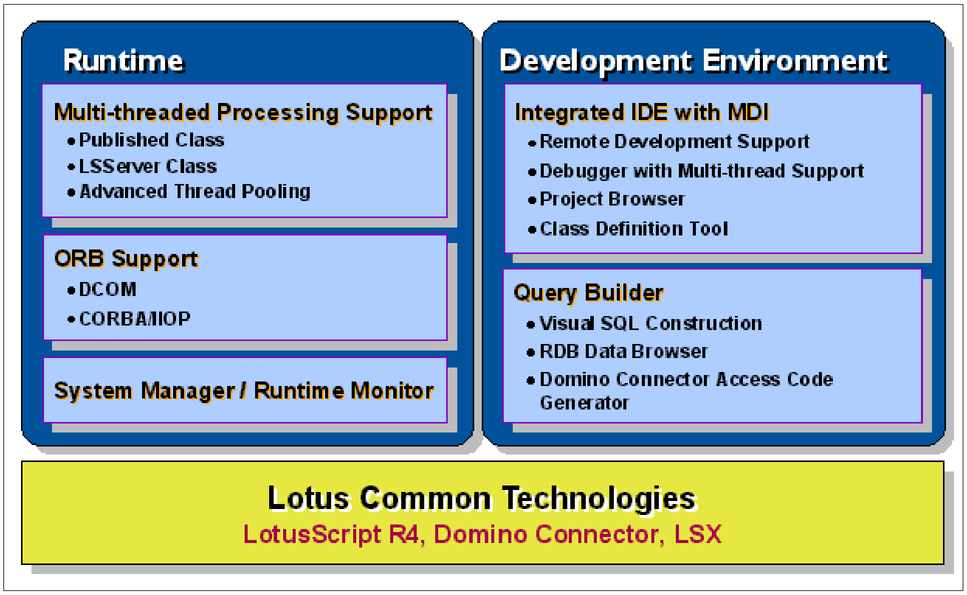

Runtime
Multi-threaded Processing Support
•Published Class
•LSServer Class
•Advanced Thread Pooling
ORB Support
•DCOM
•CORBA/IIOP
System Manager / Runtime Monitor
Development Environment
Integrated IDE with MDI
•Remote Development Support
•Debugger with Multi-thread Support
•Project Browser
•Class Definition Tool
Query Builder
•Visual SQL Construction
•RDB Data Browser
•Domino Connector Access Code Generator
Lotus Common Technologies
LotusScript R4, Domino Connector, LSX

Figure 2. ESB functions

1.4.1 Runtime environment

The following sections explain the ESB Runtime functions.

1.4.1.1 Multi-thread support

Multi-thread support includes the following enhanced functions:

- **Published class**

  The LotusScript keyword *Published*, which was newly added to ESB, is placed in front of the class keyword. This permits the Published class objects to be remotely executed or invoked from the supported ESB clients.

- **LSServer class**

  The keyword *LSServer* is also newly provided for the sharing of data between the respective threads run by multi-threads and for the serialization of various resource accesses.

- **Advanced thread pooling**

  The *thread pooling* function, which permits a variety of settings, enables an outstanding response to client requests.

1.4.1.2 ORB support

ESB supports Object Request Broker (ORB) to assure superior interactivity for connections between clients and ESB Runtime. ORB, in turn, supports Microsoft's DCOM (Windows NT version only) and CORBA/IIOP. Due to firewall and security considerations with IIOP, you may prefer to use the HTTP protocol. If you do not wish to change its settings, ESB also supports HTTP protocol combined with IBM WebSphere so that the client uses Web browsers. ORB is not used as the protocol with the client.

1.4.1.3 System Manager and Runtime Monitor

ESB provides a *System Manager* as a tool for running or operating packages created in IDE. A *Runtime Monito*r is also incorporated as a management tool that displays the operating conditions when applications are running. System Manager and Runtime Monitor are integrated as a single tool in the AIX version.

1.4.2 Development environment

ESB provides the development environment described in the following sections.

1.4.2.1 Integrated Development Environment (IDE)

The functions of ESB IDE are:

- **Remote development support**

  The ESB IDE permits the display of multiple script editors. As a result, you can simultaneously edit multiple files covering multiple projects.

- **Multi-thread support debugging function**

  A function is provided that uses the thread selector incorporated into the IDE to debug, while changing over threads.

- **Project browser**

  When simultaneously editing multiple files, you can switch files immediately by double-clicking a file name, class name, or procedure name from the tree view provided by the project browser.

- **Class Creation tool**

  The Class Creation tool provides a function for easily defining and creating user-defined classes and member procedures in applications on servers.

1.4.2.2 Query Builder

The functions of ESB Query Builder are described here:

- **Visual SQL Statement Creation function**

  Query Builder provides a graphical user interface (GUI) for creating SQL statements.

- **RDB data browsing**

  A function has been provided for the visual browsing of database information.

- **Automatic creation of access source codes using Domino Connectors**

  LotusScript skeleton codes accessed through the access class of Domino Connectors are automatically generated by using SQL statements, which are created by the visual SQL statement function.

Chapter 2. Architecture and configuration

This chapter presents an outline of the architecture and the system requirements of an ESB Runtime environment.

2.1 ESB architecture

This section describes the architecture of ESB.

2.1.1 ESB Runtime

ESB Runtime provides an enhanced LotusScript runtime environment that features such functions as thread pooling for responding quickly to simultaneous requests from multiple clients. ESB Runtime can be installed regardless of Domino installation on a same node or into another environment.

2.1.2 Control between the client and ESB Runtime

ESB Runtime provides a function for access from a client to application objects coded in LotusScript by using the Object Request Broker (ORB). ESB supports CORBA/IIOP and DCOM.

The changes in firewall settings are required when the ORB directly supports the clients beyond the firewall. To use IIOP on a Web browser, time is required to load the ORB module at the beginning. A servlet engine is required for HTTP protocol support. WebSphere 2.02, 3.0, or the servlet engine that ships with Domino 5.02b or later are recommended.

Note: The servlet engine with Domino versions earlier than 5.02b are not suitable. In this case, the WebSphere version that ships with Domino 5.0 and later should be used.

2.1.3 Client application programming interface

Three interfaces of Lotus Software eXtension (LSX), OLE automation, and applets are provided as the application interface with clients. Table 1 shows the relationship between the client type, link type, and protocol type.

Table 1. Client connectivity

Client type	Link type	High-level protocol	Remarks
Notes	LSX	IIOP or DCOM [3]	
	Applet	IIOP or HTTP [4]	

Client type	Link type	High-level protocol	Remarks
Web browser	Applet	HTTP [1] or IIOP	OLE Automation supports only MS IE
	OLE Automation	IIOP [2] or DCOM [3]	
Win32 (for example, Visual Basic)	OLE Automation	IIOP [2] or DCOM [3]	
Java application	Applet	IIOP	

Notes:

1. A servlet engine is required. WebSphere 2.02, 3.0, or Domino version 5.02b or later are recommended.
2. ESB provides a function for bridging OLE Automation and IIOP. The module of IIOP is included in the ESB package.
3. Only the Windows NT version of ESB Runtime supports this function.
4. Notes R5 or later is required.

2.1.4 Connection to backend data sources

The access methods described in the following sections are available for the connection between ESB Runtime and the backend data source.

2.1.4.1 Database system

Table 2 shows the connectivities and interfaces for the backend database. The access methods are provided as a standard component in ESB.

Table 2. Database connectivity

Data source	Link type	Access interface	Remarks
Notes/Domino	LSX	Notes Class	Backend class only
	Domino Connector	Connector Class	
DB2	Domino Connector	Connector Class	
	Native API Call	CLI	
Oracle	Domino Connector	Connector Class	
	Native API Call	OCI	

Data source	Link type	Access interface	Remarks
Sybase	Domino Connector	Connector Class	
Any ODBC Data Source	Domino Connector	Connector Class	The ODBC Driver must be completely thread-safe
	ODBC API Call	ODBC	

2.1.4.2 ERP system

Table 3 shows the connectivity of the backend system to the ERP system.

Table 3. IERP connectivity

Data source	Link type	Access interface	Remarks
SAP R/3	Domino Connector	Connector Class R/3	1, 2, 1
	LSX	LSX Class	
PeopleSoft	Domino Connector	Connector Class	1, 2
Oracle Applications	Domino Connector	Connector Class	1, 2
J.D. Edwards	Domino Connector	Connector Class	1, 2
Lawson	Domino Connector	Connector Class	1, 2
SSA	Domino Connector	Connector Class	1, 2

Notes:

1. You must verify the combination of support with ESB and so on at the Lotus Web site at `http://www.lotus.com` under Enterprise Integration.
2. These connectors are available as premium connectors at a separate package and price.

2.1.4.3 Messaging and transaction system

Table 4 on page 12 shows connectivity to the MQSeries and other transaction systems.

Table 4. IMQ and transaction system connectivity

Data source	Link type	Access interface	Remarks
MQ Series	LSX	MQ LSX Class	
CICS (directly of via MQSeries) IMS (via MQSeries) MQSeries	LSX	MQEI LSX	MQSeries Enterprise Integrator (part of the MQSeries and CICS Connections for Domino product from Lotus)
Notes: 1. You must verify the combination of support with ESB at the Lotus Web site at `http://www.lotus.com` under Enterprise Integration. 2. These connectors are available as premium connectors at a separate package and price.			

2.1.4.4 Other LSX components

You can use access components through LSX provided by sources other than Lotus to access a variety of backed data sources. Because ESB runs completely on a multi-thread environment, the LSX must be guaranteed to be thread-safe. Prior to applying to actual business deployment, it must be tested sufficiently.

Examples of other LSXs are supplied by sources in addition to Lotus include:

- Essbase LSX
- HLLAPI LSX (IBM eNetwork Personal Communication AS/400 and 3270)

Figure 3 shows an overview of ESB Runtime and the related products that work with ESB Runtime.

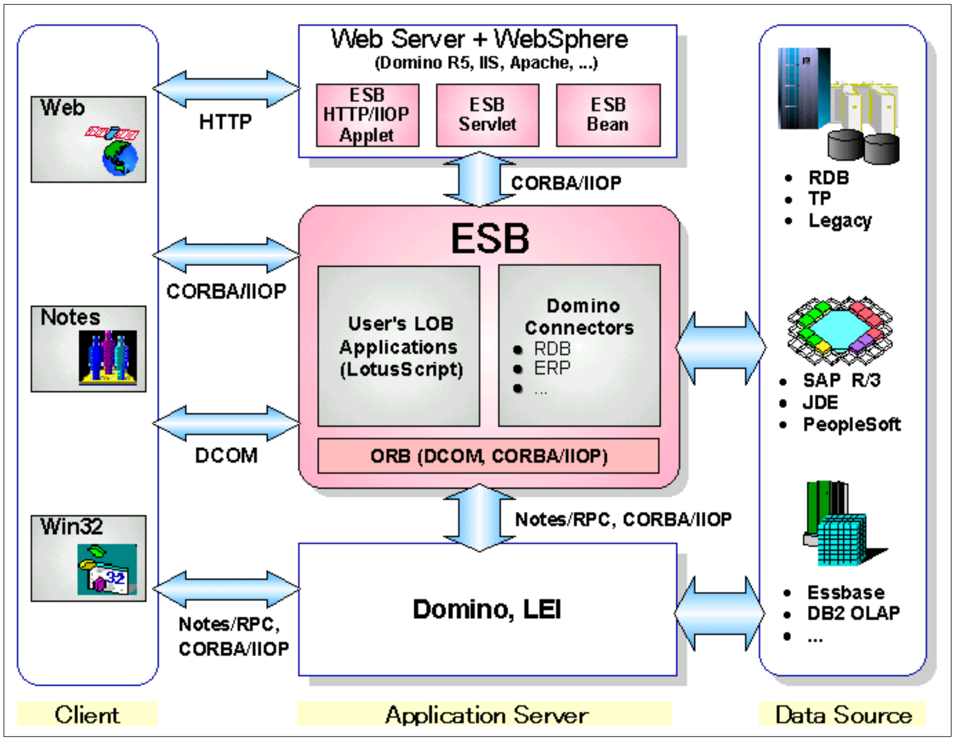


Figure 3. Overview of ESB Runtime and collaboration products

2.2 Supported platforms and system requirements

This section describes the supported platforms.

2.2.1 Supported platforms

ESB components are divided into three parts:

• **ESB Runtime**	Windows NT, AIX, and Solaris
• **ESB Developer**	Windows NT
• **ESB Client Enabler**	Windows NT, Windows 98, Windows 95, AIX, and Solaris

Refer to the system requirements in Table 5 through Table 10 on page 17 for detailed information, such as the operating system versions that are supported.

2.2.2 Windows NT version ESB Runtime

Table 5 shows the system requirements for the Windows NT version of ESB Runtime.

Table 5. Windows NT version of ESB Runtime system requirements

Item	Requirement
Hardware	IBM PC or 100% IBM compatible PC loaded with an Intel processor (Pentium 133MHz or higher is recommended) operated by Microsoft Windows NT version 4.0.
Disk capacity	100 MB of free capacity in a minimum configuration
Operating system	Microsoft Windows NT Server 4.0, Microsoft Windows NT Workstation 4.0 (Service Pack 3 or later must be applied)

2.2.3 AIX version ESB Runtime

Table 6 shows the system requirements for the AIX version of ESB Runtime.

Table 6. AIX version ESB Runtime system requirements

Item	Requirement
Hardware	RS/6000, operated by AIX (Version 4.3.2 or higher)
Disk capacity	/usr 200 MB /var 100 MB or more of free capacity
Memory capacity	128 MB (512 MB or more is recommended)
Operating system	AIX Version 4.3.2 or higher

2.2.4 Solaris version ESB Runtime

Table 7 shows the system requirements for the Solaris version of ESB Runtime.

Table 7. Solaris version ESB Runtime system requirements

Item	Requirement
Hardware	SPARC machine with Solaris version 2.6
Disk capacity	/opt 200 MB or more /var 100 MB or more
Memory capacity	128 MB or more (512 MB or more is recommended)
Operating system	Solaris 2.6
Software	Java Development Kit 1.1.8 for Solaris Production or Java Runtime Environment 1.1.8 Production Release

2.2.5 ESB Developer

Table 8 shows the system requirements for the ESB Developer.

Table 8. ESB Developer system requirements

Item	Requirement
Hardware	IBM PC or 100% IBM compatible PC loaded with an Intel processor (Pentium 133MHz or higher is recommended) operated by Microsoft Windows NT version 4.0.
Disk capacity	25 MB of free capacity in minimum configuration
Memory capacity	48 MB or more

Item	Requirement
Operating system	Microsoft Windows NT Server 4.0, Microsoft Windows NT Workstation 4.0 (SP3 or higher application) [1]
Note: 1. Verify the supported items of Windows 95 and Windows 98 at the Lotus home page for Developer package at: `http://www.lotus.com`	

2.2.6 Windows version of Client Enabler

Table 9 shows the system requirements for the Windows version of Client Enabler.

Table 9. Windows version of Client Enabler system requirements

Item	Requirement
Hardware	IBM PC or 100% IBM compatible PC loaded with an Intel processor operated by Microsoft Windows NT Workstation 4.0, Microsoft Windows 95 or Microsoft Windows 98.
Disk capacity	5 MB of free capacity (7 MB or more when installing sample)
Memory capacity	32 MB (Does not include capacity required to operate other applications)
Operating system	Microsoft Windows NT Workstation 4.0, Microsoft Windows 95, or Microsoft Windows 98
Software	• DCOM for Windows 95 Version 1.2 (Is included in ESB and is automatically installed when you install Client Enabler. The standard package of Microsoft Windows NT 4.0 and Microsoft Windows 98 includes DCOM) • LotusScript R4.5 or higher (when using Notes as the client) • Microsoft Internet Explorer 4.0 or higher (when using Web browser as the client) • Netscape Navigator 4.5 or higher (when using Web browser as the client)
Note: If the ESB HTTP Communication function is used, it is not necessary to install ESB Client Enabler.	

2.2.7 AIX version of Client Enabler

Table 10 shows the system requirement for the AIX version of the Client Enabler.

Table 10. AIX version of Client Enabler system requirements

Item	Requirement
Hardware	RS/6000, operated by AIX (Version 4.3.2 or higher)
Disk capacity	/usr 15 MB or more of free capacity (17 MB when installing sample)
Memory capacity	128 MB or more
Operating system	AIX Version 4.3.2 or higher

Chapter 3. Getting started with ESB

This chapter explains the following items in a tutorial format:

- Programming using the ESB Integrated Development Environment (IDE)
- Class Definition Tool and Client Code Creation Tool
- Creating an ESB server program that defines the Published class
- Creating a simple client program using ESB IDE
- Creating a client program using Notes
- Creating a client program using Microsoft Visual Basic
- Processing Runtime errors
- Transferring data between objects using the LSServer class
- Obtaining client information

As you read through this chapter, you should gain an understanding of the basic operations and language specifications of ESB and the procedure for creating task application programs.

3.1 Lesson 1: Creating your first ESB program

You use the Integrated Development Environment (IDE) of ESB Developer when developing ESB server programs. The ESB server application program defines Published classes, which is one of the functions that the ESB server provides. For the first step of the ESB tutorial, we create a “Hello World!” type program (for example, a statement line “Hello world!” is displayed on the console or in the message box). Although this program is not an ESB server program, because it does not define the Published class, you can obtain general knowledge of what the ESB project is or how to use the ESB IDE.

3.1.1 Starting the ESB IDE

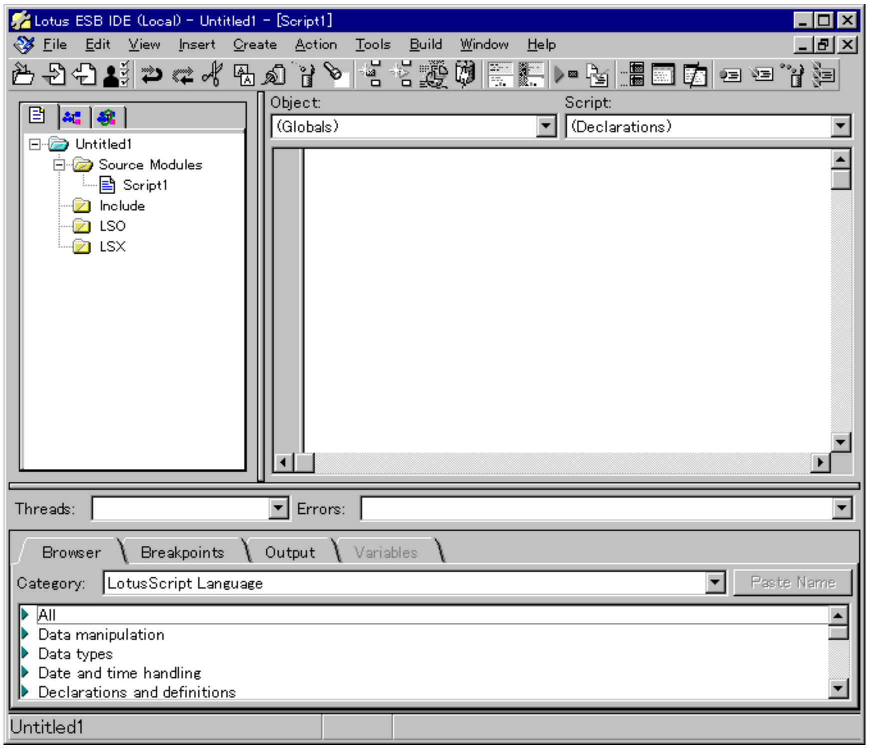
To start the ESB IDE, complete these steps:

1. Click the **Start** button on the task bar of a computer where ESP Developer was installed. Select the **Program** menu.
2. Select **Lotus ESP Developer**, and click **IDE**.
3. A dialog box asking for a user ID, password, and server name is displayed. Enter the character strings in the respective text boxes, and then click **OK.**

Hint

If ESB Runtime was installed locally, you can also start the local IDE. In this case, a dialog box asking for your user ID and so on is not displayed.

Figure 4 shows the window that appears after the IDE started.



Object: (Globals)

Script: (Declarations)

Threads:

Errors:

Category: LotusScript Language

Figure 4. IIDE initial view

3.1.2 Editing the Initialize procedure

In LotusScript, when a program starts to run, the Initialize procedure is automatically called once. You can insert a Print statement within this Initialize procedure to display the character strings.

To edit the Initialize procedure, complete these steps:

1. Click the **Script** drop-down list of the Script Editor in the upper right corner of the IDE.

2. Select **Initialize**.
3. Add the following statement:

```
Sub Initialize
   Print "Hello World!"    '<<== ADD
End Sub
```

3.1.3 Running and stopping the programs

To run and stop the programs, complete these steps:

1. Select **Build -> Run Project**.

   If it runs correctly, the character string `"Hello World!"` is displayed on the IDE Output panel.

2. Select **Debug -> Stop Running Project** to stop the program.

3.1.4 Saving the programs

Save the respective LotusScript and project files. Follow these steps:

1. Select **File-> Save Source Module as**, and save it as `Convert.lss`.
2. Select **File-> Save Project as**, and save it as `CFConv.lsp`.

3.2 Lesson 2: Defining a Published class

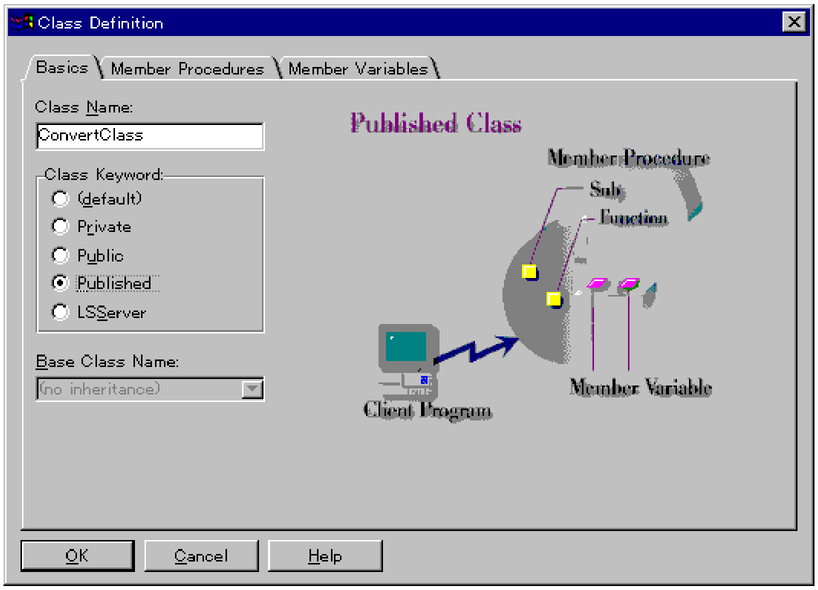
In this section, you add a Published class to the program you created in Lesson 1 to create an ESB server program. In the ESB application, ESB server programs and ESB client programs send and receive data though member procedures defined for the Published class.

3.2.1 Class Creation tool

Using the Class Creation tool of the ESB IDE facilitates the creation of class patterns when defining Published classes and other classes. Complete the following steps to create the template of the new Published class ConvertClass:

1. Select **Create -> Interface/Class**.

   Figure 5 on page 22 shows a view of the Class Creation tool.



Class Name: ConvertClass

Class Keyword:
○ (default)
○ Private
○ Public
● Published
○ LSServer

Base Class Name: (no inheritance)

Figure 5. Class Creation tool view

2. Enter the character string `ConvertClass` into the Class Name text box, and select the **Published** option from the Class Keyword group.
3. Click the **Member Procedure** tab.
4. Enter the character string `CtoF` into the Procedure Name text box. Respectively select **Function** for the Method Type and **Single** for the Return Type.
5. Click the **Parameter** button.
6. Enter the character string `c` in the Parameter text box. Select **Single** for Parameter Type. Then, click **Add**.
7. Click **OK**, and then click **Add**. The following function is displayed:

```
Function CtoF(c As Single) As Single
```

8. Repeat steps 4 through 7. Then, define the following function:

```
Function FtoC(f As Single) As Single
```

9. Click **OK**, and then click **OK** in the dialog box that asks whether you want to create a class.

3.2.2 Installing a member

To install a member, follow this process:

1. Enter the program into the Published class ConvertClass pattern created by the Class Definition tool.
2. Enter the LotusScript statement for each member procedure with the script editor:

```
Published Class ConvertClass
    Sub New()
        Print "New is called."
    End Sub

    Sub Delete()
        Print "Delete is called."
    End Sub

    Function CtoF(c As Single) As Single
        Dim f As Single
        f = c * (9/5) + 32
        CtoF = f
    End Function

    Function FtoC(f As Single) As Single
        Dim c As Single
        c = (f - 32) * (5/9)
        FtoC = c
    End Function
End Class
```

3.3 Lesson 3: Creating a client program

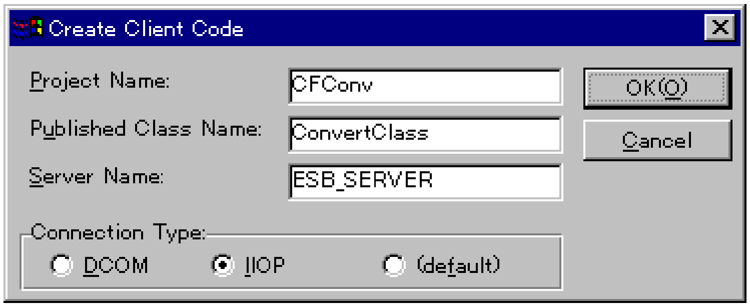
In this section, you create an ESB client program for communicating with the Published class ConvertClass of the ESB server program that you created in Lesson 2. Although ESB client programs are normally created in Domino Designer using LotusScript, this process uses the IDE to develop a simple program to confirm the operation of the Published class object.

3.3.1 Client Code Creation tool

ESB IDE includes a Client Code Creation tool for quickly creating operation confirmation programs. To use the tool, follow these steps:

1. Start the IDE again.
2. Click **Create -> Client Code**.

3. Enter `CFConv` in the Project Name box, `ConvertClass` in the Published Class Name box, and the TCP/IP host name or TCP/IP address in the Server Name box. Then, click **OK** (Figure 6).



Project Name: CFConv

Published Class Name: ConvertClass

Server Name: ESB_SERVER

Connection Type: ○ DCOM ● IIOP ○ (default)

Figure 6. Create Client Code display

4. Enter the portion described as "`<<== Add`" in the following code:

```
Sub Initialize
    Dim ORSObj As New SsClink
    Dim obj As Variant

    On Error Goto ErrorHandler
    On Event RuntimeError From ORSObj Call EventHandler

    ORSObj.ConnType = "IIOP"
    Set obj = ORSObj.CreateObject("CFConv.ConvertClass, node=ESB_SERVER")
    Print "Object is successfully created."

    '
    '   Now you can call Published class methods here.
    '   eg)
    '       Dim ret As Integer
    '       ret = obj.DoSomething(arg1, arg2)
    '

    Dim c As Single    '<<== Add
    Dim f As Single    '<<== Add
    f = 100            '<<== Add
    c = obj.FtoC(f)    '<<== Add
    Print f & " degrees Fahrenheit is " & c & " degrees Celsius." '<<== Add
    Set obj = Nothing
    Print "Object is successfully deleted."

    Exit Sub

ErrorHandler:
    Print "ErrorHandler: " & Cstr(Err)
    Print "ErrorHandler: " & Error
    Exit Sub
End Sub
```

5. To create a Published class object, specify a class name and server name. Then call the CreateObject member function of the SsClink class.

Hint

The ESB client application requires that you load the SsClink class using a USELSX statement. However, in ESB IDE, SsClink is loaded by default.

6. Select **File -> Save Source Module as**, and save it as `Client.lss`.
7. Select **File -> Save Project as**, and save it as `Client.lss`.

3.3.2 Running a program

To run a program, follow these steps:

1. Start two ESB IDEs.
2. Load the program you created in “Lesson 2: Defining a Published class” on page 21, in one IDE. Select **Build -> Run Project**, and then start the server program.
3. Load the program you created in “Lesson 3: Creating a client program” on page 23, in the other IDE. Select **Build -> Run Project**. Then, start the client program. When it runs normally, the following character string is displayed on the IDE output panel where the client program was run:

```
100 degrees Fahrenheit is 37.77778 degrees Celsius.
```

3.4 Lesson 4: Creating a client program using Notes LotusScript

Use Notes LotusScript to create a client program. The user enters a numerical value in a field on the Notes form, and then clicks the button to call the member procedure for the Published class.

Note

Two methods are provided in this lesson. One method uses Domino Designer of Notes R5, and the other method uses the design of Notes R4.

3.4.1 Creating a new client form

To create a new client form, follow this series of steps:

1. Select **File -> Database -> New** on the Notes workspace.
2. Enter `Tutorial` in the Database Name text box. Then, click **OK**.

3. Select **Create -> Design -> Form** in the Tutorial database. A new form is created, and the Untitled Form window opens.
4. Select **Design -> Form Properties** in the Untitled Form.
5. Enter `frmMain` in the Name text box in the Property Info Box. Then, click **Close**.

3.4.2 Creating a form (for Notes R5)

You can also directly create buttons and fields in forms. It is easier to create a table, create the buttons and fields within it, and then arrange the position and appearance of the form elements.

Note

You can also use a layout region to arrange the appearance of the form elements. However, the layout regions are not correctly displayed when it is published as a Web page. Tables are correctly displayed even on Web pages.

Figure 7 shows a view of designing the Form image on Notes R5 Designer.

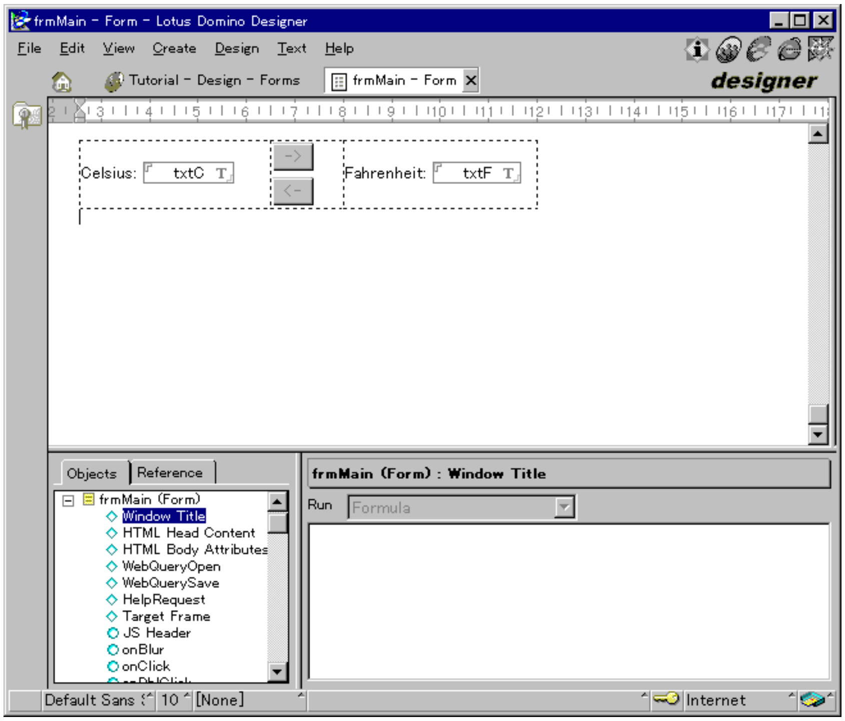


Figure 7. View of the design form image on Notes R5 Designer

3.4.2.1 Creating tables

To create the tables, complete these tasks:

1. Select **Create -> Table** on the FrmMain form.
2. Enter `1` in the Number of Rows box, and enter `3` in the Number of Columns box. Then, click **OK**.
3. Select the entire table. Then, select **Table -> Table Properties**.
4. Select **Center** in the Vertical Alignment list box in the Cell group of the Table Layout tab in the Property Info Box.
5. Click **Set all to 0** in the Cell Border Thickness group under the **Cell Borders** tab.
6. Click the **Close** button.
7. Use the ruler to arrange the size of the table cells.

3.4.2.2 Creating fields

To create fields, follow this procedure:

1. Click the leftmost cell of the table, and then enter `Celsius:`.
2. Select the **Create Field** menu.
3. Enter `txtC` in the Name box under the **Field Info** tab of the **Property Info** box. Then, click the **Close** button.
4. Click the rightmost cell of the table, and enter `Fahrenheit:`.
5. Select the **Create Field** menu.
6. Enter `txtF` in the Name box under the **Field Info** tab of the **Property Info** box. Then, click the **Close** button.

3.4.2.3 Creating buttons

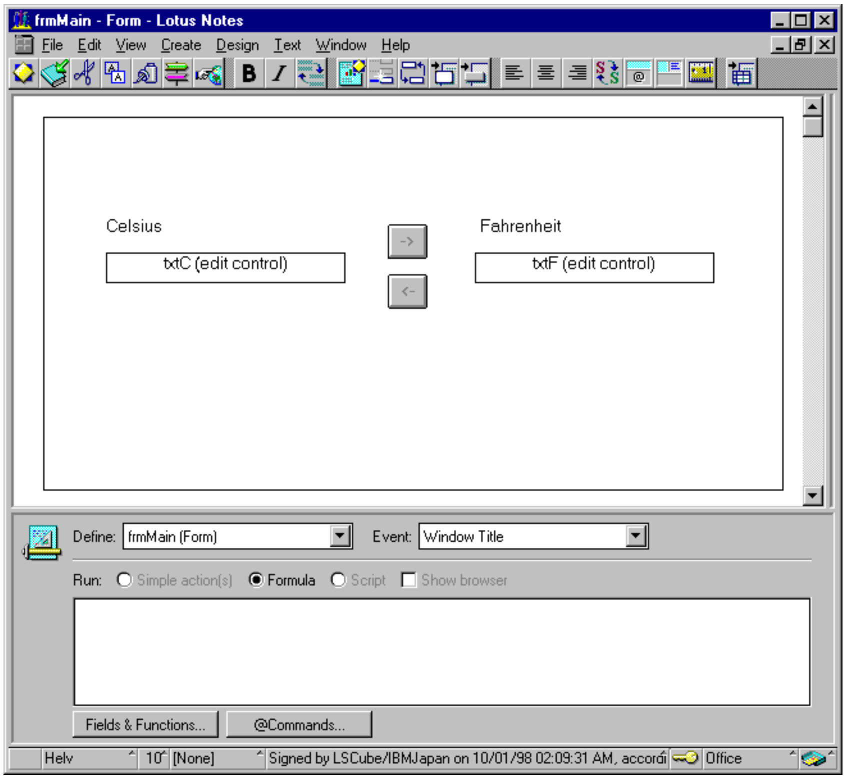
To create buttons, complete these steps:

1. Click the cell in the center of the table. Select **Create -> Hotspot -> Button**.
2. Enter `->` in the Name box under the **Button Info** tab of the **Property Info** box.
3. Press the Enter key.
4. Select **Create -> Hotspot -> Button**.
5. Enter `->` in the Name box under the **Button Info** tab of the **Property Info** box.

3.4.3 Creating a form (for Notes R4)

You can create such components as buttons and forms directly in a form. It is easier to create a layout region, create buttons and fields within it, and then arrange the position and appearance of the form fields.

Figure 8 shows designing a form image on the Notes R4.6 design panel.


frmMain - Form - Lotus Notes
File Edit View Create Design Text Window Help
B I
->
<-
Fields & Functions...
@Commands...

Celsius

txtC (edit control)

Fahrenheit

txtF (edit control)

Define: frmMain (Form)   Event: Window Title

Run: ○ Simple action(s) ● Formula ○ Script □ Show browser

Helv 10 [None] Signed by LSCube/IBMJapan on 10/01/98 02:09:31 AM, accordi Office

Figure 8. Designing a form image on the Notes R4.6 design panel

3.4.3.1 Creating layout regions

You can create layout regions by completing these steps:

1. Select **Form -> Layout Region -> New Layout Region** on the **FrmMain** form.
2. Drag the handle of the layout region and enlarge it to an appropriate size.
3. Select **Design -> Layout Properties**.
4. Click the **Basics** tab of the **Property Info** box. Turn the **Show Grid** and **Snap to Grid** check boxes to **On** under the Grid group. Then, click the **Close** button.

3.4.3.2 Creating fields

Create the fields by following these steps:

1. Select the newly created layout region on the frmMain form.
2. Select **Create -> Field**.

3. Select **Design -> Field Properties**.
4. Click the **Basics** tab in the **Property Info** Box. Enter `txtC` in the name text box. Then, click the **Close** button.
5. Repeat steps 2 through 4 to create another field. Enter `txtF` in the Name text box.

3.4.3.3 Creating buttons

To create buttons, complete these tasks:

1. Select **Create -> Hotspot -> Button**.
2. Select **Design -> Object Properties**.
3. Click the **Basics** tab of the **Property Info** box. Enter `->` in the Button Label text box. Click the **Close** button.
4. Repeat steps 1 through 3 to create another button. Enter `->` in the Button Label text box.

3.4.3.4 Creating static text

Create static text by completing these steps:

1. Select **Create -> Layout Region -> Text**.
2. Select **Design -> Object Properties**.
3. Enter `Celsius` in the **Text** text box in the **Static Text Options** group of the **Property Info** box. Click the **Close** button.
4. Repeat steps 1 through 3 to create another static text. Enter `Fahrenheit` in the **Text** text box.

3.4.4 Creating an event script

This section explains how a LotusScript sample code is produced when the the button is pressed or the form is loaded.

3.4.4.1 Loading an ESB client and an LSX file

The ESB client program uses the CreateObject member of the SsClink class to create a Published class object. Perform the following steps to load an ESB client and an LSX file:

1. Click the **Objects** of the Design panel, and select **(Globals)frmMain -> (Options)**.
2. Enter the following statement in the script editor:

```
Uselsx "*SsClink"
```

3.4.4.2 Defining a global variable

To define a global variable, complete these tasks:

1. Select **(Globals)frmMain -> (Options)** in the Design panel.
2. Enter the following statement in the script editor:

```
Dim uidoc As NotesUIDocument
Dim doc As NotesDocument
Dim myObj As Variant
```

3.4.4.3 Creating a Published class object

You can create a Published class object by following this process:

1. Select **frmMain(form) -> Postopen** in the Design panel.

> **Note**
>
> When using Notes R5 Domino Designer, select **LotusScript** in the Run box.

2. Enter the following statement with the script editor:

   Note: Replace the character string `ESB_SERVER` specified for the node parameter of the CreateObject member procedure function. Instead, use the TCP/IP host name of the computer where the ESB server is installed or the IP address in the dotted decimal notation (for example: `xxx.xxx.xxx.xxx`).

```
Sub Postopen(Source As Notesuidocument)
     Set uidoc = Source
     Set doc = uidoc.Document

     Dim ORSObj As New SsCLink
     ORSObj.ConnType = "IIOP"
     Set myObj = ORSObj.CreateObject( "CFConv.ConvertClass,
node=ESB_SERVER")
     Set ORSObj = Nothing
End Sub
```

3.4.4.4 Calling a member procedure

To call a member procedure, complete these steps:

1. Select **->(button) -> Click** in the Design panel.

Note

If you are using the Notes R5 Domino Designer, select the **LotusScript** menu in the Run box.

2. Enter the following statement in the Edit window:

```
Sub Click(Source As Button)
     Dim f As Single
     Dim c As Single

     c = CSng(uidoc.FieldGetText("txtC"))
     f = myObj.CtoF(c)
     Call uidoc.FieldSetText("txtF", Cstr(f))
End Sub
```

3. Enter the following statement in the event script for ->(button) -> Click:

```
Sub Click(Source As Button)
     Dim f As Single
     Dim c As Single

     f = CSng(uidoc.FieldGetText("txtF"))
     c = myObj.FtoC(f)
     Call uidoc.FieldSetText("txtC", Cstr(c))
End Sub
```

3.4.4.5 Deleting a Published class object

Follow these steps to delete a Published class object:

1. Select **(Globals)frmMain -> Queryclose** in the Design panel.
2. Enter the following statement in the Edit window:

```
Sub Queryclose(Source As Notesuidocument, Continue As Variant)
     Set myObj = Nothing
End Sub
```

3.4.5 Saving a form

Complete these tasks to save a form:

1. Select **File -> Save**.
2. Select **File -> Close**.

3.4.6 Running a program

To run a program, complete these steps:

1. Start IDE and load it. Run the program created in “Lesson 2: Defining a Published class” on page 21.
2. Start Notes. Click the **Tutorial** database icon from the workspace to open it. Select **Create -> frmMain**. A new document opens. Check that the ConvertClass object was created in the IDE that is running the server program.
3. Enter `0` in the Fahrenheit field. Click the **->** button. If converted correctly, `32` is displayed in the Fahrenheit field.

3.5 Lesson 5: Handling errors

In this lesson, you add a Runtime error handling routine to the client program. You add a field for specifying the server name and a button to the form. Then, you check its operation by generating a Runtime error by specifying for the server name a computer name that does not exist when running the program.

Use the LotusScript `ON ERROR` statement and SsClink class RuntimeError to perform the error handling.

3.5.1 Editing a form (for Notes R5)

This section explains how to add a connection button and the Server name input field in the form. Follow these steps to create a server name entry field:

1. Enter `Server name:` at the front of the form.
2. Select **Create -> Field**.
3. Enter `txtServer` in the **Name** box under the **Field Info** tab of the **Property Info** box.
4. Press the Enter key.
5. Select **Create -> Hotspot -> Button**.
6. Enter `Create Object` in the **Label** box under the **Button Info** tab of the **Property Info** box. Click the **Close** button.

Figure 9 on page 34 shows the newly added connection button and the Server name input field on Notes R5 Designer.

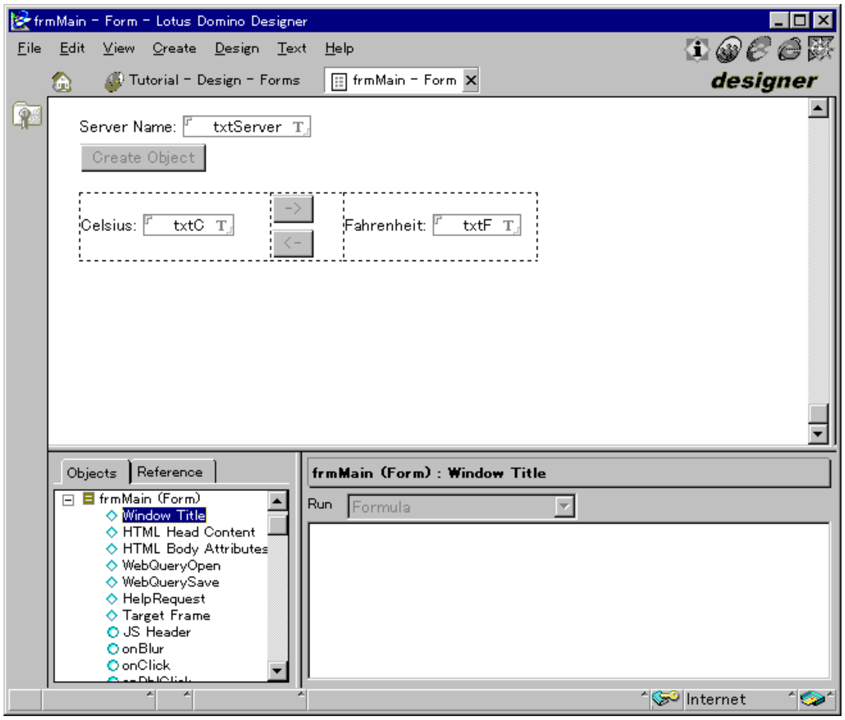


Figure 9. Notes R5 Designer connection button and server name input field

3.5.2 Editing a form (for Notes R4)

This section discusses the process for deleting a NoteBox. First, lay out the fields, buttons, and static text. Then, create a server name entry field. Follow these steps:

1. Click and select the layout region of the frmMain form.
2. Select **Create -> Field**.
3. Select **Design -> Field Properties**.
4. Click the **Basics** tab of the **Property Info** box. Enter `txtServer` in the Name text box. Click the **Close** button.
5. Select **Create -> Hotspot -> Button**.
6. Select **Design -> Object Properties**.
7. Click the **Basics** tab of the **Property Info** box. Enter `Create Object` in the Button Label text box. Click the **Close** button.
8. Select **Create -> Layout Region -> Text**.

9. Select **Design -> Object Properties**.

10. Enter `Server Name` in the Text text box in the Static Text Options group of the Property Info box. Click the **Close** button.

3.5.3 Editing an event script

With the client program of “Lesson 4: Creating a client program using Notes LotusScript” on page 25, we created a Published class object at the same time that a form was opened. However, when the user enters a server name in field and clicks the Create Object button, it changes the program, so that it creates a Published class object. Follow these steps to edit an event script:

1. Click the **Object** tab of the Design panel. Select **frmMain(Form) -> Postopen**.
2. Delete the statement that generates a Published class object from the statement within the procedure:

```
Sub Postopen(Source As Notesuidocument)
     Set uidoc = Source
     Set doc = uidoc.Document
End Sub
```

3. Select **Create Object (Button) ->Click** in the Design panel.

> **Note**
>
> Select **LotusScript** in the Run box when using Notes R5 Domino Designer.

4. Enter the following statement:

```
Sub Click(Source As Button)
     Dim ORSObj As New SsCLink
     Dim szServerName As String

     On Error Goto ErrorHandler
     On Event RuntimeError From ORSObj Call EventHandler

     szServerName = uidoc.FieldGetText("txtServer")
     ORSObj.ConnType = "IIOP"
     Set myObj = ORSObj.CreateObject( _
                 "CFConv.ConvertClass, node=" + szServerName)
     Set ORSObj = Nothing
     Exit Sub

ErrorHandler:
     Dim szMessage As String
     szMessage = "LotusScript Error : " + Cstr(Err) + ":" + Error
     Msgbox szMessage
     Exit Sub
End Sub
```

```
Sub EventHandler(ORSObj As SsClink, errorCode As Long, description As String)
     Dim szMessage As String
     szMessage = "ESB Error : " + Cstr(errorCode) + ":" + description
     Msgbox szMessage
End Sub
```

5. Select **File -> Save**.

3.5.4 Running a program

To check the Runtime error handling, run the Runtime program without starting up the server program. Follow these steps:

1. Select **Design -> Preview in Notes**.
2. Enter an appropriate character string, such as `UNKNOWN` in the Server Name field.
3. Click the **Create Object** button. When a timeout occurs, the following error message is displayed:

```
ESB Error:50017:Cannot connect to server UNKNOWN. The RPC server is not
available or not found name service.
LotusScript Error:219:Error accessing product object method
```

3.6 Lesson 6: Obtaining client information

You can obtain the user IDs and TCP/IP addresses that use the Published class from the server program and the environment information of the server where the Published class is operating. To obtain this information, invoke the GetContext function within the member procedure for the Published class.

3.6.1 Editing a server program

To edit a server program, follow these steps:

1. Start ESB IDE. Open the server program created in “Lesson 2: Defining a Published class” on page 21.
2. Add the following `<<== Add` statement to the CtoF member procedure:

```
Function CtoF(c As Single) As Single
Dim f As Single
Dim infoObj As Variant

f = c * (9/5) + 32
CtoF = f

Set infoObj = GetContext()      '<<== Add
Print "CtoF is called by " & infoObj.HostName     '<== Add
End Function
```

3.6.2 Running a program

To run the program, complete these tasks:

1. Run the server program.
2. Run the IDE client program created in “Lesson 3: Creating a client program” on page 23, or the Notes client program created in “Lesson 4: Creating a client program using Notes LotusScript” on page 25, and “Lesson 5: Handling errors” on page 33.
3. Enter `0` for the Celsius field and click the **->** button. If it runs correctly, the server displays the following statement on the output panel of the server:

   ```
   CtoF is called by XXX.XXX.XXX.XXX
   ```

3.7 Lesson 7: Creating a client program using Visual Basic

You can create an ESB client program if the language supports OLE automation, even if the script language is other than LotusScript. In this section, you use Microsoft Visual Basic (VB) to create a client program that connects to the ESB server.

Note

Use the CreateObject function of Visual Basic to create an SvClink class object. Use the CreateObject member function in SvClink to create a Published class object.

3.7.1 Creating a client program

Use VB to create a client program with the same functions as the program you created in “Lesson 5: Handling errors” on page 33.

3.7.1.1 Creating a form

To create a form, follow these steps:

1. Start Visual Basic.
2. Lay out the components as shown in Figure 10 on page 38.

   The name of the Create Object button is cmdCrateObject, the name of the -> button is cmdCtoF, and the name of the <- button is cmdFtoC.

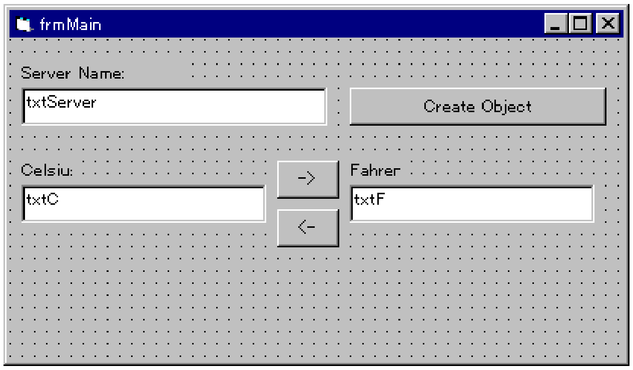


Figure 10. Form view created by Visual Basic

3.7.1.2 Editing programs

Edit the programs by following these steps:

1. Enter the following statement for the (General)-(declarations) procedure:

```
Dim myObj As Object
```

2. Enter the following statement for cmdCreateObject_Click:

```
Private Sub cmdCreateObject_Click()
    Dim ORSObj As Object
    Dim szText As String

    Set ORSObj = CreateObject("SVCLink")

    szText = "CFConv.ConvertClass, node=" & txtServerName.Text
    Set myObj = ORSObj.CreateObject(szText)

    Set ORSObj = Nothing
End Sub
```

3. Enter the following code for the cmdCtoF_Click procedure:

```
Private Sub cmdCtoF_Click()
    Dim c As Single
    Dim f As Single

    c = CSng(txtC.Text)
    f = myObj.CtoF(c)
    txtF = CStr(f)
End Sub
```

4. Enter the following code for the cmdFtoC_Click procedure:

```
Private Sub cmdFtoC_Click()
    Dim c As Single
    Dim f As Single

f = CSng(txtF.Text)
c = myObj.FtoC(f)
txtC = CStr(c)
End Sub
```

3.7.2 Running programs

Run the programs by following these steps:

1. Start IDE. Load the server program created in “Lesson 6: Obtaining client information” on page 36, and run it.
2. Select **Run -> Start** in Visual Basic.
3. Specify the TCP/IP host name where the ESB server is operating or the dot decimal type IP address (for example: `xxx.xxx.xxx.xxx`) in the Server Name text box. Click the **Create Object** button.
4. Enter `0` in the Celsius text box and click the **->** button. If it runs correctly, `32` is displayed in the Fahrenheit text box.

3.8 Lesson 8: The LSServer class

When you use the LSServer class, you can transfer data between Published classes. You can also serialize resource accesses, which is not possible with the Published class alone. Add the LSServer class MyCounter created in “Lesson 6: Obtaining client information” on page 36. Then, count the number of ConvertClass objects created once the server program has run. Output the value to the New script of the ConvertClass.

3.8.1 Editing server programs

This section decribes the procedure for adding the LSServer class definition to the server program.

3.8.1.1 Defining the LSServer class MyCounter

Use the Class Creation tool to define the LSServer MyCounter in the server program. Complete these steps:

1. Start ESB IDE. Load the program you created in “Lesson 6: Obtaining client information” on page 36.
2. Select **Create -> Interface/ Class**. Start the Class Creation tool.

3. Enter `MyCounter` in the Class Name text box. Select **LSServer** from the **Class Keyword** group.
4. Click the **Member Procedures** tab. Enter `GetNextNumber` in the Procedure Name text box. Respectively, select **Function** from the Procedure Type drop-down box and **Integer** from Return Type drop-down box. Click the **Add** button.
5. Click the **Member Variable** tab. Select the character string **counter** in the Variable Name text box and **Integer** from the Variable Type drop-down box. Click the **Add** button.
6. Click the **OK** button. You are asked whether you want to create a class. Here, click the **OK** button.

3.8.1.2 Editing the LSServer class MyCounter

Perform these tasks to edit MyCounter:

1. Enter the following statement in the New member procedure of the LSServer class MyCounter:

```
Sub New()
    Print "MyCounter::New() was called."
    counter = 0
End Sub
```

2. Enter the following statement in the Delete procedure:

```
Sub Delete()
    Print "MyCounter::Delete() was called."
End Sub
```

3. Enter the following statement in the GetNextNumber member procedure:

```
Function GetNextNumber() As Integer
    counter = counter + 1
    GetNextNumber = counter
End Function
```

3.8.1.3 Editing the Published class ConvertClass

To edit ConvertClass, enter the following statement in the New member procedure of the Published class ConvertClass:

```
Sub New()
    Dim counterObj As MyCounter
    Dim myNumber As Integer
     Set counterObj = Bind("")
    myNumber = counterObj.GetNextNumber()
    Print Cstr(myNumber) & " - ConvertClass::New() was called."
End Sub
```

3.8.2 Running a program

To run a program, follow these steps:

1. Run a server program. If run correctly, the following statement is displayed on the output panel of the server:

   ```
   'CFConv': Compiling Project Files...
   'Convert.lss': compiled successfully.
   All of the project files compiled successfully.
   MyCounter::New() was called.
   ```

2. Run the IDE client program created in “Lesson 3: Creating a client program” on page 23, or the Notes client program created in “Lesson 4: Creating a client program using Notes LotusScript” on page 25, and “Lesson 5: Handling errors” on page 33. When the ConvertClass object is created, it is displayed as follows on the Output panel:

   ```
   1 - ConvertClass::New() was called.
   ```

3. Exit the client program. Restart it and then create an object. It is displayed as follows on the Output panel:

   ```
   1 - ConvertClass::New() was called.
   ConvertClass::Delete() was called.
   2 - ConvertClass::New() was called.
   ConvertClass::Delete() was called.
   ```

Chapter 4. Server application programming

This chapter explains the following items relating to ESB Runtime programming:

- The configuration of the ESB Runtime, the environment used when developing runtime, and the tools and their main functions
- Definitions, usages and notes on Published classes, which are core elements of the ESB Runtime, and the procedures for sending and receiving the array data and user-defined data that are widely used in actual applications
- Definitions and usages of LSServer classes used in the transfer of data between objects and the serializing of processing, and the SsSharedStorage classes used for the sharing of data between projects
- Authentication and access control of ESB applications
- Notes on ESB application designs
- Topics relating to other ESB Runtime programming

This chapter simultaneously explains the ESB client programs required to confirm the operation of the ESB Runtime. For a more detailed explanation of client programming, refer to Chapter 5, “Client application programming” on page 91.

4.1 ESB Runtime

ESB Runtime defines at least one Published class and provides services through class member procedures. The Published class is a special class implemented in ESB. It can create objects outside of the runtime, particularly from computers other than those where the runtime is operating. It can call the member procedure defined in the class. The Published class can also be called from another process of the same computer. In this case, it can use a common client program without having to recreate the client program, according to the location where the ESB Runtime is running (local or remote).

Hint

When testing a created ESB application, you should first check the operation by running the runtime and the client program on the same computer. This allows you to efficiently locate errors and advance development when you lay out the server and client programs on a different computer to check their operation.

The Published class is defined in the source code (extension LSS), and is coded in LotusScript. The development and running units of ESB Runtime are referred to as *projects*. They save LotusScript information included in the project and information such as specific environment variables in project files (extension LSP). Aside from LotusScript files, projects can include several compiled LotusScript object files (extension LSO), several Include files, and several LotusScript component files (extension LSX).

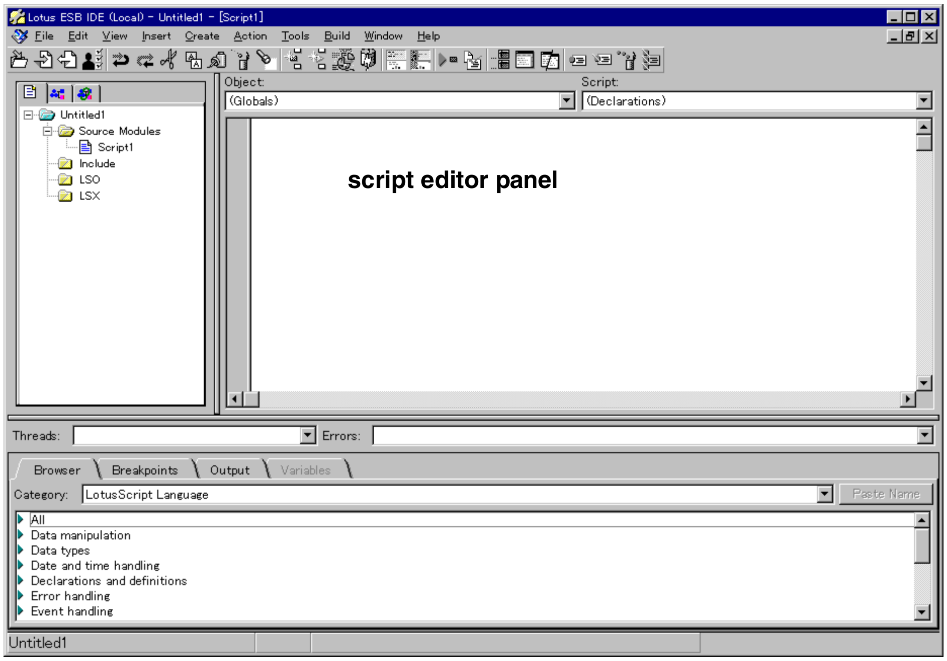
Completed ESB programs consolidate all of the files within a project into a single file, because they lack operability when configured since they are from multiple files. This is referred to as *packaging*. The created files are referred to as *package files* (extension LPK). ESB Runtime is operated and managed in the form of package files.

4.1.1 ESB Runtime development procedure

The following tasks show the ESB Runtime development procedure and the ESB tools and functions used in each step:

1. Start the ESB IDE and enter the source code of runtime in script editor panel.

   Figure 11 shows the view of ESB IDE just after it is started. The window in the right upper corner is a script editor panel.

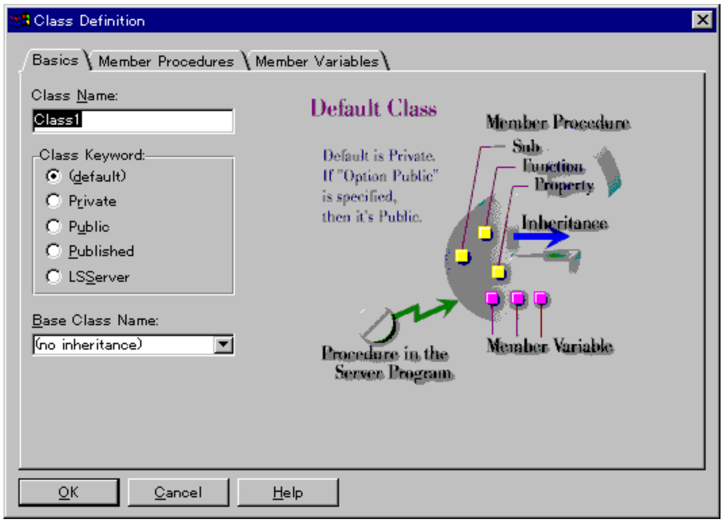


script editor panel

Figure 11. ESB IDE

2. Create a class pattern using the ESB IDE Class Wizard. Select **Create -> Interface/ Class** or press the F3 key to start up the Class Wizard.
3. Enter the required information. Click **OK**.
4. Install the respective member procedures for the Published class.

   Figure 12 on page 46 shows a view of the class definition tool. This tool is located under the Basics tab.



Class Name: Class1

Class Keyword:
● (default)
○ Private
○ Public
○ Published
○ LSServer

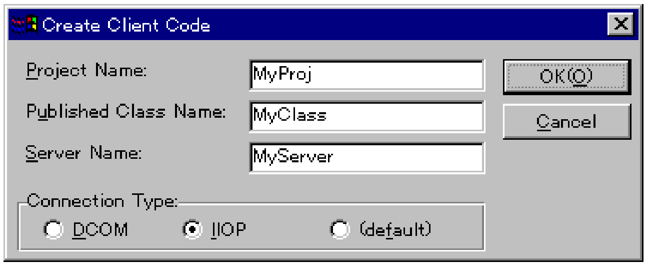
Base Class Name: (no inheritance)

Figure 12. Class Definition Tool

5. Create a client program to test the created Published class.

> **Hint**
>
> If you start another ESB IDE at this point and use it as a simple version client program, you can quickly test and debug the runtime. Select **Create -> Create Client Code**, and a template of client code is created.

Figure 13 shows the client code creation tool, which can create a simple version client program.



Project Name: MyProj

Published Class Name: MyClass

Server Name: MyServer

Connection Type:
○ DCOM ● IIOP ○ (default)

Figure 13. Client Code Creation Tool

6. Test your server's application program by using IDE debugger. You can run your code line-by-line by using the step function of the IDE debugger.

Or, you can force a branch by changind the values of the variables for the conditional branch.

7. Create and test the ESB client program.

8. Distribute the completed projects by packaging them in the ESB IDE, and operate it using the system manager.

   Figure 14 shows a view of the ESB System Manager. In this figure, the CFConv project was just loaded.

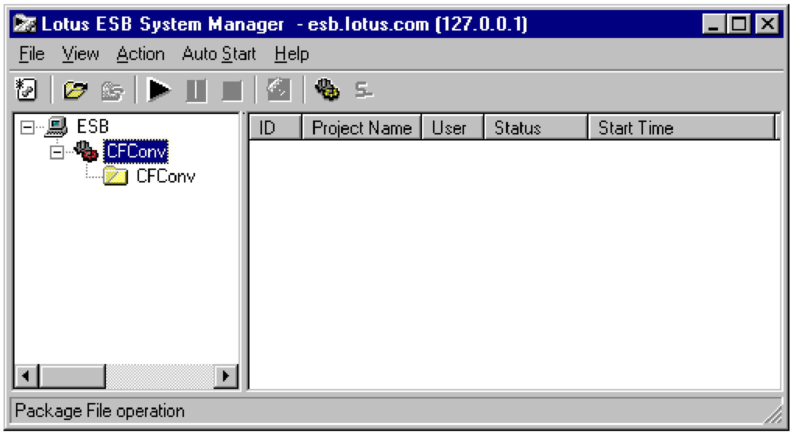


Figure 14. ESB System Manager

Figure 15 on page 48 shows the flow of the ESB Runtime development and the configuration of the files that are created.

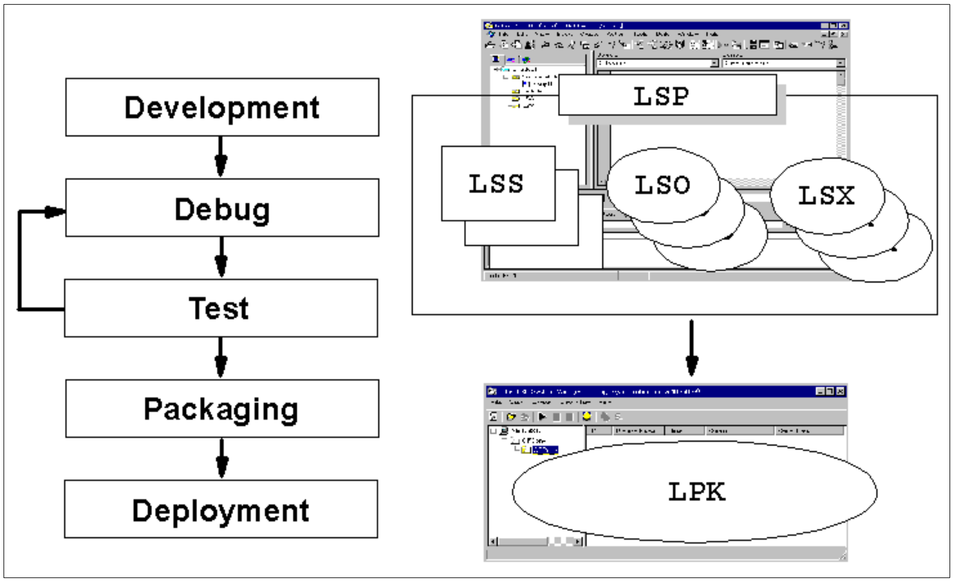


Figure 15. File extension naming and the ESB program development procedure

4.2 Published class

The *Published class* is a special class that is newly implemented in ESB. In ESB applications, server and client programs send and receive the data using the Published class and the member procedure defined in the program.

The Published class is defined in the Declarations script of Globals object, which is the same as usual user-defined classes. See the following example:

```
Public Published Class MyFirstESB
    Sub SayHelloToWorld()
       Print "Hello World!"
    End Sub
End Class
```

Hint

The scope of the Published class is always public. Because the LotusScript files created in ESB IDE are declared as standard to be Option Public in (Globals) - (Options), it can be described as an abbreviation as follows:

```
Published Class MyFirstESB
```

4.2.1 Example of the Published class

In this section, we create an ESB Runtime that defines the Published class. The Published class Average calculates the average value and the total value, which was entered from the client. We explain the corresponding client program and how to run the program.

4.2.1.1 Creating a server side program

Complete these steps to create a server side program:

1. Start the ESB IDE.
2. Select **Globals -> Declarations**. Enter the following code in the script editor:

```
'   Average.lss
Published Class Average
    '--- member variables
    total As Integer
    counter As Integer

    Sub New
        total = 0
        counter = 0
    End Sub

    Sub AddNumber(i As Integer)
        total = total + i
        counter = counter + 1
    End Sub

    Function GetTotal() As Integer
        GetTotal = total
    End Function

    Function GetAverage() As Integer
        GetAverage = total / counter
    End Function
End Class
```

> **Hint**
>
> You can quickly create a template of class by using the IDE Class Wizard. Select **Create -> Interface/ Class** or press the F3 key to start the Class Wizard.

3. Save the LotusScript files as `Average.lss` and `Math.lsp`.

Note

Here, we affix the same file name as the Published class name to the LotusScript file. However, this is not mandatory. You are free to assign a name that is easy to remember. You can also assign the project file name of your choice. *The project file name will be used when the client specifies a class name*.

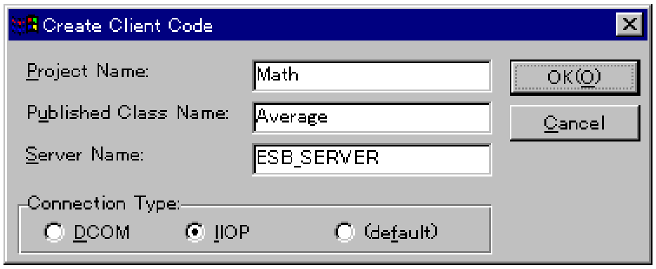
In this example, if the project is saved with a name other than Math.lsp, you should replace the project name of the client program to be created in following section with the specified project name.

4.2.1.2 Creating client programs

This process starts another ESB IDE and creates a client program for confirmation for a simple test, the runtime that was created in the previous section. We can use the Client Code Creation tool to easily create a confirmation program. Follow these steps:

1. Start another new ESB IDE.
2. Select **Create Client Code**.

   Figure 16 shows a view of the Client Code Creation tool.



Project Name: Math

Published Class Name: Average

Server Name: ESB_SERVER

Connection Type: ○ DCOM ● IIOP ○ (default)

Figure 16. Client Code Creation tool

3. Enter `Math` in the Project Name box, `Average` in the Published Class box, and the TCP/IP host name of the ESB server in the Server Name box.

Note

If the project file of the runtime was saved with a name other than *Math.lsp*, you should specify the project file name that was saved in the Project Name box.

4. Click **OK**.
5. Add the call for the member procedure:

```
'   MathClient.lss
Sub Initialize
    Dim ORSObj As New SsClink
    Dim obj As Variant

    On Error Goto ErrorHandler
    On Event RuntimeError From ORSObj Call EventHandler

    ORSObj.ConnType = "IIOP"
    Set obj = ORSObj.CreateObject("Math.Average, node=ESB_SERVER")
    Print "Object is successfully created."

    '
    '   Now you can call Published class methods here.
    '   eg)
    '       Dim ret As Integer
    '       ret = obj.DoSomething(arg1, arg2)
    '
    obj.AddNumber(100)    '<<== ADD
    obj.AddNumber(50)    '<<== ADD
    Print "Total : " & obj.GetTotal()    '<<== ADD
    Print "Average : " & obj.GetAverage()     '<<== ADD

    Set obj = Nothing
    Print "Object is successfully deleted."

    Exit Sub

ErrorHandler:
    Print "ErrorHandler: " & Cstr(Err)
    Print "ErrorHandler: " & Error
    Exit Sub
End Sub
```

4.2.1.3 Running the sample program

Complete these steps to run the sample program:

1. Select **Build -> Run Project** in the IDE where you created the runtime to run the runtime.
2. Select **Build -> Run Project** in the IDE where you created the client program to run the client program.

If it is run correctly, the following sample code is displayed in the IDE output panel on the client side:

```
'MathClient': Compiling Project Files...
'MathClient.lss': compiled successfully.
All of the project files compiled successfully.
Object is successfully created.
Total : 150
Average : 75
Object is successfully deleted.
```

4.2.2 Object initialization and deletion

Each Published class has the defined special functions *New* and *Delete*. They are called automatically at the creation and deletion time. This section explains the New and Delete functions.

4.2.2.1 New Member procedure and Delete Member procedure

The *New Member procedure* is a special procedure that runs automatically at the creation of Published class object in the New Member procedure. ESB executes the required pre-processing and variable initialization within the Published class object. The member variables of the Published class object initialized in this execution can be referenced only while the Published class object exists.

The *Delete Member procedure* is a special procedure that runs automatically at the deletion of a Published class object. In the Delete Member procedure, ESB executes the required post-processing for the termination of the Published class object.

In the following example, a log is written into the file each time an object is created and deleted:

```
'   NewAndDelete.lss
Published Class NewAndDelete
    Sub New()
        Dim fileNumber As Integer
        fileNumber = Freefile
        Open "NEW_DEL.LOG" For Append As fileNumber
        Write #fileNumber, "New is called."
        Close fileNumber
    End Sub

    Sub Delete()
        Dim fileNumber As Integer
        fileNumber = Freefile
        Open "NEW_DEL.LOG" For Append As fileNumber
        Write #fileNumber, "Delete is called."
        Close fileNumber
    End Sub
End Class
```

Hint

Upon creating the template of a class by using Class Wizard, the New and Delete procedures are defined automatically.

4.2.3 Transmitting data from ESB Runtime

There are two ways to transfer the data from ESB Runtime to an ESB client program:

- Use the **return value** of the Published class member procedure.
- Use the **ByRef argument** of the Published class member procedure.

We previously saw an example using the return value in the initial example. Define the Published class member procedure as a function, and substitute the return value for the function name:

```
'   GetBack.lss
Published Class GetBack
    Function ReturnServerData() As String
        ReturnServerData = "This is Server data"
    End Function
End Class
```

To return data to the client using a ByRef argument, specify the **ByRef** argument to the argument of the Published class member procedure. Assign a value to this variable within the member procedure.

Note

If nothing is specified for the definition of the member procedure, the argument becomes a *ByRef* argument. The following two-member procedures have the same definition.

```
Sub GetServerData(data As String)
Sub GetServerData(ByRef data As String)
```

Here is an example of defining the member procedure by using a *ByRef* argument:

```
'   GetBack.lss
Published Class GetBack
    Function GetServerData(data As String) As Integer
        data = "This is Server data"
    End Function
End Class
```

The following example is part of a sample client program calling *ReturnServerData* and *GetServerData*:

```
'   GetBackClient.lss
Dim data As String
Dim rc As Integer
data = obj.ReturnServerData()
Print data                  '"This is Server data" is displayed.

data = "This is Client data"
Print data                  '"This is Client data" is displayed.
rc = obj.GetServerData(data)
Print data                  '"This is Server data" is displayed.
```

4.2.4 Transferring array data

The transfer of array data is performed differently by the category of array. This section describes the process.

4.2.4.1 Sending and receiving fixed-length array data

Define the Published class member procedure as shown in the following example to transfer fixed-length array data:

```
'   ArrayServer.lss
Function FixedArray(arraydata() As String) As Integer
    Dim i As Integer
    '--- print client sent data
    For i = LBound(arraydata) to UBound(arraydata)
        Print "client data #" & CStr(i) & " : " & arraydata(i)
    Next
    '--- assign new data
    For i = LBound(arraydata) to UBound(arraydata)
        arraydata(i) = "new data #" & CStr(i)
    Next
End Function
```

The client program can specify array type variables for the member procedure argument. In this example, the elements of the array variables are replaced by new data by the server:

```
'   ArrayClient.lss
Dim data(2) As String
data(0) = "0th data"
data(1) = "1st data"
data(2) = "2nd data"
Print data(0)              '"0th data" is displayed.
rc = obj.FixedArray(data)
Print data(0)              '"new data #0" is displayed.
```

4.2.4.2 Sending and receiving variable-length array data

When sending and receiving variable length array data, particularly when the number of array elements returned from the server are unknown, transfer the data using the VARIANT type variable. In the following example, `REDIM` is used to redefine the array:

```
'   ArrayServer.lss
Function VariableArray(arraydata As Variant) As Integer
    Redim arraydata(100) As String
    arraydata(100) = "new data #100"
End Function
```

The client receives data using the VARIANT type variable. It can also send data through the VARIANT type variable:

```
'   ArrayClient.lss
Dim data(2) As String
data(0) = "0th data"
data(1) = "1st data"
data(2) = "2nd data"
Dim arrayVariant As Variant
arrayVariant = data
Print arrayVariant(0)          '"0th data" is displayed.
rc = obj.VariableArray(arrayVariant)
Print arrayVariant(0)          '"" is displayed.
Print arrayVariant(100)        '"new data #100" is displayed.
```

Note

If `REDIM` is used to redefine an array, the data sent from the client is lost. In the previous example, for the VARIANT type variable `arrayVariant` following the `VariableArray` call, the value `new data #100` is set only for the arrayVariant(100). All the values other than that become zero-length character strings ("").

You can also specify a VARIANT type variable for the *return value* to send or receive variable length array data:

```
'   ArrayServer.lss
Function ReturnArray() As Variant
    Dim i As Integer
    '--- assign return data
    Dim returnData(100) As Integer
    For i = 0 to 100
        returnData(i) = i * 10
    Next
    ReturnArray = returnData
End Function
```

The client receives the variable length array of the return value by the VARIANT type variable.

```
'   ArrayClient.lss
Dim variantReturn As Variant
variantReturn = obj.ReturnArray()
Print variantReturn(2)       '"20" is displayed.
```

4.2.5 Transferring user-defined data

You *cannot specify* direct user-defined data for the argument of the Published class member procedure. When sending or receiving user defined data, you either send and receive each member as an individual argument, or send and receive by using the array of the VARIANT type variable.

4.2.5.1 Using the member procedure argument

When there are a few user-defined members, specify each member as the argument of the Published class member procedure:

```
'   UserDefine.lss
Type Employee
    ID As Integer
    lastName as String
    firstName As String
End Type

Function QueryEmployee(ID As Integer, _
                        lastName as String, _
                        firstName as String) As Integer
    '--- construct Employee data
    Dim oneData As Employee
    oneData.ID = ID
    oneData.lastName = lastName
    oneData.firstName = firstName

    '--- do something with Employee data.
    oneData.lastName = "Wilkinson"
    oneData.firstName = "May"

    '--- back Employee data to each arguments
    ID = oneData.ID
    firstName = oneData.firstName
    lastName = oneData.lastName
End Function
```

The client program specifies each of the user-defined members in the member procedure:

```
'   UserDefineClient.lss
Dim myData As Employee
myData.ID = 1000
rc = obj.QueryEmployee(myData.ID, myData.lastName, myData.firstName)
```

4.2.5.2 Using a VARIANT type array

Because the number of arguments in member procedure is limited, you cannot use a member procedure argument to send or receive user-defined data, including a large number of members. In such cases, you should use a VARIANT type array.

Hint

A maximum of *30* Published class member procedure arguments is allowed.

In the following example, the respective elements of the QueryEmployee2 argument correspond to the respective elements of the user-defined Employee:

```
'   UserDefine.lss
Function QueryEmployee2(varEmployee() As Variant)
    '--- construct Employee data
    Dim onedata As Employee
    onedata.ID = varEmployee(0)
    onedata.lastName = varEmployee(1)
    onedata.firstName = varEmployee(2)

    '--- do something with Employee data.
    oneData.lastName = "Robinson"
    oneData.firstName = "Bill"

    '--- back Employee data to each element
    varEmployee(0) = onedata.ID
    varEmployee(1) = onedata.lastName
    varEmployee(2) = onedata.firstName
End Function
```

The client program uses the VARIANT type array to send user-defined data to the runtime:

```
'   UserDefineClient.lss
Dim myData As Employee
Dim varEmployee(3) As Variant
varEmployee(0) = myData.ID
varEmployee(1) = myData.lastName
varEmployee(2) = myData.firstName
rc = obj.QueryEmployee2(varEmployee)
myData.ID = varEmployee(0)
myData.lastName = varEmployee(1)
myData.firstName = varEmployee(2)
```

Hint

You can code concise, relatively error-free source code by defining the conversion from a user-defined variable to a VARIANT type array variable. Or you can define the conversion in reverse, which would be to convert from a VARIANT type array variable to a user-defined variable as a subroutine (or function).

4.3 Sharing data and resources between Published class objects

Use the *LSServer* class when sharing data and resources between Published class objects. Use the *SsSharedStorage* class when sharing data between different projects.

4.3.1 ESB threads

An ESB Runtime can simultaneously process in parallel the request from multiple clients using the multi-thread function. ESB uses the following two threads:

- Client threads
- Global threads

Client threads are created for each Published class object generated by a request from a client. Member variables and global variables declared with the Published class have their own characteristic values for each of these client threads. You cannot directly reference a member variable in another thread from a given thread.

Global threads are special threads that are present from the start-up to the termination of the ESB Runtime. These are created not only for running initialize and terminate procedures, but are also created for each LSServer class object, which is described later.

Next, we show the relationship between client threads and global threads. In Figure 17, the Published class objects for an ESB Runtime from two clients are created. Note that the global variables with a characteristic value reside in each thread.

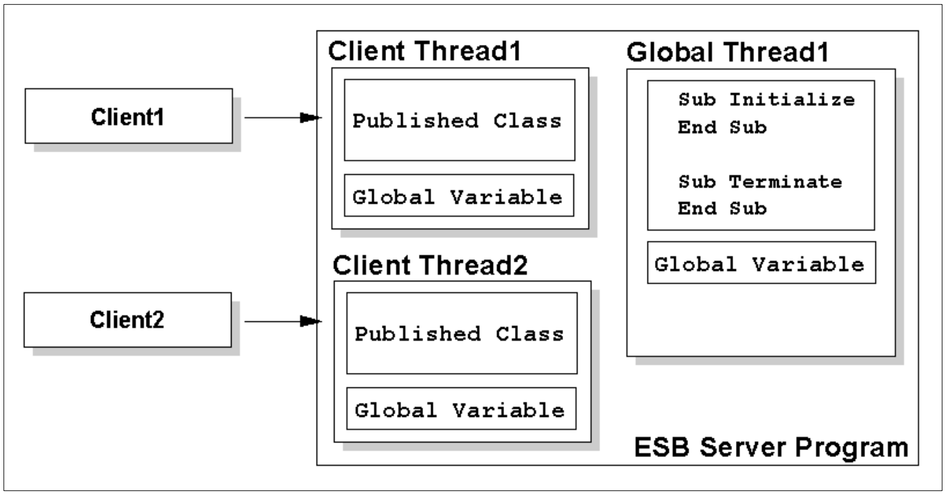


Figure 17. Client threads and global threads

4.3.2 Global variables and Published classes

Consider the relationship between the global variables and the threads in the following sample program:

```
'   ThreadTest.lss
Dim gCounter As Integer
Published Class ThreadTest
    Sub New()
        Print "New(): gCounter = " & gCounter
    End Sub

    Sub Delete()
        Print "Delete(): gCounter = " & gCounter
    End Sub
End Class

Sub Initialize
    gCounter = 100
    Print "Initialize: gCounter = " & gCounter
End Sub

Sub Terminate
    Print "Terminate: gCounter = " & gCounter
End Sub
```

The global variable gCounter is set to 100 by the Initialize procedure. ThreadTest objects are created and deleted, and then a value is output by the Terminate procedure. At first glance, it appears that 100 is displayed for every PRINT statement. However, in actual fact, it happens as shown here:

```
Initialize: gCounter = 100
New: gCounter = 0
Delete: gCounter = 0
Terminate: gCounter = 100
```

Note

In this example, 0 is displayed in the New procedure and the Delete procedure. However, this value is indefinite. A different value is displayed depending on the runtime environment.

This is because it has a different value of global variable in each client thread and global thread. Consequently, global variables cannot be used when sharing data between Published class objects.

Hint

Global variables can be used when sharing data extending across the member procedures of a Published class object.

The following list summarizes global variables:

- Global variables cannot share data between Published class objects.
- The initial value of the global variable is indefinite. It is not always 0.
- Initialize with the New procedure to use a global variable in a Published class object. Even if a value has been initialized with the Initialize procedure, it will not be valid for a Published class object.

4.3.3 LSServer class

The LSServer class is used to transfer data between Published class objects. The LSServer class is a special class provided in ESB that can only be created in one instance on a system. Data can be transferred between Published class objects by accessing data through the LSServer class. You can also use the LSServer class for safe accessing when using such resources as external files.

The LSServer class is executed by the declaration scripts of global objects, which is the same as Published classes and normal user-defined classes.

```
Public LSServer Class MyFirstLSServerClass
```

Hint

The scope of the LSServer class is normally public. However, because Option Public is declared as standard, the description of the LotusScript file for creating ESB IDE can be omitted:

```
LSServer Class MyFirstLSServerClass
```

4.3.3.1 LSServer class and global threads

Figure 18 on page 62 shows the relationship among the Published class, the LSServer class, client threads, and global threads in an ESB Runtime. A client thread is created for every Published class. Here, Published classes A and B each have three threads, for a total of six client threads. The LSServer classes X and Y are run on their respective distinct global threads. Only one LSServer class object is created. Since there are global threads for the Initialize procedure and the Terminate procedure, in addition to the global thread for the LSServer class, there is a total of three global threads.

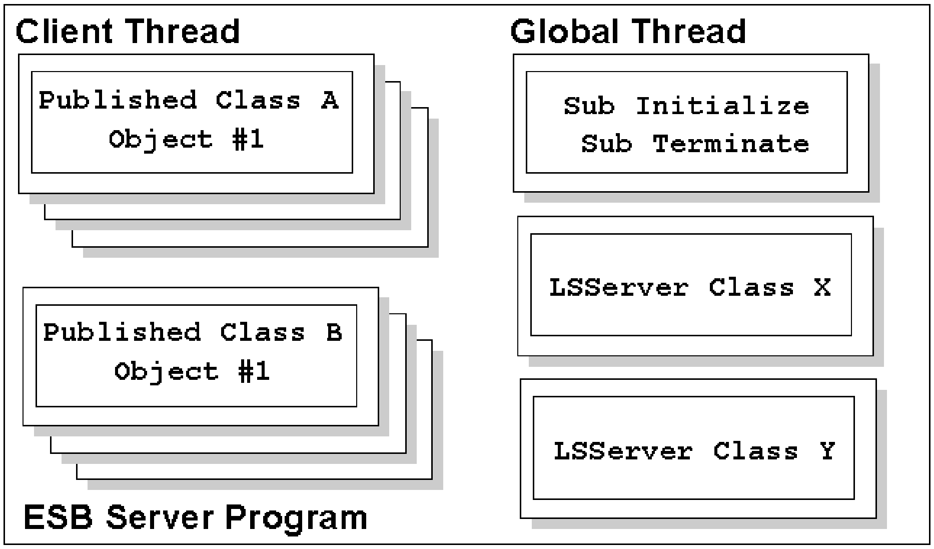


Figure 18. Global threads and LSServer class

When the ESB Runtime begins to run, it is initialized based on the following sequence of events:

1. An LSServer class object is created by a global thread, and a new procedure is called. When multiple LSServer classes are defined, the LSServer class objects are created in the sequence in which they were loaded into the IDE and in the sequence coded in the module.
2. The Initialize procedure is called by a global thread. When multiple script files are defined for a project, the call sequence is in the order in which they were loaded into the IDE.

When the ESB Runtime is stopped, the termination process is done in the following sequence of steps:

1. The Terminate procedure is called by the global thread. When multiple script files are defined for a project, they are called in the reverse order in which they were loaded into the IDE.
2. The Delete procedures for the LSServer class objects are called by the respective global threads. When multiple LSServer classes are defined for a project, they are called in the reverse order in which they were created.

4.3.3.2 Example where an LSServer class has been used

The following sample program writes the user name that created the Published class object to the log file. The Published class LSServerTest obtains the client information using the GetContext function within the New member procedure and writes the user name into the file using the LSServer class UserLog. The LSServer class UserLog possesses an opened file handle (gFileNumber) of the external file USER.LOG as a member variable and a counter (gCounter) that holds the number of objects that were created. When the Log member procedure is called, it writes them to files.

```
'   LSServerTest.lss
Lsserver Class UserLog
    gCounter As Integer
    gFileNumber As Integer

    Sub New()
        gCounter = 0
        gFileNumber = Freefile
        Open "USER.LOG" For Append As gFileNumber
    End Sub

    Sub Delete()
        Close gFileNumber
    End Sub

    Sub Log(userName As String)
        gCounter = gCounter + 1
        Write #gFileNumber, gCounter, userName
    End Sub

End Class

Published Class LSServerTest
    Sub New()
        '--- get client user name
        Dim userName As String
        Dim context As Variant
        Set context = GetContext()
        userName = context.userID

        '--- write log through LSServer class
        Dim objLog As UserLog
        Set objLog = Bind("")
        objLog.Log(userName)
    End Sub
End Class
```

Perfor this series of steps:

1. Create a new project and describe the above code for **(Globals) - (Declaration)**. Save the project name as `LSServerTest.lsp`.
2. Start another ESB IDE, and create the client program for operation confirmation.

3. Select **Create -> Client Code**. Enter the `LSServerTest.lsp` for the project name and class name, and the TCP/IP host name of the ESB server for the server name. Click **OK**. The following code is automatically created:

```
'   LSServerTestClient.lss
Sub Initialize
    Dim ORSObj As New SsClink
    Dim obj As Variant

    On Error Goto ErrorHandler
    On Event RuntimeError From ORSObj Call EventHandler

    ORSObj.ConnType = "IIOP"
    ORSObj.Userid = Inputbox$("Type your name")
    Set obj = ORSObj.CreateObject("LSServerTest.LSServerTest, node=ESB_SERVER")

    Print "Object is successfully created."
    Set obj = Nothing
    Print "Object is successfully deleted."

    Exit Sub

ErrorHandler:
    Print "ErrorHandler: " & Cstr(Err)
    Print "ErrorHandler: " & Error
    Exit Sub
End Sub
```

4. After repeating the starting and stopping of the client program, stop the runtime and refer to the USER.LOG file.

Note

The USER.LOG cannot be accessed from another program until the runtime has stopped. Temporarily stop the runtime when confirming it.

4.3.4 Global threads and serialization

The LSServer class is executed in a *global thread*. When the ESB Runtime begins the execution, ESB creates global threads equal to the number of LSServer classes and creates one LSServer class object on each global thread.

The *member procedure* of LSServer class can be called from another thread. However, only one thread can be used at a time. When a member procedure of LSServer class is called from a certain thread, a member procedure of LSServer class is called from another thread. This member procedure call automatically goes into the standby state until the initial member procedure call ends. The exclusive control of this Server class is not controlled by the basis of a method. It is controlled by the LSServer class basis. That means, if the first thread is calling the member procedure fnShare1, even if another thread calls other than the member procedure fnShare1, it enters a standby state. As shown in Figure 19, Client Thread1 called the member function fnShare1 of LSServer. Client Thread2 is entered into the wait state, where it calls another member function, fnShare2, of LSServer.

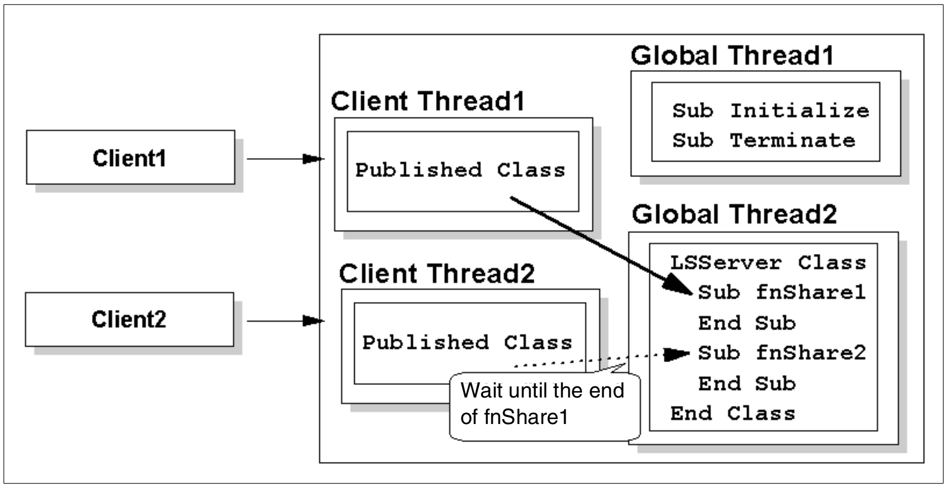


Figure 19. Global threads and serialization

4.3.5 SsSharedStorage class

The Published class can transfer data with the LSServer class of same project, but cannot transfer data with the LSServer class of another project. A *SsSharedStorage class* can be used to share the data beyond the project. The SsSharedStorage class holds String type data with the combination of key and value in shared memory. The held data can be referenced from any project. Figure 20 on page 66 shows that multiple projects access the data of SsSharedStorage class.

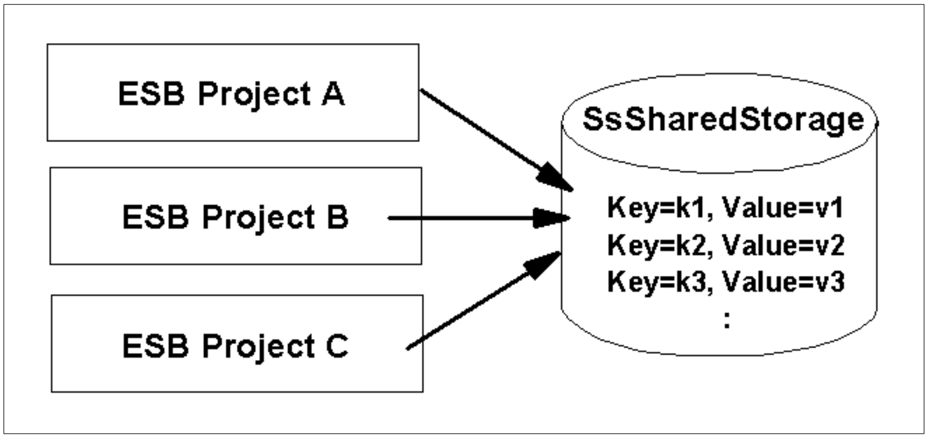


Figure 20. SsSharedStorage class

4.3.5.1 Example of SsSharedStorage class usage

The following sample program transfers data between multiple projects. Start two ESB IDEs, and enter the following codes for the respective (Globals)-(Declarations). Save the LotusScript file with the name `SharedClass.lss`, and save the respective projects with the names `SharedProject1.lsp` and `SharedProject2.lsp`.

```
'   SharedClass.lss
Published Class SharedStorageClass
    shrStorage As SsSharedStorage

    Sub New()
        Set shrStorage = New SsSharedStorage(1000)
    End Sub

    Sub Delete()
        Set shrStorage = Nothing
    End Sub

    Sub WriteToSharedStorage(data As String)
        Call shrStorage.setValue("TheKEY", data)
    End Sub

    Function ReadFromSharedStorage() As String
        ReadFromSharedStorage = shrStorage.getValue("TheKEY")
    End Function
End Class
```

Furthermore, start another ESB IDE. Then, create a client program for the operation confirmation. Enter the following code for the subroutine. Be sure to enter the TCP/IP host name for the node=parameter of the CreateObject member function (there are two places).

```
'   SharedClient.lss
Sub Initialize
```

```
    Dim ORSObj As New SsClink
    Dim obj1 As Variant
    Dim obj2 As Variant

    ORSObj.ConnType = "IIOP"
  Set obj1 = ORSObj.CreateObject("SharedProject1.SharedStorageClass, node=ESB_SERVER")
'<== Change
  Set obj2 = ORSObj.CreateObject("SharedProject2.SharedStorageClass, node= ESB_SERVER")
'<== Change

    Call obj1.WriteToSharedStorage("Data wrote by obj1")
    Print obj2.ReadFromSharedStorage()

    Call obj2.WriteToSharedStorage("Data wrote by obj2")
    Print obj1.ReadFromSharedStorage()

    Set obj1 = Nothing
    Set obj2 = Nothing
End Sub
```

The value is written in the SharedProject1 project. Read it in SharedProject2. SharedProject1 reads the value written in SharedProject2.

4.4 Error handling

When an error occurs while LotusScript is executing, if the error handling routine is not coded, the program execution is interrupted and suspended:

```
'   ErrorHandle.lss
Published Class ErrorHandle
    Function Divide1(x As Integer, y As Integer) As Integer
        Dim answer As Integer
        answer = x / y
        Divide1 = answer
    End Function
End Class
```

In this example, when 0 is specified for the second argument `y` of the member procedure `Divide1`, a runtime error occurs that interrupts the processing. The following message is displayed on the output panel:

```
 [0010:000002] Division by zero
(Domain:XX.XX.XX.XX,Project:ErrorHandle,Module:ERRORHANDLE,Class:ERRORHAND
LE,Method:DIVIDE1,Line:13)
```

The ESB Runtime uses an ON ERROR statement and a RESUME statement to embed the error handling routine.

4.4.1 ON ERROR statement and RESUME statement

ESB uses the ON ERROR statement and RESUME statement the same way that LotusScript for Notes and Domino uses them for error handling.

4.4.1.1 Error handling using the ON ERROR GOTO label

Use the ON ERROR GOTO label to specify the error handling routine where jump to, at an error, occurred. In the following example, when a `0` is substituted to the variable `y`, a `0` division error occurs. The execution jumps to the error handling routine ErrorExit. It outputs an error message and terminates.

```
'   ErrorHandle.lss
Function Divide2(x As Integer, y As Integer) As Integer
    Dim answer As Integer
    On Error Goto ErrorExit
    answer = x / y
    Divide2 = answer
    Exit Function
ErrorExit:
    Print "Line Number:" & Erl
    Print "Error Number:" & Err
    Print "Error Description:" & Error
End Function
```

Hint

In case there is no Exit Function statement on the sixth line, it runs up to the Print statement for the error handling routine, even if it operates normally.

4.4.1.2 Error handling using RESUME

You can use the RESUME statement to specify the location of execution to be resumed after error handling is done. Altering the example in the previous section appears as shown in the following code. In case `0` is specified for the variable `y`, assign `1` to it temporarily. Then, continue the calculation.

```
'   ErrorHandle.lss
Function Divide3(x As Integer, y As Integer) As Integer
    Dim answer As Integer
    On Error Goto ErrorExit
Calculate:
    answer = x / y
    Divide3 = answer
    Exit Function
ErrorExit:
    y = 1
    Resume Calculate
End Function
```

4.4.1.3 Error handling using ON ERROR RESUME NEXT

Use ON ERROR RESUME NEXT to ignore an error and to execute the next statement. In the following example, when a 0 division error occurs, the initial value `1` of the variable `answer` is returned:

```
'   ErrorHandle.lss
Function Divide4(x As Integer, y As Integer) As Integer
    Dim answer As Integer
    On Error Resume Next
    answer = -1
    answer = x / y
    Divide4 = answer
End Function
```

4.4.2 Runtime error handling when client program ended abnormally

When the Published class object is deleted, the Delete member procedure is called. In a situation where the client program ends abnormally, it rolls back the data prior to committing it in the Delete member procedure of the Published class object and closes the opened resources.

4.5 Security

This section explains the security used in ESB and the configuration. It also covers the creation and the usage of its own security logic.

4.5.1 Authentication and access control

The ESB security function is performed based on *authentication* and *access control*. In *authentication*, when a Published class object is created, ESB confirms the identity of the client that requested its creation. A user ID and password that has been clearly or implicitly set by the client is used to confirm the identity. The following options are available for ESB authentication:

- Anonymous authentication (no authentication)
- Operating system level authentication
- Authentication using LDAP
- Authentication using a user defined Exit Routine

The authentication of ESB is done in the Exit Routine. The Exit Routine is included in the ESB installer, although you can also use it by creating, registering, or setting an Exit Routine. Use the *ESB Configuration Tool* for Windows NT and *SMIT* for AIX to register or setup an Exit Routine.

Setting in Windows NT

To set the exit routine in ESB Runtime of Windows NT, the ESB configuration Tool is started with following procedure:

1. Start the ESB Configuration Tool.
2. Click the **Exit Routine** tab.

Setting in AIX

To set the exit routine in ESB Runtime under an AIX system, the ESB Configuration Tool is started by this procedure:

1. Start **SMIT**.
2. Select **Applications -> ESB -> System Configuration -> Change/Show the Client Authentication**.

Select the Exit Routine to be used for the authentication in the opened dialog box or menu, and change the parameters settings.

With *access control*, ESB controls who can access a Published class object and which Published class object can be accessed. The following options are available for ESB access control:

- Programmable access control using LotusScript
- Declarative access control using project environment variables
- Access control using the ACL of the Notes database

These authentications and access controls can be used in combination.

Figure 21 shows the category of security that is usable in ESB, the valid combinations, and the sequence in which access control and authentification are checked.

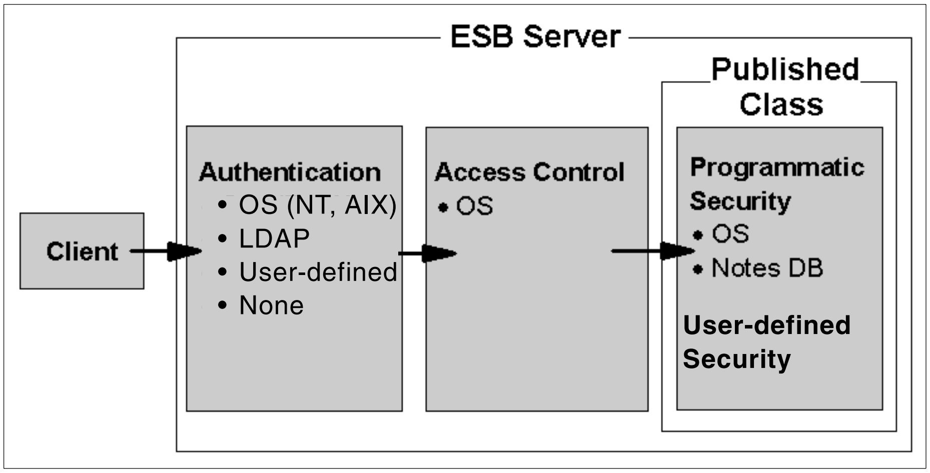


Figure 21. ESB security components and the flow

Note

For Web client, ESB performs basic authentication on a Web server in addition to the process previously described. It can perform access control using the authenticated user ID. See Chapter 6, “Using WebSphere” on page 129, for the details.

4.5.2 Authentication when using DCOM

If DCOM is used as a communication protocol between the client and ESB server, for the authentication when a Published class object is created, the DCOM level authentication is executed first. Then, ESB is authenticated. The user ID and password are used for DCOM level authentication. They are also used at the network logon of Windows.

Note

The values set in the UserID property of SSClink class and SVClink class are not used in DCOM authentication.

The DCOM level authentication is done prior to ESB authentication. When authentication of the DCOM level fails, an error is returned to the client without ESB authentication taking place. To omit authentication at the DCOM level, validate the *Guest account* of the computer on which the ESB Runtime is running.

On the DCOM level, authentication is successfully completed. ESB level authentication takes place. At this point, you can select a user ID from the following options to be used in the ESB level authentication:

- The values set for the UserID property of SSClink class and SVClink class
- The user ID used in the DCOM level authentication
- The domain name and user ID used in the DCOM level authentication (for example, Domain or user ID)

To change the user ID, start the ESB Configuration Tool. Click the **Exit Routine** tab.

4.5.3 Authentication when IIOP is used

When IIOP is used as the communication protocol between the client and ESB server, ESB-level authentication is only performed at Published class object creation. ESB-level authentication uses the UserID property and password of SSClink class and SVClink class.

4.5.4 Anonymous authentication

The setting procedure to omit ESB level authentication is shown in this section.

> **Note**
>
> When IIOP is used, any user can create a Published class object with this procedure. However, if DCOM is used, the DCOM level authentication must be disabled in addition to this procedure. Void the Guest account to disable DCOM level authentication.

For Windows NT, add only Anonymous Security in the Exit Routine box of Configuration Tool.

For AIX, specify the full path of `libhpwdsec.a` in Exit Routine #1 of SMIT, and specify **none** for the other Exit Routine.

4.5.5 OS authentication

OS level authentication is performed to authenticate the client by the user ID and password combination registered in the operating system.

For Windows NT, select **OS security** on the **Exit Routine** tab of Configuration Tool.

For AIX, specify the fullpath of `libhpwssec.a` in Exit Routine of SMIT.

4.5.6 Authentication using LDAP

You can also use an external LDAP directory for authentication. The following settings can be made to perform LDAP authentication.

For Windows NT, add **LDAP Security** in the **Exit Routine** tab of Configuration Tool. Click the **Properties** button and specify the name of the LDAP directory server.

For AIX, specify the fullpath of `libhpwssec.a` in Exit Routine of SMIT. Specify the name of the LDAP directory server for the LDAP Server Name.

Note

To authentificate using LDAP on ESB of AIX, IBM eNetwork LDAP Directory must be installed. The IBM eNetwork LDAP Directory is included in IBM AIX Install CDs.

The client specifies the LDAP user name (Distinguished Name) and password in the UserID property of SsClink class and SvClink class:

```
'   LDAPClient.lss
Sub Initialize()
    Dim ORSObj As New SsClink
    Dim obj As Variant
    ORSObj.UserID = "cn=Sandra Smith, o=Acme"
    ORSObj.Password = "secret"
    Set obj = ORSObj.CreateObject("MyProj.MyClass, node=MySvr")
    '--- do something ---
    Set obj = Nothing
End Sub
```

4.5.7 Authentication using a user-defined exit routine

All authentications introduced up to this point were done through the exit routine provided by ESB. However, you can create your own exit routine, and then register it and perform authentication. Moreover, you can use your Exit Routine to access an authentication system outside ESB and to perform user authentication.

4.5.7.1 Creating a user-defined exit routine

In this section, you create the exit routine *MySec*, which permits access when the client user ID and password are the same. Complete these steps:

1. Enter the following code using a text editor and save it as the file name `mysec.cpp`:

```
//  mysec.cpp
#include <string.h>
struct secinfo {
    long version;
    char *userid;       // userID
    char *password;     // password
    char *winname;
    char *ipaddr;
    char *projectname;
    char *classname;
};
extern "C"
{
    long security_exit(void *secinfo);
}
long security_exit(void *pInfo)
{
    struct secinfo *info = (struct secinfo *)pInfo;
    if ((info == NULL) || (info->userid == NULL) || (info->password == NULL))
    {
        return -1;
    }
    if (strcmp(info->userid, info->password) == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
```

2. For Windows, prepare the following module definition file and save it with the file name `mysec.def`:

```
;   userlog.def (Windows only)
LIBRARY     mysec
EXPORTS     security_exit
```

3. Compile and link the routine as shown here:

   For Windows NT (Microsoft Visual C++)

   ```
   cl /c mysec.cpp
   link /dll mysec.obj /def:mysec.def
   ```

   For AIX (IBM VisualAge C++ for AIX)

   ```
   xlc_r -c mysec.cpp
   makeC++SharedLib_r -bmodtype:SRE -bnoentry -p \
                       -100 -o libmysec.a mysec.o
   ```

4.5.7.2 Registering and testing a created exit routine

Set the created Exit routine into the ESB Runtime registration, and test it as described here:

1. Register the created exit routine as shown here:

   For Windows NT, click **Add** on the **Exit Routine** tab of Configuration Tool. Then, click **Others** and specify the created exit routine `mysec.dll`.

   For AIX, specify the created exit routine `libmysec.a` in the Exit Routine of SMIT with the full path.

2. Create a Server Application Program and save the project name as `UserExit.lsp`:

   ```
   '   UserExit.lss
   Published Class UserExit
       Sub New
           Print "UserExit object is created."
       End Sub
   End Class
   ```

3. Create a client program. Specify the TCP/IP host name of the ESB server for node=parameter of the CreateObject member function:

   ```
   '   UserExitClient.lss
   Sub Initialize()
       Dim ORSObj As New SsClink
       Dim obj1 As Variant

       '--- success case. UserID is equivalent to Password ---
       ORSObj.UserID = "Bob"
       ORSObj.Password = "Bob"
       Set obj = ORSObj.CreateObject("UserExit.UserExit, node=ESBServer")
       Set obj = Nothing

       '--- error case. UserID is not equivalent to Password ---
       ORSObj.UserID = "Alice"
   ```

```
    ORSObj.Password = "Ecila"
    Set obj = ORSObj.CreateObject("UserExit.UserExit, node=ESBServer")
    Set obj = Nothing
End Sub
```

Authentication was successful for the creation of the first object because the same value, *Bob*, was set for both. However, the authentication failed for the creation of the second object because the UserID property and password property are different.

4.5.8 Access control

The access control includes *programmable access control* and *declarative access control*.

Programmable access control is a method that dynamically controls the access by external information, such as client information and project environment variables. In ESB, you can perform this method by using the client information obtained from the *method* and property of SsContext class taken out through the GetContext function.

Declarative access control involves registering a list of users in advance that can be accessed and users whose access is prohibited for each project and class basis. ESB declarative access control includes access control using project environment variables and access control using the Notes database ACL.

4.5.9 Programmable access control using LotusScript

You can limit a call to member procedures, by means of the user name and host name that created the Published class object and the information registered in the project environment function. For example, you can limit the updates of the corporate human resource database, so that only certain determined users can do it.

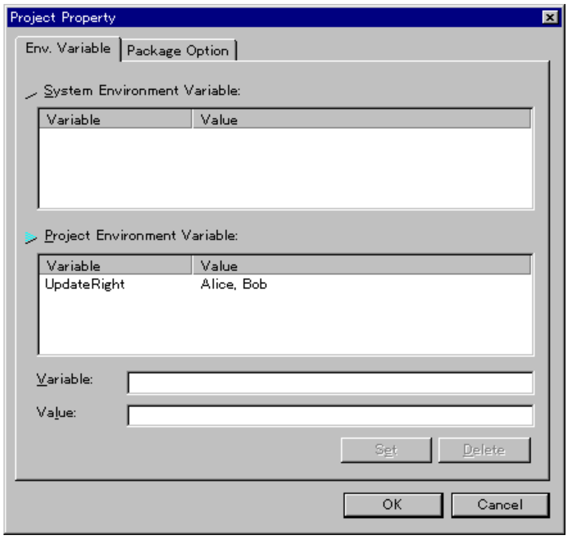
Note

To use this function, the users who start the Server Application Program must belong to the Administrators group.

4.5.9.1 Creating a programmable access control

This section shows an example of access control performed by using the values set in the project environment function, and the member functions of the SsContext class. Follow this process:

1. Start ESB IDE. Select **File -> New Project**.
2. Select **File -> Project Property**. Figure 22 shows a view of the project property.



Variable	Value

Variable	Value
UpdateRight	Alice, Bob

Figure 22. Project Property

3. Click the **Project Environment Variable** list box and move the blue arrow in front of the **Label Project Environment Variable**.
4. Enter `UpdateRight` in the Variable text box. Enter `Alice, Bob` in the Value text box. Then, click **Set**.
5. Click **OK**. Then, close the dialog box.
6. Select **Globals -> Declarations**. Enter the following code in the script panel and then save it as project name `CustomAccessControl.lsp`:

```
'   CustomAccessControl.,_,ì,ì
Published Class CustomAccessControl
    Function CanIUpdate() As Integer
        '--- get client user name ---
        Dim userName As String
        Dim context As Variant
        Set context = GetContext()
        userName = context.UserID

        '--- check whether the client has an update right
        Dim rc As Integer
        rc = context.IsAllowed("UpdateRight")
        If rc = 1 Then
            Print userName & " : OK (registered user)"
        Else
            '--- check whether the client is belongs to "system" group
            Dim groupNames As Variant
```

```
                groupNames = context.getGroups()
                If Datatype(groupNames) = 0 Then
                    Print userName & " : NG (no group)"
                Else
                    Dim i As Integer
                    For i = Lbound(groupNames,1) To Ubound(groupNames,1)
                        Print groupNames(i)
                        If groupNames(i) = "system" Then
                            Print userName & " : OK (system member)"
                            rc = 1
                            Exit For
                        End If
                    Next
                    If rc <> 1 Then
                        Print userName & " : NG (is not belongs to system)"
                    End If
                End If
            End If
            CanIUpdate = rc
        End Function
    End Class
```

7. Start another ESB IDE. Then, create a client program for the operation confirmation. Enter the following code:

```
'   CustomAccessControlClient.lss
Sub Initialize
    Dim ORSObj As New SsClink
    Dim obj As Variant
    Dim rc As Integer

    ORSObj.UserID = Inputbox("Type your user name.")
    Set obj = ORSObj.CreateObject("CustomAccessControl.CustomAccessControl,
node=ESB_SERVER")    '<<== Change
    rc = obj.CanIUpdate()
    Set obj = Nothing
End Sub
```

8. Set **Anonymous Authentication** using the ESB Configuration tool or SMIT, so that another authentication mechanism does not work.

9. Run the server program on ESB Runtime. Then, run the client program. Select **Alice**, **Bob** or a user that belongs to the System group at the input dialog box of client. The value of *CanIUpdate* shows “1” as a returned value.

4.5.10 Declarative access control using project environment variables

You can control the access by using project environment variables to request Published class object creation from the client.

When there is a project environment variable named *$classname_ACCESS$*, ESB uses this value as an access control list when creating objects. The user can enter a permissible user ID or the group name delimited by commas (,).

Note

The user who starts the Server Application Program must belong to the Administrators group to use this function.

4.5.10.1 Access control with project environment variables

This section shows an example of authentication by creating Published class object using project environment variables. Follow this process:

1. Start ESB IDE. Select **File -> New Project**.
2. Select **File -> Project Properties**.
3. Enter `$ClassAccessControl_ACCESS$` in the Variable text box. Enter `Alice,Bob` in the Value text box. Then click **Set**.
4. Click **OK**.
5. Select **Globals -> Declarations**. Enter the following code in the script panel, and save it as project name `ClassAccessControl.lsp`:

```
'   ClassAccessControl.lss
Published Class ClassAccessControl
    Sub New()
        Dim context As Variant
        Set context = GetContext()
        Print context.UserID & " was authenticated."
    End Sub
End Class
```

6. Start another ESB IDE. Then, create a client program for operation confirmation. Enter the following code:

```
'   ClassAccessControlClient.lss
Sub Initialize
    Dim ORSObj As New SsClink
    Dim obj As Variant
    ORSObj.UserID = Inputbox("Type user name.")
    Set obj =
ORSObj.CreateObject("ClassAccessControl.ClassAccessControl,
node=ESB_SERVER")     <<== Change
    Set obj = Nothing
End Sub
```

7. Set **Anonymous Authentication** using the ESB Configuration tool or SMIT, so that another authentication mechanism does not work.
8. Run the Server Application Program first. Then, run the client program. If you select **Alice** or **Bob** at the input dialog box of client, an object is

created normally. If another name is entered in the input dialog box, the authentification error is returned.

4.5.11 Declarative access control using Notes database ACL

You can use the Notes database ACL to perform access control. Query the ACL of a specified Notes database with the user ID taken out from the GetContext function.

4.5.11.1 Declarative access control using the Notes database ACL

This section shows an example of access control using the Notes database ACL. In this example, only a user can call a method, who is set as an *Administrator* in the database access authority list.

Now we assume that Notes is installed on the computer where the ESB Runtime is running and ESB is in an accessible state. Follow this procedure:

1. Right-click the Notes database for access control on the Notes work space. Select **Database -> Access Control**.
2. Click **Add**. Enter `Alice`, and then click **OK**.
3. Select **Alice** from the list box, and select **Manager** in the Access list box.
4. Likewise, add **Bob** as a **Reader**.
5. Select **Globals -> Declarations**. Enter the following code in the script panel and save it as project name `NotesACL.lsp`. Replace the values of the function `dbServer` and `dbFileName` in the source code with an appropriate Notes server name (local Notes DB by specifying "") and the Notes database file name that set the access authority:

```
'   NotesACL.lss
Const ACLLEVEL_NOACCESS  = 0
Const ACLLEVEL_DEPOSITOR = 1
Const ACLLEVEL_READER    = 2
Const ACLLEVEL_AUTHOR    = 3
Const ACLLEVEL_EDITOR    = 4
Const ACLLEVEL_DESIGNER  = 5
Const ACLLEVEL_MANAGER   = 6

Published Class NotesAccessControl
    Function CanIUpdate() As Integer
        Dim rc As Integer
        Dim dbServer As String
        Dim dbFileName As String

        '--- setup Notes Database used fro AccessControl
        dbServer = ""               '"" means local server
        dbFileName = "c:\lotus\notes\data\MyACLDB.nsf"
        rc = SetACLDBInfo(dbServer, dbFileName)
        If rc <> 0 Then
            Print "Cannot find DB Server or DF File"
            Exit Function
        End If

        '--- check your ACL ---
        rc = IsAllowed()
        If rc <> ACLLEVEL_MANAGER Then
            Print "You must have a manager right for this operation"
            Exit Function
        End If
        '--- do something
    End Function
End Class
```

6. Start another ESB IDE. Then, create a client program for operation confirmation. Enter the following code:

```
'   NotesACLClient.lss
Sub Initialize
    Dim ORSObj As New SsClink
    Dim obj As Variant
    Dim rc As Integer

    ORSObj.UserID = Inputbox("Type user name.")
    Set obj = ORSObj.CreateObject("NotesACL.ClassAccessControl, node=ESBServer")
    rc = obj.CanIUpdate()
    Set obj = Nothing
End Sub
```

7. Set Anonymous Authentication using the ESB Configuration Tool or SMIT, so that another authentication mechanism does not work.
8. Run the ESB Server Application Program first. Then, run the client program. If you enter `Alice`, who has manager authorization, at the input dialog box of client, it is called correctly. However, if you enter `Bob`, which only has reader authorization, an error is displayed.

4.6 Designing an ESB application

This section explains the ESB Server Application program design point to be considered.

4.6.1 Programming model for ESB applications

The following two programming models are available for ESB applications depending on whether the state on the server exists. Each model has their respective advantages and disadvantages:

- State-full programming model
- State-less programming model

State refers to the application information held on the server. Client IDs and connection handles for databases, for example, are included in the state.

The *state-full programming model* refers to a programming model that has a state on the server. Since the essential information is held on the server in advance, when you call a method for a Published class object from a client, it can transfer just the information required for the execution of the member procedure, thus reducing network traffic. Also, you can adopt a natural programming style for client program, because it specifies only the specific argument to the member procedure. However, because it must constantly be in existence while the object is providing the service, a large amount of memory and resources on the server are used to maintain the state.

In the *state-less model*, the application information *is not saved* on the server. It creates and deletes an object each time it calls a member procedure for a Published class object, and passes all the required information as a member procedure argument. Since the object only exists while a certain member procedure is running, it uses only a small amount of memory and resources on the server. However, the network traffic increases, because it must provide all the necessary information every time a member procedure is called. The processing is also complicated due to the fact that the client program creates and deletes an object before and after each member procedure call.

4.6.2 Synchronizing Notes user and ESB user authentication

If the ESB client application is a Notes application and DCOM is used for the connection type, at least two authentications must be confirmed for the ESB client user:

- The authentication performed on a Notes client and a Domino server
- The DCOM authentication performed on an ESB client and an ESB server

For this reason, a problem of redundant user management arises, namely *Notes and Domino* and the *ESB server and the client*.

Generally, the ESB application creates a Published class object using LotusScript from the Notes form. A Notes form must be opened to run LotusScript. The fact that a user can access a form means that it has confirmed Notes authentication. Consequently, the Guest account of ESB server is set to active, and then the authentification of DCOM is released. A user who has received Notes authentication can automatically create a Published class object. In short, the Notes authentication and the ESB authentication can be synchronized.

Note

Be careful not to disclose the form code for the client user. If the code is disclosed to the public, since the DCOM authentication did not work, any user can create a Published class object. However, in such a case, the possibility of the object being accessed directly still remains. In this case, use it in combination with other authentications and access controls.

4.6.3 Designing a Published class in a distributed environment

This section explains the design considerations of a specific distribution environment.

4.6.3.1 Considering the number of network transmissions

In contrast to the fact that a normal class member procedure coded in LotusScript is called within the same process space, network traffic is generated each time a member procedure for an ESB server application is called from an ESB client. When creating Published classes, you should design them to reduce the number of transmissions between client and server.

For example, consider the following two Published classes, *Rectangle1* and *Rectangle2*, which show quadrangles:

```
Published Class Rectangle1
    width As Integer
    height As Integer
    Sub SetWidth(w As Integer)
        width = w
    End Sub
    Sub SetHeight(h As Integer)
        height = h
    End Sub
    Function Area() As Integer
        Area = width * height
    End Function
    Function Is
End Class

Published Class Rectangle2
    Function Area(width As Integer, height As Integer) As Integer
        Area = width * height
    End Function
End Class
```

To determine the size of area of *Rectangle1*, a client must call the method on three occasions:

```
Call obj.setWidth(w)
Call obj.setHeight(h)
area = obj.Area()
```

On the other hand, in the case of *Rectangle2,* it determines the size of the area at one calling:

```
area = obj.Area(w,h)
```

The *Rectangle1* alternative can flexibly respond to data changes, but the *Rectangle2* alternative is superior from the perspective of execution speed.

4.6.3.2 Considering the network traffic volume

Be careful not to send meaningless network data back and forth. Consider the following client code for example. This is a program that obtains 100 sets of data from the server. From the second loop on, it sends the data obtained in the preceding loop back to the server.

```
Dim szData As String
Dim i As Integer
For i = 1 to 100
    Call obj.GetDataFromServer(szData)
    Print szData
Next
```

You can reduce the network traffic volume by clearing unnecessary data before calling a member procedure:

```
Dim szData As String
Dim i As Integer
For i = 1 to 100
    szData = ""
    Call obj.GetDataFromServer(szData)
    Print szData
Next
```

4.6.4 ESB project design related hints

The following sections offers hints to make an efficient ESB Server Application Program.

4.6.4.1 Divide large script files into multiple files

Projects can be composed of multiple script files. Dividing script files appropriately based on function facilitates source code management and maintenance.

4.6.4.2 Use IDE to manage the run sequence for multiple script files

There are multiple script files for a certain project. Compiling and running sequences of the script files become important. When running a project on ESB, first the LSO file, then the include file, and finally the script file are loaded in the order in which they appear in the IDE file viewer. The procedure defined in the previously loaded file can be referenced in files loaded subsequently. The reverse process results in an error. Consequently, you should move script files and similar files containing common routines to the front of the script file in the file viewer. To change the order sequence in the IDE file viewer, drag the file name to be changed to the desired position.

4.6.4.3 Manage constants in another script file

When using numbers, you can flexibly respond to the changes of project by consolidating their management in another file. When referencing constants from a program, you can reference them by adding a file that stores the constants in the Include folder of the ESB IDE or by using an INCLUDE statement in the respective script file.

4.6.4.4 Use a few changed script files as object files

Programs with relatively few changes that are frequently used as common routines, utility functions and so on can reduce compiling time when you use them. First, they are compiled and converted into an object file form. Then, they are added to the LSO folder in the ESB IDE. However, you must be careful that the LSO file is loaded prior to any script file loading and that the LSO is loaded in the sequence displayed in the ESB IDE file viewer.

4.6.4.5 Do not use multiple Published classes at one time from a single client

The ESB server manages clients by Published class object basis and assigns their respective characteristic thread. For example, when it creates two Published class objects from a single client, two threads are used. Consequently, you should design the number of Published classes, which are simultaneously created from one client, so that the ESB server resources are efficiently utilized. If you want to use multiple classes at the same time, define them as Public classes and refer them from one Published class.

4.6.4.6 Adopt a reasonable number of projects to run at one time

We recommend that you do not divide projects into small units and run multiple projects (100 or more for example) simultaneously. You should choose a reasonable number of projects to run.

4.6.4.7 Avoid creating dependencies between projects

To run multiple projects on ESB, avoid creating dependency between the respective projects to be run. Dependency occurs, for example, when you create a Published class object in another project (B) from a certain project (A). If there is no dependency, you can start and stop each project. However, in this example, when you stop project B, you must also stop project A. If no dependency exists, a certain project on ESB can be stopped and updated.

4.6.4.8 Set the parameter to be updated during operation as the project environment variable

When operating a project, the project is packaged in a packaged file. The created packaged file is run on the system manager. You may want to change the project parameters, for example, for a project for accessing the data source layer. The User ID of the account user and the password for data source access may sometimes be changed at the time of operation.

In case these parameters are coded in a program file, you must modify the source code and redo the packaging when you change the parameters. To avoid the redundant operations, define the parameters in the project environment variable. The parameters are intended to change at the time of operation and refer the project environment variable from the program file. Since project environment variables cannot only be changed from the ESB IDE, but from the system manager as well, you can change project parameters without redoing the packaging at the time of operation.

4.6.4.9 Minimize PRINT statement usage

Frequent use of PRINT statements to display messages in a program affects the running speed of the server. Minimize the usage of PRINT statements during operation.

4.7 Other topics

This section summarizes and explains other items related to server application programming.

4.7.1 Timer

ESB provides an *SsTimer* class that generates events at a definite period of time. You can use the SsTimer class to periodically poll the external resources from the ESB application and process the scripts every so often at a certain period of time.

The following example displays the character string `I am alive!` three times per second and subsequently disables the event of the SsTimer class object:

```
'   Timer.lss
Dim gTimerObj As Variant
Dim gCounter As Integer

Sub Initialize
    gCounter = 0
    Set gTimerObj = New SsTimer(1)
    gTimerObj.Enabled = True
    On Event Alarm From gTimerObj Call timerHandler
End Sub

Sub timerHandler(obj As SsTimer)
    If gCounter < 3 Then
        gCounter = gCounter + 1
        Print "I am alive!"
    Else
        obj.Enabled = False
    End If
End Sub
```

4.7.2 Calling the DLL function

To call the function defined in the external dynamic link library (DLL) from the LotusScript running on Windows, use a DECLARE statement to define the function. Then, call it the same as a normal function.

4.7.2.1 Example of calling a simple function

The following example calls the *GetVersion* function of Windows. The GetVersion function returns the Windows version as the return value.

```
'   GetVersion.lss
Declare Function GetVersion Lib "Kernel32" () As Integer
Sub Initialize
    Dim lVersion As Long
    lVersion = GetVersion()
    Print "GetVersion() = " & CStr(lVersion)
End Sub
```

4.7.2.2 Example of calling a complex function

The following example calls the *GetVersionExA* function of Windows. The GetVersionEx function returns the information of Windows version in the member of the structure *OSVERSIONINFO*:

```
Type OSVERSIONINFO
    dwOSVersionInfoSize As Long
    dwMajorVersion As Long
    dwMinorVersion As Long
    dwBuildNumber As Long
    dwPlatformId As Long
    szCSDVersion As String*128
End Type

Declare Function GetVersionExA Lib "Kernel32" (version As OSVERSIONINFO) As Long

Sub Initialize
```

```
    Dim version As OSVERSIONINFO
    Dim bReturn As Long
    version.dwOSVersionInfoSize = 148
    bReturn = GetVersionExA(version)
    Print "GetVersionEx() = " & version.dwMajorVersion & "." & version.dwMinorVersion & "
"& version.szCSDVersion
End Sub
```

4.7.3 Obtaining access logs using the exit routine

The ESB exit routine can be used as a function for logging the access of the Published class objects aside from the use of authentication. The following exit routine example writes, into the log, the client user ID, IP address, project name, and class name that had to be created each time an object is created. See 4.5.7.1, “Creating a user-defined exit routine” on page 74, for more information about compiling, links, and usage.

Note

Do not use this exit routine in the actual operation.

```
//  userlog.cpp
#include <stdio.h>

struct secinfo {
    long version;
    char *userid;     // userID
    char *password;
    char *winname;
    char *ipaddr;     // client IP address
    char *projectname;     // ESB project name
    char *classname;     // Published class name
};

extern "C"
{
    long security_exit(void *secinfo);
}

long security_exit(void *pInfo)
{
    FILE *fp;
    fp = fopen("access.log", "a+");

    struct secinfo *info = (struct secinfo *)pInfo;

    if (secinfo)
    {
        fprintf(fp, "User:%s at %s creates %s.%s\n",
                    info->userid, info->ipaddr,
                    info->projectname, info->classname);
    }
    else
    {
        fprintf(fp, "secinfo is null.\n");
    }

    fclose(fp);
    return 0;
}
```

The following file is required when compiling on Windows:

```
;   userlog.def (Windows only)
LIBRARY     userlog
EXPORTS     security_exit
```

Chapter 5. Client application programming

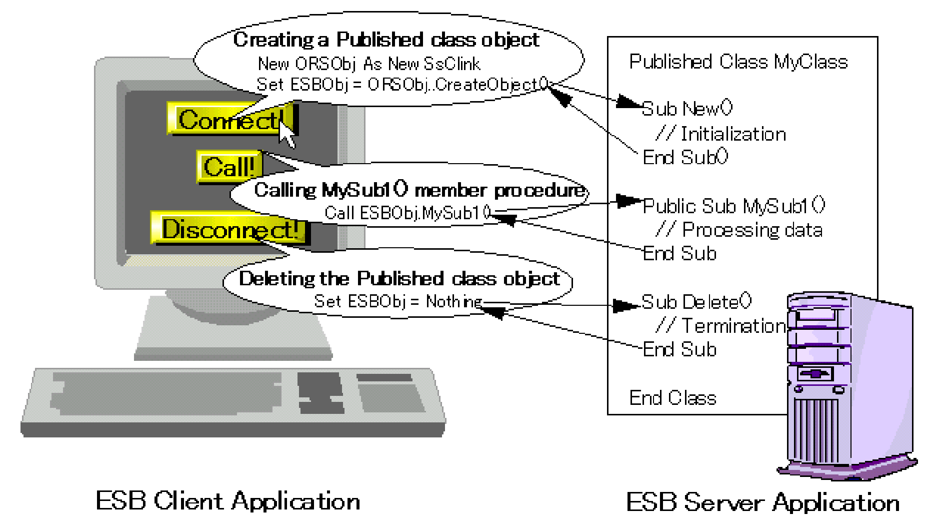
ESB supports a wide variety of connectivity with backend systems and also supports a diverse range of front end systems (clients). This chapter uses the example of a typical client to code the procedure for creating an ESB client program.

5.1 Overview

The highlight of ESB server programing was defining the Published class, which is described in Chapter 4, “Server application programming” on page 43. By defining the Published class, ESB can reference the member procedure of a class from a remote client.

In contrast to server programming, the highlight of ESB client programming is the creation of a Published class object. Creating a Published class object establishes the connection with the ESB Runtime. It means the same as preparing to use a server program. Once a Published class object is created, you can call the member procedure in the server program with a dot notation, same as other object oriented languages.

Figure 23 shows the relationship between the server program and the client program. It also shows three buttons that are created in the client program. Each button is respectively described and coded to the related functions.



Creating a Published class object
Connect!
Call!
Calling MySub1() member procedure
Disconnect!
Deleting the Published class object
ESB Client Application
ESB Server Application

```
New ORSObj As New SsClink
Set ESBObj = ORSObj.CreateObject()
```

```
Call ESBObj.MySub1()
```

```
Set ESBObj = Nothing
```

```
Published Class MyClass

Sub New()
  // Initialization
End Sub()

Public Sub MySub1()
  // Processing data
End Sub

Sub Delete()
  // Termination
End Sub

End Class
```

Figure 23. Relationship between the server program and client program

The three buttons are further explained here:

- **Creating a Published class object**

  The code creating Published class object is described in the *Connect!* button within Figure 23 on page 91. When you click the Connect! button, it establishes a connection with ESB Runtime. Then, *New()* member procedure of the Published class is called and the object is initialized.

- **Calling a member procedure**

  The code that calls MySub1 of the member procedure is described in the *Call!* button. If you click the Call! button, MySub1 is processed on the server and the result is returned to the client application.

- **Deleting a Published class object**

  The code that deletes the Published class object is described in the *Disconnect!* button. If you click the Disconnect! button, the Delete() subroutine is called on the server side, the object is managed, and the connection with ESB Runtime is disconnected.

5.1.1 Clients supported by ESB

ESB client programming refers to the creating of a Published class object published on the server and the calling of its member procedure. In effect, the ability to create a Published class object is a prerequisite for becoming an ESB client. Published class objects can be created with any of the following languages:

- LotusScript
- JavaScript or VBScript
- OLE automation support language (for example, Visual Basic)

Any environment that can execute these languages can be an ESB client. Web browsers that cannot create Published class objects by themselves can become ESB clients through the JSP programming of the ESB HTTP communication function.

Note

It is possible to become an ESB client if the conditions discussed above are met. However, if you use a client not formally supported by ESB, you should carefully test it in advance.

In this chapter, we use the following typical applications to explain ESB client programming:

- Lotus Notes applications (language used: LotusScript)
- Web applications (language used: JavaScript)
- Microsoft Visual Basic applications (language used: Visual Basic (VB))

5.1.2 Client application creation flow

To make the client application the same as all the clients, there are six stages you need to follow:

1. Create a client form.

   You create a form for receiving user inputs and displaying the results from ESB Runtime.

2. Create a Published class object.

   To use an ESB server program, you first create a Published class object. In a normal application, you create a Published class object when the form you created in step 1 is displayed. However, there are also circumstances where it is alright to create and delete the Published class objects for every procedure invocation, such as applications that leave a form displayed for an extended period of time.

3. Call the procedure.

   After you obtain the variable to be input and verify its compatibility (data type and range), set the argument and call the procedure.

4. Display the result.

   Receive and display the result of the called procedure from the return value and the argument of the referenced transfer.

5. Error handling.

   Describe the handling for when a connection has failed or the calling of a procedure was unsuccessful.

6. Delete a Published class object.

   Lastly, delete the Published class object and disconnect the connection with ESB Runtime. When you create a Published class object along with a form display, delete it when you close the form.

This chapter explains client applications along the lines of these six stages.

5.1.3 The server program to be used

A server program is required to create and run client programs. In this chapter, we use the following server programs in all the clients:

```
Published Class Conversion

Sub New()
  Print "Conversion Class: New()"
End Sub

Sub Delete()
  Print "Conversion Class: Delete()"
End Sub

Public Function ToDoubleValue(myLong As Long, myString As String, _
  myDoubles() As Double) As Long
  Dim i%, LB%, UB%
  On Error Goto ErrorHandler
  Print "Conversion Class: ToDouble()"

  myLong = myLong * 2
  myString = myString & myString
  LB% = Lbound(myDoubles)
  UB% = Ubound(myDoubles)
  For i = LB% To UB%
    myDoubles(i) = myDoubles(i) * 2
  Next
  ToDoubleValue = 0
  Exit Function
ErrorHandler:
  Print "Error" & Str(Err) & ": " & Error$
  ToDoubleValue = -1
  Exit Function
End Function

End Class
```

This project has a Published class Conversion, and the Conversion class has the member function ToDoubleValue. This member function ToDoubleValue doubles and returns all the Long variable, String variable, and Double array values entered by the user. The return value 0 implies that a normal termination has occurred and -1 that a Runtime error (for example, overflow) has occurred.

Our main objective in this chapter is to create a Published class Conversion object and to call the member function ToDoubleValue. Before creating a client program, save the above code as project name `chap05` (file name: `chap05.lsp`), and run it on the machine where you installed ESB Runtime.

5.2 Notes application

In this section, we use a Lotus Notes example to explain how to create a client program using LotusScript. The software required for the client machine is:

- Lotus Notes R4.5 or higher
- ESB Client Enabler

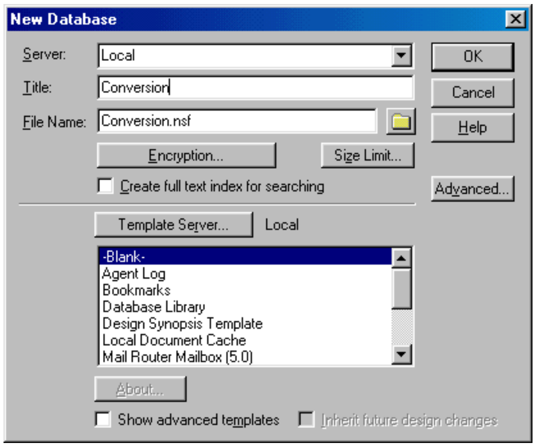
> **Note**
>
> When using Lotus Notes R5, you must use Domino Designer to create applications.

5.2.1 Creating a client form

In this section, we use Lotus Notes R5 for our explanation. The basic operation is the same in Lotus Notes R4. Complete these steps:

1. Start Domino Designer.
2. Select **File -> Database -> New**.

   The New Database dialog box is displayed (Figure 24).



New Database

Server: Local

Title: Conversion

File Name: Conversion.nsf

□ Create full text index for searching

Template Server... Local

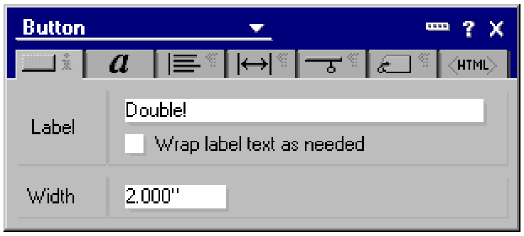
-Blank-
Agent Log
Bookmarks
Database Library
Design Synopsis Template
Local Document Cache
Mail Router Mailbox (5.0)

□ Show advanced templates □ Inherit future design changes

Figure 24. New Database dialog box

3. Enter an appropriate name (for example, Conversion) in the Database Name field. Then, click the **OK** button. Leave the template blank.
4. Select **Create -> Design -> Form**.

5. Select **Create -> Hotspot -> Button** and create a button to call the procedure. Enter `Double!` in the label field in the Information box as shown in Figure 25.

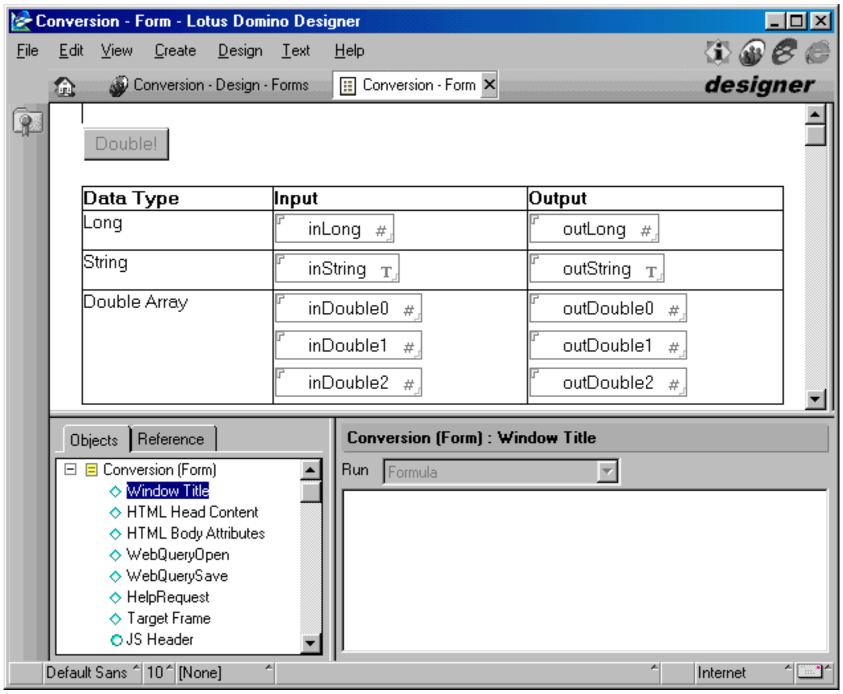


Label: Double!
□ Wrap label text as needed
Width: 2.000"

Figure 25. Information box of a button

6. Create a table for laying out texts and fields. Select **Create -> Table**. Then, create a table with four lines and three columns.

7. Create text and fields within the table and lay them out as shown in Figure 26.



Data Type	Input	Output
Long	inLong #	outLong #
String	inString T	outString T
Double Array	inDouble0 # inDouble1 # inDouble2 #	outDouble0 # outDouble1 # outDouble2 #

Conversion (Form)
- Window Title
- HTML Head Content
- HTML Body Attributes
- WebQueryOpen
- WebQuerySave
- HelpRequest
- Target Frame
- JS Header

Conversion (Form) : Window Title
Run: Formula

Figure 26. Layout of the client application using Notes

The fields to be created are shown in Table 11.

Table 11. Details of the created fields

Field	Type	Description
inLong	Number, editable	Long type user entry field.
outLong	Number, editable	Long type result display field.
inString	Text, editable	String type user entry field.
outString	Text, editable	String type result display field.
inDouble*n*	Number, editable	Double type array user entry field. *n* is an integer starting from 0. The form shows up to two, but you can create any number you like.
outDouble*n*	Number, editable	Double type array result display field. Create the same number as the user entry field.

8. Select **File -> Save** to save the form. When it asks for a name for the form, enter `Conversion`.

5.2.2 Creating a Published class object

It is useful to create Published class objects when you open a form. Use the PostOpen event for form objects to accomplish this in the Notes Form.

You must prepare for creating a Published class, before entering the script in the PostOpen event. First, open the form. Then, write the following line in the (Options) script of the (Globals) object:

```
Uselsx "*SsClink"        ' Load ESB client LSX
```

Use the SsClink class provided by ESB to create a Published class object. The Uselsx statement means that it will load the LSX and define the SsClink class. Next, define two variables as global variables. Describe the following two lines in the (Declarations) script of the (Globals) object:

```
Dim ORSObj As SsClink    ' SsClink class object
Dim ESBObj As Variant    ' Published class object
```

ORSObj is the variable that holds the SsClink object. ESBObj is the one that holds the Published class object. Define ESBObj as a global variable, so that it can be referenced from all the objects within the form. ORSObj is an object that is necessary when creating Published class objects. However, it has the important function of generating a RuntimeError event and providing error information, when a runtime error has occurred while a Published class object is in use.

This completes the preparations. Let us describe the script to create the Published class object. First, create an SsClink class object. Then, use the CreateObject member function of the SsClink class to create a Published class object:

```
Set ORSObj = New SsClink
Set ESBObj = ORSObj.CreateObject("chap05.Conversion, node=myServer")
```

The host name (myServer), where ESB Runtime are operating, the project name (chap05), and class name (Conversion) are set. Change the host name as appropriate in conformity with the actual environment.

Hint

When you want to change the ORB to be used for connection or set the user ID and password to be used for authentication, specify it before calling the CreateObject member function. For example, when changing ORB to DCOM, use:

```
ORBObj.ConnType = "DCOM"
```

5.2.3 Calling a procedure

Let us try calling a procedure using the Published class object that you created. Enter a script for the Click event of the Double! button object, so that the procedure is called when you click the Double! button on the form.

First, obtain the value entered in the entry field. Then, substitute it into the variable.

```
myLong& = Clng(uidoc.FieldGetText("inLong"))
myString$ = uidoc.FieldGetText("inString")
For i% = 0 To 2
  myDoubles#(i%) = Cdbl(uidoc.FieldGetText("inDouble" & Cstr(i%)))
Next
```

Use the FieldGetText function of the NotesUIDocument class to obtain the field value. Define the class object, uidoc, of the NotesUIDocment class as a global variable. Obtain the instance in the PostOpen event of the form object. Since the return value of the FieldGetText function is a string type, you should expressly perform type conversion when necessary.

Finally, set the argument to call the procedure:

```
rc = ESBObj.ToDoubleValue(myLong&, myString$, myDoubles# )
```

5.2.4 Displaying the result

The return value of the ToDoubleValue function is an error code. If it is not 0, some error has occurred on the server side, which requires you to interrupt the process without displaying the result.

```
If rc <> 0 Then
  Msgbox "Some error has occurred on ESB Server."
  Exit Sub
End IF
```

All of the arguments of the ToDoubleValue function are pass-by references. In other words, when the function call finishes, the result is stored in the variable specified in the argument. Use these variables to display the results in the field, as shown here:

```
Call uidoc.FieldSetText("outLong", Cstr(myLong&))
Call uidoc.FieldSetText("outString",myString$)
For i% = 0 To 2
  Call uidoc.FieldSetText("outDouble" & Cstr(i%), _
    Cstr(myDoubles#(i%)))
Next
```

Use the FieldSetText subroutine of the NotesUIDocument class to set the value for a field. Since the second argument of the FieldSetText subroutine is a string type, you should use the Cstr function as required to convert the type.

5.2.5 Error handling

The SsClink class generates a RuntimeError event when a runtime error occurs in a Published class object. You can easily describe the error handling by catching this event.

Use the On Event statement to catch the RuntimeError event. Add the following line to the PostOpen event of the form object:

```
On Event RuntimeError From ORSObj call RunErrHandler
```

This code signifies that if a RuntimeError event occurs, it will call the RunErrHandler subroutine. The subroutine called here is referred to as an *event handling subroutine*. Although the subroutine name and argument name are optional, the number and type of the arguments are determined in advance, as shown here:

```
Sub RunErrHandler (obj As SsClink, errCode As Long, errMsg As String)
```

Let us actually define an event handling subroutine. Enter the following code in the (Declarations) script of the (Globals) object to permit referencing from any object on the form:

```
Sub RunErrHandler(obj As SsClink, errCode As Long, errMsg As String)
  MsgBox "Error!! rc = " & CStr(errCode) & " : " & errMsg
End Sub
```

This event handling subroutine is called when a RuntimeError event is generated in ORSObj and the characteristic ESB error code and error message are displayed.

Handling code is also necessary when a runtime error has occurred in other than a Published class object. Use the On Error statement for the runtime error handling. Add the following code to the beginning of the Click event of the Double! button:

```
On Error Goto ErrorHandler
```

This code signifies that it will move the control to the error handling routine ErrorHandler, when a Runtime error has occurred within a procedure. Now, enter the content of the error handling routine at the end of the Click event of the Double! button:

```
ErrorHandler:
  MsgBox "Error!! rc = " & CStr(Err) & " : " & Error
  Exit Sub
```

The error handling routine ErrorHandler displays the error code and the error message in the message box and terminates the subroutine.

5.2.6 Deleting objects

It is appropriate to delete the Published class object created when you display the form and when the form is closed. Use the QueryClose event of the form object to accomplish this with Notes Form. Use the Nothing constant to delete the object. You should delete the Published class object and the SsClink class object at the same time:

```
Set ESBObj = Nothing   'Delete Published class object
Set ORSObj = Nothing   'Delete SsClink class object
```

5.2.7 Summary

Now you have finished creating the client application using Notes. The following sections describe the entire code for the client application and how to execute it.

5.2.7.1 Entire source code

The entire code for the client program you created in this section is shown here:

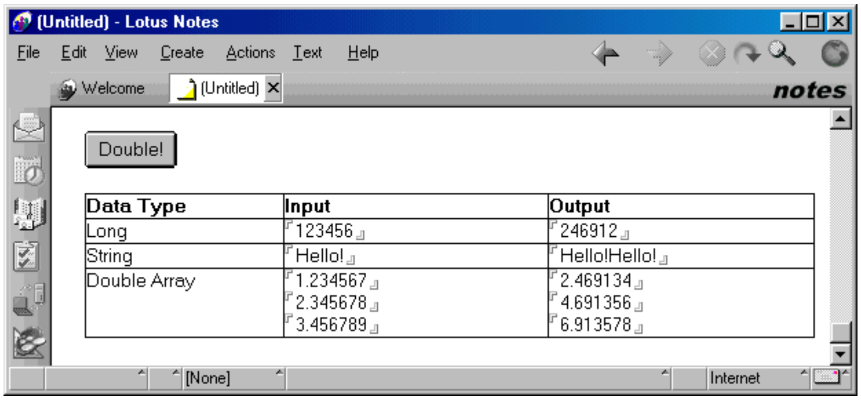
```
 (Globals) - (Options)
Option Public
Option Explicit
Uselsx "*SsClink"       ' Load ESB client LSX

(Globals) - (Declarations)
Dim ORSObj As SsClink   ' SsClink class object
Dim ESBObj As Variant   ' Published class object
Dim uidoc As NotesUIDocument

'--- Error handling procedure
Sub RunErrHandler(obj As SsClink, errCode As Long, errMsg As String)
  MsgBox "Error!!: rc = " & errCode & " : " & errMsg
End Sub

(Form) - (PostOpen)
Sub Postopen(Source As Notesuidocument)
  On Event From ORSObj Call RunErrHandler
  Set uidoc = Source        ' Set a NotesUIDocument class object
  Set ORSObj = New SsClink ' Create an SsClink class object

  '--- Create a Published class object
  Set ESBObj = ORSObj.CreateObject("chap05.Conversion, node=myServer")
End Sub

(Form) - (QueryClose)
Sub Queryclose(Source As Notesuidocument, Continue As Variant)
  Set ESBObj = Nothing   'Delete the Published class object
  Set ORSObj = Nothing   'Delete the SsClink class object
End Sub

(Double) - (Click)
Sub Click(Source As Button)
  On Error Goto ErrorHandler
  Dim myLong&, myString$, myDoubles#(2)
  Dim i%, rc&

  '--- Get field values
  myLong& = Clng(uidoc.FieldGetText("inLong"))
  myString$ = uidoc.FieldGetText("inString")
  For i% = 0 To 2
    myDoubles#(i%) = Cdbl(uidoc.FieldGetText("inDouble" & Cstr(i%)))
  Next

 '--- Call a member function of Published class
  rc = ESBObj.ToDoubleValue(myLong&, myString$, myDoubles#)
  If rc <> 0 Then
    Msgbox "Some error has occurred on ESB Server."
    Exit Sub
  End IF

  '--- Set results to fields
  Call uidoc.FieldSetText("outLong", Cstr(myLong&))
  Call uidoc.FieldSetText("outString",myString$)
  For i% = 0 To 2
    Call uidoc.FieldSetText("outDouble" & Cstr(i%), _
      Cstr(myDoubles#(i%)))
  Next
  Exit Sub

'--- Error handling routine
ErrorHandler:
  MsgBox "Error!! rc = " & CStr(Err) & " : " & Error
  Exit Sub
End Sub
```

5.2.7.2 Running a sample program

In this section, we execute the application you created. Follow these steps:

1. Select **File -> Save** to save the form.
2. Select **Design -> Preview in Notes**.
3. Enter an appropriate value for the entry field, and click the **Double!** button.

   The result returned from ESB for the result display field is displayed. (Figure 27).



Data Type	Input	Output
Long	123456	246912
String	Hello!	Hello!Hello!
Double Array	1.234567 2.345678 3.456789	2.469134 4.691356 6.913578

Figure 27. Executed results on Notes

5.3 Web applications

In this section, we use JavaScript as an example to explain how to create a Web client program using an ESB HTTP applet. You must have one of the following software applications on the client machine:

- Netscape Communicator R4.5 or higher
- Microsoft Internet Explorer 4.0 SP1 or higher
- Lotus Notes R5 or higher

Hint

- Refer to Chapter 12, “HTTP Communication Programming,” in the *ESB User’s Guide* for server side settings, such as WebSphere.
- You do not need to install ESB Client Enabler on the client machine.

5.3.1 Creating a client page

There are three primary methods for creating a client page for a Web application:

- **Using an HTML authoring tool**

  This is currently the creation method most generally considered. Tools such as IBM NetObjects TopPage, NetObjects Fusion, and Microsoft FrontPage Express are available. Some tools allow you to enter script after a page has been created, and other tools do not. If your tool does not permit this, you need to enter the script using a script development tool, such as a text editor or NetObjects ScriptBuilder.

- **Using Domino Designer**

  Domino Designer allows you can create a Web client page using the same method as Notes Form. You can also edit the script with a conventional IDE. In addition, you can create a client application that simultaneously supports Notes and Web.

- **Editing HTML directly**

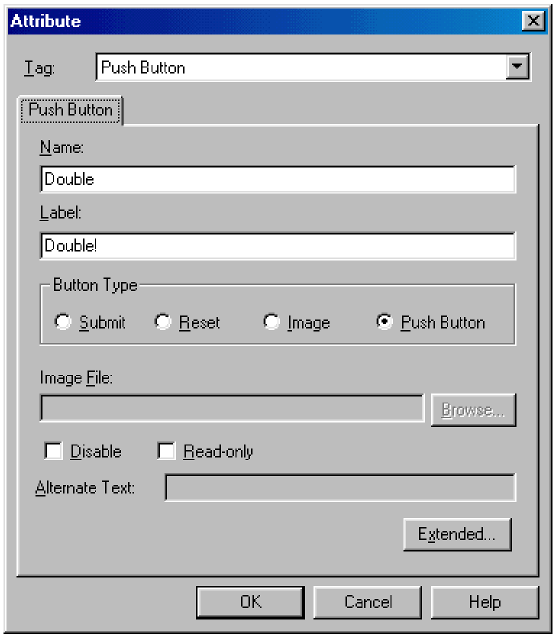
  For a simple page, you can create the HTML document on an appropriate text editor and enter the script at the same time. When creating a page using a text editor, refer to the source code incorporated at the end of the following section.

5.3.1.1 Creating a page using IBM NetObjects TopPage

To create a page using IBM NetObjects TopPage, follow these steps:

1. Start TopPage.
2. Select **File -> New** to create a blank page.
3. Select **Insert -> Form and Input Fields -> Form**.
4. Select **Insert -> Form and Input Fields -> Push Button -> Button**.

   The Attributes dialog box is displayed as shown in Figure 28 on page 104.
5. Enter `Double` in the Name field and `Double!` in the Label field. Select **Push Button** for the Button Type.



Tag: Push Button

Name: Double

Label: Double!

Button Type: ○ Submit ○ Reset ○ Image ● Push Button

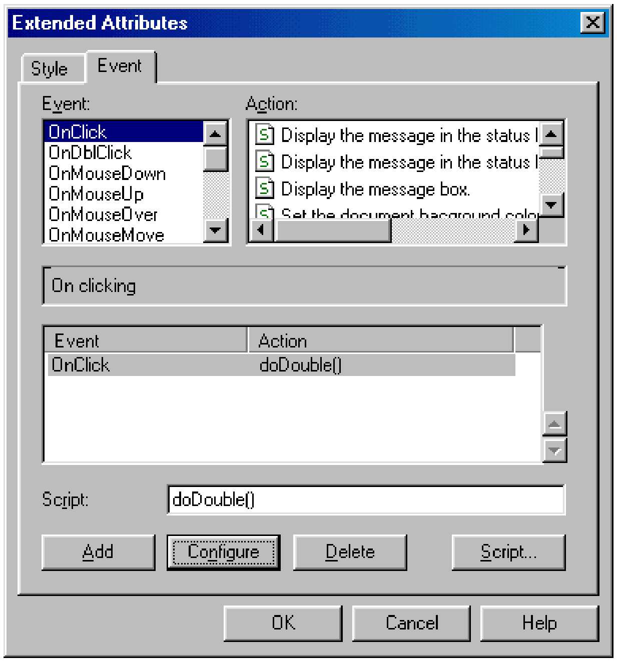
Image File:

□ Disable □ Read-only

Alternate Text:

Figure 28. Attributes dialog box

6. Click the **Extended** button, and select the **Event** tab on the Extended Attribute dialog box as shown in Figure 29.



On clicking

Event	Action
OnClick	doDouble()

Script: doDouble()

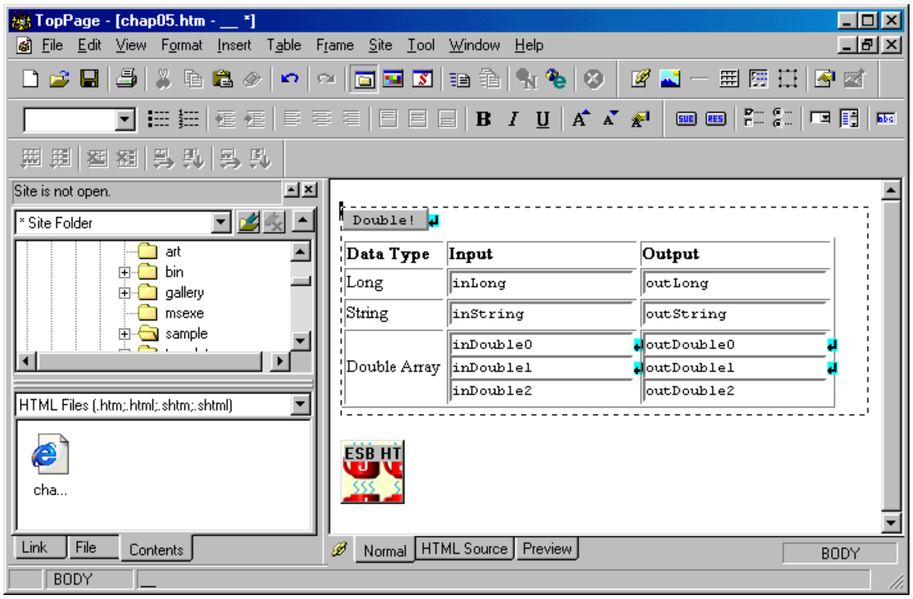
Figure 29. Extended Attribute dialog box

7. Select the **OnClick** event. Click the **Add** button. Enter `doDouble()` in the Script field. Click the **Configure** button. Enter `Procedure call` and `Result Display` for the content of the doDouble() function.

8. Click the **OK** button. Then, close the Attributes and Extended Attributes dialog boxes.

9. Select **Insert -> Table**, and create a three-column by four-row table.

10. Insert the text and text fields. Create them by selecting **Insert -> Form and Input Fields -> Text Field**. Align them as shown in Figure 30 on page 106.



TopPage - [chap05.htm - __ *]
File Edit View Format Insert Table Frame Site Tool Window Help
Site is not open.
* Site Folder
art
bin
gallery
msexe
sample
HTML Files (.htm;.html;.shtm;.shtml)
cha...
Link File Contents
Normal HTML Source Preview
BODY
BODY

Double!

Data Type	Input	Output
Long	inLong	outLong
String	inString	outString
Double Array	inDouble0 inDouble1 inDouble2	outDouble0 outDouble1 outDouble2

Figure 30. Layout of the client application using TopPage

The text fields to be created are shown in Table 12.

Table 12. Details of the created fields

Name	Entry format	Application
inLong	Text	Long type user entry field
outLong	Text	Long type result display field
inString	Text	String type user entry field
outString	Text	String type result display field
inDouble*n*	Text	Double type array user entry field. *n* is an integer starting from 0. The page shows up to two, but you can create any number you like.
outDouble*n*	Text	Double type array result display field. Create the same number as the user entry field.

11. Place the cursor outside of the form, and select **Insert -> Java Applet**.

12. Enter the values shown in Table 13 into the Attributes dialog box. Then, click the **OK** button.

Table 13. Input values for the applet attributes

Field name	Input value	Explanation
Code	com.lotus.esb.applet. SvClientApplet	ESB HTTP applet class name
Code Base	(Example) /esb_applet	URL of ESB HTTP applet
Substitute Text	ESB HTTP applet	Character string displayed when the applet cannot be run
Archives	SvClientApplet.jar	Jar file that stores the ESB HTTP applet class
Size	Both height & width 0	Applet display size. The ESB applet does not have a GUI, so it does not need to be displayed.

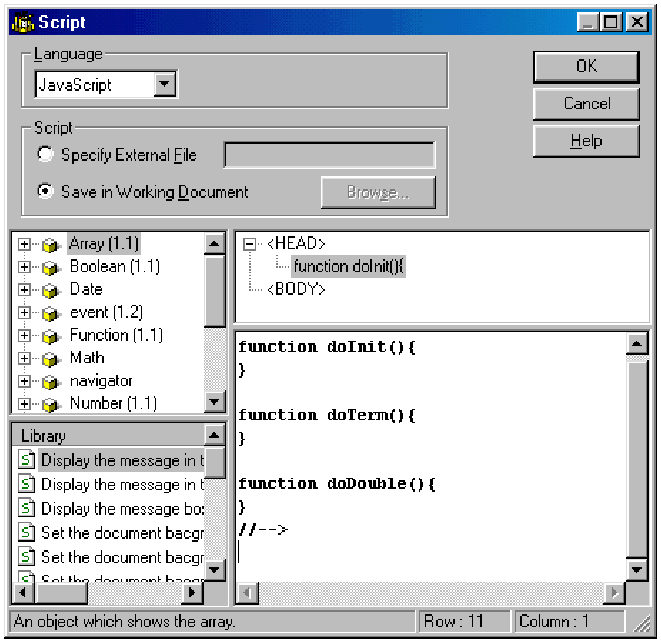
Hint

When you want to use the ESB IIOP applet to create a client application, enter `com.lotus.esb.iapplet.SvIIOPApplet` in the code field and `SvIIOPApplet.jar` in the archive. The other methods are the same as for the ESB HTTB applet.

13. Select **Edit -> Document Properties**. Then click the **Extended** button in the displayed Attributes dialog box. Click the **Event** tab in the Extended Attributes dialog box. Set **doInit()** for the OnLoad event and **doTerm()** for the OnUnload event, as explained in step 7.

14. Click the **Script** button to display the Script dialog box (Figure 31 on page 108). Add the following code to the text box at the lower right, and then click the **OK** button:

```
function doInit(){

}

function doTerm(){

}

function doDouble(){

}
```



Language: JavaScript

Script

○ Specify External File

● Save in Working Document

```
function doInit(){
}

function doTerm(){
}

function doDouble(){
}
//-->
```

An object which shows the array. Row : 11 Column : 1

Figure 31. Script dialog box

15. Save the file with the file name `chap05.htm`.

The source code of the HTML document created in this procedure appears as shown in the following example. You should consult this example when using the text editor to create a page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM NetObjects TopPage V4.0.3  for Windows">
<TITLE></TITLE>
<SCRIPT language="JavaScript">
<!--
function doInit(){
}

function doTerm(){
}

function doDouble(){
}
//-->

</SCRIPT></HEAD>
<BODY onload="doInit()" onunload="doTerm()">
<FORM><INPUT type="button" name="Double" value="Double!"
  onclick="doDouble()"><BR>
<TABLE border="1">
  <TBODY>
```

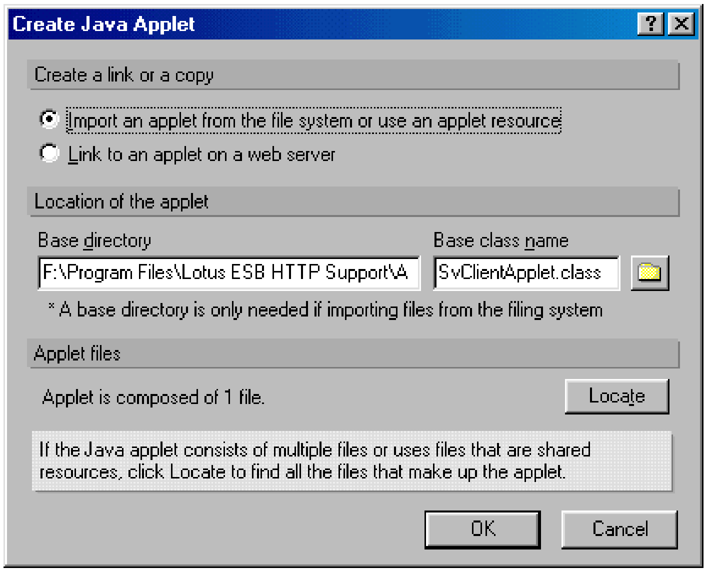
```
        <TR>
          <TD><B>Data Type</B></TD>
          <TD><B>Input</B></TD>
          <TD><B>Output</B></TD>
          </TR>
        <TR>
          <TD>Long</TD>
          <TD><INPUT size="20" type="text" name="inLong" value="inLong">
          </TD>
          <TD><INPUT size="20" type="text" name="outLong" value="outLong">
          </TD>
          </TR>
        <TR>
          <TD>String</TD>
          <TD>
          <INPUT size="20" type="text" name="inString" value="inString">
          </TD>
          <TD>
          <INPUT size="20" type="text" name="outString" value="outString">
          </TD>
        </TR>
        <TR>
          <TD>Double Array</TD>
          <TD>
          <INPUT size="20" type="text" name="inDouble1" value="inDouble1">
          <BR>
          <INPUT size="20" type="text" name="inDouble2" value="inDouble2">
          <BR>
          <INPUT size="20" type="text" name="inDouble3" value="inDouble2">
          </TD>
          <TD>
          <INPUT size="20" type="text" name="outDouble1"
            value="outDouble1"><BR>
          <INPUT size="20" type="text" name="outDouble2"
            value="outDouble2"><BR>
          <INPUT size="20" type="text" name="outDouble3"
            value="outDouble2">
          </TD>
          </TR>
      </TBODY>
    </TABLE>
    </FORM>
    <P><APPLET code="com.lotus.esb.applet.SvClientApplet"
     codebase="/esb_applet" alt="ESB HTTP applet" width="0" height="0"
     archive="SvClientApplet.jar"></APPLET></P>
    </BODY>
    </HTML>
```

5.3.1.2 Creating a form using Lotus Notes R5

The basic creation procedure is the same as the process explained in 5.2, “Notes application” on page 95. The difference is that you enter the ESB HTTP applet required to create the Published class object inside the form. Follow these steps:

1. Perform steps 1 through 8 in 5.3.1.1, “Creating a page using IBM NetObjects TopPage” on page 103.
2. Place the cursor at the bottom of the form, and select **Create -> Java Applet**.

The Create Java Applet dialog box appears as shown in Figure 32.



Create Java Applet

Create a link or a copy

● Import an applet from the file system or use an applet resource

○ Link to an applet on a web server

Location of the applet

Base directory: F:\Program Files\Lotus ESB HTTP Support\A

Base class name: SvClientApplet.class

* A base directory is only needed if importing files from the filing system

Applet files

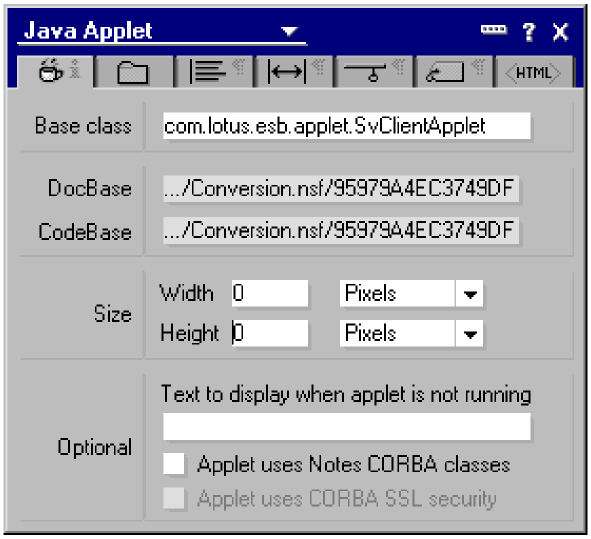
Applet is composed of 1 file.

If the Java applet consists of multiple files or uses files that are shared resources, click Locate to find all the files that make up the applet.

Figure 32. Create Java Applet dialog box

3. Select **Import an applet from the file system or use an applet resource**.
4. Click the 📁 button. Specify `SvClientApplet.jar`. Copy SvClientApplet.jar to a local directory, to create a client program on a computer where the ESB HTTP communication function has not been installed.
5. Click the **OK** button.
6. Select **Java Applet -> Java Applet Properties**.
7. Click the **Information** tab (Figure 33). In the Basic Class Name field, enter:

```
com.lotus.esb.applet.SvClientApplet
```



Base class: com.lotus.esb.applet.SvClientApplet

DocBase: .../Conversion.nsf/95979A4EC3749DF

CodeBase: .../Conversion.nsf/95979A4EC3749DF

Size: Width 0 Pixels; Height 0 Pixels

Optional: Text to display when applet is not running

□ Applet uses Notes CORBA classes

□ Applet uses CORBA SSL security

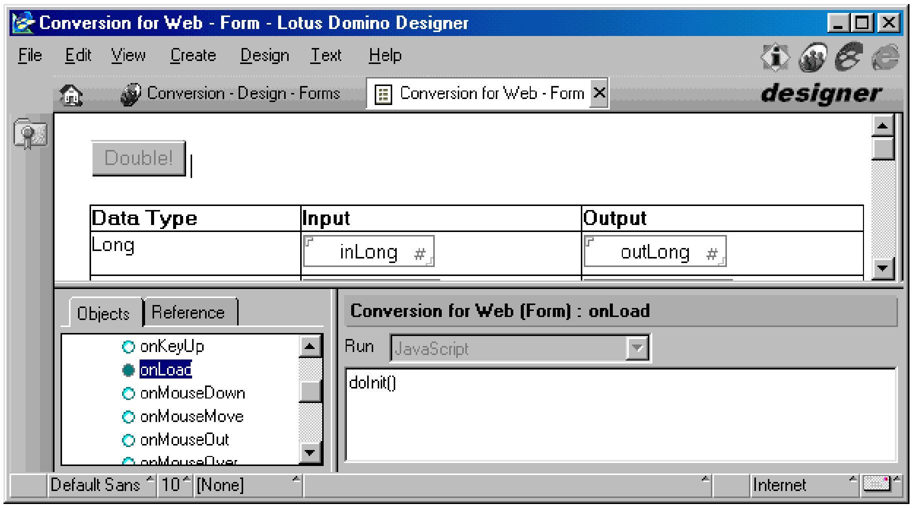
Figure 33. Information box of a Java applet

8. Click the **<HTML>** tab. In the other field, enter:

```
width="0" hight="0"
```

9. Close the InfoBox, and save the form.

10. Prepare a description location in advance for the script to be entered. Enter `doInit()` for the onLoad event of the form object. Enter `doTerm()` for the onUnload event. Then, enter `doDouble()` for the onClick event of the Double! button (Figure 34 on page 112).


Conversion for Web - Form - Lotus Domino Designer
File Edit View Create Design Text Help
Conversion - Design - Forms
Conversion for Web - Form
designer
Double!
Objects
Reference
onKeyUp
onLoad
onMouseDown
onMouseMove
onMouseOut
onMouseOver
Default Sans
10
[None]
Internet

Data Type	Input	Output
Long	inLong #	outLong #

Conversion for Web (Form) : onLoad

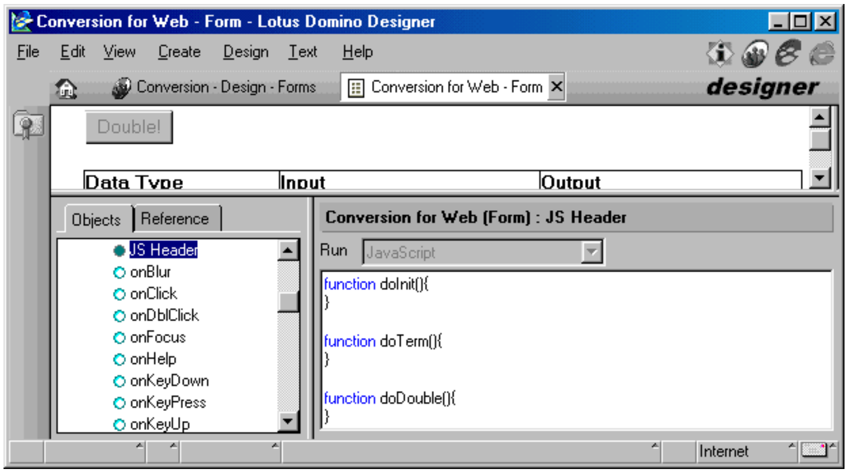
Run: JavaScript

```
doInit()
```

Figure 34. OnLoad event of the form object

11. Add the following code to the JS Header script of the form object as shown in Figure 35.

```
function doInit(){

}

function doTerm(){

}

function doDouble(){

}
```


Conversion for Web - Form - Lotus Domino Designer
File Edit View Create Design Text Help
Conversion - Design - Forms
Conversion for Web - Form
designer
Double!
Data Type
Input
Output
Objects
Reference
JS Header
onBlur
onClick
onDblClick
onFocus
onHelp
onKeyDown
onKeyPress
onKeyUp
Conversion for Web (Form) : JS Header
Run
JavaScript
Internet

```
function doInit(){
}

function doTerm(){
}

function doDouble(){
}
```

Figure 35. JS Header script of the form object

12.Save the form.

5.3.2 Creating a Published class object

Create a Published class object when an HTML document has been loaded. It was set for creating the client page, so that the doInit function is called when an HTML document is loaded. You should describe the creation of a Published class object in the doInit function.

1. Define the following variable as a global variable prior to entering the script in the doInit function. Enter the following code in the uppermost part of the JavaScript code description section (the JS Header script of the form object in Domino Designer, or the part surrounded by the SCRIPT tags).

```
var ESBObj;    // ESB HTTP applet
```

   ESBObj is the variable that holds the ESB HTTP applet. In this section, we use this ESB HTTP applet to create a Published class object. You must be able to refer to it in all functions, so make it a global variable.

2. Enter the script that creates the Published class object in the doInit function. First, obtain the ESB HTTP applet, and then assign it to the ESBOj.

```
ESBObj = document.applets[0];   // Get ESB HTTP applet
```

3. Set the URL of the ESB HTTP servlet, the host name of ESB Runtime, the project name, and the class name. Modify the URL and host name as appropriate in conformity with the actual environment.

```
ESBObj.SetParameter("servlet_name","/servlet/SvSessionServlet");
ESBObj.SetParameter("server_name","myServer");
ESBObj.SetParameter("project_name","chap05");
ESBObj.SetParameter("class_name","Conversion");
```

Hints

- You can also set the parameters using the PARAM tag within the HTML document. For example, when specifying the project name, consider this example:

  ```
  <PARAM NAME="project_name" VALUE="chap05">
  ```

- Displaying the trace on a Java console is useful for debugging. Add the following line:

  ```
  ESBObj.setParameter("trace_level","5");
  ```

4. Create the Published class object. Use the createObject function of the ESB HTTP applet to create a Published class object in the Web application.

   ```
   rc = ESBObj.createObject();
   ```

   The return value `rc` of the createObject function is an error code. Refer to 5.2.5, “Error handling” on page 99.

Hint

Unlike other clients, the Published class object that was created is held within an ESB HTTP applet. It cannot be directly touched from a client program. For this reason, such things as procedure calls are to be made using ESB HTTP applets, rather than Published class objects.

5.3.3 Calling a procedure

You call a procedure when you click the Double! button. Since the page was created for calling the doDouble function for an onClick event of the Double! button, you should describe the invocation steps of the procedure in the doDouble function.

1. Obtain the value entered in the text box. Use the Document Object Model (DOM) to obtain the value in the text box:

   ```
   thisForm = document.forms[0];         // Get Form object
   myLong = thisForm.inLong.value;
   ```

```
myString = thisForm.inString.value;
for (i=0; i<3; i++){
    myDoubles[i] = thisForm.elements[i+5].value;
}
```

2. Set the obtained value as the argument. Use the setArgument method for setting the argument in the Web application.

   ```
   ESBObj.setArgument(1,"(long)" + myLong);
   ESBObj.setArgument(2,myString);
   ESBObj.setArgument(3,"(array)(double)" + myDoubles.join(","));
   ```

   The first argument of the setArgument method indicates the index of the argument (an integer starting from 1). You can also specify the variable name used for the procedure definition, for example:

   ```
   ESBObj.setArgument("myString", myString);
   ```

   The second argument of the setArgument method is the value to actually be passed to the server. With the ESB HTTP applet, you cast it by adding a “(data_type)” character string to the front of the variable. Also, when setting the array for an argument, pass the character string that is delimited by a comma (,) by casting it to the array.

3. Once you have completed preparing the argument, use the callMethod method to call the Published class procedure. The argument of the callMethod method is the procedure name.

   ```
   rc = ESBObj.callMethod("ToDoubleValue");
   ```

5.3.4 Displaying the results

You can obtain the results of the procedure from the return value and argument of the pass-by reference. Use the getReturnValue of the ESB HTTP applet to obtain the return value. Use the getArgument to obtain the argument. Use DOM to display the obtained results in the output text box.

```
thisForm.outLong.value = ESBObj.getArgument(1);
thisForm.outString.value = ESBObj.getArgument(2);
for (i=0; i<3; i++){
  thisForm.elements[i+8].value = ESBObj.getArgument(3,i);
}
```

The first argument of the getArgument method is the argument index (an integer starting from 1). Instead of the index, you can specify the argument name used by the server for a procedure definition, for example:

```
thisForm.outString.value = ESBObj.getArgument("myString");
```

If the argument is an array, the return value of the getArgument method becomes a character string delimiting the data by commas (,). You can also obtain specific data alone by specifying an array index (an integer starting from 0) for the second argument.

5.3.5 Error handling

Functions that perform such tasks as creating Published class objects and calling procedures return error codes. The code 0 indicates a normal termination, and there is no problem with continuing on to the next process. However, a code other than 0 indicates that a problem has occurred, for example with an ESB Runtime or client, or a network, that requires some handling.

In this sample, if an error occurs, it displays an error message and interrupts the processing. Add the following code after the createObject method or the callMethod:

```
if (rc != 0) {
  alert("Error!! rc = " + ESBObj.getErrorCode() + "\n"
       + ESBObj.getErrorMessage());
  return;
}
```

When an error has been generated, this code displays the error code and message in the warning dialog box and interrupts the process of the function itself where the error was generated. You can use the getErrorNumber method and getErrorMessage of the ESB HTTP applet to obtain the code and the message of the error that occurred previously.

5.3.6 Deleting an object

Enter the script for deleting a Published class object in the doTerm argument called when the HTML document is unloaded. Use the deleteObject method of the ESB HTTP applet to delete a Published class object. Add the following line within the doTerm function:

```
ESBObj.deleteObject();
```

5.3.7 Summary

Now you have finished creating the client application using Web browsers. This section describes the entire code for the client application and how to execute the application.

5.3.7.1 Entire source code

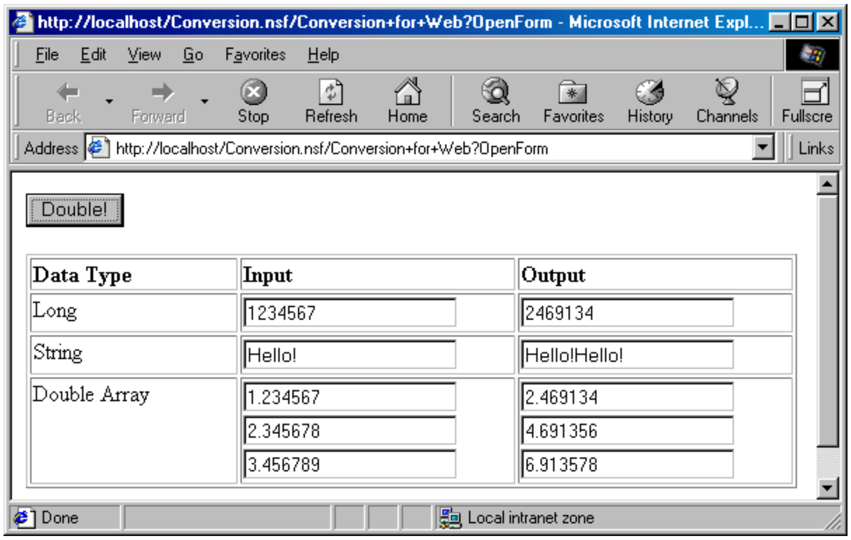
The entire code for the sample application code created in this section is shown here:

```
var ESBObj;                      // ESB HTTP applet

function doInit(){
  //--- Get ESB HTTP applet
  ESBObj = document.applets[0];
  //--- Set ESB HTTP applet parameters
  ESBObj.setParameter("servlet","/servlet/SvSessionServlet");
  ESBObj.setParameter("server_name","myServer");
  ESBObj.setParameter("project_name","Chap05");
  ESBObj.setParameter("class_name","Conversion");

  //--- Create Published class object
  rc = ESBObj.createObject();
  if (rc != 0) {
     alert("Error at CreateObject()!! rc = " +
       ESBObj.GetErrorNumber() + " : " + ESBObj.GetErrorMessage());
     return;
  }
}

function doTerm(){
  //--- Delete Published class object
  ESBObj.deleteObject();
}

function doDouble(){
  var thisForm;
  var myLong, myString, myDoubles;
  var i, rc, result;

  myDoubles = new Array(2);
  thisForm = document.forms[0];   // Get Form object
  //--- Get text box values
  myLong = thisForm.inLong.value;
  myString = thisForm.inString.value;
  for (i=0; i<3; i++){
    myDoubles[i] = thisForm.elements[i+5].value;
  }

  //--- Set arguments
  ESBObj.setArgument(1,"(long)" + myLong);
  ESBObj.setArgument(2,myString);
  ESBObj.setArgument(3,"(array)(double)" + myDoubles.join(","));

  //--- Call member procedure of Published class
  rc = ESBObj.callMethod("ToDouble");
  if (rc != 0) {
   alert("Error at callMethod()!! rc = " + ESBObj.getErrorNumber() +
   " : " + ESBObj.getErrorMessage());
   return;
  }

  //--- Get return value
  rc = ESBObj.getReturnValue();
  if (rc != null || rc != 0) {
   alert("Some error has occurred on ESB Runtime!!");
   return;
  }
  //--- Get and set results
  thisForm.outLong.value = ESBObj.getArgument(1);
  thisForm.outString.value = ESBObj.getArgument(2);
  for (i=0; i<3; i++){
    thisForm.elements[i+8].value = ESBObj.getArgument(3,i);
  }
}
```

5.3.7.2 Running a sample program

Let us run the application program you created:

1. Save the client application.
2. Display the client page on a Web browser:
   - When using IBM NetObjects TopPage, select **Tools -> Browser -> Internet Explorer** or **Tools -> Browser -> Netscape**.
   - When using Domino Designer, select **Design -> Preview in Notes**, **Design -> Preview in Web Browser -> Internet Explorer** or **Design -> Preview in Web Browser -> Netscape Version 4**.
   - When using the text editor, open the URL that has the HTML document created on the Web browser.
3. Set the appropriate value for the input field and click the **Double!** button as shown in Figure 36. The value returned from ESB for the result display field is shown.


http://localhost/Conversion.nsf/Conversion+for+Web?OpenForm - Microsoft Internet Expl...
File Edit View Go Favorites Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Channels
Fullscre
Address
http://localhost/Conversion.nsf/Conversion+for+Web?OpenForm
Links
Double!
Done
Local intranet zone

Data Type	Input	Output
Long	1234567	2469134
String	Hello!	Hello!Hello!
Double Array	1.234567 2.345678 3.456789	2.469134 4.691356 6.913578

Figure 36. Executed results on a Web browser

5.4 VB application

This section uses the VB example to explain the creation of a client application using the OLE automation support language. The following software is required for the client computer:

- Run module of Microsoft Visual Basic (the creator of the client application also requires a VB development environment)
- ESB Client Enabler

Creating a client application in the OLE automation support language is almost the same as the method for creating it in LotusScript. There are two differences. It uses the SvClink class to create the Published class object, and the error handling is only for runtime errors.

Hint

Since LotusScript also supports OLE automation, it can create a Published class object using the SvClink class. However, since it is compared to the SsClink class, the SvClink class is limited to such tasks as arguments and error handling. We recommend that you use SsClink with LotusScript.

5.4.1 Creating a client screen view

To create a client screen view, follow these steps:

1. Start VB.
2. Select **File -> New Project,** and create a Standard EXE.
3. Create buttons and text boxes. Lay them out as shown in Figure 37 on page 120.

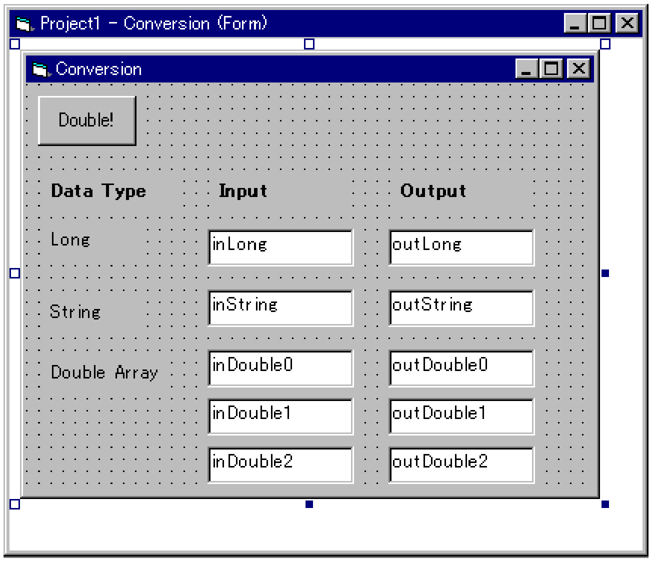


Figure 37. Layout of the client application using VB

The buttons and text box properties are shown in Table 14.

Table 14. Details of the created fields

Object	Type	Application
ToDoubleValue	CommandButton and Caption are "Double!"	Button that calls a procedure.
inLong	Textbox, Anumbers and editing permissible	Long type user entry field.
outLong	Textbox, Anumbers and editing permissible	Long type result display field.
inString	Textbox, Atext and editing permissible	String type user entry field.
outString	Textbox, Atext and editing permissible	String type result display field.
inDouble*n*	Textbox, Anumbers and editing permissible	Double type array user entry field. *n* is an integer starting from 0. The screen view shows up to 2, but you can create any number you like.
outDouble*n*	Textbox, Anumbers and editing permissible	Double type array result display field. Create the same number as the user entry field.

4. Select **File -> Save Form1 As**, and save it with the file name `conversion.frm`. Also, select **File -> Save Project As**, and save it with the file name `chap05.vbp`.

5.4.2 Creating a Published class object

It is useful to create a Published class object when you open a window. Use the Load event of the Form object (Form_Load subroutine) to accomplish this with VB:

1. Define the following variable as a global variable. Enter the following code in the (Declarations) script of the (Globals) object:

   ```
   Dim ESBObj As Object             ' Published class object
   ```

   ESBObj is the variable that holds the Published class object. ESB defines it as a global variable to permit referencing from all the objects within the form.

2. Enter the script for creating for creating Published class objects in the Form_Load subroutine. Use the SvClink class provided by ESB to create a Published class object using the OLE automation function.

   a. Use the CreateObject function incorporated into VB to create an SvClink class object.

   b. Use the CreateObject member function to create a Published class object.

   ```
   Set ORSObj = CreateObject("SvClink")
   Set ESBObj = ORSObj.CreateObject("chap05.Conversion, node=myServer")
   Set ORSObj = Nothing
   ```

3. The project name (Chap05), the class name (Conversion), and the host name (myServer) where ESB Runtime is operating are set for the argument of the CreateObject function. Change the host name as appropriate in conformity with the actual settings. ORSObj is no longer necessary after the Published class object is created, so delete it immediately.

Hint

When you want to change an ORB to be used for connection, or set a user ID and password to be used for authentication, specify it before invoking the CreateObject member function. For example, when changing ORB to DCOM, consider this example:

```
ORBObj.ConnType = _gDCOM_h
```

5.4.3 Calling a procedure

Let us call a procedure using the Published class object you created. When you click the Double! button on the screen view, you want to call a procedure. Therefore, the script you must enter is the Click event (ToDoubleValue_Click subroutine) of the ToDoubleValue button object.

1. Obtain the value entered in the entry field and assign it to the variable:

   ```
   myLong = CLng(inLong.Text)
   myString = inString.Text
   myDoubles(0) = CDbl(inDouble0.Text)
   myDoubles(1) = CDbl(inDouble1.Text)
   myDoubles(2) = CDbl(inDouble2.Text)
   ```

   Use the Text Property of the text box to obtain the field value. This is a string type property, so you should perform a type conversion when necessary.

2. Set the arguments, and call the procedure:

   ```
   rc = ESBObj.ToDoubleValue(myLong, myString, myDoubles)
   ```

5.4.4 Displaying the result

The return value of the ToDoubleValue function is an error code. If it is other than 0, some error has occurred on the server side. Therefore, the result is not displayed, and it is necessary to interrupt the processing. Accordingly, you should add the following code after the ToDoubleValue function value:

```
If rc <> 0 Then
  MsgBox "Some error has occurred on ESB Server."
  Exit Sub
End IF
```

All the ToDoubleValue function arguments are pass-by references. In other words, when the calling of the function finishes, the result is stored in the variable specified for the argument. Consequently, you use these arguments to display the results in the output text box.

```
outLong.Text = CStr(myLong)
outString.Text = myString
outDouble0.Text = CStr(myDoubles(0))
outDouble1.Text = CStr(myDoubles(1))
outDouble2.Text = CStr(myDoubles(2))
```

5.4.5 Error handling

Prepare for cases where a runtime error has occurred in a Published class object and at other locations, and write the error handling. Add the following code to the beginning of the Form_Load subroutine and the ToDoubleValue subroutine:

```
On Error Goto ErrorHandler
```

This code shifts the control to the error handling routine ErrorHandler when a runtime error has occurred. Next, enter the content of the ErrorHandler. Add the following code to the end of the Form_Load subroutine and the ToDoubleValue subroutine:

```
ErrorHandler:
  MsgBox "Error!! rc = " & CStr(Err) & " : " & Error
  Exit Sub
```

When a runtime error occurs within a subroutine, the processing jumps to the ErrorHandler and an error code, and messages are displayed in the message box.

5.4.6 Deleting an object

It is appropriate to delete the Published class object created when a window is displayed when you have closed the window. Use the Form_Unload subroutine to accomplish this with VB. Use the Nothing constant to delete the object:

```
Set ESBObj = Nothing
```

5.4.7 Summary

Now you have completed creating the client application using VB. This section describes the entire code for the client application and how to run the application.

5.4.7.1 Entire source code

The entire source code for the sample application created in this section is shown here:

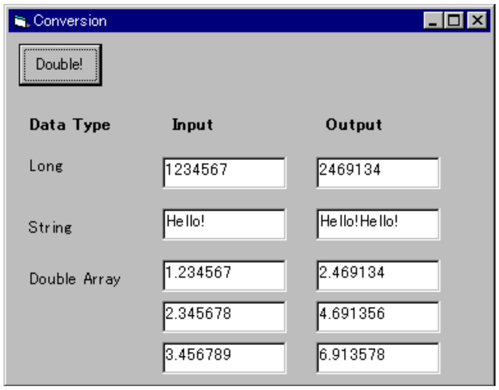
```
Dim ESBObj As Object            ' Published class object

Private Sub Form_Load()
On Error Goto ErrorHandler     ' Error Handling
Dim ORSObj As Object            ' SvClink class object

'--- Create SvClink class object
Set ORSObj = CreateObject("SvClink")
'--- Create Published class object
Set ESBObj = ORSObj.CreateObject("chap05.Conversion, node=myServer")
Set ORSObj = Nothing            ' Delete SvClink class object
Exit Sub

'--- Error handling routine
ErrorHandler:
  MsgBox "Error!! rc = " & CStr(Err) & " : " & Error
  Exit Sub
End Sub

Private Sub Form_Unload(Cancel As Integer)
'--- Delete Published class object
Set ESBObj = Nothing
End Sub

Private Sub ToDoubleValue_Click()
On Error Goto ErrorHandler
Dim myLong As Long
Dim myString As String
Dim myDoubles(2) As Double
Dim rc As Long

'--- Get textbox values
myLong = CLng(inLong.Text)
myString = inString.Text
myDoubles(0) = CDbl(inDouble0.Text)
myDoubles(1) = CDbl(inDouble1.Text)
myDoubles(2) = CDbl(inDouble2.Text)

'--- Call member function of Published class
rc = ESBObj.ToDoubleValue(myLong, myString, myDoubles)
If rc <> 0 Then
  MsgBox ("Some error has occurred on ESB Runtime.")
  Exit Sub
End If

'--- Set results to textboxes
outLong.Text = CStr(myLong)
outString.Text = myString
outDouble0.Text = CStr(myDoubles(0))
outDouble1.Text = CStr(myDoubles(1))
outDouble2.Text = CStr(myDoubles(2))
Exit Sub

'--- Error handling routine
ErrorHandler:
  MsgBox "Error!! rc = " & CStr(Err) & " : " & Error
  Exit Sub
End Sub
```

5.4.7.2 Running a sample program

Let us run the application you created:

1. Save the form and the project.
2. Select **Run -> Start**.
3. Substitute an appropriate value into the entry text box. Then, click the **Double!** button.

   The value returned from ESB to the result display field appears as shown in Figure 38 on page 125.



Data Type	Input	Output
Long	1234567	2469134
String	Hello!	Hello!Hello!
Double Array	1.234567	2.469134
	2.345678	4.691356
	3.456789	6.913578

Figure 38. Results on VB application

5.5 Programming hints

Refer to Chapter 4, “Server application programming” on page 43, which also describes programming that relates to the transfer of array data and security.

5.5.1 Load distribution of the ESB program

The CreateObject member function of the SvClink class and SsClink class can specify the multiple ESB Runtimes by delimiting them through the use of semicolons (;). Here is an example of the chap05 project run on three hosts: esb1, esb2, and esb3:

```
Set ESBobj =ESBObj.CreateObject _
  ("chap05.Conversion, node=esb1;esb2;esb3")
```

When multiple host names are specified, it attempts to create an object until it succeeds, from the left side server to the right side server. It is necessary to make the random order for attempting connections into ESB Runtime by specifying the SHUFFLE option, to achieve a load distribution for ESB Runtime. Also, if you specify the RETRY option at the same time, it uniformly attempts to connect to all the hosts, regardless of whether it has failed to connect in the past.

```
Set ESBobj = ESBObj.CreateObject _
  ("chap05.Conversion, node=esb1;esb2;esb3, option=SHUFFLE;RETRY")
```

5.5.2 Internet Explorer applications

When using Microsoft Internet Explorer as the client, you can use VBScript and JScript, as well as JavaScript as the programming language. When you use VBScript and JScript, you can also use the programming style by using the SvClink class. Consider the following example for VBScript:

```
Set ORSObj = createObject("SVClink")
Set ESBObj = ORSObj.CreateObject("chap05.Conversion, node=myServer")
```

5.5.3 Receiving arrays with a Web client

In a simple application, we obtained arrays using the getArgument(int, int) method. However, you can also obtain arrays with either the getArgument(int) or the getReturnValue() methods. At this time, the arrays are transferred as string type variables connecting the data with commas as the delimiters. A handy approach is to use the split method of the String object of JavaScript to convert these variables into arrays.

For example, when the return value has called an array function, refer to this code:

```
var i, resultString, resultArray;
resultString = ESBObj.getReturnValue();
resultArray = resultString.split(",");
for (i = 0; i < resultArray.length; i++) { alert(resultArray[i]); }
```

5.5.4 Applications using multiple forms, pages, or screen views

In this chapter, we created a Published class object along with a screen display. However, there are times where this programming style may not be not appropriate, depending on the application. When using multiple forms in Notes and VB applications in particular, multiple Published class objects are created, which may possibly affect the server performance. Add processing so that it will close new forms and transfer the Published class objects to subforms.

When using multiple screens in a Web application, it is convenient to create a non-displayed frame (a form with 0 width or height) and download the ESB HTTP applet to the HTML document loaded there. To reference an applet from a separate frame at this time, obtain the frame containing the applet from the new window object to obtain the applet object.

For example, a menu frame and an invisible applet frame are included in frame.html. The ESB HTTP applet downloaded to the applet frame is referenced from the menu frame.

frame.html

```
<html>
<head>
<title>Frame Test</title>
```

```
</head>
<frameset cols="0,*">
  <frame name="applet"" src="applet.html" noresize>
  <frame name="menu" src="menu.html">
</frameset>
</html>
```

applet.html

```
<html>
<body>
<applet CODEBASE="/esb_applet" ARCHIVE="SvClientApplet.jar"
  CODE="com.lotus.esb.applet.SvClientApplet" NAME="applet1" WIDTH=0
  HEIGHT=0>
</applet>
</body>
</html>
```

menu.html

```
<html>
<HEAD>
<SCRIPT>
  var ESBObj;
  ESBObj = parent.applet.applets[0];
…
```

Chapter 6. Using WebSphere

Lotus ESB supports Web clients using HTTP. This chapter describes how to create and manage ESB applications using IBM WebSphere.

6.1 Overview

ESB uses DCOM or IIOP as communication protocols between servers and clients. This communication system is not suited to extranet or Internet environments because it has normal firewall limitations. Moreover, the fact that modules supporting these communications must be distributed to all the client computers has been a major obstacle to its popularization.

ESB provides specialized servlets, beans, and applets to enable access from Web browsers through HTTP. It enables the use of ESB in an extranet and with the Internet by integrating these components with WebSphere.

An ESB servlet and ESB bean are loaded onto WebSphere, which receives requests from clients through HTTP and sends those requests to ESB Runtime through IIOP. Moreover, an ESB servlet provides a variety of functions for the effective operation of such tasks as client object management, used object pool functions, setting the maximum number of client objects, session management, and security functions.

Web client programming methods can be classified into two types. The first type is *applet programming*, such as JavaScript using an ESB applet. The second type is *JSP programming*, which uses the JSP provided by WebSphere for programming. The following sections describe their respective characteristics.

6.1.1 Applet programming

The operational flow of applet programming involves this series of actions:

1. The ESB HTTP applet is embedded in an HTML document and downloaded to a Web browser.
2. The Web browser communicates with the ESB servlet through script (JavaScript, VBScript, and so on).
3. The ESB servlet communicates with the ESB server through IIOP whose function is provided by the ESB bean.
4. The results are obtained through the *method* of the ESB applet. It is possible to create client programs in formats similar to normal ESB

programming, such as creating objects using the *createObject method* and calling a method using the *callMethod method*.

This programming method displays the outputs in the field of the same page when the browser receives the result from the server because the field is treated as one of the objects. As a result, the entire display does not refresh. This is a significant benefit to suit the job for the Line of Business type. The amount of traffic between client and server can be reduced drastically.

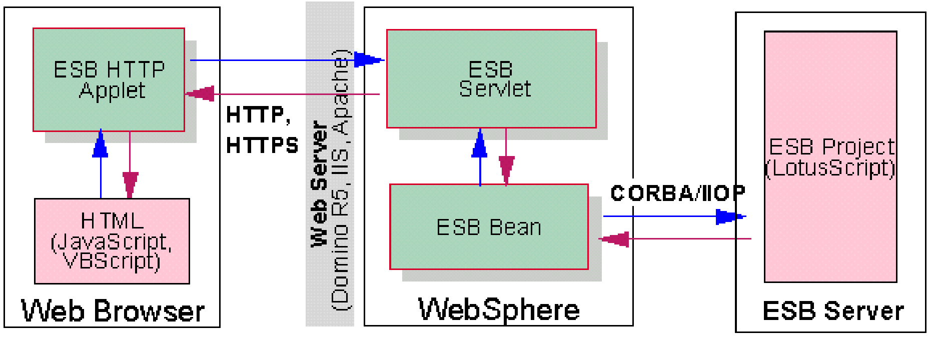

ESB HTTP Applet
HTML (JavaScript, VBScript)
Web Browser
HTTP, HTTPS
Web Server (Domino R5, IIS, Apache)
ESB Servlet
ESB Bean
WebSphere
CORBA/IIOP
ESB Project (LotusScript)
ESB Server

Figure 39. Relationship among the ESB applet, ESB servlet, and ESB bean

6.1.2 JSP programming

The client may request the creation of an object and method call, for example, using the parameters from the HTML form for ESB servlets. The HTML file must be created by using the JSP file matched to the respective results because the results are received using the JSP file. It is possible to code using Java in the JSP as well as HTML. The Java code is interpreted on the server side. Then, it is downloaded to the client. The ESB servlet and ESB beans provide two types of beans, *servlet beans* and *client beans*. The benefit of the JSP programming style is that Java can be used from non-Java VM Web browsers.

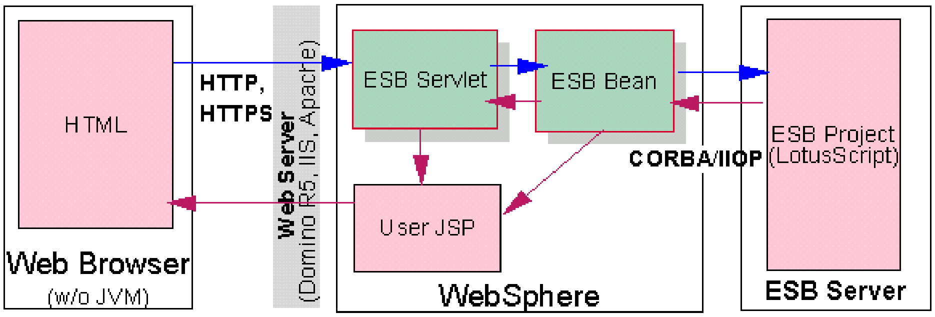


Figure 40. Relationship of the JSP programming type

6.2 Programming using ESB applets

This section describes programming an applet in ESB.

6.2.1 Application creation example

Let us try creating a client application that converts Celsius to Fahrenheit by calling the *CtoF(c As Single) As Single* of the sample CFConv.

The CFConv server program is located in the samples\CFConv (/usr/lpp/esb/samples/En_US/CFConv for AIX) in the directory, where ESB Runtime is installed. The following information is included in the CFConv project:

- **Project name**: CFConv
- **Class name**: ConvertClass
- **Method**: CtoF(c As Single) As Single

The client code appears as shown in the following example. The bold portions are key programming elements. Each element is explained later in the same sequence as the functions that are highlighted with a boxed number.

Note

The following example is actually coded in the sequence written in the sample. The number “1” appears at the bottom of this sample code although it is executed first. The number “8” does not appear in a boxed number. This means that unloading an applet is done explicitly when another URL is executed or the browser is exited.

```
<html>
<head>
<title>Simple Temperature Conversion Program</title>
<script language="JavaScript">
<!--
function calculate(){
  var lsobj;
  var TheForm;
  var rc;
  var c;
  var f;
  var ServerName;
  <!-- myServer is the host name of ESB Server -->
  var C_ServerName = "myServer" ;
  var C_ProjectName = "CFConv";
  var C_ClassName = "ConvertClass";

  TheForm = document.CFConv;
  lsobj = document.myobj1;

2 lsobj.setParameter("server_name", C_ServerName);
  lsobj.setParameter("project_name", C_ProjectName);
  lsobj.setParameter("class_name", C_ClassName);

3 rc = lsobj.createObject();
if (rc != 0)
  {
    alert("Connection to ESB Server has failed.");
    return;
  }
  c = TheForm.txtC.value ;

4 lsobj.setArgument("c", c);

5 rc = lsobj.callMethod("CtoF");

  if (rc != 0) {
    alert("Calling a method has failed.");
    return;
  }

6 f = lsobj.getReturnValue();

  TheForm.txtF.value = f;

7  lsobj.deleteObject();

  lsobj = "";
}

//-->
</script>
</head>
<body><form name="CFConv">
Celsius<input type="text" size="8" name="txtC" value="0">
<input type="button" value="Calculate" onclick="calculate()">
Fahrenheit<input type="text" size="8" name="txtF" value="0">


1 <applet codebase="/esb_applet" archive="SvClientApplet.jar"
   code ="com.lotus.esb.applet.SvClientApplet"
   name="myobj1" width=0 height=0>
```

```
<!-- webserver is the host name of the web server. -->

<param name="servlet"
        value="http://webserver/servlet/SvSessionServlet">
</applet>
</form>
</body>
</html>
```

The processing details are described as follows:

1. Load ESB HTTP applets.

   The applet programming style begins by downloading the ESB HTTP applet into a Web client. The downloading of the applets is normally invoked by means of <APPLET> tags.

   Specify `SvClientApplet.jar` for the archive attribute value and `com.lotus.esb.applet.SvClientApplet` for the code attribute value. Enter the virtual directory of `SvClientApplet.jar` for the CODEBASE. Specify the object name of `SvClientApplet` for the NAME. Enter the URL of the servlet for the *servlet* parameter. One applet makes one Published class object respectively. When creating multiple Published class objects, it must be coded the same number of <APPELET> tags as objects.

2. Set the ESB Runtime to be connected.

   Set the host name of ESB Runtime, the project name, and the Published class name to be connected for the downloaded applet. It is possible to set other option parameters according to the conditions. There are two types of setting methods. One method is to use <PARAM> tags within the <APPLET> tags of the HTML document. The other method is to use a setParameter method in the script.

   An example of the <PARAM> tag is shown here:

   ```
   <param name="server_name" value="myServer">
   ```

   An example of the setParameter method is shown here:

   ```
   lsobj.setParameter("server_name", "myServer");
   ```

3. Create the Published class objects.

   Once settings are completed, such as for the server name, a Published class object can be created. Use the **createObject** method for the creation.

4. Set the argument for a Published class object.

   The argument of a Published class must be set prior to calling a Published class method. Use the **setArgument** method for setting the argument. It is not necessary to define all of the arguments defined by the Published class method. Set only the argument that sends the data to ESB Runtime.

5. Call a Published class method.

   Once the arguments are set, the Published class method can be called using the method callMethod.

6. Obtain the result.

   Upon successful completion of the calling procedure, the result can be obtained. Use the **getArgument** method to obtain the ByRef argument, and use the **getReturnValue** method to obtain the return value.

7. Delete the object.

   If the connection to ESB Runtime becomes unnecessary, use the **deleteObject** method to delete the Published class object. Even if it is not explicitly deleted, it will inevitably be deleted when the applet is unloaded.

8. Unload the applet.

   The applet is unloaded along with the unloading of the HTML document and the ESB applet program ends.

6.2.2 Running the application

Run the code created by following these steps:

1. Run the project CFConv on ESB Runtime.
2. Open the HTML document created on the Web browser.

   Figure 41 shows the Browser (IE) view of the sample program that was created by using ESB Applet.

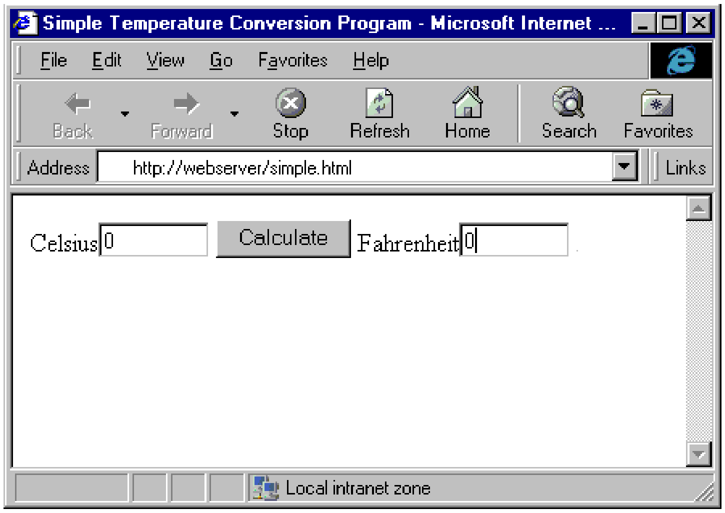

Simple Temperature Conversion Program - Microsoft Internet ...
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites
Address http://webserver/simple.html
Links
Celsius 0
Calculate
Fahrenheit 0
Local intranet zone

Figure 41. Browser (IE) view of sample program (CFConv) using ESB Applet

3. Enter an appropriate numerical value in the Celsius file. Click the **Calculate** button. The corresponding Fahrenheit value is displayed.

6.3 Programming using JSP

JSP programming performs the request for the creation of a Published class object or the request for the calling of a Published class method in an HTML document, and then receives the result in a JSP file. Thereafter, you can recursively perform this series of tasks from the JSP file.

6.3.1 An example of application creation

This section shows an example of the creation of JSP programming using a Celsius and Fahrenheit conversion program. It requests the conversion from Celsius to Fahrenheit on the initial page (jsptest.html) and outputs the result to the JSP (result1.jsp). Furthermore, result1.jsp can request conversion from Fahrenheit to Celsius or from Celsius to Fahrenheit. The former result is output to result2.jsp, and the latter is output to the same result1.jsp.

The CFConv server program is included in samples\CFConv (/usr/lpp/esb/samples/En_US/CFConv for AIX) in the directory where you installed ESB Runtime.

The following information is included in the CFConv project:

- **Project name**: CFConv
- **Class name**: ConvertClass
- **Method1**: CtoF(c As Single) As Single
- **Method2**: FtoC(f As Single) As Single

The files to be created are jsptest.html, result1.jsp, and result2.jsp for the client application. First, create the CFConv object in jsptest.html. Then, determine the Fahrenheit for the Celsius and output the calculation result to result1.jsp. The calculation result is output to result1.jsp. Create a button to enable further calculation from Fahrenheit to Celsius and from Celsius to Fahrenheit. Set it up so that it outputs the result of the Celsius to Fahrenheit calculation to the result1.jsp file and the result of the Fahrenheit to Celsius calculation to the result2.jsp file.

6.3.1.1 Creating jsptest.html

Create a CFConv object. Then, call the CtoF method. The result is output to result1.jsp. Name the client bean `cfconv`. At this point, enter the data to be sent to the ESB servlet, and create a button for sending it.

Hint

Upon deleting type="hidden", the data is displayed on the browser.

```
<html>
<head>
<title>Test the JSP</title>
</head>
<body>
<h1>Using the CFConv sample with JSP.</h1>
<form method="get" action="/servlet/SvSessionServlet">

<!-- specify the published class object information. -->
<!-- myServer is the host name of the ESB server. -->
<input type="hidden" name="server_name" value="myServer">
<input type="hidden" name="project_name" value="CFConv">
<input type="hidden" name="class_name" value="ConvertClass">
<!-- specify the information for method call. -->
<input type="hidden" name="method_name1" value = "CtoF">
Celsius<input type="text" name="argument1-1" value="0">
<input type="hidden" name="bean_name" value="cfconv">
<!-- the result will be output into result1.jsp. -->
<input type="hidden" name="jsp_name"
        value="/esb_samples/cfconv/result1.jsp">

<input type="submit" name="testJSP" value="Celsius -> Fahrenheit">
</form>

<form method="get" action="/servlet/SvSessionServlet">
<!-- specify the published class object information. -->
<!-- myServer is the host name of the ESB server. -->
```

```
<input type="hidden" name="server_name" value="myServer">
<input type="hidden" name="project_name" value="CFConv">
<input type="hidden" name="class_name" value="ConvertClass">
<input type="hidden" name="method_name1" value = "FtoC">
Fahrenheit<input type="text" name="argument1-1" value="0">
<input type="hidden" name="bean_name" value="cfconv">
<!-- the result will be output into result2.jsp. -->
<input type="hidden" name="jsp_name" value="/esb_samples/cfconv/result2.jsp">
<input type="submit" name="testJSP" value="Fahrenheit -> Celsius"></form>
</body>
</html>
```

6.3.1.2 Creating result1.jsp

To create result1.jsp, follow these steps:

1. Set the beans.

   Define the servlet bean and the client bean. Name the object name of client bean from the name specified in the bean_name parameter of the jsptest.html.

2. Display the result of the procedure.

   Obtain the calculation result using getReturnValue() or getArgument() of the client bean object.

3. Prepare for a recall.

   Create a button so that you can recall the two methods in the following part. Set the result of the CtoF method outputs to the same JSP file and the result of the FtoC method outputs to result2.jsp. At this time, the name of the bean specified in the following two examples must be specified as the same one.

   The URL of the SvSessionServlet servlet, the server name, the project name, and the class name of the ESB Runtime can use respectively the getURL() and getServer() methods of the SvServletBean. They can also use the getServerName(), getProjectName(), and getClassName() methods of SvClientBean.

Hint

In case the bean name specified in the bean_name parameter is already active in the same session, this bean can be used for the calling of the procedure. In other words, the server_name, project_name, and class_name parameters can be omitted.

```
<html>
<head>
<title>Result of JSP Test</title>
</head>
<!-- Declare bean  -->
```

```
1 <bean name="SvServletBean" type="com.lotus.esb.servlet.SvServletBean"
    introspect="no" create="no" scope="request">
  </bean>
  <bean name="cfconv" type="com.lotus.esb.bean.SvClientBean"
    introspect="no" create="no" scope="request">
  </bean>

  <body><h1>result1.jsp</h1>
  <!-- Display result. -->

2 <p>Celsius <b><%= cfconv.getArgument(1) %></b> = Fahrenheit <b><%=
  cfconv.getReturnValue() %></b>


3 <!-- Prepare next call (Call CtoF method.) -->
  <!-- Output ersult to result1.jsp -->
  <form method="get" action="<%= SvServletBean.getURL() %>">
  <input type="hidden" name="method_name1" value = "CtoF">
  Celsius <input type="text" name="argument1-1">
  <input type="hidden" name="bean_name" value="cfconv">

  <input type="hidden" name="jsp_name"
          value="/esb_samples/cfconv/result1.jsp">
  <input type="submit" name="testJSP" value="Celsius -> Fahrenheit"></form>

4 <!-- Prepare next call  (Call FtoC method.) -->
  <!-- Output result to result2.jsp. -->
  <form method="get" action="<%= SvServletBean.getURL() %>">
  <input type="hidden" name="method_name1" value = "FtoC">
  Fahrenheit<input type="text" name="argument1-1">
  <input type="hidden" name="bean_name" value="cfconv">
  <input type="hidden" name="jsp_name"
          value="/esb_samples/cfconv/result2.jsp">
  <input type="submit" name="testJSP" value="Fahrenheit -> Celsius"></form>
  </body>
  </html>
```

6.3.1.3 Creating result2.jsp

The content is the same as result1.jsp, other than the output of the Fahrenheit to Celsius calculation. Since the same bean as result1.jsp is contained in the same session, the server_name, project_name, and class_name parameters can be omitted.

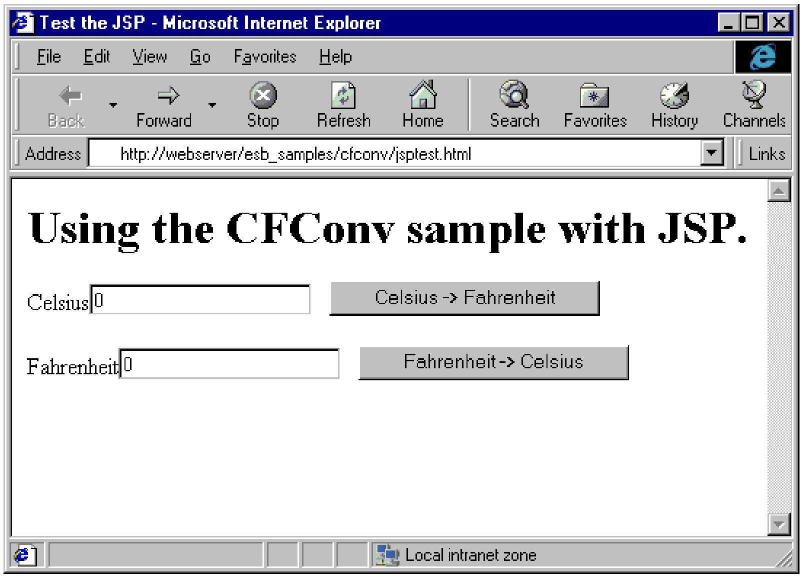
```
<html>
<head>
<title>Result of JSP Test</title>
</head>
<!-- Declare bean. -->
<bean name="SvServletBean" type="com.lotus.esb.servlet.SvServletBean"
  introspect="no" create="no" scope="request">
</bean>
<bean name="cfconv" type="com.lotus.esb.bean.SvClientBean"
  introspect="no" create="no" scope="request">
</bean>
<body><h1>result2.jsp</h1>
<!--  Display result. -->
<p>Fahrenheit  <b><%= cfconv.getArgument(1) %></b> = Celsius <b><%=
cfconv.getReturnValue() %></b>

<!-- Prepare next call (Call CtoF method.) -->
```

```
<!-- Output ersult to result1.jsp -->
<form method="get" action="<%= SvServletBean.getURL() %>">
<input type="hidden" name="method_name1" value = "CtoF">
Celsius <input type="text" name="argument1-1">
<input type="hidden" name="bean_name" value="cfconv">
<input type="hidden" name="jsp_name" value="/esb_samples/cfconv/result1.jsp">
<input type="submit" name="testJSP" value="Celsius -> Fahrenheit"></form>

<!-- Prepare next call  (Call FtoC method.) -->
<!-- Output result to result2.jsp. -->
<form method="get" action="<%= SvServletBean.getURL() %>">
<input type="hidden" name="method_name1" value = "FtoC">
Fahrenheit<input type="text" name="argument1-1">
<input type="hidden" name="bean_name" value="cfconv">
<input type="hidden" name="jsp_name" value="/esb_samples/cfconv/result2.jsp">
<input type="submit" name="testJSP" value="Fahrenheit -> Celsius"></form>
</body>
</html>
```

6.3.2 Running the application

Run the application that was created by following these steps:

1. Run the project CFConv on ESB Runtime.
2. Copy the three files—jspts.html, result1.jsp, and result2.jsp—to the samples\CFConv directory where the HTTP communication function (/usr/lpp/esb/samples/En_US/servlet/CFConv for AIX) directory was installed. Figure 42 shows the Browser (IE) view of the sample program created by using JSP.


Test the JSP - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Channels
Address
http://webserver/esb_samples/cfconv/jsptest.html
Links

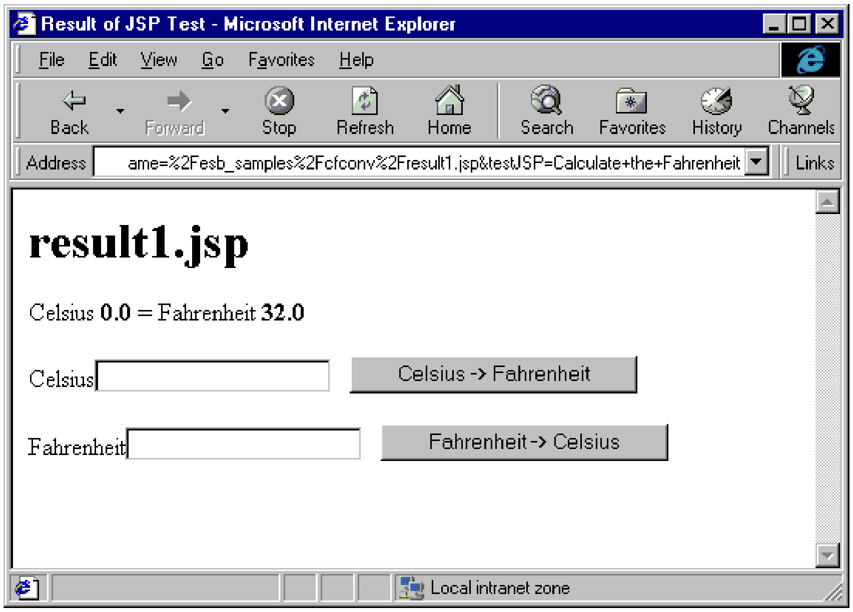
Using the CFConv sample with JSP.

Celsius 0 Celsius -> Fahrenheit

Fahrenheit 0 Fahrenheit -> Celsius

Figure 42. Browser (IE) view of the sample program created by using JSP

3. Open jsptst.html on the Web browser.
4. Click the **Celsius->Fahrenheit** button.
5. result1.jsp is read in. The Fahrenheit value corresponding to 0 degrees Celsius is displayed (Figure 43).



Result of JSP Test - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Channels
Address
ame=%2Fesb_samples%2Fcfconv%2Fresult1.jsp&testJSP=Calculate+the+Fahrenheit
Links

result1.jsp

Celsius **0.0** = Fahrenheit **32.0**

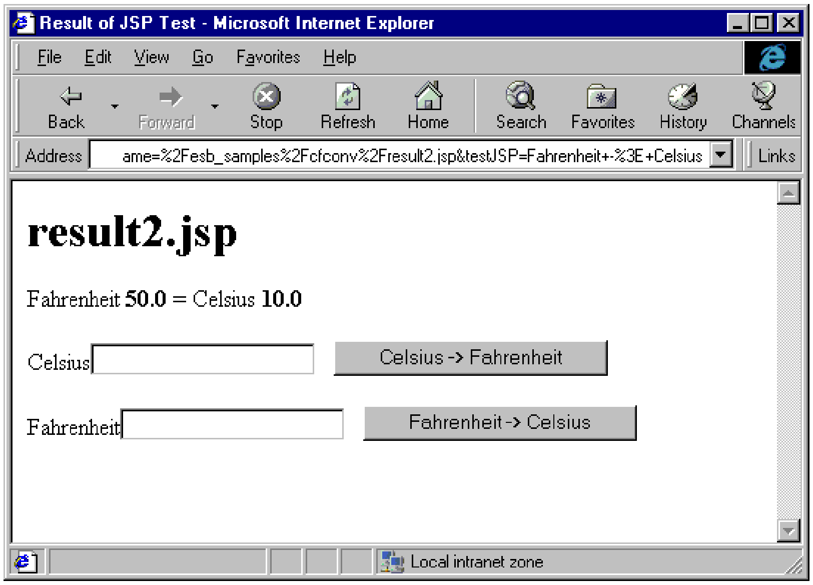
Celsius

Fahrenheit

Fahrenheit-> Celsius
Local intranet zone

Figure 43. View of result.jsp

6. Enter `50` in the Fahrenheit field. Click the **Fahrenheit -> Celsius** button.
7. result2.jsp is read. The Celsius value corresponding to 50 degrees Fahrenheit is displayed (Figure 44).



Result of JSP Test - Microsoft Internet Explorer

Address ame=%2Fesb_samples%2Fcfconv%2Fresult2.jsp&testJSP=Fahrenheit+-%3E+Celsius

result2.jsp

Fahrenheit **50.0** = Celsius **10.0**

Celsius

Fahrenheit

Figure 44. View of result2.jsp

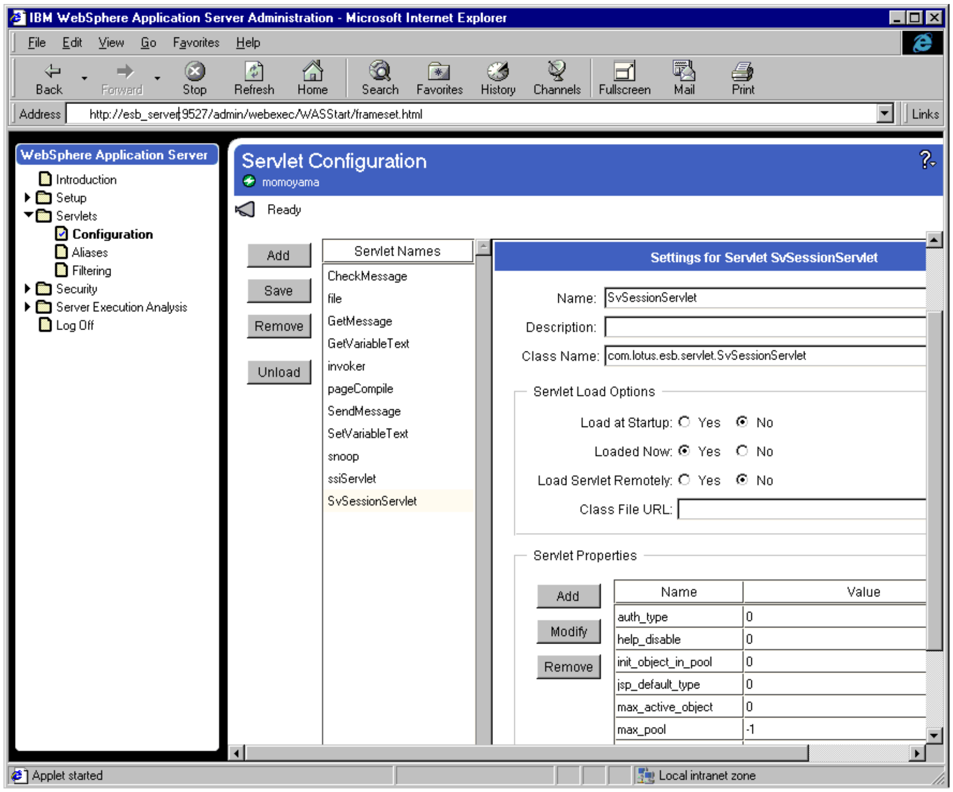
6.4 Setting the HTTP communication function

This section describes the various settings for HTTP communication.

6.4.1 Setting the ESB servlet

Set the following ESB properties using WebSphere Application Server Administration Tool:

1. Start the WebSphere Application Server Administration Tool.
2. Select **Servlet -> Configuration** to display the Servlet Configuration page.
3. Select **SvSessionServlet** from the Servlet names. Then, the property registered in the Servlet Properties field and its value are displayed (Figure 45 on page 142).


IBM WebSphere Application Server Administration - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print
Address http://esb_server:9527/admin/webexec/WASStart/frameset.html
Links
WebSphere Application Server
Introduction
Setup
Servlets
Configuration
Aliases
Filtering
Security
Server Execution Analysis
Log Off
Servlet Configuration
momoyama
Ready
Add
Save
Remove
Unload
Applet started
Local intranet zone

Servlet Names
CheckMessage
file
GetMessage
GetVariableText
invoker
pageCompile
SendMessage
SetVariableText
snoop
ssiServlet
SvSessionServlet

Settings for Servlet SvSessionServlet

Name: SvSessionServlet

Description:

Class Name: com.lotus.esb.servlet.SvSessionServlet

Servlet Load Options

Load at Startup: ○ Yes ● No

Loaded Now: ● Yes ○ No

Load Servlet Remotely: ○ Yes ● No

Class File URL:

Servlet Properties

Name	Value
auth_type	0
help_disable	0
init_object_in_pool	0
jsp_default_type	0
max_active_object	0
max_pool	-1

Figure 45. Servlet configuration on WebSphere Application Server Administration

4. Modify the window as required. Then, click the **Save** button, and reload.

6.4.1.1 Setting the maximum value for the ESB object

Once a large number of clients are accessed, the performance may be reduced depending on the speed of the computer in which the servlet runs. Considering such a situation, the ESB servlet can limit the number of ESB objects that can be accessed at the same time. When placing a limit, set the maximum value of the ESB object in the property max_active_object of the ESB servlet, where the object can be accessed simultaneously.

6.4.1.2 Remote setting of servlet properties

Setting the remote_control property of the ESB servlet to “1” enables you to remotely set the following properties from a Web browser.

A malicious user may cause problems with the Remote setting turned “on” in the deployment of the system. Therefore, we strongly recommend that you turn the Remote setting to “off.”

- auth_type
- jsp_default_type
- max_pool
- max_active_object
- trace_level

Send the following command to the ESB servlet to remotely set these properties:

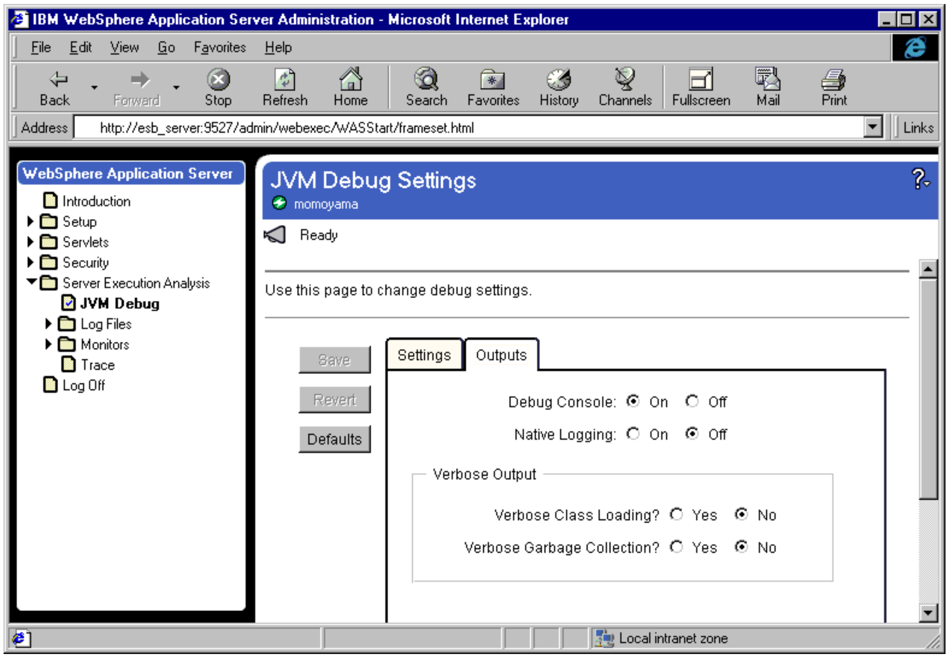
```
http://your_web_server/servlet/SvSessionServlet?property=value
```

Here, `your_web_server` is the host name of the Web server being used by WebSphere. For example, set the trace level to `5`:

```
http://your_web_server/servlet/SvSessionServlet?trace_level=5
```

When the trace output destination (trace_file property of the ESB servlet) in the ESB Servlet Properties is set to stdout (default), the trace log can be referred to in real time on the Java Debug Console of WebSphere.

To start the Java Debug Console, select **Server Execution Analysis -> JVM Debug** to display the JVM Debug Settings page (Figure 46 on page 144). Then, click the **Output** tag and turn on the Java Debug Console.



Address: http://esb_server:9527/admin/webexec/WASStart/frameset.html

JVM Debug Settings

Use this page to change debug settings.

Debug Console: ● On ○ Off

Native Logging: ○ On ● Off

Verbose Output

Verbose Class Loading? ○ Yes ● No

Verbose Garbage Collection? ○ Yes ● No

Figure 46. JVM debug setting

Hint

Increasing the trace level decreases the performance. We recommend that you set the level to “0” when deploying the system.

6.4.1.3 Displaying servlet information

Set the remote_control property of the ESB servlet to “1”. This allows you to check the ESB server information (such as maximum pool number and client authentication) on a remote computer. Send the following URL to the ESB servlet for displaying the information of servlet setting:

```
http://your_web_server/servlet/SvSessionServlet?servlet_info=
```

6.4.1.4 Displaying help

Set the help_disable property of the ESB servlet to “0”. This allows remote referencing of the ESB servlet and the Help applet. Send the following URL to the ESB servlet to refer the Help applet:

```
http://your_web_server/servlet/SvSessionServlet?help=servlet
http://your_web_server/servlet/SvSessionServlet?help=applet
```

6.4.2 Setting security

Using the security function of ESB or Web server, it is possible to set a limit on the access to the project. The security settings may be set using one of the following options:

- Using the user ID and password properties of ESB
- Using basic Web server authentication
- Using both

6.4.2.1 Using the user ID and password parameters

Use the user ID and password parameters of ESB applets and servlets. Since it is not mandatory to set the security of a Web server, this method applies when the HTML document is disclosed to the public for any user, and the connections to ESB Runtime is limited to a certain number of users.

The following series of actions describes how to set security for an ESB HTTP applet using a sample configuration:

1. Open CFConv.html in the text editor.

   For Windows NT, it is located in samples\CFConv under the directory installed NT ESB HTTP communication function.

   For AIX, it is in /usr/lpp/esb/samples/En_US/servlet/CFConv.

2. Correct the cmdConnect_OnClick() function as shown here.

   Any user can connect to the server if the parameter of the user ID and the password are defined in the HTML document. They should be entered by the user. This example shows the use of the prompt function.

```
function cmdConnect_OnClick(){
  var rc;
  var TheForm;
  var ServerName;
  var UserID;
  var Password;
  TheForm = document.CFConv;
  lsobj = document.myobj1;

  if (Connected == true){
    alert(ERROR_CONNECTED);
    return;
  }

  ServerName = prompt(PROMPT_SERVER, C_ServerName);
  if (ServerName == "" || ServerName == null ){
    ServerName = "" ;
    return;
  }

  UserID  =  prompt("Enter user name.",  UserID);
  Password  =  prompt("Enter password",  Password);
  C_ServerName = ServerName
  //Set server information
  lsobj.setParameter("server_name", ServerName);
  lsobj.setParameter("project_name", C_ProjectName);
  lsobj.setParameter("class_name", C_ClassName);
  lsobj.setParameter("userid", UserID);
  lsobj.setParameter("password", Password);

  //Call createObject method
  rc = lsobj.createObject();
  :
  :
}
```

If you are using JSP, it is possible to send the user ID and password parameters when creating an object in an HTML document:

```
<html>
:
  <form method="GET" action="/servlet/SvSessionServlet">
  <input type="hidden" name="server_name" value="myServer">
  <input type="hidden" name="project_name" value="myProject">
  <input type="hidden" name="class_name" value="myClass">
  <p>user id:
  <input type="text" name="userid">
  <p>password:
  <input type="password" name="password">
  <input type="hidden" name="method_name1" value="Method1">
  <input type="hidden" name="argument1-1" value="0">
  <input type="hidden" name="argument1-2" value="1">
  <input type="hidden" name="argument1-3" value="2">
  <input type="hidden" name="bean_name" value="myBean">
  <input type="hidden" name="jsp_name" value="/beanresult.jsp">
  <input type="submit" name="go" value="Call Method1">
   :
</html>
```

3. Use the WebSphere Application Server Administration Tool to set the auth_type property of the ESB servlet to 0 and reload the ESB servlet. The user IDs and passwords can then be sent to ESB Runtime.
4. Register the users in the computer in which ESB Runtime is running.

   For Windows NT, follow these steps:

   a. Select **Start -> Program -> Management Tool -> Domain User Manager**.
   b. Set the user names to be given access authority and their passwords.

   For AIX, follow these steps:

   a. Run SMIT or SMITTY after the log-in by the root user.
   b. Set the users to be given access authority and their passwords in **Security & Users -> Users -> Add a User**.
5. Use the ESB Configuration Tool to set the exit routine for user authentication to **OS Security.**

For Windows NT, follow these steps:

a. Open the ESB **Configuration Tool.**

b. Click the **Exit Routine** tab.

c. Click the **Add** button, and then select **OS Security.**

For AIX, follow these steps:

a. Run SMIT or SMITTY after the log-in by the root user.

b. Select the settings for **Application -> ESB -> System Configuration -> Change/ Show the Client Authentication.**

c. Enter `libhpwssec.a` into the Exit Routine text box.

6. Restart ESB Runtime.

   For Windows NT, stop the Lotus ESB Engine Service in the control panel services, and then start it up.

   For AIX, complete these tasks:

   a. Run SMIT or SMITTY after the log-in by the root user.

   b. Run **Stop Subsystem** in **Application -> ESB -> System Management**, and then run **Start Subsystem.**

7. Run `CFConv.lpk` in the ESB System Manager.

8. Start the Web browser on the client side and open CFConv.html.

9. Click the **Connect** button. Enter the user name and password.

   When the connection is successful, the user name of the connected user is output to the system manager on the ESB Runtime side or inside the Status tag in the Runtime monitor.

Entering an unregistered user name or wrong password causes the return of security error. In case the authentification by user ID and password is used with basic authentication, it can be implemented with more strict security by setting the security of ESB Runtime and HTML document independently.

Note

The evaluation of the user ID and password depends on your security set up. For more information, see 4.5, “Security” on page 69.

6.4.2.2 Using basic authentication

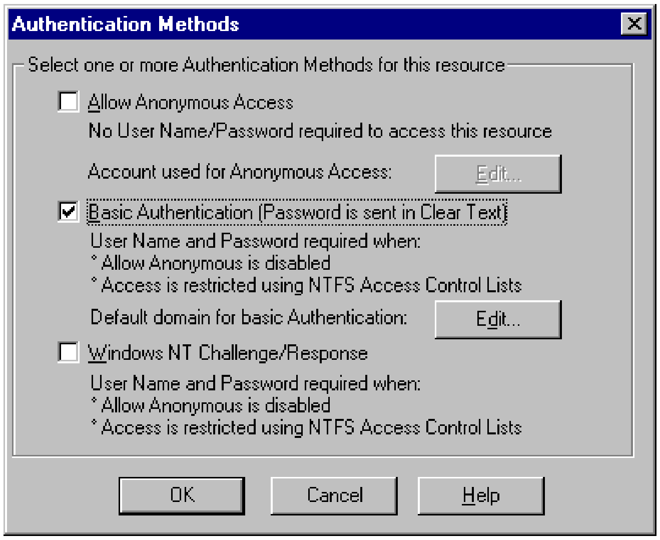
It is possible to use the basic authentication function of the Web server without using the user ID or password parameter. In addition, after basic authentication, it can be authenticated again in ESB Runtime by passing the user name and password to ESB Runtime.

Although the setting of the basic authentication for each resource differs depends on the Web server, we shows examples of the settings in Microsoft Internet Information Server and Domino Go Server (AIX version).

Example of the settings in Microsoft Internet Information Server

Follow this process:

1. Start the Internet Service Manager.
2. Right-click **Predetermined Web Site**, and open **Properties.**
3. Open the **Directory Security** tag. Click the **Edit** button of **Anonymous Access and Approval Control.**
4. Deselect **Allow Anonymous Access**. Then, select **Basic Authentication** as shown in Figure 47.



Select one or more Authentication Methods for this resource

□ Allow Anonymous Access
No User Name/Password required to access this resource
Account used for Anonymous Access:

☑ Basic Authentication (Password is sent in Clear Text)
User Name and Password required when:
* Allow Anonymous is disabled
* Access is restricted using NTFS Access Control Lists
Default domain for basic Authentication:

□ Windows NT Challenge/Response
User Name and Password required when:
* Allow Anonymous is disabled
* Access is restricted using NTFS Access Control Lists

Figure 47. Authentification for Microsoft IIS

5. Click the **OK** button to exit the setting of basic authentication.
6. Using the WebSphere Application Server Administration Tool, set the auth_type property of the ESB servlet to `1` and reload the ESB servlet. The user ID and password entered in the basic authentication are then sent to ESB Runtime.

7. Register the user to the operating system in which ESB Runtime is running.
8. Also register the same user to the operating system where Web Server is running. This step can be omitted when Web Server and ESB Runtime are on the same computer.
9. Start System Manager on ESB Runtime, and run `CFConv.lpk`.
10. Start the Web browser on the client, and read CFConv.html.

    Note: There is no need to use the corrected CFConv.html. Use the user ID and password parameters.
11. When CFConv.html is opened, a user name and password are requested, so enter these.
12. Click the **Connect** button.

When the connection is successful, the user name of the connected user is output to the system manager on the ESB Runtime side or inside the Status tag in the Runtime monitor.

Examples of the settings in the Domino Go Webserver

Follow this process:

1. Open the Web browser, and set the Go Webserver.

   After opening `"http://Go_Webserver/"`, click **CONFIGURATION AND ADMINISTRATION FORMS**.

   In this URL, `Go_Webserver` is the name of the host where Go Webserver is installed.
2. Click **Add User** in **Administration of Users** to register the users and the group. An example is shown in Table 15.

Table 15. Sample user parameters for Go Webserver

Field	Value
User name	esbuser
Password	esbuser
Group	esb
Password file	/usr/lpp/internet/server_root/protect/esb.passwd
Group file	/usr/lpp/internet/server_root/protect/esb.group

3. After the user is registered, click **Document Protection** in **Access Control**.
4. Set Document Protection as shown in Table 16 on page 150. Then, click the **Apply** button.

Table 16. Sample access control parameters

Field	Value
URL request template	/* (Define basic authentication for all documents)
Authentication options	Password or user/group authentication
Define protection settings	In-line

5. Set **Protection Setup** as shown in Table 17. Then, click the **Apply** button.

Table 17. Sample parameters of protection

Field	Value
Protection realm	esb
Password file	/lpp/internet/server_root/protect/esb.passwd
Group file	/lpp/internet/server_root/protect/esb.group
Users with Read permission (GET or POST)	esbuser

When the settings in Table 17 are completed, they are saved as follows to /etc/httpd.conf:

```
Protect  /*  {
      GroupFile    /usr/lpp/internet/server_root/protect/esb.group
      PasswdFile   /usr/lpp/internet/server_root/protect/esb.passwd
      ACLOverride  Off
      Mask          Anybody@(*)
      PostMask     esbuser
      GetMask      esbuser
      AuthType     Basic
      ServerID     esb
}
```

6. Click the **Restart Server** button or log in to the **root user** from the computer terminal where Go Webserver was installed. Enter the following commands:

```
stopsrc -s httpd
startsrc -s httpd
```

   Once you complete this task, the basic settings of the Go Webserver are complete.

7. Using the WebSphere Application Server Administration Tool, set the auth_type property of the ESB servlet to `1` and reload the ESB servlet. This sets it up so that the user ID and password entered in the basic authentication are then sent to ESB Runtime.

8. Register the users (users and passwords registered in the Go Webserver; in this example, it is esbuser/esbuser) to the operating system where ESB Runtime is running.

9. Start the **System Manager** in ESB Runtime, and run `CFConv.lpk`.

10. Start the Web browser on the client, and read **CFConv.html**. There is no need to use the CFConv.html corrected in [Use userid/password parameters].

11. When CFConv.html is opened, a user name and password are requested. Enter `esbuser/esbuser`.

12. Click the **Connect** button.

When the connection is successful, the user name of the connected user is output to the system manager on the ESB Runtime side or inside the Status tag in the Runtime monitor.

6.4.3 Setting the session management

This section explains how to set session management by using JSP.

6.4.3.1 Setting time-outs

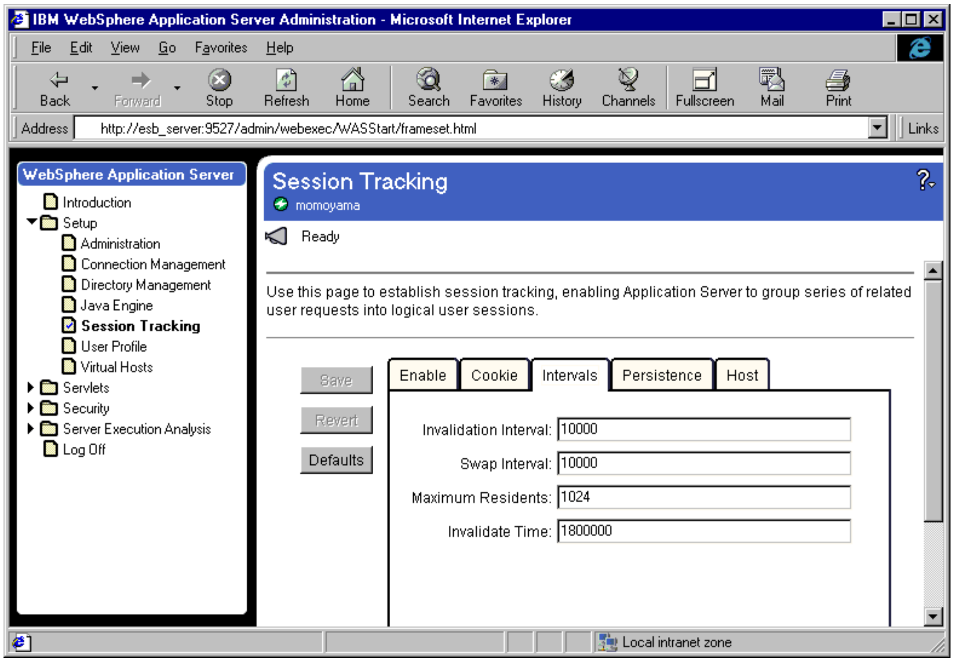
Once a client object is created by using JSP, the client object will not be deleted even though the Web browser is terminated. However, the WebSphere session time-out function can be used to automatically delete the object if there has not been any request from a client for a certain time. Use the WebSphere Application Server Administration Tool to make this setting. Follow these steps:

1. Start the WebSphere **Manager Tool.**

2. Select **Setup -> Session Tracking** to display the Session Tracking page as shown in Figure 48 on page 152.

3. Click the **Intervals** tag. Set **Invalidation Interval** and **Invalidate Time.** The units are in milliseconds.

4. Click the **Save** button.



Address: http://esb_server:9527/admin/webexec/WASStart/frameset.html

Use this page to establish session tracking, enabling Application Server to group series of related user requests into logical user sessions.

Invalidation Interval: 10000

Swap Interval: 10000

Maximum Residents: 1024

Invalidate Time: 1800000

Figure 48. Session Tracking Intervals setting

6.4.3.2 Using cookies

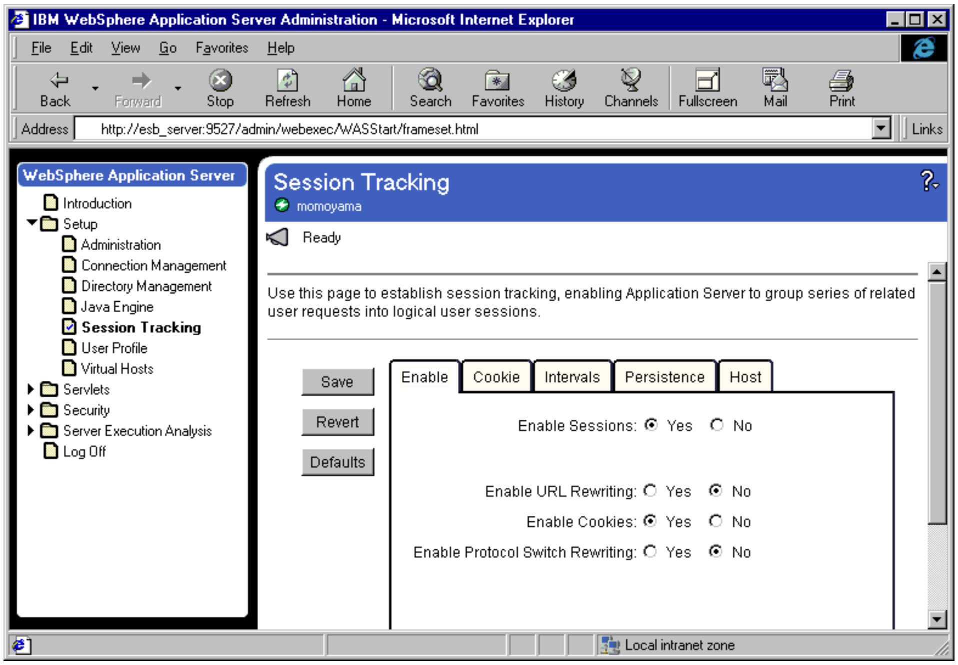
To manage sessions using cookies, both the server and client must be set so that cookies can be used. To use cookies, follow this procedure:

1. Start the WebSphere Application Server Administration tool.

2. Select **Setup -> Session Tracking** to display the Session Tracking page as shown in Figure 49 on page 153.

3. Click the **Enable** tag.

4. Set **Enable Cookies** to “on”.


IBM WebSphere Application Server Administration - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print
Address http://esb_server:9527/admin/webexec/WASStart/frameset.html
Links
WebSphere Application Server
Introduction
Setup
Administration
Connection Management
Directory Management
Java Engine
Session Tracking
User Profile
Virtual Hosts
Servlets
Security
Server Execution Analysis
Log Off
momoyama
Ready
Save
Revert
Defaults
Enable
Cookie
Intervals
Persistence
Host
Local intranet zone

Session Tracking

Use this page to establish session tracking, enabling Application Server to group series of related user requests into logical user sessions.

Enable Sessions: ● Yes ○ No

Enable URL Rewriting: ○ Yes ● No

Enable Cookies: ● Yes ○ No

Enable Protocol Switch Rewriting: ○ Yes ● No

Figure 49. Session Tracking Cookies setting

5. Click the **Save** button to save it.

6. Open **Properties** in the Web browser on the client side. Set it so cookies can be used.

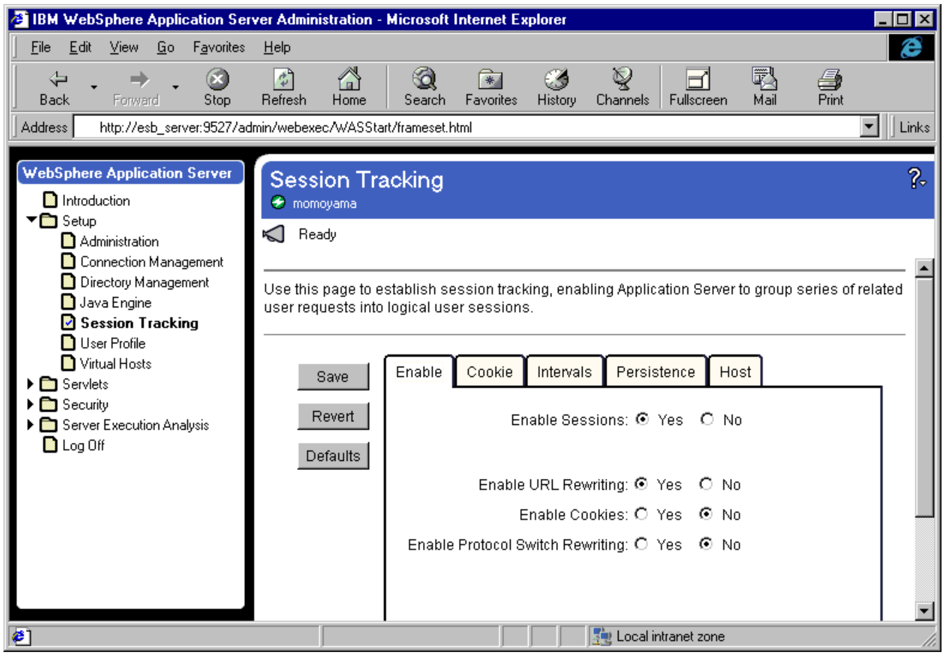
6.4.3.3 Using the rewriting function

It is possible to maintain the session on the browser that is not compliant to the cookies by using a URL re-write function. When using this function, it must be written as shown in the following JSP file to obtain the part specifying the ESB servlet using getURL() method of the servlet bean. When the URL rewrite function and the getURL() method are used, the session ID is added to the URL of the SvSessionServlet. The session is maintained by using this procedure for calling the next ESB servlet.

```
<form method="get" action="<%= SvServletBean.getURL() %>">
<input type="hidden" name="bean_name" value="myBean">
<input type="hidden" name="jsp_name" value="/output.jsp">
<input type="submit" name="delete" value="Exit">
</form>
```

Complete these steps to use the rewrite function:

1. Start the WebSphere Application Server Administration tool.
2. Select **Setup -> Session Tracking** to display the Session Tracking page as shown in Figure 50.
3. Click the **Enable** tab.
4. Set **Enable URL Rewriting** to “on”.
5. Set **Enable Cookies** to “off”, or disable the cookies on the client.


IBM WebSphere Application Server Administration - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print
Address http://esb_server:9527/admin/webexec/WASStart/frameset.html
Links
WebSphere Application Server
Introduction
Setup
Administration
Connection Management
Directory Management
Java Engine
Session Tracking
User Profile
Virtual Hosts
Servlets
Security
Server Execution Analysis
Log Off
Session Tracking
momoyama
Ready
Save
Revert
Defaults
Enable
Cookie
Intervals
Persistence
Host
Local intranet zone

Use this page to establish session tracking, enabling Application Server to group series of related user requests into logical user sessions.

Enable Sessions: ● Yes ○ No

Enable URL Rewriting: ● Yes ○ No

Enable Cookies: ○ Yes ● No

Enable Protocol Switch Rewriting: ○ Yes ● No

Figure 50. Session Tracking enable URL rewriting

6. Click the **Save** button to save it.
7. Select **Servlet -> Configuration** to display the Servlet Configuration page.
8. Select **SvSessionServlet** from the Servlet Names field.
9. Select the **url_rewriting** property from the Servlet Properties field. Then, click the **Modify** button. If this property is not available, add it.
10. Enter `1` in the Value field. Then, click the **Modify** button.
11. After clicking the **Save** button, click the **Unload** and then **Load** buttons.

6.5 Application creation hints

This section offers helpful hints that you can use when creating the application:

- The ESB servlet has a function that pools used client objects and holds the connection with ESB Runtime. However, if ESB Runtime is stopped while client objects are pooled, a mismatch occurs and it does not operate normally. In case ESB Runtime stops periodically, we recommend that you set the max_pool property to `-1` (Do not pool). Also, if ESB Runtime is stopped while the client objects are being pooled, reload the ESB servlet.
- When using JSP to create client objects, the client objects are not deleted even if the Web browser terminates. Use the WebSphere session time-out function. Or, set the function for deleting objects in HTML or JSP to operate so that it explicitly deletes them from the Web browser.
- When using JSP to recursively call ESB servlets, we recommend that you use the **getURL()** method of the ESB servlet bean instead of explicitly specifying the URL of the ESB servlet. It is possible to use both of the session managements using the cookies and the URL rewriting function.

  ```
  <form method="get" action="<%= SvServletBean.getURL() %>">
  ```

- When using JSP to call ESB servlets, we recommend that you use the **getServer()** method of the ESB servlet bean. You can also use the **getServerName()** method of the client bean, instead of explicitly specifying the server_name parameter. By doing so, when ESB Runtime moves to another server or when the host name is changed, you do not need to make any corrections.

  ```
  <input type="hidden" name="server_name" value="<%=
  SvServletBean.getServer() %>">
  ```

- Enabling the Remote setting (setting the remote_control property to “1”) in the employment of system can result in trouble by a malicious user. Therefore, we strongly recommend that you disable the Remote setting. That is, set the remote_control property to “0” in such situations.
- When the trace level is increased, the performance deteriorates. Under those circumstances, we recommend that you set the trace level to "0". That is, set the trace_level property to “0”.
- In JSP programming, the loop can be created by using the argument of the client bean. For example, assume that the second argument of the method is an array element and that the first argument is its number of elements. It appears as shown here:

```
<% for(int i; i < new Integer(myBean.getArgument(1)).intValue(); i++)
{
%>
<b>Element[<%= i %>] = <% myBean.getArgument(2, i) %></b>
<%
}
%>
```

- After calling a procedure in JSP programming, set the delete_object parameter to `1` to delete the client bean used previously.

```
<form method="get" action="<%= SvServletBeangetURL() %>">
<input type="hidden" name="delete_object" value="1">
<input type="hidden" name="bean_name" value="myBean">
<input type="hidden" name="jsp_name" value="/endJSP.jsp">
<input type="submit" name="delete" value"Exit">
</form>
```

- ESB applets do not support Variant types and two dimensional or more arrays as arguments of procedures and return values. Consequently, when creating applications using a Web client, design on the server application side that does not use such arguments, or return values must be adopted.

Chapter 7. Accessing from and to Notes and Domino

ESB can use the functions of Lotus Notes and Lotus Domino by using the Notes classes of LotusScript. ESB can also access data on the Notes databases, create and save documents, and use ESB functions from the server agent of Lotus Domino. This chapter explains ESB programming using Notes and Domino through the development of a sample program.

7.1 Outline

This section describes the relationship between ESB and Notes and Domino, as well as the prerequisites for developing a sample application.

7.1.1 ESB and Notes or Domino

A Notes client is a typical ESB client. The primary reason for this is the high degree of affinity with ESB because it has the same language as LotusScript for its platform. The same may also be said for Lotus Domino. There is no problem for ESB to invoke an application on Lotus Domino or Lotus Domino invokes an ESB server application. The Lotus Software eXtension (LSX) makes this mutual affinity possible. Loading the ESB client LSX by Lotus Domino and loading Notes LSX by the ESB Runtime enables them to use each other's functions.

In the sense that Lotus Domino bears the second tier of the three-tier client-server application, the properties are similar to ESB. Consequently, when ESB is collaborating with Lotus Domino, it is essential to use both system’s advantages and balance the load carefully. Performance efficiency and network traffic must be considered as well. Generally, data processing is apportioned to the ESB, while data display and document creation is allocated to Lotus Domino.

The creation and modification of documents on a Notes database should be done by Lotus Domino. This way, efficient performance can be obtained rather than performing such operations by ESB. It is also important to access the database locally, unless it requires exclusive control.

On the contrary, when the application processes data from a RDB, such as the calculation of annual interest or combined data with other RDBs, then the processed data is sent to the client. The process logic should be executed in ESB because it offers better performance, as well as flexibility and maintainability in actual deployment.

7.1.2 Prerequisites

You need the following prerequisites to use Notes or Domino from ESB, or to use ESB from Domino respectively:

- When using Lotus Notes or Lotus Domino functions from ESB

  When ESB invokes the function of Lotus Notes or Domino (in other words, the Notes class of LotusScript is called), it is required that Lotus Notes R4.6 or higher or Lotus Domino R4.6 is up and running with ESB Runtime on the same node. These Notes and Domino programs do not need to be up and running when running an ESB application. Figure 51 shows an ESB project accessing a Domino server.

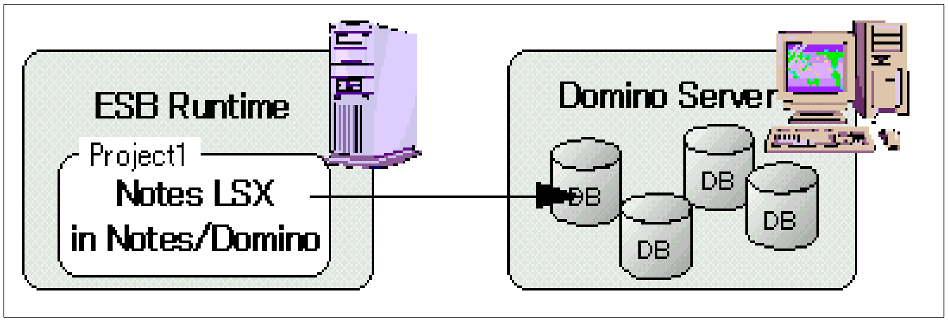


Figure 51. Accessing a Domino server from an ESB project

> **Note**
>
> ESB supports Notes R4.5 as a client. However, since Notes LSX of Notes Domino R4.5 is not thread-safe, it should not be used for collaboration with ESB Runtime for a backend connection.

- If Lotus Notes or Lotus Domino invokes an ESB function

  When using an ESB function from Lotus Notes or Domino, ESB Client Enabler must be installed on the node where Notes or Domino is running. Figure 52 shows a Domino agent accessing an ESB project.

> **Hint**
>
> ESB Client Enabler is automatically installed on a machine where ESB Runtime or ESB Developer was installed. It is not necessary to do a re-installation.

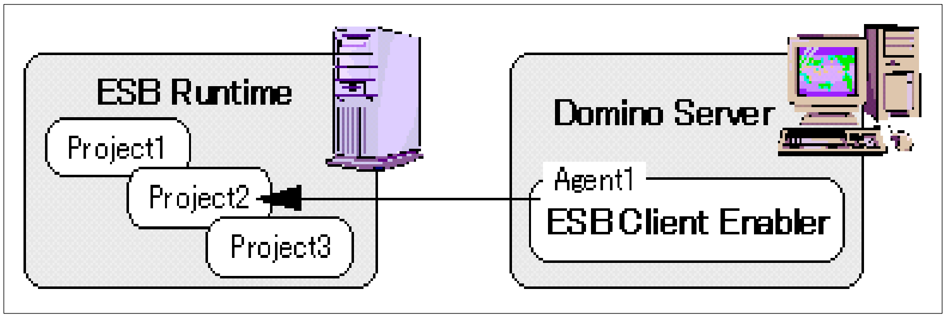


Figure 52. Accessing an ESB project from a Domino agent

7.2 Connecting to Notes or Domino

While creating a sample program, we explain how to use Notes or Domino from an ESB server application.

7.2.1 Flow of the sample program

The scenario of a sample program is that, a name entered by the client is searched for using a Notes database to obtain telephone numbers. The result is returned to the client using mail. The specific program flow is shown here:

1. Request from the client.

   The client invokes a request for a telephone number to the project being run on ESB Runtime. The project name is chap07, the Published class name is AddressBook, and the name of the function that processes the client request is getPhoneNumber. The client passes the name of the person to be searched and the mail address for sending the result as arguments to the getPhoneNumber function.

2. Search the database.

   The Address Book database (Domino Directory in R5) on Lotus Domino is searched for the telephone number based on the name of the person. The getPhoneNumber function that received the request from the client first accesses the Address Book database. Then, it compares the fields with the documents saved in the database for the name data of the argument, and thus obtains the telephone numbers of the target persons.

3. Create the results document.

   The database is used for result document creation, where the database is created from the discussion template, which has the name chap07.nsf. First, chap07.nsf is accessed, and a new document is created. Then, the results of step 2 are entered into a subject and body text of the document.

4. Send the document.

   We send the document created in step 3 to the client by using mail.

5. Save the document.

   The sent document is saved on the chap07.nsf database as a sent record. The process of the getPhoneNumber function is completed. Figure 53 shows the entire process flow of this sample program.

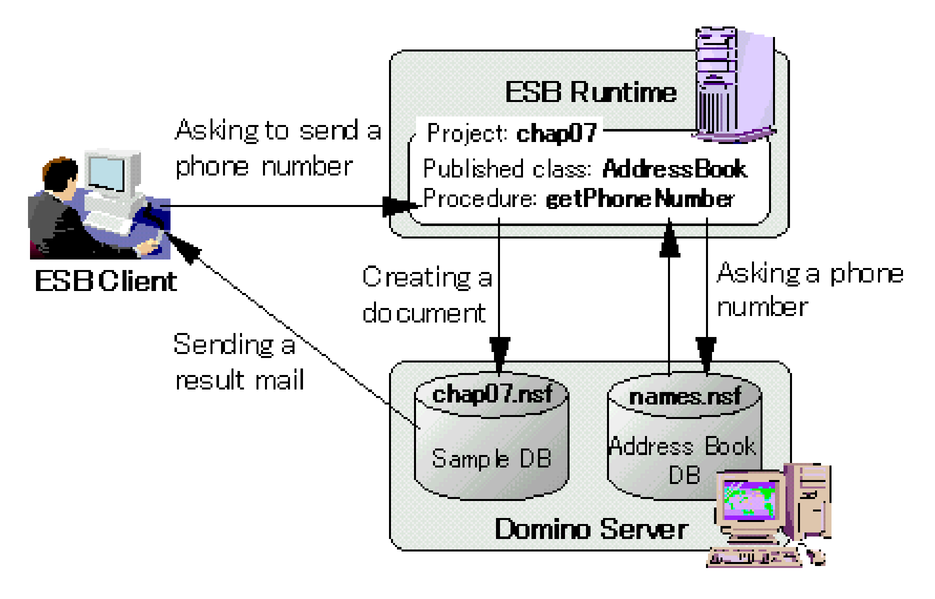


Figure 53. Process flow of the sample program

Hint

If a suitable Lotus Domino server cannot be provided on another node, the development and testing of this sample application can be performed with a local Lotus Domino of the same node where the ESB Runtime is running. In this case, the mail server of Notes ID used in the sample program must be up and running.

7.2.2 Preparation

You should run the following process before actually creating a sample application:

1. Confirm the Address Book.

   Confirm the file name of the Address Book located on Lotus Domino. Usually, it is names.nsf.

2. Create a sample database to use in creating a document.

   a. Start **Lotus Domino**.

   b. Select **File -> Database -> Create**.

   c. Enter a server name that is mounted on the Address Book database in the Server text box.

   d. Enter `Chap07` in the Database Name text box.

   e. If the template list is not displayed, select **Template Server**, and select the server where the template is saved.

   f. Select **Discussion Template** (discuss4.nsf or discsw50.nsf).

   g. Deselect the **Inherit future design changes** check box.

   h. Click **OK**.

   i. After the database is created, close the policy document where it is displayed.

3. Prepare a Notes ID for connecting to the Notes database.

   Copy the Notes ID file onto the ESB Runtime machine. This Notes ID must be authenticated by the target database connection (Address Book and the database created in step 2).

> **Note**
>
> While a project collaborating with Notes is running, the Notes ID that is used by ESB Runtime should never be used by Notes client.

4. Prepare for creating a server application as explained here:

   a. Start ESB IDE.

   b. Select **Create -> Interface/Class**.

   c. Enter the Class Name: `AddressBook`. Then, select **Published** for the class keyword.

d. Click the **Member Procedure** tab. Then, add the following member function:

```
Public Function getPhoneNumber(firstName As String, _
   lastName As String, sendTo As String)
```

e. Click the **OK** button to create a class.

f. Save the script file name as `AddressBook.lss` and the project file name as `chap07.lsp`.

7.2.3 Creating a sample application

Now, create a sample application program. Start the IDE and open the file chap07.lsp created in the previous section. Then, follow these steps:

1. Load Notes LSX.

   To use the Notes class in ESB, first you have to use the UseLSX statement to load Notes LSX. Add the following code to the (Options) script of the (Globals) object:

   ```
   UseLSX "*Notes"
   ```

   Note

   Special settings are required to use Notes LSX in AIX. Before proceeding to the next step, refer to the *Lotus ESB User's Guide* and read the section “Using Notes Classes” in Appendix A on page A-4.

2. Connect to the Lotus database.

   There are several types of connecting methods for the Notes database. However, the most generic one is explained.

   a. Create a NotesSession class object. Add the following code:

   ```
   Set session = New NotesSession
   ```

   b. Read the Notes ID file for connecting to the Notes Domino Server. Then, set the password. For reading (switching) the Notes IDs file, it is helpful to use the NotesRegistration class:

   ```
   Set registration = New NotesRegistration
      userName = registration.SwitchToID("d:\lotus\notes\data\user.id",_
         "Password")
   ```

   c. The first argument of the SwitchToID method is the path to the Notes ID file. The second argument is the password for that ID. You enter the password as simple text, so you should be careful when handling the source file being developed.

Hint

The NotesRegistration class has been provided since Lotus Notes R4.6. For those of you who cannot use this class, refer to the *Lotus ESB User's Guide*, and read the section "Connecting with a Notes Database" in Appendix A on page A-2.

d. Use the GetDatabase method of the NotesSession class to connect to the database. This method establishes a database connection at the same time it obtains the NotesDatabase object, and permits the code to be made succinctly.

```
Set addressDB = session.GetDatabase("esb/Lotus", "names.nsf" )
```

e. Specify the server name in the first argument of the GetDatabase method and specify the database name in the second argument. When connecting to a local database, make the server name an empty string.

Hint

You can also obtain the Address Book database by using the AddressBooks property of the NoteSession class.

3. Search for documents and reference the field values.

a. Let us try to search for a document containing the names that were entered and to obtain the telephone numbers. Once we are connected to the database, the subsequent programming is the same as with other Notes applications.

The Person form documents, where individual information is stored, are displayed in the People view. Obtain this view first:

```
Set view = addressDB.GetView("People")
```

b. It retrieves the document contained in the view one-by-one and checks whether the entered first name and last name match the values stored in the field. If either matches, it obtains the value in the OfficePhoneNumber field where the telephone number is stored, creates a document for the mail body text, and stores it in the string variable:

```
Set doc = view.GetFirstDocument
  While Not(doc Is Nothing)
    If doc.FirstName(0) = firstName Then
      If doc.LastName(0) = lastName Then
```

```
        bodyText = firstName & " " &lastName & "'s phone number is "
&_
         doc.OfficePhoneNumber(0) & "."
      End If
    End If
    Set doc = view.GetNextDocument(doc)
Wend
```

c. Hereafter, the Address Book is not used. Disconnect the database:

```
Set addressDB = Nothing
```

Hint

It is possible to use the GetDocumentByKey method of the NotesView class or the Search method of the NotesDatabase class for coding more succinctly and efficiently. However, in this redbook, the previous codes are used to absorb the differences between platforms and Notes releases.

4. Create a document.

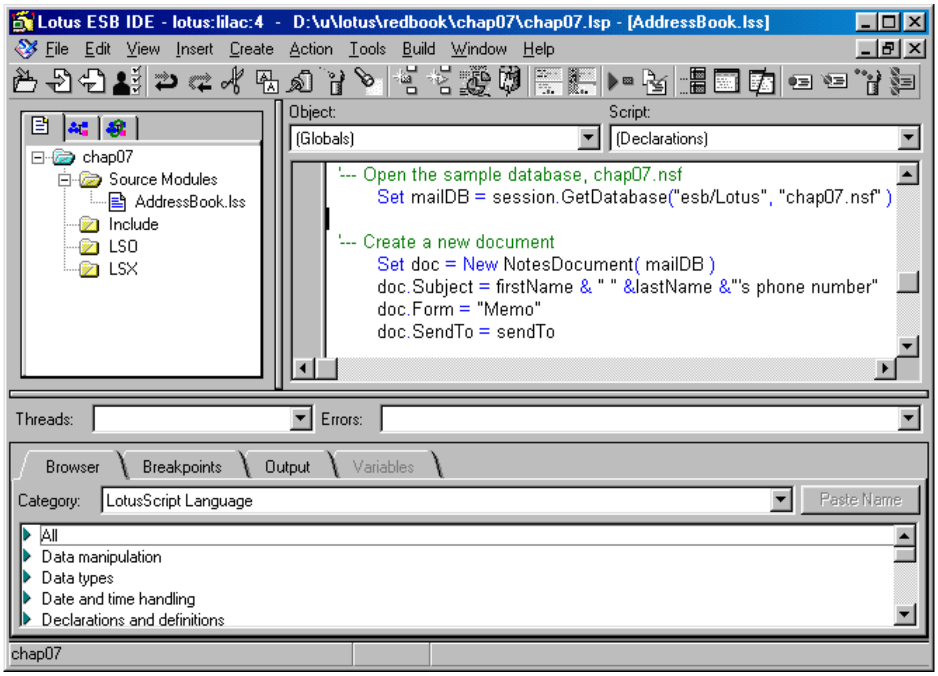
a. Connect to the database (chap07.nsf) that was prepared for creating the mail document:

```
Set mailDB = session.GetDatabase("esb/Lotus", "chap07.nsf")
```

b. Create a document. Since this document will be sent as mail later, set the form to “Memo” and set the result in the Subject field and Body field, where the result was obtained from the previous search operation:

```
  Set doc = New NotesDocument( mailDB )
  doc.Form = "Memo"
  doc.Subject = firstName & " " &lastName &"'s phone number"
  If bodyText = "" Then
    doc.Body = firstName & " " &lastName &_
      " is not registered the address book."
  Else
    doc.Body = bodyText
  End If
```

Figure 54 shows server program creation on ESB IDE.


Lotus ESB IDE - lotus:lilac:4 - D:\u\lotus\redbook\chap07\chap07.lsp - [AddressBook.lss]
File Edit View Insert Create Action Tools Build Window Help
chap07
Source Modules
AddressBook.lss
Include
LSO
LSX
Threads:
Errors:
Browser
Breakpoints
Output
Variables
Paste Name
All
Data manipulation
Data types
Date and time handling
Declarations and definitions
chap07

Object: (Globals)

Script: (Declarations)

```
'--- Open the sample database, chap07.nsf
     Set mailDB = session.GetDatabase("esb/Lotus", "chap07.nsf" )

'--- Create a new document
     Set doc = New NotesDocument( mailDB )
     doc.Subject = firstName & " " &lastName &"'s phone number"
     doc.Form = "Memo"
     doc.SendTo = sendTo
```

Category: LotusScript Language

Figure 54. Server program creation

5. Send mail.

   Send the document created as mail. Set the mail address entered in the sendTo field and call Send method of the NotesDocument class:

   ```
     doc.SendTo = sendTo
     Call doc.Send( False )
   ```

   The first argument of the Send method is a parameter that indicates whether the form is stored and sent along with the document. Specify `False` since you do not need to send a form at this time. The sender of the mail will be the Notes user ID used for the database connection.

6. Save the documents.

   Since this database does not contain a Memo form, change the form name, and then save it.

   ```
   Doc.Form = "Main Topic"
   Call doc.Save( False, False )
   ```

   The first argument of the Save method is a parameter that indicates the process contention with another user. The second argument is a parameter that indicates whether to save it as a reply document. In this case, the main topic was created as a new document, so specify `False` for both arguments.

7.3 Connecting from Notes or Domino

Up to now, the Notes or Domino functions are called from ESB. Now create an agent for connecting to the ESB server from Domino. Since this means connecting to the ESB server from another process, it is the same as a client application from the view of an ESB server.

An agent will be created on the Notes database used in the previous section for saving the mail. It uses the server application that was also developed in previous section.

Note

The Notes client AIX version is not formally supported as an ESB client. It only supports connection from Domino agents. When connecting to ESB Runtime from a Domino agent, you should set the ESB communication environment according to the following procedure prior to starting Lotus Domino:

1. Log on with the user running Lotus Domino.
2. Edit **hpwclset.ksh** (in case of a C shell, edit **hpwclset.csh**) located in the /usr/lpp/esb.cb/bin directory. Set the IP address of TCP/IP to the environment variable HOSTNAME.
3. Run **hpwclset.ksh** within the session that starts Lotus Domino.
4. Start **Lotus Domino**.

Follow this process to connect from Notes or Domino:

1. Start **Domino Designer**, and open **chap07.nsf** on Lotus Domino.
2. Select **Create -> Design -> Agent**.
3. Enter `Send Phone Number` in the Name field, and check the **Shared Agent** check box.
4. In "Which document(s) should it act on?", select **Run once (@Commands may be used)**.
5. Select **LotusScript** in the Run drop-down box. This completes the preparations for creating an agent.

6. Using the ESB server application, the ESB client LSX must be loaded. Enter following codes in the (Options) script:

```
Uselsx "*SsClink"
```

7. Create a Published class object. Enter the following Initialize subroutine:

```
Dim ORSObj As New SsClink
Dim ESBObj As Variant

Set ESBObj = ORSObj.CreateObject("chap07.AddressBook, _
   node=myServer")
```

   Change the host name to be specified for the node in conformity with the actual environment. You can also set the ConnType property or the UserID property in the same way as other client applications.

8. Call the getPhoneNumber function of the server application. Modify the three arguments of the getPhoneNumber function in conformity with the actual environment.

```
ret = ESBObj.getPhoneNumber("John", "Smith", _
   "Kaori Namba/esb/Lotus")
```

   Once the calling of the function is completed, delete the Published class object:

```
Set ESBObj = Nothing
Set ORSObj = Nothing
```

9. Describe the error handling, after defining of the ORSObj variable in the getPhoneNumber function. Add the following two lines:

```
On Error Goto ErrorHandler
On Event RuntimeError From ORSObj Call EventHandler
```

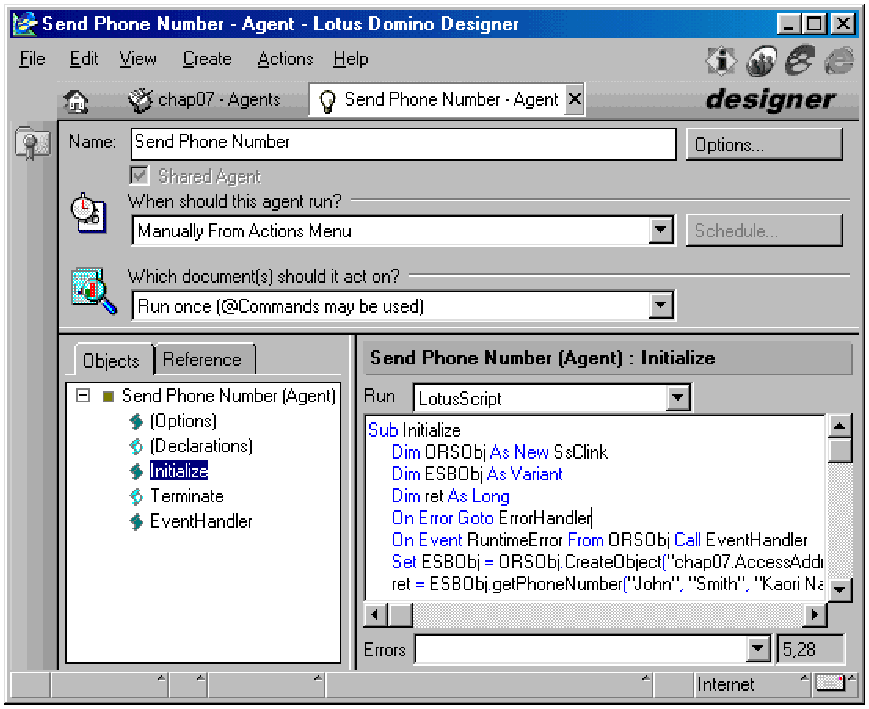
   Describe the error handling routine. Enter the following code at the bottom of the Initialize subroutine:

```
ErrorHandler:
  Print "ErrorHandler: " & Cstr(Err)
  Print "ErrorHandler: " & Error
  Exit Sub
End Sub
```

   Now, enter the handling routine for the RuntimeError event. Enter the following code in the (Declarations) script:

```
Sub EventHandler(ORSObj As SsClink, errorCode As Long, _
  description As String)
  Print "EventHandler:" & Cstr(errorCode) & " or 0x" & Hex(errorCode)
  Print "EventHandler:" & description
End Sub
```

10. Save the agent. Figure 55 shows the Domino agent creation on Domino Designer.


Send Phone Number - Agent - Lotus Domino Designer
File Edit View Create Actions Help
chap07 - Agents
Send Phone Number - Agent
designer
Options...
Schedule...
Objects
Reference
Send Phone Number (Agent)
(Options)
(Declarations)
Initialize
Terminate
EventHandler
Internet

Name: Send Phone Number

☑ Shared Agent

When should this agent run? Manually From Actions Menu

Which document(s) should it act on? Run once (@Commands may be used)

Send Phone Number (Agent) : Initialize

Run LotusScript

```
Sub Initialize
    Dim ORSObj As New SsClink
    Dim ESBObj As Variant
    Dim ret As Long
    On Error Goto ErrorHandler
    On Event RuntimeError From ORSObj Call EventHandler
    Set ESBObj = ORSObj.CreateObject("chap07.AccessAddı
    ret = ESBObj.getPhoneNumber("John", "Smith", "Kaori Na
```

Errors 5,28

Figure 55. Domino agent creation

7.4 Summary

This section describes the entire code that was created in this chapter and how to execute this code.

7.4.1 Entire code

The entire code of the server application is as follows:

(Globals) - (Options)

```
Option Public
Option Explicit
'--- Load Notes LSX
Uselsx "*Notes"'NT
' Uselsx "/opt/lotus/latest/ibmpow/liblsxbe_r.a"'AIX
```

(Globals) - (Declarations)

```
Published Class AddressBook

Public Function getPhoneNumber(firstName As String, _
  lastName As String, sendTo As String) As Long
  On Error Goto ErrorHandler
  Dim session As NotesSession
  Dim registration As NotesRegistration
  Dim addressDB As NotesDatabase
  Dim mailDB As NotesDatabase
  Dim view As NotesView
  Dim doc As NotesDocument
  Dim bodyText As String
  Dim userName As String

  GetPhoneNumber = -1
  bodyText = ""

  '--- Get a new session
  Set session = New NotesSession

  '--- Change a Note ID file
  Set registration = New NotesRegistration
  userName = registration.switchToID("D:\lotus\notes\data\user.id",_
    "Password")

  '--- Open the Address Book database, names.nsf
  Set addressDB = session.GetDatabase("esb/Lotus", "names.nsf" )
  Set view = addressDB.GetView("People")
  '--- Search a person whose name is "fisrtName lastName"
  Set doc = view.GetFirstDocument
  While Not(doc Is Nothing)
    If doc.FirstName(0) = firstName Then
      If doc.LastName(0) = lastName Then
        bodyText = firstName & " " &lastName & "'s phone number is " & _
         doc.OfficePhoneNumber(0) & "."
      End If
    End If
    Set doc = view.GetNextDocument(doc)
  Wend
  '--- Close the Address Book database
  Set addressDB = Nothing

  '--- Open the sample database, chap07.nsf
  Set mailDB = session.GetDatabase("esb/Lotus", "chap07.nsf" )

  '--- Create a new document
  Set doc = New NotesDocument( mailDB )
  doc.Subject = firstName & " " &lastName &"'s phone number"
  doc.Form = "Memo"
  doc.SendTo = sendTo
  If bodyText = "" Then
      doc.Body = firstName & " " &lastName &_
       "is not registered the address book."
    Else
      doc.Body = bodyText
    End If
  '--- Send the document
  Call doc.Send( False )
  '--- Change the form and save the document
  doc.Form = "Main Topic"
  Call doc.Save(False,False)
  Set doc = Nothing
  Set maildb = Nothing

  Set session = Nothing
  Set registration = Nothing
  GetPhoneNumber = 0
  Exit Function

'--- Error handling routine
ErrorHandler:
    Print "Error!! code = " & Cstr(Err) & " : " & Error$
    GetPhoneNumber = Err
    Exit Function
End Function

End Class
```

The code created for the agent is shown in the following example. It will operate even if you enter it in the event handling routine within the form.

```
 '--- Load ESB client LSX
Uselsx "*SsClink"'NT

Sub Initialize
  Dim ORSObj As New SsClink
  Dim ESBObj As Variant
  Dim ret As Long

  On Error Goto ErrorHandler
  On Event RuntimeError From ORSObj Call EventHandler

  '--- Create a Published class object
  Set ESBObj = ORSObj.CreateObject _
    ("chap07.AddressBook, node=myServer")

  '--- Call getPhoneNumber function
  ret = ESBObj.getPhoneNumber("John", "Smith", _
    "ESB Lotus/esb/Lotus")
  If ret <> 0 Then
    MsgBox "Some error has occurred on ESB Runtime."
  End If

'--- Delete the Published class object
  Set ESBObj = Nothing
  Set ORSObj = Nothing
  Exit Sub

'--- Error handling routine
ErrorHandler:
  Print "ErrorHandler: " & Cstr(Err)
  Print "ErrorHandler: " & Error
  Exit Sub
End Sub

'--- RuntimeError event handling subroutine
Sub EventHandler(ORSObj As SsClink, errorCode As Long, _
  description As String)
  Print "EventHandler: " & Cstr(errorCode) & " or 0x" & Hex(errorCode)
  Print "EventHandler: " & description
End Sub
```

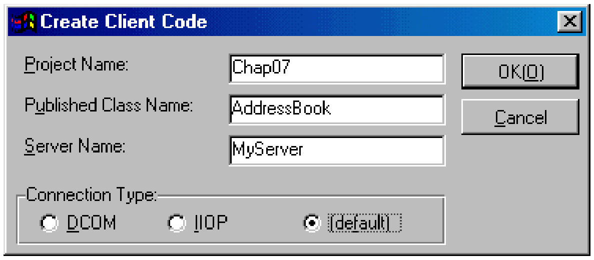
7.4.2 Running the sample application

Run the sample application program that you just created.

7.4.2.1 Connecting to Notes or Domino

In 7.2.3, “Creating a sample application” on page 162, you only created a server application. In this section, you use the IDE as a client to run the application. Because it includes a client code creation tool, you can easily create client applications in the IDE.

1. Verify that Lotus Domino is operating where the Address Book and chap07.nsf are located.
2. Run project **chap07** on the ESB Runtime side.
3. Start the IDE, and create a new project.
4. Select **Create-> Client Code**.
5. Respectively, enter `Chap07` and `AddressBook` for the Project Name and the Published Class Name. Enter the host name where ESB Runtime is operating in the Server Name. Click the **OK** button. A client code is created in the Initialize subroutine (Figure 56).


Create Client Code

Project Name: Chap07

Published Class Name: AddressBook

Server Name: MyServer

Connection Type: ○ DCOM ○ IIOP ● (default)

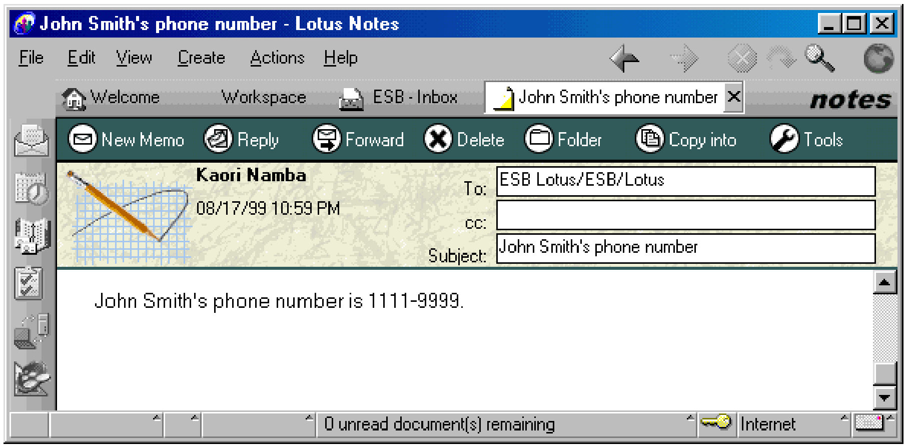
Figure 56. Create Client Code dialog box

6. Add the following code after creating a Published class object to call the getPhoneNumber function:

```
Dim ret As Long
ret= obj.getPhoneNumber("John", "Smith", "ESB Lotus/esb/Lotus")
```

   Change the three arguments as appropriate in conformity with the actual environment.

7. Select **Build -> Run Project**. When it ends normally, a new document is created in the chap07 database. The mail reporting the result (Figure 57 on page 172) is delivered to the mail address specified within the client code.


John Smith's phone number - Lotus Notes
File Edit View Create Actions Help
Welcome
Workspace
ESB - Inbox
John Smith's phone number
notes
New Memo
Reply
Forward
Delete
Folder
Copy into
Tools

Kaori Namba
08/17/99 10:59 PM
To: ESB Lotus/ESB/Lotus
cc:
Subject: John Smith's phone number

John Smith's phone number is 1111-9999.

0 unread document(s) remaining
Internet

Figure 57. Mail reporting a result

7.4.2.2 Connection from Notes or Domino

To run the application you created in 7.2.3, “Creating a sample application” on page 162, complete the following steps:

1. Verify that Lotus Domino is operating where the Address Book and chap07.nsf are located.
2. Run project **chap07** on the ESB Runtime side.
3. Open **chap07.nsf** on Lotus Domino.
4. Select **Action -> Send Phone Number**.

When the application ends normally, a new document is created in the chap07 database. The mail reporting the result is delivered to the mail address specified within the agent.

7.4.3 Application creation hints

This section describes hints to develop an application using Notes or Domino:

- **Shifting Notes or Domino applications**

  It is easy to shift Notes or Domino applications created with LotusScript to ESB server applications. You can readily operate it as an ESB server application, even if the loading of Notes LSX is described. However, the following precautions are in order:

  – Unlike Notes, there is no “Current” concept. Therefore, the CurrentDatabase properties of the UI type class (NotesUIDocument and so on) and of the NotesSession class cannot be used. When these are used, they must be appropriately rewritten.

  - When developing a program, such as updating documents, exclusive control is required, so that contention never happens due to the ESB multi-thread function. We recommend that you use the LSServer class.

- **Accessing a local database**

  It is possible to connect to a local database using the GetDatabase method of the NotesSession class. In this case, specify the Notes or Domino data directory as a base reference for specifying the database with a relative path. Be careful because it is not an ESB Runtime run directory. An absolute path specification case does not apply.

- **Unable to connect to a database**

  Even though the server name and Notes ID authentication have been set correctly, you may be unable to connect a database. The reason may be because the notes.ini where ESB Runtime is running is not correct (location setting and so on). In such cases, start Notes, authorize the access of target database for the Notes ID used in ESB, and specify the correct location. Then, exit Notes.

- **Searching a field**

  In this chapter, the operational flow is for a field search. One document is opened at a time, and then the field values are compared. However, due to the fact that this method results in redundant script descriptions, performance problems may occur if the number of documents has increased. Generally, you can search more efficiently by creating a view for searching (one where the key of the search target is displayed in the first column). Then, use the GetDocumentByKey method of the NotesView class. Also, consider using the Search method of the NotesDatabase class. Remember that the @ function for the Search method argument cannot be used in the ESB Runtime for AIX.

Chapter 8. Connecting to a relational database

The ESB can connect and link in real time to a variety of database management systems (DBMS) and collaborate with those systems. This chapter explains several methods for accessing the relational databases (RDB), including DBMS.

8.1 Overview

The following items are supported as interfaces for database access from ESB:

- **Domino Connectors** and **LSX LC**: Domino Connectors to various backend systems executed with LotusScript code using the LotusScript Extension for Lotus Domino Connectors (LSX LC)
- **Custom LSXs**: LotusScript Extensions for systems such as LS:DO, DB2 LSX, and MQSeries LSX
- **CLI**: The native interface of DB2
- **OCI**: The native interface of Oracle databases
- **ODBC**: An interface for connecting to various databases

With ESB, you can select the optimum connection configuration based on the developer's needs and skill for each of the methods, which are explained in the following sections.

8.1.1 Differences between using Domino Connectors with the LSX LC and other LSX, CLI, and OCI

With Domino Connectors used with the LSX LC (which has a standard API access to backend systems), the development is straightforward. LotusScript Extensions (LSXs) that are designed as specialized DBMS, specifically for CLI and OCI, and LSXs that have an original API are more complicated. Consequently, there is no need to remember the API that is determined for a specific DBMS. There is almost no overhead incurred when shifting to a different DBMS. Also, because the initialization procedure required in normal programming is processed internally, the developer does not need to perform any complex processing for initialization or management of variables.

On the other hand, because the CLI and OCI achieve more native access, a skilled developer can make very small settings for precise processing. Refer to 8.4, “Performance comparison” on page 199, for a comparison of both.

Figure 58 shows the relationship between the ESB, Domino Connectors, and the systems to which they can be connected.

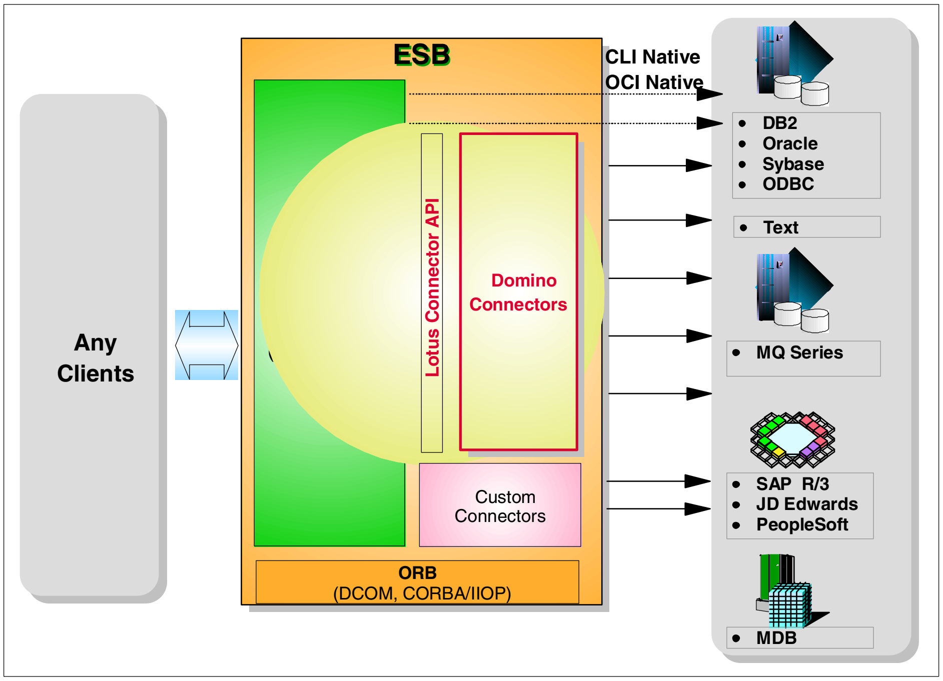


Figure 58. Generic ESB backend connectivity

8.2 Domino Connectors and the LSX LC

This section explains how to use the Domino Connectors with the LSX LC.

8.2.1 What a Domino Connectors are

Domino Connectors are dynamic link (.dll) or shared library files. They provide connecting and authenticating functionality with backend systems. To access these .dlls from LotusScript, a special set of extensions has been created called the LotusScript Extension for Lotus Domino Connectors. The same classes, methods, and properties of the Connector LSX can be used with any connector .dll file, regardless of the backend system that is desired. The following list shows the Domino Connectors that are currently available:

- DB2/UDB
- Notes
- ODBC

- Oracle
- Sybase
- File/Text
- SAP
- PeopleSoft
- JD Edwards
- Oracle applications

In this section, we incorporate specific examples for the connection to DB2/UDB and Oracle. Figure 59 shows the access from ESB and Domino to DBMS using Domino Connectors.

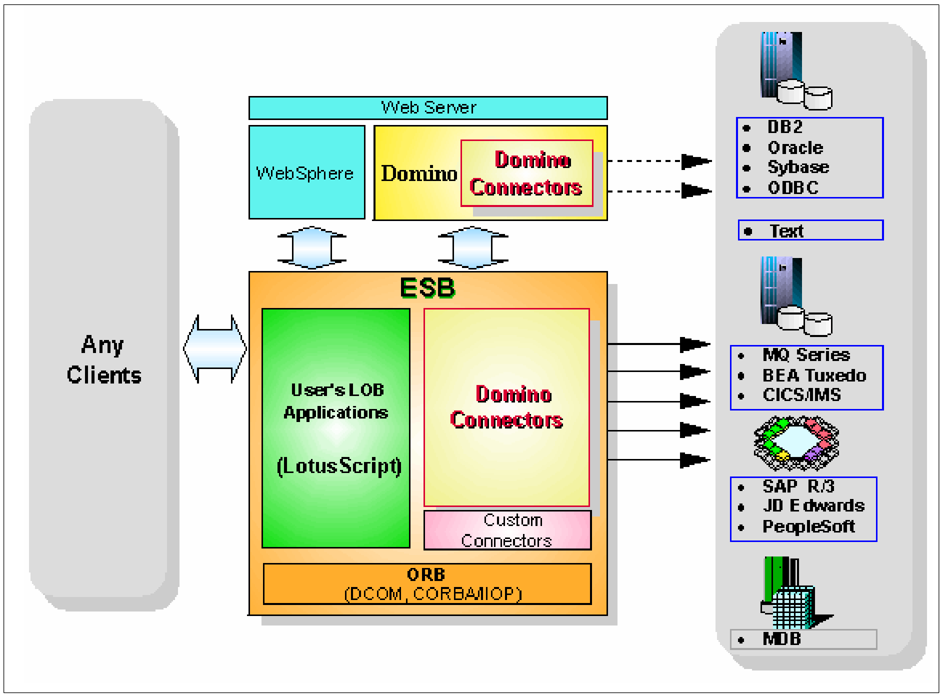


Figure 59. RDMS access from ESB and Domino using Domino Connectors

8.2.2 Development using Domino Connectors, Lotus Connector LSX

The following sections explain the step-by-step development using Domino Connectors with the LSX LC.

8.2.2.1 Loading the LSX LC

The IDE requires that you load the LSX LC to use the Domino Connector functions. Enter the (Options) script of the (Globals) object to load this LSX LC as shown here:

```
Uselsx "*lsxlc"
```

When ESB is installed, the `lsxlc` keyword is registered, which enables its use.

8.2.2.2 Classes that are provided

The following types of classes are provided in `lsxlc`:

- **LCSession**

  This class manages the Domino Connector running environment. When a runtime error has occurred, it holds the error information. It obtains the error information from this class object to perform error handling.

- **LCConnection**

  This class manages the connection with the database. This class method is used to perform collaboration with the DBMS for such operations as querying or updating data.

- **LCFieldlist**

  This class stores field groups to be queried and updated when a database is accessed. It facilitates data operations and linking by consolidating the individual fields to be processed.

- **LCField**

  This class is used in storing data. It can store all data types of the four classes described in the following bullet.

- **LCStream/LCNumeric/LCCurrency/LCDatetime**

  These classes are used for storing and the operation of a specific data type.

8.2.2.3 Connection pooling

The following process is normally performed for accessing databases:

1. Connect to the database.
2. Run business logic, such as database queries.
3. Disconnect from the database.

In complex cases, such as where the business logic performed in step 2 queries or updates large volume data encompassing multiple tables, step 2 occupies a high proportion of the overall processing. In the current situation, in most cases, connection pooling consumes even more time than running the business logic. Under the connection pooling mechanism, when an ESB application for connecting to a database is run, first (immediately after running the ESB application), an appropriate amount of connections are created. They are then allocated each time there is a connection request. Even when an application requests a disconnection from the database, it merely returns to the connection pool without actually disconnecting. For this reason, even if there is another connection request, it just allocates a pooled connection, which permits extremely fast connection processing. Figure 60 shows schematic view of connection pooling.

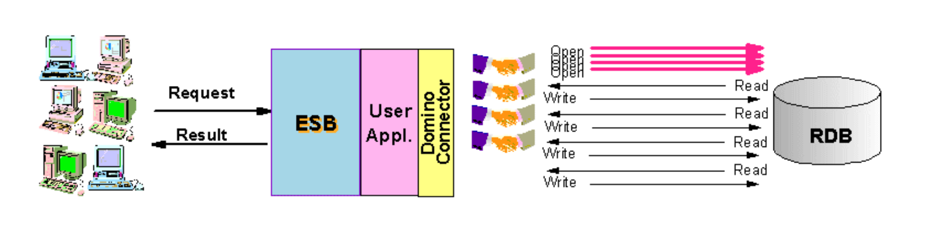


Figure 60. Connection pooling of Domino Connectors

The connection pooling procedure is further explained here.

To create a connection pool with LSX LC code, you must set the ConnectionPooling property to “True”. With this setting, the Domino Connector creates as many connections as necessary, and then ends. Although the Disconnect method is called, if the ConnectionPooling property has been set to True, the connection within the pool is not lost.

Normally, connection pooling should be created when you begin to run a project. Therefore, you should enter that logic into the Initialize subroutine of the ESB project. The following example shows how to create connection pooling for DB2:

```
Sub Initialize()
    Dim i As Integer
    Dim sesObj As New LCSession
    sesObj.ClearStatus
    sesObj.ConnectionPooling = True 'Enable connection pooling

    Dim conObjs() As LCConnection
    Const poolNum=10 'Number of pooling connection is 10
    Redim conObjs(poolNum)

    Print "Initializing pooling for " & Cstr (poolNum) & " connections"
    For i = 0 To (poolNum - 1)
        Set conObjs(i) = New LCConnection("db2")
        conObjs(i).Database = "TESTDB"
        conObjs(i).userID = "userid"
        conObjs(i).Password = "password"
        conObjs(i).Connect
    Next

    For i = 0 To (poolNum - 1)
        conObjs(i).Disconnect
    Next

    Print "Connection pool initialization complete."
    Exit Sub
End Sub
```

Here, the required number of connections and disconnections is made. Even after disconnections, the connections within the connection pool are not lost. Therefore, pooling has been executed. However, because connection pooling is optional, it does not necessarily have to be executed.

Be aware of the following points when running it:

- In DBMS, there is a fixed upper limit on the number of connections that the pooling cannot exceed.
- Because it occupies a fixed number of connections, there may be an impact when the database is accessed from another application. Consequently, you should check, in advance, the upper license and resource related limit value on the number of DBMS connections. Specify an upper limit value in the INI file when necessary.
- When the database stops momentarily and then restarts, the handle of the database being pooled will be invalidated. In other words, if an ESB-related application uses an invalidated handle to connect to the database, an error results. In this case, the connection pooling must be recreated.

In the LEI.INI file located in the ESB run directory, enter the following statement to set the maximum number of connection pools in the DBMS:

```
ConnectionPool=[db_type,poolMax[,singleMax]][,db_type …]
```

Consider this example:

```
ConnectionPool=oracle,10,db2,20,5
```

In this example, the maximum number of connection pools in Oracle/DB2 is set to 10 and 20 respectively. In DB2, the maximum number of connection pools per database is set to 5. This setting is made to prevent the inability to create a connection pool for another database, such as when the connection pool once created for one database is not used often thereafter and occupies high portions of the maximum number of connections.

The connection processing to the DBMS in the code previously described is discussed later in this chapter.

8.2.2.4 Transaction processing

In LSX LC, the Action method of the LCConnection class can be used for Commit/Rollback transaction control. The properties necessary for each DBMS have been provided, and values referred to as a token have been assigned. The number “1” has been assigned as the token of the CommitFrequency property that decides the mode to be used for the transaction processing. Refer to the *Enterprise Integrator Domino Connector LotusScript Extension Guide*, which is included with the Domino Connector package, for details on the supported properties and tokens. Check whether the CommitFrequency property is supported in the target DBMS. If it is, follow the procedure for setting Manual-Commit. Check whether it is supported with the LookupProperty method of the LCConnection class.

```
If(conObj.LookupProperty(COMMIT_FREQUENCY)) Then
    .......
End If
```

Hint

DB2 and Oracle both support the CommitFrequency property.

If it is supported, True is returned. If it is not supported, False is returned. If True is returned, MANUAL_COMMIT is set as the value of the property in the SetPropertyInt method. Enter the following statement to process transactions with Manual-Commit:

```
Const  COMMIT_FREQUENCY = 1 'CommitFrequency property token is 1
Const  MANUAL_COMMIT = 0
Dim conObj As LCConnection
If(conObj.LookupProperty(COMMIT_FREQUENCY)) Then
    CallconObj.SetPropertyInt(COMMIT_FREQUENCY, MANUAL_COMMIT)
End If

.......

conObj.Action (LCACTION_COMMIT) 'Commit
   or,
conObj.Action (LCACTION_ROLLBACK) 'Rollback
```

8.2.2.5 Connecting to and disconnecting from the DBMS

You must set various properties to connect to the DBMS. These properties vary depending on the type of DBMS. Refer to the *Enterprise Integrator*

Domino Connector LotusScript Extension Guide for details. Here, we describe the cases for DB2 and Oracle.

In DB2, the Database property is necessary, but in Oracle it does not need to be specified. Conversely, in Oracle, the Server property is necessary. However, if you use the default service name, no particular specification is required.

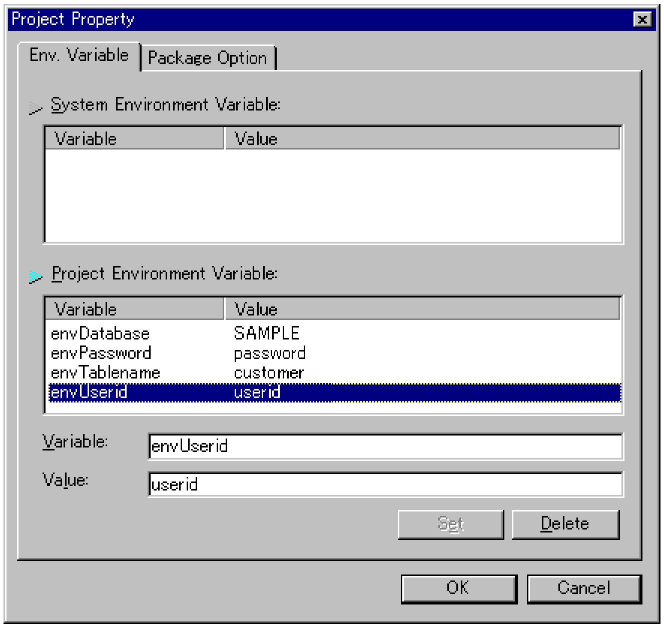
The Metadata property indicates the aggregate configuration of the data on the database to be connected. For DB2 and Oracle, this corresponds to tables and views. Connect by using the Connect method when the setting of the required properties is finished:

```
Dim conObj As New LCConnection (connector)
If connector = "db2" Then
    conObj.Database = "SAMPLE"
Elseif connector = "oracle" Then
    conObj.Server = "orcl" 'not required when accessing default service
End If

'Set properties to connect to data source
conObj.UserID = "userid"
conObj.Password = "password"
conObj.Metadata = "customer"
conObj.Connect
```

When connecting to a database, if you enter the database name, and the user ID, and the password required for access directly into the source file, and those settings are different in the development and deployment environments, you must change the source code. It may be better to use property environment variables for setting the database name, the user ID, and the password, from the perspective of separating the development and deployment of the ESB project.

```
Const connector = "db2"
Dim conObj As New LCConnection (connector)

Dim context As Variant
Set context = getContext()

'Set properties to connect to data source
conObj.Database = context.ProjectEnvValue("envDatabase")
conObj.UserID = context.ProjectEnvValue("envUserid")
conObj.Password = context.ProjectEnvValue("envPassword")
conObj.Metadata = context.ProjectEnvValue("envTablename")
conObj.Connect
```

This way, you obtain the project environment variable with the ProjectEnvValue method after obtaining the context information. After the developer has entered the code as indicated above in the source file, the operator or the manager sets project environment variables corresponding to envDatabase/envUserid/envPassword/envTablename. To set the project environment variables, select **File->Project Property** on the IDE and set the appropriate values as shown in Figure 61. Since the content set here is entered into the 1sp file, it can be managed apart from the source file.



System Environment Variable:

Variable	Value

Project Environment Variable:

Variable	Value
envDatabase	SAMPLE
envPassword	password
envTablename	customer
envUserid	userid

Variable: envUserid

Value: userid

Figure 61. Environment Variable panel in Project Property of ESB IDE

Normally when designing ESB applications, the security and access control for the respective clients is done when the Published class object is created or in the logic within that class. When an ESB application connects to a database, it is normal for a dedicated ID to be used, which provides adequate authority for the processing to be done by the application.

Otherwise, there are cases where you may want to use the access control function of DBMS based on the user ID information of the respective clients. At such times, you should set the user ID and password when creating the Published class object on the client side and use it for connection to the database on the server side. In this case, the connection pooling function cannot be used effectively.

```
Dim context As Variant
```

```
Set context = getContext()
'Set properties to connect to data source
conObj.UserID = context. Userid
conObj.Password = context. Password
```

Note

When using ESB Runtime for AIX, enter the argument (type of DBMS) for the New method of the LCConnection class in lower-case letters:

Incorrect: `Dim conObj As New LCConnection ("DB2")`

Correct: `Dim conObj As New LCConnection ("db2")`

8.2.2.6 Creating or deleting tables

With the LSX LC, when you run SQL on a database, you can select whether to run SQL directly (using the Execute method of the LCConnection class) or to use the specialized method provided by the LSX LC. Here, we show the two methods and look at their differences.

Table 18 shows the relationship between the five fields for the CustomerNo, Name, E-mail, Phone, and Age, and the attributes and the type.

Table 18. Field, attribute, and type

Field	Field attribute	Field type
CustomerNo	Long	LCTYPE_INT
Name	String	LCTYPE_TEXT
E-mail	String	LCTYPE_TEXT
Phone	String	LCTYPE_TEXT
Age	Long	LCTYPE_INT

- When using the Execute method

  In this case, you create a string type variable to store the SQL statement, make that variable the argument. Then, run the Execute method of the LCConnection class.

  ```
  Const STREAM_SIZE = 20
  Const DECIMAL_SIZE = 4
  Dim stmt As String
  stmt = "create table customer( " &_
    "CustomerNo DECIMAL(DECIMAL_SIZE,0), " &_
    "Name CHAR("& STREAM_SIZE &"),  " &_
    "Email CHAR("& STREAM_SIZE &"),  " &_
    "Phone CHAR("& STREAM_SIZE &"),  " &_
    "Age DECIMAL(DECIMAL_SIZE,0))"
  ```

```
  Call conObj.Execute(stmt, Nothing)
  Print "Table was successfully created!"
Enter
  stmt = "drop table customer"
to delete the table.
```

- When using the Create/Drop method

  To create a table, define the table to be created. Then, use the Append method to add it to the object of the LCFieldList class. To create a string type field, you must use the SetFormatStream method of the LCField class to set the format. Lastly, create the table with the Create method.

```
  'Build field list "CustomerNo, Name, Email, Phone, Age"
  Const STREAM_SIZE = 20
  Dim flds As New LCFieldList
  Dim fld As LCField
  Set fld = flds.Append("CustomerNo",LCTYPE_INT)
  Set fld = flds.Append("Name",LCTYPE_TEXT)
  Call fld.SetFormatStream(LCSTREAMF_NO_CASE,STREAM_SIZE, _
    LCSTREAMFMT_UNICODE)
  Set fld = flds.Append("Email",LCTYPE_TEXT)
  Call fld.SetFormatStream(LCSTREAMF_NO_CASE,STREAM_SIZE, _
    LCSTREAMFMT_UNICODE)
  Set fld = flds.Append("Phone",LCTYPE_TEXT)
  Call fld.SetFormatStream(LCSTREAMF_NO_CASE,STREAM_SIZE, _
    LCSTREAMFMT_UNICODE)
  Set fld = flds.Append("Age",LCTYPE_INT)
  Call conObj.Create(LCOBJECT_METADATA,flds)
  Print "Table : "&conObj.Metadata & " was created sucessfully!"
Enter
  Call conObj.Drop(LCOBJECT_METADATA)
to delete the table.
```

8.2.2.7 Inserting data

We use the following example to explain the procedure for inserting five records:

- When using the Execute method

  For this method, you create a string type variable to store the SQL statement, make that variable the argument, and then run the Execute method of the LCConnection class.

```
  Const NUM_RECORD=5
  Dim stmtBase As String
  Dim stmt(NUM_RECORD) As String
  stmt(0)="3909, 'Bill Johnson', 'Bill@aaa.com', '521-146-3233',40"
  stmt(1)="4245, 'Dave Jackson', 'Dave@aaa.com', '352-496-4962',20"
  stmt(2)="5252, 'Phil Sims', 'Phil@aaa.com', '232-787-5866',58"
  stmt(3)="5486, 'Magic Johnson', 'Magic@aaa.com', '443-402-6661',57"
  stmt(4)="5556, 'Janet Thomas', 'Janet@aaa.com', '644-618-1562',26"
  Dim i As Integer
  For i=0 To NUM_RECORD-1
    stmtBase = "INSERT INTO CUSTOMER (CUSTOMERNO, NAME, EMAIL,) " &_
      " PHONE, AGE VALUES ( "& stmt(i) &" ) "
    Call conObj.Execute(stmtBase, Nothing)
  Next
  Print NUM_RECORD & " records were inserted!"
```

- When using the Insert method

  To insert data, prepare the string type and long type arrays matched to the attributes for each field, and store the data. Create a field list by adding fields to an LCFieldList object, which is the object of the LCFieldList class. At that time, you should copy the values into the value property of the LCField class object for each field to store the values of this array into the five fields where you will insert the data. Lastly, you should insert the data with the Insert method of the LCConnection class.

```
Const NUM_RECORD=5
'Field list to be inserted into table
Dim flds As New LCFieldList(NUM_RECORD)
Dim fld As LCField
Dim sData(NUM_RECORD-1) As String
Dim iData(NUM_RECORD-1) As Long
Dim msg As String

Print "Inserting Data ..."

Set fld = flds.Append("CustomerNo",LCTYPE_INT)
idata(0) =3909
idata(1) =4245
idata(2) =5252
idata(3) =5486
idata(4) =5556
fld.value = idata

Set fld = flds.Append("Name",LCTYPE_TEXT)
sdata(0) ="Bill Johnson"
sdata(1) ="Dave Jackson"
sdata(2) ="Phil Sims"
sdata(3) ="Magic Johnson"
sdata(4) ="Janet Thomas"
fld.value = sdata

Set fld = flds.Append("Email",LCTYPE_TEXT)
sdata(0) ="Bill@aaa.com"
sdata(1) ="Dave@aaa.com"
sdata(2) ="Phil@aaa.com"
sdata(3) ="Magic@aaa.com"
sdata(4) ="Janet@aaa.com"
fld.value = sdata

Set fld = flds.Append("Phone",LCTYPE_TEXT)
sdata(0) ="521-146-3233"
sdata(1) ="352-496-4962"
sdata(2) ="232-787-5866"
sdata(3) ="443-402-6661"
sdata(4) ="644-618-1562"
fld.value = sdata

Set fld = flds.Append("Age",LCTYPE_INT)
idata(0) =40
idata(1) =20
idata(2) =58
idata(3) =57
idata(4) =26
fld.value = idata
```

```
'Insert Data
Print conObj.Insert(flds,1,NUM_RECORD)& " records were inserted!"
```

8.2.2.8 Searching for data

In the following example, we explain the procedure of how the information on CustomerNo 3909 is retrieved from the table:

- When using the Execute method

  For this method, you create a string type variable to store the SQL statement, make that variable the argument, and then run the Execute method of the LCConnection class. After running the Execute method, use the Fetch method to obtain the data.

```
Dim flds As New LCFieldList
Dim stmt As String
stmt = "Select * from customer Where CustomerNo=3909"
Call conObj.Execute(stmt, flds)

Dim count As Integer
count = flds.FieldCount
Dim resline As String
resLine = ""
Dim i As Integer

'Code below is for fetching data
For i = 1 To count
  resLine = resLine & flds.GetName (i) & ",  "
Next
Print resline

Dim Fetched As Long
Fetched = 0
Dim FetchOK As Integer
FetchOK = 1

While FetchOK = 1
  FetchOK = conObj.Fetch(flds)
  resline =""
  If FetchOK <> 0 Then
    Fetched = Fetched + 1
    For i = 1 To count
      resLine = resLine & flds.GetField(i).text(0)& ",  "
    Next
      resLine = resLine
      Print resline
  End If
Wend
Print Fetched & "records were successfully retrieved from the table!"
```

- When using the Select method

  To create a search key, add a CustomerNo field to the object of the LCFieldList class. Next, specify the value that will be the search key and the conditions in the Value property and the Flags property. In this case, you should specify `LCFieldF_KEY` for the Flags property, so that it will search for items having an equal value. After running the Select method of the LCConnection class, use the Fetch method to obtain the data.

```
Dim flds As New LCFieldList
Dim keys As New LCFieldList
Dim fld As LCField
Dim iCustomerNo As Long
iCustomerNo= 3909

'Create key to find certain records to select
Set fld = keys.Append ("CustomerNo", LCTYPE_INT)
fld.value = iCustomerNo
fld.Flags =  LCFieldF_KEY

conObj.FieldNames = "CustomerNo,Name,Email,Phone,Age"

'Search data
Call conObj.Select (keys, 1, flds)

'Fetch procedure is quite the same as above case.
 .....
```

The Flags property that becomes the search condition is set as:

```
fld.Flags =  LCFieldF_KEY
```

However, the Flags property is used as shown in Table 19 when setting other conditions.

Table 19. Flag property values of a search condition

Search condition	Flags property value
>=	LCFIELDF_KEY + LCFIELDF_KEY_GT
<=	LCFIELDF_KEY + LCFIELDF_KEY_LT
<>	LCFIELDF_KEY + LCFIELDF_KEY_NE
>	LCFIELDF_KEY + LCFIELDF_KEY_GT + LCFIELDF_KEY_NE
<	LCFIELDF_KEY + LCFIELDF_KEY_LT + LCFIELDF_KEY_NE

Be absolutely sure that the LCFIELDF_KEY is included. Refer to the *Domino Connector* manual for other details.

8.2.2.9 Updating data

In the following example, we explain the procedure where the information on CustomerNo 3909 is updated on the table, using the specified values:

- When using the Execute method

  In this case, you create a string type variable to store the SQL statement, make that variable the argument, and then run the Execute method of the LCConnection class.

  ```
  Dim fldLst As New LCFieldList
  Dim stmt As String
  ```

```
stmt = "UPDATE CUSTOMER " &_
    " SET Name = 'newName, Email='newEmail', Phone='11111', Age=20 "&_
    " Where CustomerNo = 3909"
Dim count As Integer
count = conObj.Execute(stmt, fldLst)
Print count & " data was updated!"
```

- When using the Update method

  Create an LCFieldList class object in the same manner as when searching, and set the search field property and the update data. Then, run the Update method of the LCConnection class, and make this LCFieldList class object the argument.

```
Dim flds As New LCFieldList
Dim keys As New LCFieldList
Dim fld As LCField
Dim rc As Long
Dim msg As String
Dim iCustomerNo As Integer
iCustomerNo= 3909

'Create key to find certain records to update
Set fld = keys.Append ("CustomerNo", LCTYPE_INT)
fld.value = iCustomerNo
fld.Flags =  LCFieldF_KEY
conObj.FieldNames = "CustomerNo,Name,Email,Phone,Age"

'Search data
'Build field list to update
Set fld = keys.Append ("Name", LCTYPE_TEXT)
fld.text = "Bill Johnson"
Set fld = keys.Append ("Email", LCTYPE_TEXT)
fld.text = "Bill@aaa.com"
Set fld = keys.Append ("Phone", LCTYPE_TEXT)
fld.text = "521-146-3233"
Set fld = keys.Append ("Age", LCTYPE_INT)
fld.value = 40

'Update data
Dim count As Integer
count = conObj.Update(keys)
Print count & " data was updated successfully!"
```

8.2.2.10 Deleting data

In the following example, we explain the procedure where the information on CustomerNo 3909 is deleted from the table:

- When using the Execute method

  For this method, you create a string type variable to store the SQL statement, make that variable the argument, and then run the Execute method of the LCConnection class.

```
Dim fldLst As New LCFieldList
Dim stmt As String
stmt = "delete from customer Where CustomerNo = 3909"
Dim count As Integer
count = conObj.Execute(stmt, fldLst)
Print count & " data was removed!"
```

- When using the Remove method

  Create an LCFieldList class object in the same manner as when searching, and set the search field property. Then, run the Remove method of the LCConnection class, and make this LCFieldList class object the argument.

```
Dim keys As New LCFieldList
Dim fld As LCField
Dim iCustomerNo As Long
iCustomerNo= 3909

'Create key to find certain records to remove
Set fld = keys.Append ("CustomerNo", LCTYPE_INT)
fld.value = iCustomerNo
fld.Flags =  LCFieldF_KEY

'Remove data
Print "Removing data ..."
Dim count As Integer
count = conObj.Remove(keys)
Print count & " record were removed successfully!"
```

8.2.2.11 Error handling

When a Runtime error occurs, you can use the GetStatusText method of the LCSession class to obtain the error message. You can also use the Status property to obtain the error number.

You must initialize the object of the LCSession class calling the ClearStatus method for the Runtime error to be handled correctly.

```
Function Connect() As Integer
  '// Set error handler
  On Error Goto ErrorHandler
  Dim sesObj As New LCSession
  sesObj.ClearStatus

  'Connection procedure

  '// -----------------------------------------------------------
  '// Error handler for 'Connect' method.
  '// -----------------------------------------------------------
ErrorHandler:
  If (SesObj.Status <> LCSUCCESS)Then
    lastMessage = "[DB] " & SesObj.GetStatusText
    Print lastMessage
  Else
    lastMessage = "[DB] No error specific message is available."
    Print lastMessage
  End If
  ConnectFlag = False
  Connect = -1
  Exit Function
End Function
```

8.3 CLI/ODBC and OCI

When you want finer control than the Domino Connectors with the LSX LC in the connections and SQL statement processing and when you must have multiple query result sets for a single connection, you can directly use the API of CLI/ODBC and OCI. In this section, we explain how to develop database access programs on ESB using CLI (DB2) and OCI (Oracle), which are the native interfaces of DB2 and Oracle.

8.3.1 CLI native call programming

This section provides an outline of the ESB project development procedure using CLI native calls. Refer to the DBACC sample attached to ESB for the detailed code.

8.3.1.1 Flow and outline of a basic CLI program

CLI programming is broadly divided into three steps: initial setting, transaction processing, and termination. See Figure 62.

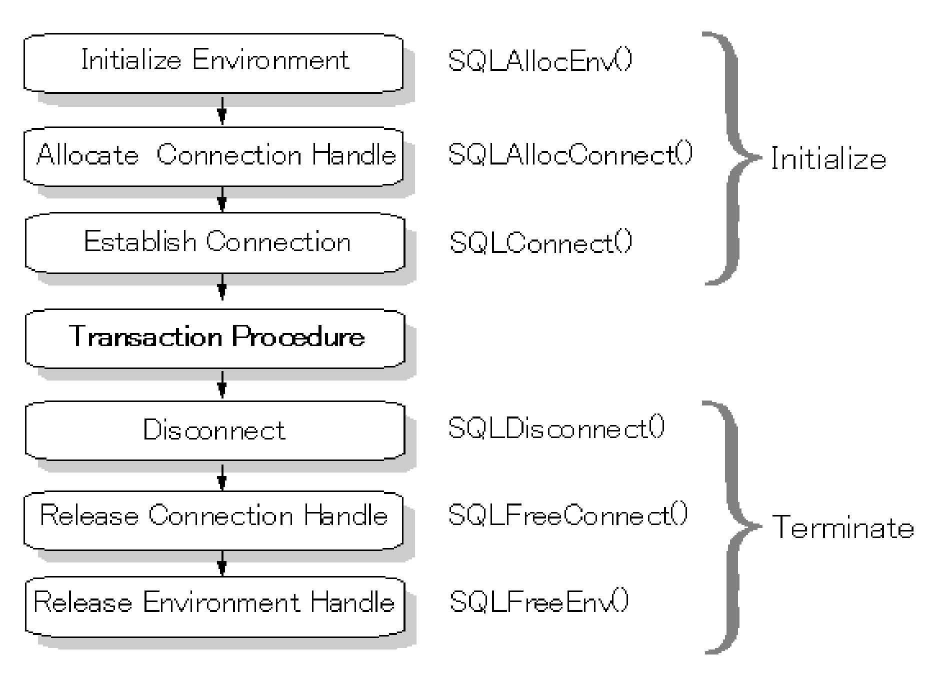


Figure 62. Initialization and termination of CLI

In the initial setting, the environment handle and the connection handle are assigned, and the application is connected to the database. The handle is the variable that refers to the data object to be controlled by DB2 CLI. There are four types of handles: environment, connection, statement, and descriptor.

- **Environment handle**

  The environment handle refers to the data objects containing the information pertaining to the global status of the applications. They are allocated by SQLAllocEnv() and are released by SQLFreeEnv(). To allocate a connection handle, you must first allocate the environment handle.

- **Connection handle**

  The connection handle refers to the data objects containing the information pertaining to specific database connections. This information includes connection options, general status information, traffic conditions, and diagnosis information. This connection handle is allocated by calling SQLAllocEnv() and released by calling SQLFreeEnv().

Transaction processing runs the main SQL statement of the application. Figure 63 shows a sample of the transaction processing that runs the SELECT statement.

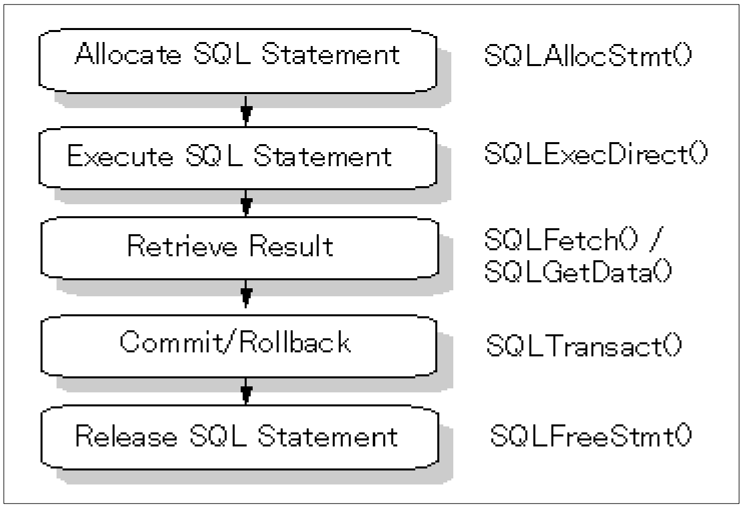


Figure 63. SELECT SQL process

The termination process disconnects from the database and releases the handle.

Hint

You can enter the code for ODBC in the same step as DB2 CLI. Changing both can be facilitated by switching the Include file on ESB. For example, in the DBACC sample attached to ESB, remove the comment of the unnecessary line in the (Options) script of the (Globals) object:

```
%INCLUDE "lsdbcli.lss"' Data access include file for CLI
'%INCLUDE "lsdbodbc.lss"' Data access include file for ODBC
```

8.3.1.2 Connection pooling

To use a connection pooling mechanism in the CLI programming on ESB, enter its logic in the Server class. This is because connection pooling must be shared between Published class objects. In the Server class, you first obtain the number of connection handles pooled after initialization of the environment handle. The environment handles are stored in Long type variables, and the connection handles are stored in arrays for long type elements. Create and then initialize the array that indicates the usage status of the handle (False: Unused), to have exclusive control of the usage of the handle. In addition, you should respectively provide a function for lending the connection handle to clients and a function for returning the connection handle. In the following example, we show the GetConHandle() function for obtaining the connection handle and the ReleaseConHandle() for returning the connection handle:

```
'arghdbc : Connection handle
Function GetConHandle(arghdbc As Long) As Variant

  Dim i As Integer
  Const Con_MaxConnect=5

  'Use one of the UNUSED connection handles,
  'and turn the status of this handle to USED.
  For i = 0 To (Con_MaxConnect - 1)
    If hdbc_state(i) = False Then
      arghdbc = hdbc(i) 'hdbc(i) : Connection handle
      hdbc_state(i) = True ' hdbc_state(i) : Status handle
      GetConHandle = True
      Exit Function
    End If
  Next i

  GetConHandle = False ' UNUSED connection handle is not available.
End Function
```

The GetConHandle() checks the usage status of the connection handle, allocates usage if it is unused, and then changes the usage status of the handle to True. If there are no unused connection handles, it returns False, and then reports that the handle could not be obtained.

```
Function ReleaseConHandle(arghdbc As Long) As Variant
  Dim i As Integer
  Const Con_MaxConnect=5

  'Turn the status of the used connection handle to UNUSED.
  For i = 0 To (Con_MaxConnect - 1)
    If arghdbc = hdbc(i) Then
      hdbc_state(i) = False
      ReleaseConHandle = True
      Exit Function
    End If
  Next i

  'The used connection handle cannot be set to UNUSED.
  ReleaseConHandle = False
End Function
```

By simply setting the usage status of the connection handle to False, the ReleaseConHandle() function arranges for the corresponding connection to be reallocated when the GetConHandle() function is again called. Figure 64 shows the mechanics of the CLI programming model.

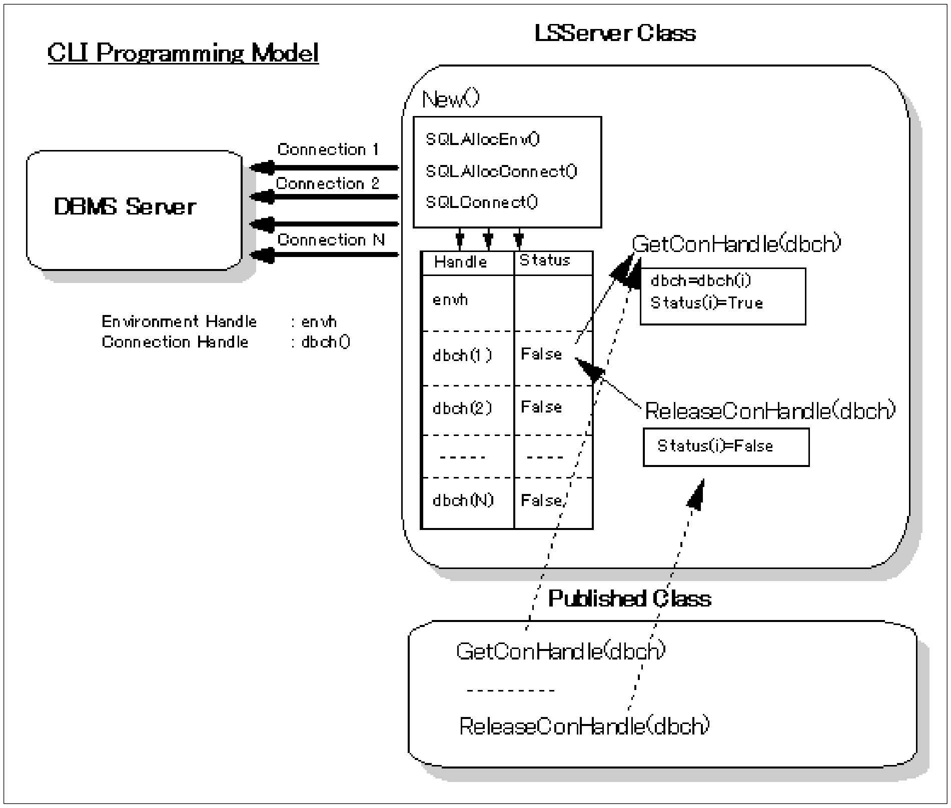


Figure 64. Mechanics of the CLI programming model

The Published class object can obtain unused handles among the previously pooled connection handles by binding Server classes and calling the GetConHandle() function. At this time, it sets the usage status of the connection handle to True so that no other Published class object can use that handle. Subsequently, it uses the obtained connection handle to run the CLI function for the SQL processing. Once the processing ends, it calls the ReleaseConHandle() function. It then sets the usage status of the connection handle to False and release the connection handle to other Published class objects. The method calling of the server class is serialized, so handles cannot be lent redundantly and no erroneous statuses are set.

8.3.2 OCI native call programming

The section outlines the development process of the ESB project using OCI native calls (when using Oracle8). Refer to the OCI8ACC sample attached to the ESB.

Hint

If you are using Oracle7, refer to the OCIACC sample attached to ESB.

8.3.2.1 Flow and outline of the basic OCI program

The typical process flow of OCI is showed in Figure 65 on page 196.

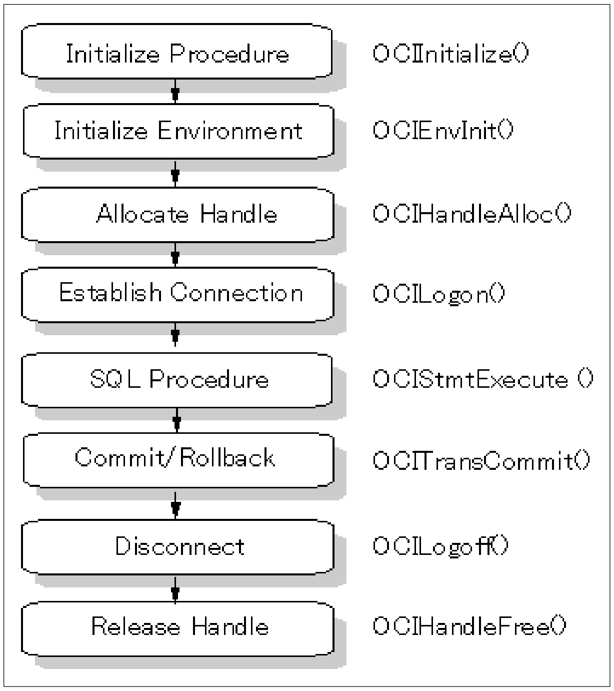


Figure 65. OCI process flow

The handle concept is used in several processes in OCI. The handles are pointers to data structures, which comprehensively define the necessary parameter group for each function when calling an OCI function. This handle is passed when an OCI function is called. Handles serve to alleviate programmers from such burdens imposed by parameter variable declarations, data holds, parameter enumeration upon function calls, and work area memory allocation for OCI functions. The primary handles are explained here:

- **Environment handle**

  The environment handle defines the environment that calls all the OCI functions. The environment handle has a memory cache area. In the case of multi-thread applications, exclusive control is implemented in environment handle units. When using a single environment handle in multiple threads, when one thread is accessing a resource within the environment handle, such as the memory cache, for example, access by others is blocked. Since ESB operates applications with multi-threads, a handle is created for each thread in an effort to improve performance and assure stability.

- **Error handle**

  The error handle is passed as a parameter for almost all OCI calls and handles the information on errors generated during OCI operation.

- **Service context handle**

  The service context handle defines the OCI operating environment for the server. It holds the pointer to the server handle, the user session handle, and the transaction handle.

8.3.2.2 Connection pooling

To use the connection pooling mechanism in the programming on ESB, you describe its logic in the Server class. This is because connection pooling must be shared between Published class objects. After running the initialization (OCIInitilize() function) of the OCI process, the Server class repeats the initialization of the environment handle, the allocation of the error handle, and the starting of the session the number of times that it pools the connection. Each handle is stored in a long type element array. You must also create and initialize an array indicating the usage status of the handle (False: unused), to exclusively control the usage of the handle. In addition, you should respectively provide a function for lending handles to the clients and a function for returning handles. Here, we show an example of a GetHandle() function for obtaining handles and a ReleaseHandle() for returning handles.

The GetHandle() checks the usage status of the handle, allocates it if it is unused, and then changes the handle usage status to “True”. If there are no unused handles, it returns “False” and reports that no handle could be obtained.

```
'hnumber : Index of a handle
'envh    : Environment handle
'errh    : Error handle
'svch    : Service context handle
Function GetHandle(hnumber As Integer, envhout As Long, _
    errhout As Long, svchout As Long) As Variant
  Dim i As Integer
  Const Con_MaxConnect=5

  ' Use one of the UNUSED connection handles,
  ' and change the status of this handle to USED.
  For i = 0 To (Con_MaxConnect - 1)
    If h_state(i) = False Then
      hnumber = i
      h_state(i) = True ' h_state(i) : Status handle
      envhout = envh(i) 'envh(i) : Environment handle
      errhout = errh(i) 'errh(i) : Error handle
      svchout = svch(i) 'svch(i) : Service context handle
      GetHandle = True
      Exit Function
    End If
  Next i
  ' UNUSED connection handle is not available.
  GetHandle = False
End Function
```

By simply setting the handle usage status to “False”, the ReleaseHandle() function arranges for the concerned handle to be reallocated when the GetHandle() function is called.

```
Sub ReleaseHandle(hnumber As Integer)
  Dim i As Integer
  ' Set the using handle to UNUSED.
  h_state(hnumber) = False
End Sub
```

The mechanics of the OCI programming model are shown in Figure 66.

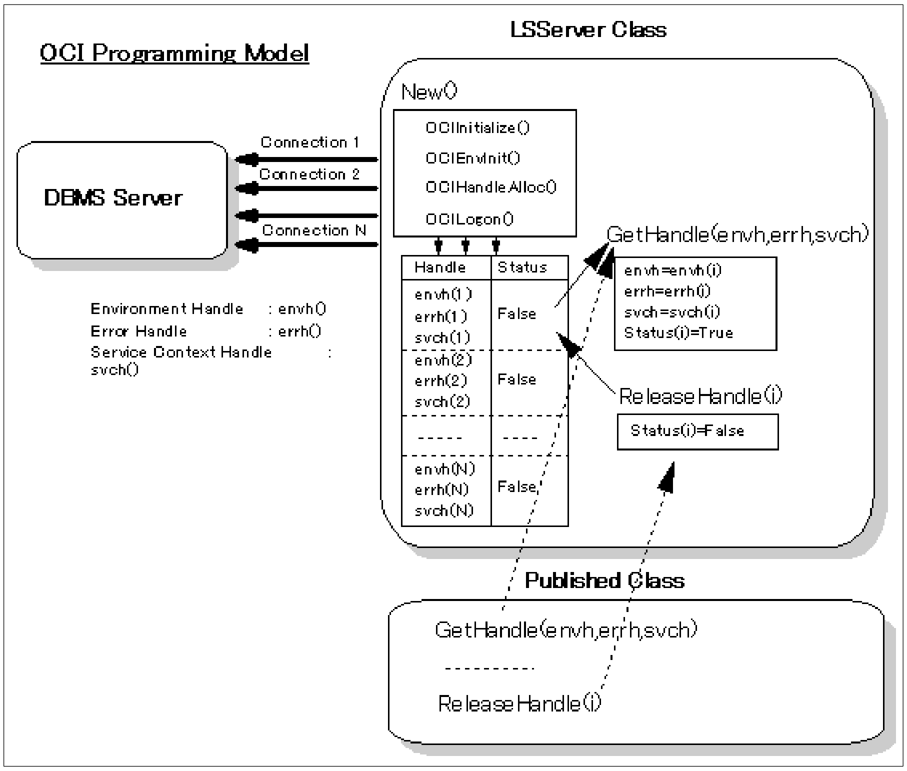


Figure 66. OCI programming model

The Published class object can obtain an unused handle among the previously pooled handles, by binding the Service class and calling the GetHandle() function. At this time, it sets the usage status of the connection handle to “True” so that no other Published class object can use that handle. Subsequently, it uses the obtained connection handle to run the CLI function for the SQL processing. Once the processing ends, it calls the ReleaseConHandle() function, and then sets the handle to the unused status, releasing the connection handle to other Published class objects. The method calling the Server class is serialized. Therefore, handles cannot be lent redundantly, and no erroneous statuses are set.

8.4 Performance comparison

In this section, we measure the performance for each of several programming methods for accessing databases. Then, we describe the results. After we consider this performance and the relative difficulty of the programming, we identify what programming method should be used.

Note

In the following section, “LSX LC” implies the use of the classes, methods, and properties of the LSX LC with a Domino Connector.

8.4.1 Test environment

We prepared a total of three platforms: a database server, an ESB Runtime server, and an ESB client. They are based on the configurations shown in Table 20 and Table 21 for the performance test. For the database server, we adopted the AIX environment and configured it for connection from the ESB client through an ESB Runtime server using the Windows NT environment.

Table 20. System hardware configuration for performance testing

Hardware configuration	CPU	Memory
DB Server	PowerPC3 200MHz	2GB
ESB NT Server	Pentium-2 233Mhz	160MB
ESB Client	Pentium 120Mhz	64MB

Table 21. System software configuration for performance testing

Software configuration	Products
DB Server	AIX V4.3.2.0
- DB2	DB2 UDB EE V5.2
- Oracle	Oracle8 Release 8.0.5 for AIX

Software configuration	Products
ESB NT Server	Windows NT Server + SP4
ESB Client	Windows NT Workstation + SP3

8.4.2 Search processing comparison

For the search processing test, we create a database with data for ten thousand records and search for records matching the search key. The table structure is shown in Table 22. We set a random numerical value in the client program and search for records with IDs located in the Primary Key of the table that equal this numerical value. We create an SQL based on this value in ESB Runtime (or if we use the Select method of the LSX LC, we use it as the method argument), and measure the time from the calling of the method to the database server until it is returned. After one hundred measurements, we calculate the average value, which is made the measurement value.

Table 22. Search comparison between DB2 and Oracle

Column	DB2	Oracle
ID(Primary Key)	INTEGER	NUMBER
A	CHAR(5)	VARCHAR2(5)
B	CHAR(10)	VARCHAR2(10),
C	VARCHAR(40)	VARCHAR2(40),
D	DECIMAL	NUMBER

The table image is shown here:

```
1, "A0001", "B000000001", "C(**************)  1",  10001
2, "A0002", "B000000002", "C(**************)  2",  10002
3, "A0003", "B000000003", "C(**************)  3",  10003
4, "A0004", "B000000004", "C(**************)  4",  10004
....................
100000, "A0000", "B000100000", "C(**************)  100000",  10000
```

In the server program, the LSX LC method is compared with the method using the OCI/CLI native call. However, there are two description methods for the LSX LC. On this occasion, multipoint processing is performed according to the method that is used for search processing to measure the respective performances.

```
Dim fldLst as New LCFieldList

Select Case whichway
Case 0  'This method is to execute SQL directly
```

```
  Dim stmt as string
  stmt = "SELECT * FROM "& Userid+"."& Table &" where ID = " & keyVal
  call conObj.Execute(stmt, fldLst)
Case 1  ' This method is to call domino connector unique method
  Dim iCustomerNo As Long
  Dim keys As New LCFieldList
  Dim fld As LCField
  'Create key to find certain records to select
  Set fld = keys.Append ("ID", LCTYPE_INT)
  fld.value = keyVal
  fld.Flags =  LCFieldF_KEY

  conObj.FieldNames = "ID,A,B,C,D"

  'Search data
  call conObj.Select (keys, 1, fldLst)
End Select

'Data fetch procedure
Dim count As Integer
count = flds.FieldCount
Dim resline As String
resLine = ""
Dim i As Integer

For i = 1 To count
  resLine = resLine & flds.GetName (i) & ",  "
Next

Dim Fetched As Long
Fetched = 0
Dim FetchedOK As Integer
FetchOK = 1

While FetchOK = 1
    FetchOK = conObj.Fetch(flds)
    resline =""
    If FetchOK <> 0 Then
        Fetched = Fetched + 1
        For i = 1 To count
            resLine = resLine & flds.GetField(i).text(0)& ",  "
        Next
        resLine = resLine & Chr(10)
    End If
Wend
```

The most remarkable difference lies in the amount of code. We find that, compared to the case where SQL is run directly with the Execute method, the case where the specialized method provided by the LSX LC is used consists of a substantially greater amount of code.

8.4.2.1 DB2

The results of a comparison of the CLI native call and the two LSX LC methods are shown here and in Table 23:

```
Units: seconds
```

Table 23. DB2 Access comparison by CLI and Domino Connector

(1) CLI Direct Call	(2) LSX LC: Execute Direct	(3) LSX LC: Select Method
0.0119	0.0105	0.0115

Items (2) and (3) in Table 23 represent the cases where the LSX LC was used. Execute Direct in item (2) refers to the case where SQL is run directly with the Execute method. The Select method in (3) indicates cases where the LSX LC Select method was used. Here, we learn that the methods indicated in (2) and (3) using the LSX LC were slightly faster than the CLI native call. However, if we consider the measurement error in (1) and (3), there is virtually no significant difference. Although the best performance was obtained in the case of (2), when running SQL directly with the Execute method, we find that it not only can be described relatively simply compared to the voluminous code description required for the CLI native call, but also operates faster. For example, in the CLI native call programming, several APIs must be called, such as:

```
SQLAllocStmt()
SQLExecDirect()
SQLFetch()
SQLGetData() or SQLGetDataStr()
SQLFreeStmt()
```

As arguments, many handles must be respectively provided and managed. On the other hand, in the case of (2), as we saw previously, the description is done in a few lines. If we consider the hazard of bugs being produced and the ease of description in code, the Domino Connector LSX code would appear to be superior to CLI.

In the two LSX LC description methods indicated in (2) and (3), there is no great difference in performance. Significant differences appear in the respective description methods when running SQL in complex conditions. In the previously described example, we searched for records matching a key. However, when combining two or more tables for this or when taking data within the range of certain conditions, in the approach using the LSX LC method, we must master the peculiar description method. For example, when searching for records matching a key, you set fld.Flags = LCFieldF_KEY.

However, if you change the search conditions, you must change the values as shown in Table 24.

Table 24. Flag property values of search conditions

Search conditions	Flag property values
>=	LCFIELDF_KEY + LCFIELDF_KEY_GT
<=	LCFIELDF_KEY + LCFIELDF_KEY_LT
<>	LCFIELDF_KEY + LCFIELDF_KEY_NE
>	LCFIELDF_KEY + LCFIELDF_KEY_GT + LCFIELDF_KEY_NE
<	LCFIELDF_KEY + LCFIELDF_KEY_LT + LCFIELDF_KEY_NE

This is as described in how to search for data with the Select method provided by the Domino Connector LSX. From this perspective, it appears that it would be preferable to run SQL with the Execute method as indicated in (2) for accessing an RDB that can use SQL.

8.4.2.2 Oracle

The results of the comparison of the case of the OCI native call and the two Domino Connector methods are shown here and in Table 25:

```
Units: seconds
```

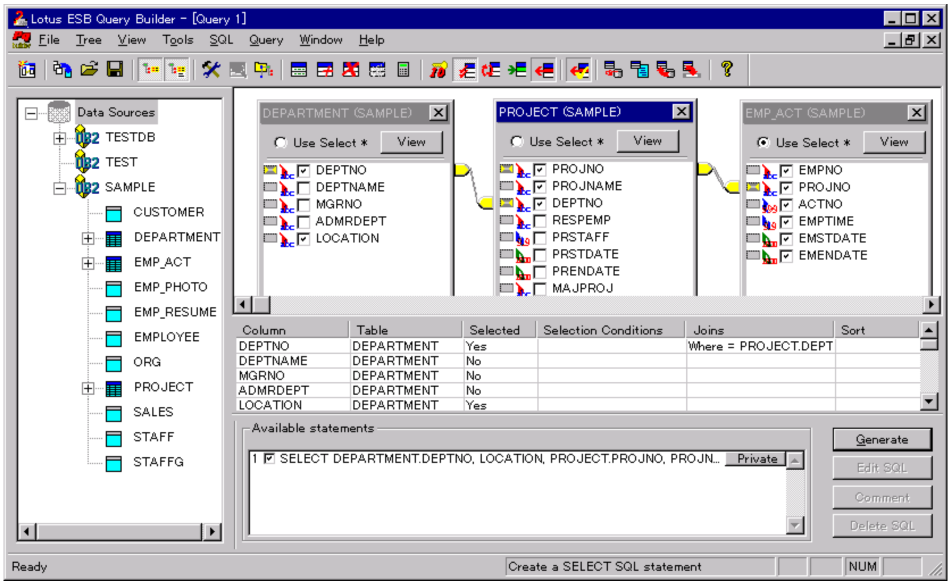
Table 25. Performance comparison of OCI direct call and Domino Connector

(1) OCI Direct Call	(2) Domino Connector - Execute Direct	(3) Domino Connector - Select Method
0.0102	0.0095	0.0098

As in the case of DB2, the methods indicated in (2) and (3), in Table 25, using the LSX LC, appear to be slightly faster than the OCI native call. However, it can be said that if measurement error is taken into consideration, there is no significant difference. Since the best performance was obtained in the case of (2), when running SQL directly with the Execute method, it appears to be preferable to run the SQL statement with the Execute method as indicated in (2) for accessing an RDB that can use SQL.

8.5 Linking with Query Builder

This section briefly describes accessing the DBMS from Query Builder of ESB. Figure 67 on page 204 shows the view of Query Builder Code Generator connected to the DB.



DEPARTMENT (SAMPLE): ○ Use Select *
- ☑ DEPTNO
- □ DEPTNAME
- □ MGRNO
- □ ADMRDEPT
- ☑ LOCATION

PROJECT (SAMPLE): ○ Use Select *
- ☑ PROJNO
- ☑ PROJNAME
- ☑ DEPTNO
- □ RESPEMP
- □ PRSTAFF
- □ PRSTDATE
- □ PRENDATE
- □ MAJPROJ

EMP_ACT (SAMPLE): ● Use Select *
- ☑ EMPNO
- ☑ PROJNO
- ☑ ACTNO
- ☑ EMPTIME
- ☑ EMSTDATE
- ☑ EMENDATE

Column	Table	Selected	Selection Conditions	Joins	Sort
DEPTNO	DEPARTMENT	Yes		Where = PROJECT.DEPT	
DEPTNAME	DEPARTMENT	No			
MGRNO	DEPARTMENT	No			
ADMRDEPT	DEPARTMENT	No			
LOCATION	DEPARTMENT	Yes			

Available statements

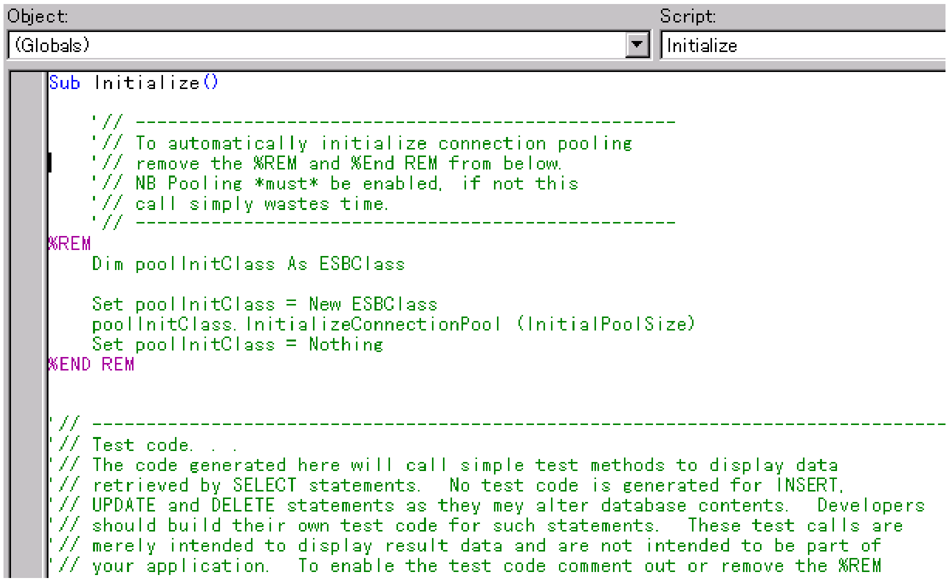
1 ☑ SELECT DEPARTMENT.DEPTNO, LOCATION, PROJECT.PROJNO, PROJN... Private

Create a SELECT SQL statement

Figure 67. Query Builder Code Generator connected to a sample database

8.5.1 Creating and using a source file by Code Generator

The codes are generated by the Query Builder Code Generator. Figure 68 shows the codes displayed in the Script panel of IDE.



```
Sub Initialize()

    '// --------------------------------------------------
    '// To automatically initialize connection pooling
    '// remove the %REM and %End REM from below.
    '// NB Pooling *must* be enabled, if not this
    '// call simply wastes time.
    '// --------------------------------------------------
%REM
    Dim poolInitClass As ESBClass

    Set poolInitClass = New ESBClass
    poolInitClass.InitializeConnectionPool (InitialPoolSize)
    Set poolInitClass = Nothing
%END REM


'// -------------------------------------------------------------------------------
'// Test code. . .
'// The code generated here will call simple test methods to display data
'// retrieved by SELECT statements.  No test code is generated for INSERT,
'// UPDATE and DELETE statements as they mey alter database contents.  Developers
'// should build their own test code for such statements.  These test calls are
'// merely intended to display result data and are not intended to be part of
'// your application.  To enable the test code comment out or remove the %REM
```

Figure 68. Codes generated by Query Builder Code Generator

In ESB, you can use Query Builder Code Generator to create skeleton source files for DBMS connections.

To create a source file on the IDE, select **Tools->Create New Query** from the IDE and start Query Builder. When you create a source file with Code Generator, it is automatically added to a project on the IDE and can then be utilized. Refer to Chapter 5 of the *ESB Users Guide* for detailed usage of Code Generator. Code generated with Code Generator supports connection pooling and transaction processing. The following section briefly explains how to use such generated code.

8.5.2 Connection test

The Initialize subroutine of a file created by Code Generator contains the test code for the DBMS connection written as “'// Test code. . .”. The following example shows the code generated when a user creates a class called an ESBClass using the Code Generator. Deleting the %REM and %END REM statements enables the running of the code for the connection test. If you run the project file just as it was created, no result will be displayed and you will not know whether it was successful. It is a good idea to insert a Print statement for the Connect member and Disconnect member subroutines within the (Declarations) script, so that the appropriate message is displayed.

```
'// -------------------------------------------------------
'// Test code. . .
'// The code generated here will call simple test methods to display data
'// retrieved by SELECT statements. No test code is generated for INSERT,
   . . .
'// your application. To enable the test code comment out or remove the
'// %REM and %End REM statements below.
'// --------------------------------------------------------
%REM
  Dim shObj As ESBClass

  Set shObj = New ESBClass

  shObj.Connect
  shObj.Disconnect
  Set shObj = Nothing

%END REM
```

Note

The class on the code generated by the Code Generator is generated as a Public class rather than a Published class. Consequently, it cannot be called directly from a client program. You can call it from within the same project (Initialize event) as in the above case. The following two methods are available for calling from a client program:

- Create the Published class separately, and then use the Public class generated by the Code Generator indirectly.
- Change the Public keyword to the Published keyword, and substitute the Get Property statement in the normal Function statement. The Published class does not support the Property statement.

8.5.3 Connection pooling

The Initialize subroutine of the file created by the Configuration Tool contains the following code. Deleting the %REM and %END REM statements enables the running of the code for the connection test.

```
'// -------------------------------------------------
'// To automatically initialize connection pooling
'// remove the %REM and %End REM from below.
'// NB Pooling *must* be enabled, if not this
'// call simply wastes time.
'// -------------------------------------------------
%REM
  Dim poolInitClass As ESBClass

  Set poolInitClass = New ESBClass
  poolInitClass.InitializeConnectionPool (InitialPoolSize)
  Set poolInitClass = Nothing
%END REM
```

To create the connection pool explained in the preceding item, you must set the ConnectionPooling property of the LCSession class object to “True”. In the source file crated by the Code Generator, you can set the ConnectionPooling property in the New() member subroutine of the created class.

```
Sub New()
  '// Set error handler
  On Error Goto ErrorHandler

  '// ------------------------------------------------
  '// Create Domino Connector Objects
  '// ------------------------------------------------
  '// Set session object
  Set sesObj = New LCSession
  sesObj.ClearStatus

  '// Deal with connection pooling. . .
  '// Uncomment the line below to enable connection pooling
  '// sesObj.ConnectionPooling = True
   ...
End Sub
```

Remove the comment character within the New() member subroutine in the line described as:

```
sesObj.ConnectionPooling = True
```

Make it effective. We understand, that as a result, the ConnectionPooling property is set appropriately in the New() member subroutine by executing the “Set poolInitClass = New ESBClass” line. Therefore, the correct preprocessing can be performed before the InitializeConnectionPool method is called. The connection pool default value has been set to 5. However, if you change it, you should change the following description to an appropriate value in the (Options) script of the (Globals) object:

```
Const  InitialPoolSize = 5
```

8.5.4 Setting the transaction processing

Transaction processing is set up to be done in the Manual-Commit (no Auto-Commit) mode in the source file created by the Code Generator. To change this, set an appropriate value in the following section, in the (Options) script of the (Globals) object within the code:

```
Const  COMMIT_FREQUENCY = 1
Const  MANUAL_COMMIT = 0
```

The explanation covering such items as the COMMIT_FREQUENCY constant and the token of the DBMS property is redundant. Therefore, it will be omitted. Verify whether the COMMIT_FREQUENCY property is supported by the DBMS to be connected. If it is, the procedure to be followed for changing to Manual-Commit mode is described as shown in the Connect member function:

```
Function Connect() As Integer
  ...
  If(conObj.LookupProperty(COMMIT_FREQUENCY)) Then
    Call conObj.SetPropertyInt(COMMIT_FREQUENCY, MANUAL_COMMIT)
  End If
  ...
End Function
```

Explicit Commit and Rollback processing is required when setting to the Manual-Commit mode. Therefore, you should appropriately call the Commit() and Rollback() member subroutines within the class created by the Code Generator:

```
'Commit Method
Sub Commit()
  If connectFlag = True Then
    conObj.Action (LCACTION_COMMIT)
  End If
End Sub

'Rollback Method
Sub Rollback()
  If connectFlag = True Then
    conObj.Action (LCACTION_ROLLBACK)
  End If
End Sub
```

8.5.5 Setting for an Oracle connection

When connecting to an Oracle database, you must set the value for the Server property of the LCConnection class to connect to the specific Oracle database service, rather than if you do not connect to the default service. The following description is contained within the New() member subroutine within the class created by the Code Generator:

```
'// -------------------------------------------------
'// Set database parameters.
'// -------------------------------------------------
'// The default server was used to connect to oracle.
'// This means that no database or service name was used. To use
'// a specified database or service change the 'Default oracle
'// source' in the conObj.Server entry below and uncomment the
'// line.
'// conObj.Server = Default oracle source
```

Besides the default service, you can connect to a service name by changing the value of the Server property described on the last line to the service name to be connected:

```
conObj.Server = "orcl"
```

Chapter 9. Accessing transaction systems

As an information processing system in a corporation, the enterprise business management system are built to run as a production management system, service management system, and a corporate management system for personal and financial information. These systems are built on a host computer, and the information is stored on a database such as DB2. A Customer Information Control System (CICS) or Information Management System (IMS) is used as the transaction interface system of the database. Multiple systems on different platforms in the corporation are often connected through message-oriented middleware systems such as IBM MQSeries. The latest information is used to improve business performance. These mission critical business applications are referred to as “backend” applications, which process large volumes of data at high speeds. Under these circumstances, customers demand reliability, availability, and serviceability to provide constant service without causing the problem of a system shutdown.

When the server application program of ESB is developed to work with the backend application, a typical way to connect the backend application is to use the emulator program for the main frame interactive or use the application program interface directly provided by CICS or DB2. This chapter introduces the MQSeries link LotusScript Extension (MQLSX) method, which is provided by MQSeries.

9.1 Integration with mission-critical business applications using MQLSX

MQSeries is the product which collaborates with the application on the Transaction Management Systems. By defining the message (data) unique to an application on the MQSeries, it can be exchanged between different applications, on multiple platforms. Lotus provides the MQSeries link LotusScript Extension (MQLSX) to use MQSeries from LotusScript. It can be one of the most effective selections for developing a server application with ESB, which collaborates with the critical business applications. In this section, we explain how to use MQLSX using ESB with the example of UseMQLSX included in the ESB.

9.1.1 What MQLSX is

The MQSeries link LotusScript Extension (MQLSX) enables Domino applications to integrate with mission-critical business applications on a host computer. Here, it works with applications coded in LotusScript on Lotus Notes or ESB.

MQLSX provides a series of classes with methods and properties designed to facilitate the use of the message queue interface (MQI) provided by the MQSeries from LotusScript. The classes include:

- MQSession Class
- MQQueueManager Class
- MQQueue Class
- MQMessage Class
- MQGetMessageOptions Class
- MQPutMessageOptions Class
- MQProcess Class

Moreover, through the use of On Event and On Error statements, the MQLSX isolates error processing from the normal processing flow and provides each class with a property Completion Code and Reason Code that saves the event MQWARNING, MQERROR, and error information to permit concise programming.

9.1.2 Usable platforms

MQLSX can be used on a variety of platforms:

- Intel platforms (OS/2, Windows 3.1, Windows 95, Windows 98, and Windows NT)
- UNIX platforms (AIX, HP-UX, Sun Solaris)

Note: ESB must use MQLSX for Windows NT or AIX.

9.1.3 How to obtain MQLSX

You can obtain MQLSX from the SupportPac provided by IBM. You can also get it as a tool in the MQSeries and CICS Connections for Domino tool from the Lotus Enterprise Integration Web site:

- MA6D: MQSeries for AIX link LotusScript Extension
  `http://www.software.ibm.com/ts/mqseries/txppacs/ma6d.html`
- MA7D: MQSeries for Windows 32-bit platforms link LotusScript Extension
  `http://www.software.ibm.com/ts/mqseries/txppacs/ma7d.html`
- Lotus Enterprise Integration Web site: `http://www.lotus.com/dominoei`

The most recent version of the MQLSX ships with MQSeries V5.1 and is called “MQLSX5.1”.

Note: These Web addresses were valid at the time that this redbook was written.

9.1.4 Prerequisites

An MQSeries Server or MQSeries Client is required on the same machine that is running ESB Runtime.

Note: The latest information on the respective MQSeries components, including MQLSX, is available at the following Web site:
`http://www.software.ibm.com/ts/mqseries/support/fixes/`

The latest MQLSX version at the time this redbook was written was 5.1, whihch ships with MQSeries V5.1. The program temporary fix (PTF) and Corrective Service Diskettes (CSD) of the MQSeries Server and MQSeries Client used with it are subject to revision at any time. Please refer to the readme.txt file of the MQLSX for a description of the PTF and CSD, which is a prerequisite for using MQLSX.

The following environments were used in preparing this example:

For Windows NT:

- MQSeries for Windows NT V5.0 + PTF U200095
- MQSeries for Windows 32-bit platforms link LotusScript Extension Release 1.3.3

For AIX:

- MQSeries for AIX V5.0 + PTF U461602
- MQSeries for AIX link LotusScript Extension Release 1.3.3

9.2 Examples of MQLSX

In this sample, integrated processing is not performed with an actual host application. Instead, it is composed of three classes:

- **Published class UseMQLSXClass**: Receives text data send or receive requests from an ESB client program created in a Notes database, accesses the queue manager, and then sends or receives data (messages).
- **LSServer class HostSimulatorClass**: Independently accesses the queue manager, and then automatically performs the reply processing for the messages sent from the UseMQLSXClass.
- **Public class MQAccessClass**: Provides message sending and receiving functions to the UseMQLSXClass and HostSimulatorClass to simplify the use of MQLSX.

The queue manager accessed by the UseMQLSXClass and the HostSimulatorClass can also be set to operate on a different machine in which the queue interval has been remotely set, even if the setting circumstances are the same. Therefore, the UseMQLS XClass, for example, can be used as a substitute for the functions performed by the HostSimulatorClass in the CICS program operating on the MVS/ESA.

By expanding the HostSimulatorClass, you can create server applications that automatically perform a series of tasks. Such tasks include receiving messages sent using the MQSeries, updating the DB2 database, and sending the results in Notes Mail.

Finally, the MQAccessClass is a generic class created with the objective of easily sending and receiving messages, by using a very small part of the class methods and properties provided by MQLSX. Consequently, you can develop other applications, by reusing MQCommon.lss, which includes the MQAccessClass. In addition to the MQAccessClass, MQCommon.lss also includes the LSServer class MessageIDClass that creates dedicated MessageIDs for messages to be sent.

Figure 69 shows the relationship of each component to the process flow.

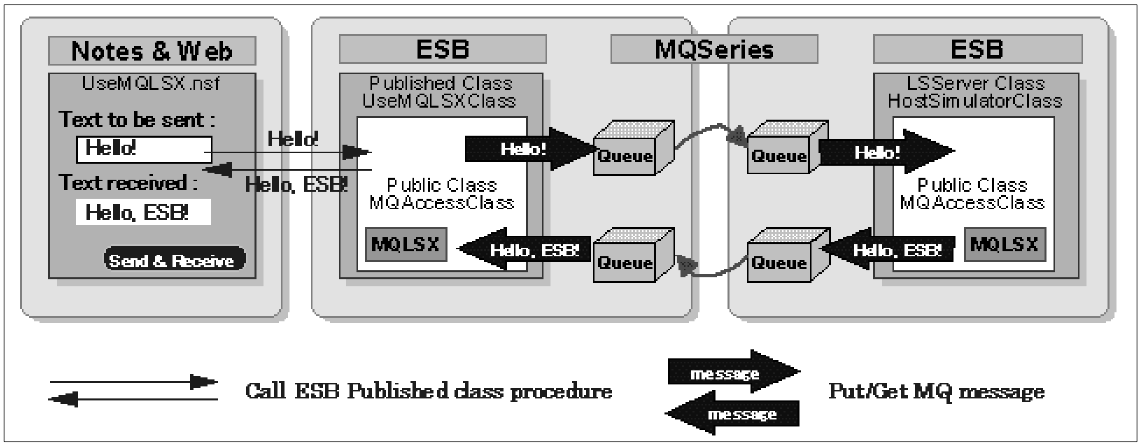


Figure 69. Overview of the relationship between ESB and MQSeries

The following functions can be run from client program of this sample:

- **Object creation**

  Creates Published class UseMQLSXClass objects.

- **Object deletion**

  Deletes Published class UseMQLSXClass objects.

- **Sending and receiving**

  Sends text entered in [`send text`] to the message queue and receives messages sent back by the LSServer class HostSimulatorClass from the message queue.

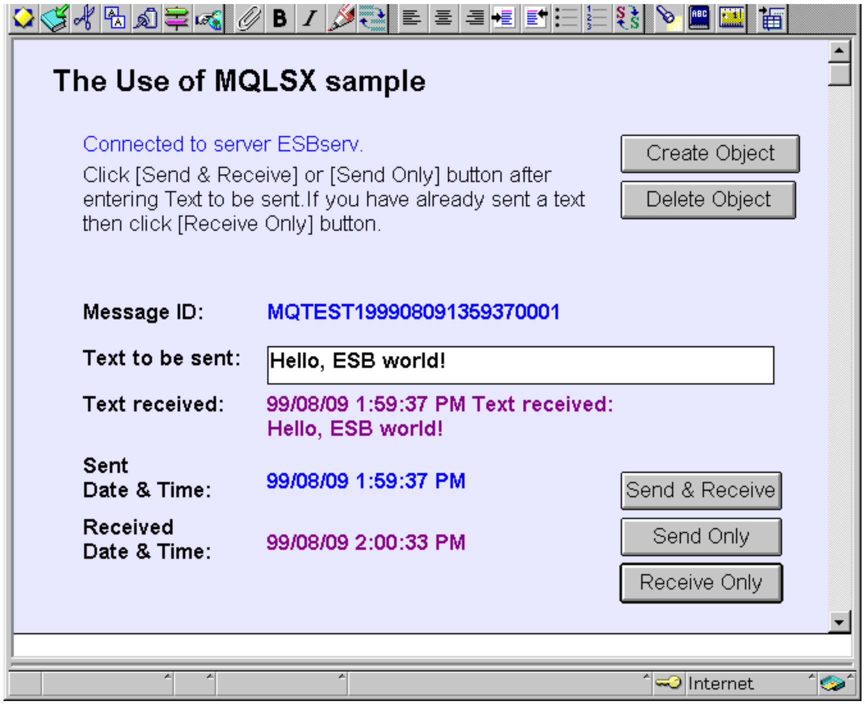
- **Sending only**

  Sends text entered in [`send text`] to the message queue. It does not run the receiving of messages sent back from the LSServer class HostSimulatorClass. It can complete this process asynchronously, even when the LSServer process is not running.

- **Receiving only**

  After the send process is completed, it runs only the receive process for the messages that were sent back from the LSServer class HostSimulatorClass. The receive process runs the message ID automatically created at sending time as the key. The process can be completed asynchronously, when the LSServer class has already run the reply process for that message.

Figure 70 on page 214 shows a view of a sample program for MQSeries on Notes client.



The Use of MQLSX sample

Connected to server ESBserv.

Click [Send & Receive] or [Send Only] button after entering Text to be sent.If you have already sent a text then click [Receive Only] button.

Message ID: MQTEST199908091359370001

Text to be sent: Hello, ESB world!

Text received: 99/08/09 1:59:37 PM Text received: Hello, ESB world!

Sent Date & Time: 99/08/09 1:59:37 PM

Received Date & Time: 99/08/09 2:00:33 PM

Figure 70. Sample program on Notes Client

9.2.1 Process flow

The sample UseMQLSX includes the Notes database UseMQLSX.nsf for clients and the following three source files as the server programs:

- **MQCommon.lss**: Source file including Public class MQAccessClass
- **PUBLISH.lss**: Source file including Published class UseMQLSXClass
- **SERVER.lss**: Source file including Server class HostSimulatorClass

The following sections outline the MQAccessClass, UseMQLSXClass, and HostSimulatorClass, which are the main elements in the server program. It also explains the flow of their process.

9.2.1.1 Outline of the Public class MQAccessClass

MQAccessClass is a component class that consolidates a series of processes performed on the MQLSX to facilitate the sending and receiving of text (messages). By reusing MQCommon.lss, which includes the MQAccessClass, you can send and receive messages without consciousness of the MQLSX.

Loading MQLSX

To use MQLSX, you must load MQLSX using a UseLSX statement.

	Object: (Globals)	**Script: (Options)**
For Windows NT, enter:	`Uselsx "mqlsx"`	`’use MQLSX (NT)`

For AIX, enter:	`Uselsx "/usr/lpp/mqm/`
	`mqlsx/lib/libmqlsx.a"`
		`’use MQLSX (AIX)`

> **Hint**
>
> When [**Insert -> Insert LSX File**] is selected to insert the MQLSX for a project, a statement is unnecessary.

Definition of the MQLSX class variables

The class provided by the MQLSX is defined as the member variable of the MQAccessClass. At this time, each member variable is defined as a private variable to prevent direct referral from the outside.

Object: (Globals) **Script: (Declarations)**

```
Public Class MQAccessClass
…
Private itsSess                  As MQSession
Private itsQMgr                  As MQQueueManager
Private itsPutOpt                As MQPutMessageOptions
Private itsGetOpt                As MQGetMessageOptions
Private itsQueue                 As MQQueue
Private itsQMsg                  As MQMessage
…
```

Creating MQAccessClass objects

Sub New() is defined to create the object of the MQAccessClass. The setting of the source is enabled by calling the following member variable values to make the MQAccessClass a generic class:

- Queue manager name (`Private itsQMgrName As String`)
- Sending queue name (`Private itsSendQName As String`)
- Receiving queue name (`Private itsRecvQName As String`)
- Receiving waiting time (`Private itsWaitTime As Long`)

Object: (Globals) **Script: (Declarations)**

```
Sub New(sQMgrName As String, _
  sSendQName As String, _
  sRecvQName As String, _
  lWaitTime As Long)
  …
```

Connecting to the queue manager

To send and receive data (messages) through MQLSX, you must connect to the queue manager that manages the queue for sending or receiving the messages.

To do this, first create an MQSession object, which is the root class of the MQLSX. Then, create the MQQueueManager by calling the AccessQueueManager provided by the MQSession class. The connection to the specific queue manager is completed when you set the queue manager name (itsQMgrName) specified by the calling source as the argument.

Object: (Globals) **Script: (Declarations)**

```
Private Sub Connect()
  Set itsSess = New MQSession
  Set itsQMgr = itsSess.AccessQueueManager(itsQMgrName)
  …
```

Sending a message

Complete the following process to send messages:

1. Set the send option.

   The message send option creates the MQPutMessageOptions object and sets it according to the option properties.

   Object: (Globals) **Script: (Declarations)**

   ```
   Private Sub OpenSendQueue ()
     Set itsPutOpt = New MQPutMessageOptions
     itsPutOpt.Options = _
     MQPMO_FAIL_IF_QUIESCING + MQPMO_NO_SYNCPOINT
     …
   ```

2. Connect to the sending queue.

   To connect the sending queue, create an MQQueue object by calling the AccessQueue provided by the MQQueueManager. Then, specify `MQOO_OUTPUT` to open it as a sending queue.

   Object: (Globals) **Script: (Declarations)**

   ```
   Continuation of Private Sub OpenSendQueue ()
     …
     Dim lOpenOpts   As Long
     lOpenOpts = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING
     Set itsQueue = itsQMgr.AccessQueue(itsSendQName,lOpenOpts, _
       "","","")
     …
   ```

3. Send a message.

   You send a message by the MQQueue class Put method, by creating an MQMessage object. Then, set the text data that will be conveyed from the call source using the WriteString member. It deletes the MQMessage object once the send is completed.

Hint

- If you explicitly set a value according to the properties of the MessageID property and the CorrelationID property of the MQMessage class, you can send the specific data possessed by that value.
- In the UseMQLSX sample, the LSServer class MessageIDClass is defined to create a dedicated MessageID.
- You can set MQMessage class Expiry property to automatically delete from the queue any messages that have not been received within the set time.

Object: (Globals) Script: (Declarations)

```
Private Sub PutMsg(sInput As String)
  Set itsQMsg = New MQMessage
  itsQMsg.WriteString sInput
  itsQMsg.Format = MQFMT_STRING
  itsQMsg.Persistence = MQPER_PERSISTENT
  itsQMsg.Messageid = itsMsgID 'Set MQ message ID
  itsQMsg.Expiry = kEXPIRY 'Set expiration time
  itsQueue.Put itsQMsg, itsPutOpt
  Delete itsQMsg
  …
```

4. Release the connection to the sending queue.

   After completing the sending of the message, release the connection to the sending queue, by deleting the now unnecessary MQQueue object and MQPutMessageOptions object.

Object: (Globals) Script: (Declarations)

```
Private Sub CloseSendQueue()
  Delete itsQueue
  Delete itsPutOpt
  …
```

Receiving a message

Complete the following steps to receive messages:

1. Set the receive option.

   The message receive option creates the MQGetMessageOptions object and sets it according to the option properties. It also sets the waiting time set according to the call source in the WaitInterval property.

   Object: (Globals) **Script: (Declarations)**

   ```
   Private Sub OpenReceiveQueue ()
     Set itsPutOpt = New MQGetMessageOptions
     itsGetOpt.Options = MQGMO_WAIT + MQGMO_FAIL_IF_QUIESCING + _
       MQGMO_NO_SYNCPOINT
     itsGetOpt.WaitInterval = itsWaitTime * 1000
     …
   ```

2. Connect to the receiving queue.

   To connect the receiving queue, create an MQQueue object by calling the AccessQueue provided by the MQQueueManager. Then, specify `MQOO_INPUT_SHARED` to open it as a receiving queue.

   Object: (Globals) **Script: (Declarations)**

   ```
   Continuation of Private Sub OpenReceiveQueue ()
     …
     Dim lOpenOpts   As Long
     lOpenOpts = MQOO_INPUT_SHARED + MQOO_FAIL_IF_QUIESCING
     Set itsQueue = itsQMgr.AccessQueue(itsRecvQName,lOpenOpts,_
       "","","")
     …
   ```

3. Receive a message.

   You receive messages by the Get method of the MQQueue class when you create an MQMessage object. You can receive only messages that have a specific message ID by using the MessageID property. When the receiving process ends normally, it acquires the text data of the length returned by the MessageLength property using the ReadString of the MQMessage class and returns it to the call source. Lastly, it deletes the MQMessage object.

   Object: (Globals) **Script: (Declarations)**

   ```
   Private Function GetMsg() As String
     Set itsQMsg = New MQMessage
     itsQMsg.MessageID = itsMsgID   ' Set ID of message being received
     itsQueue.Get itsQMsg, itsGetOpt
     sMsg = itsQMsg.ReadString(itsQMsg.MessageLength)
   ```

```
  Delete itsQMsg
  …
```

4. Release the connection to the receiving queue.

   After completing the receiving of the message, release the connection to the receiving queue by deleting the now unnecessary MQQueue object and MQPutMessageOptions object.

   Object: (Globals) **Script: (Declarations)**

   ```
   Private Sub CloseReceiveQueue ()
     Delete itsQueue
     Delete itsGetOpt
     …
   ```

Releasing the connection to the queue manager

After completing the message sending or receiving processes, call the Disconnect method for the MQQueueManager class to release the connection to the queue manager. Delete the unnecessary MQQueueManager object. This process is called from the Delete() procedure, which deletes the MQQueueManager object.

Object: (Globals) **Script: (Declarations)**

```
Sub Delete()
  Call Disconnect()
  …
End Sub

Private Sub Disconnect ()
  itsQMgr.Disconnect
  Delete itsQMgr
  …
End Sub
```

Sending data

The MQAccessClass provides a text data send function to an external resource as the Public procedure Send(). Send() executes the process by sequentially calling the Private procedure of the MQAccessClass that was explained previously.

Object: (Globals) **Script: (Declarations)**

```
Public Sub Send( _
  sText As String, _           'Input: text to be sent
  lRC As Long _                'Output: return code
  )
  Call Connect()               'Connect to MQ
  Call OpenSendQueue()         'Open send queue
```

```
  Call PutMsg(sText)              'Send MQ message
  Call CloseSendQueue()           'Close send queue
  …
```

Receiving data

The MQAccessClass provides a text data receiving function to an external routine as the Public procedure Receive(). Receive() executes the processing by sequentially calling the Private procedures of the MQAccessClass that was explained previously.

Object: (Globals) **Script: (Declarations)**

```
Public Function Receive( _
  lRC As Long _                  ''Output: return code
  ) As String                    ''Return: text received
  Call Connect()                  'Connect to MQ
  Call OpenReceiveQueue()        ''Open receive queue
  Receive = GetMsg()             ''Receive MQ message
  Call CloseReceiveQueue()       ''Close receive queue
  …
```

9.2.1.2 Outline of the Published class UseMQLSXClass

The Published class UseMQLSXClass is defined in PUBLISH.lss and provides the function for sending and receiving messages to the ESB client (currently the Notes client) using the MQSeries.

Definition of the UseMQLSXClass class variable

The UseMQLSXClass defines the MQAccessClass as a member variable.

Object: (Globals) **Script: (Declarations)**

```
Published class UseMQLSXClass
  itsMQObj As MQAccessClass
```

Creating UseMQLSXClass objects

Open the Notes form **Use of the MQLSX sample** and click the [**CreateObject**] button. When the creation of a UseMQLSXClass object is requested, Sub New is called. It then creates an MQAccessClass object and saves it to the member variable itsMQObj, which in turn enables the use of the member provided by the UseMQLSXClass. The argument for creating an UseMQLSXClass object is defined in the following ways as a constant in PUBLISH.lss and can be changed to a conformity running environment:

- Queue manager name

```
Private Const QMGR_SERVER = "SIMPLE"
```

- Sending queue name

  ```
  Private Const Q_PUBLISHER_TO_SERVER = "PUBLISHER.TO.SERVER"
  ```

- Sending queue name

  ```
  Private Const Q_SERVER_TO_PUBLISHER = "SERVER.TO.PUBLISHER"
  ```

- Receiving wait time

  ```
  Private Const WAIT_TIME = 10
  ```

Object: (Globals) **Script: (Declarations)**

```
Sub New()
  Set itsMQObj = New MQAccessClass( _
    QMGR_SERVER, _
    Q_PUBLISHER_TO_SERVER, _
    Q_SERVER_TO_PUBLISHER , _
    WAIT_TIME)
  …
```

Sending and receiving data

Enter the text to be sent in the Notes form **Use of the MQLSX sample**. Click the [**Send & Receive**] button to call the Public procedure SendText() of the UseMQLSXClass. Calling the member and property provided by the MQAccessClass in the following sequence causes SendText() to send the entered text and receive the text returned by the HostSimulatorClass.

1. Send the text.
2. Obtain a message ID, which is automatically generated when the text is sent.
3. Receive the text.

> **Hint**
>
> In addition to SendText(), the UseMQLSXClass provides SendOnly() for data sending alone and ReceiveOnly() for data receiving alone.

Object: (Globals) **Script: (Declarations)**

```
Function SendText( _
  sSendText As String, _             ''Input: text to be sent
  sMsgID As String, _                ''Output: message ID
  sSendDate As Variant, _            ''Output: sent date & time
  sRecvDate As Variant, _            ''Output: received date & time
  lRC As Long _                      ''Output: return code
  ) As String                        ''Return: text received
  Call itsMQObj.Send(sSendText, lRC)    ''Send text
```

```
sMsgID = itsMQObj.MessageID           ''Get message ID
sSendDate = Now                       ''Get sent date & time
SendText = itsMQObj.Receive(lRC)      ''Receive text
sRecvDate = Now                       ''Get received date & time
…
```

Reusing MQLSX class objects

This sample creates, deletes, and calls members of MQLSX class objects within a Published class object (client instance) by providing MQAccessClass objects to the Published class UseMQLSXClass as member variables. When you wish to reuse MQLSX client objects between multiple client threads, you can do so by holding the MQAccessClass object as a member constant and making single or multiple definitions for the LSServer class that provides the same Public procedure as the UseMQLSXClass. This method is effective, for example, when using a queue that is managed by means of an MQLSX server, which uses an MQSeries client to operate on another machine.

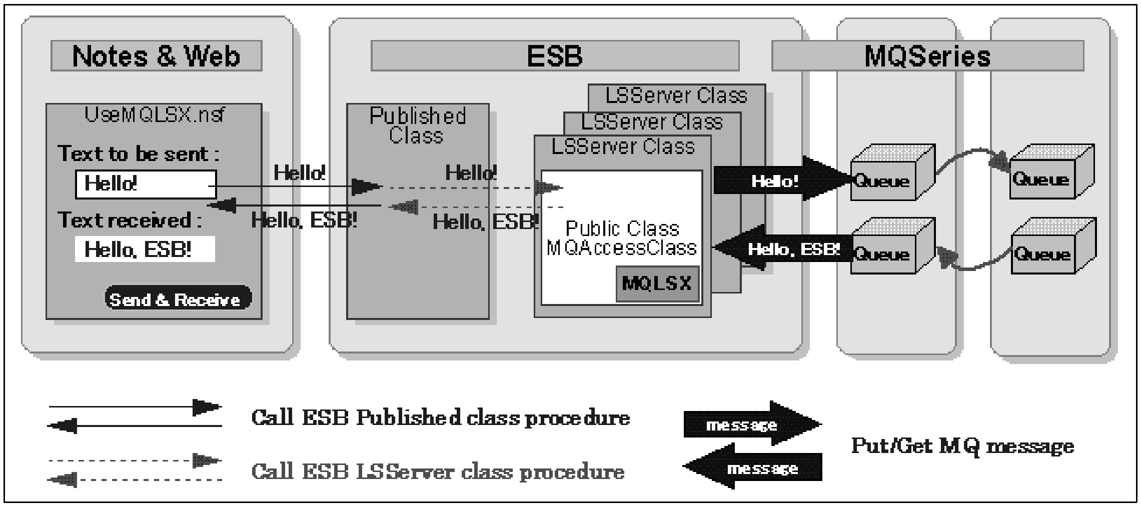


Figure 71. Flow among each component for MQSeries and ESB

9.2.1.3 Outline of the LSServer class HostSimulatorClass

The LSServer class HostSimulatorClass is defined in the SERVER.lss. When it receives a message sent by UseMQLSXClass, it immediately creates a receipt confirmation message. Then, it replies using the MQSeries. The HostSimulatorClass uses the SsTimer class to perform constant message monitor processing.

Defining the HostSimulatorClass class variables

The HostSimulatorClass defines the MQAccessClass object as a member variable. It defines the SsTimer class object as a member variable for continuously running the message monitor processing.

Object: (Globals) Script: (Declarations)

```
Published class HostSimulatorClass
  Private itsTimer                As SsTimer
  Private itsMQObj As MQAccessClass
  …
```

Creating HostSimulatorClass objects

LSServer class HostSimulatorClass objects are created when projects are run and called by Sub New(). The HostSimulatorClass object starts the procedure GetMessage() for constantly processing the monitoring of messages sent by the UseMQLSXClass.

Object: (Globals) Script: (Declarations)

```
Sub New()
  Set itsTimer = New SsTimer(1)
  On Event Alarm From itsTimer Call GetMessage
  …
```

Message monitor processing

The GetMessage() procedure called in the SsTimer class function runs through the following process:

1. Create the MQAccessClass object.
2. Wait for the message receipt.
3. Receive a message reply.
4. Repeat the process two to three times.

The argument for creating a MQAccessClass object is defined as follows as the constant SERVER.lss. Therefore, it can be changed in conformity with the running environment.

- Queue manager name

  ```
  Private Const QMGR_SERVER = "SIMPLE"
  ```

- Sending queue name

  ```
  Private Const Q_PUBLISHER_TO_SERVER = "PUBLISHER.TO.SERVER"
  ```

- Receiving queue name

  ```
  Private Const Q_SERVER_TO_PUBLISHER = "SERVER.TO.PUBLISHER"
  ```

- Receiving wait time

```
Private Const WAIT_TIME = 600
```

Object: (Globals) **Script: (Declarations)**

```
Sub GetMessage(source As SsTimer)
  source.Enabled = False               'Disable timer
  Set itsMQObj = New MQAccessClass( _
    QMGR_SERVER, _
    Q_SERVER_TO_PUBLISHER, _
    Q_PUBLISHER_TO_SERVER, _
    WAIT_TIME)
  Do While lRC = CC_OK
    itsMQObj.MessageID = ""            'Receive any message
    sText = itsMQObj.Receive(lRC)      'Wait & receive message
    sText = Format$(Now()) & " Text received: " & _
    Chr(13) & Chr(10) & sText          'Make reply
    Call itsMQObj.Send(sText, lRC)     'Reply to the message
  Loop
  Set itsMQObj = Nothing
  …
```

9.2.2 Setup procedure

This section explains the setup procedure of the UseMQLSX example.

9.2.2.1 Server side

Open the command prompt, and enter the following series of commands. Before entering them, verify that the definition file (*.tst) of the MQSC command is in the current directory.

For Windows NT, check:

```
> CD c:\Program Files\Lotus ESB Runtime\Samples\UseMQLSX
```

For AIX, check:

```
> mkdir /home/mqm/usemqlsx
> cd /home/mqm/usemqlsx
> cp /usr/lpp/esb/samples/En_US/UseMQLSX/* .
```

In case one queue manager is used in a single machine, perform the following steps:

1. Create a queue manager named SIMPLE:

```
> crtmqm SIMPLE
```

2. Start running the queue manager SIMPLE:

```
> strmqm SIMPLE
```

3. Create two queues, PUBLISH.TO.SERVER and SERVER.TO.PUBLISHER, to be used for sending and receiving messages:

```
> runmqsc SIMPLE < SIMPLE.tst
```

In case individual queue managers are used on two machines, apply the following steps to perform the tasks on different machines by separating them into PUBLISHER and SERVER:

1. Create a queue manager named PUBLISHER

```
> crtmqm PUBLISHER
```

2. Start running the queue manager PUBLISHER:

```
> strmqm PUBLISHER
```

3. Edit the MQSC command file PUBLISH.tst. Set the correct TCP/IP host name for that machine with the channel definition command `DEFINE CHANNEL` and the parameter CONNAME.
4. Create a remote queue PUBLISHER.TO.SERVER, a local queue SERVER.TO.PUBLISHER, a transmission queue, and a channel for sending and receiving messages:

```
> runmqsc PUBLISHER < PUBLISH.tst
```

5. Create a queue manager named SERVER:

```
> crtmqm SERVER
```

6. Start running the queue manager SERVER:

```
> strmqm SERVER
```

7. Create a remote queue SERVER.TO.PUBLISHER, a local queue PUBLISHER.TO.SERVER, a transmission queue, and a channel for sending and receiving messages:

```
> runmqsc SERVER < SERVER.tst
```

8. Start running the listener program on the SERVER side:

   For Windows NT, run the following program:

```
> runmqlsr -m SERVER -t TCP
```

   For AIX, check the following settings. Refer to the MQSeries document for details.

a. Check for the following settings in /etc/services. If they are not present, add them.

```
MQSeries  1414/tcp  # MQSeries channel listener
```

b. Check for the following settings in /etc/inetd.conf. If they are not present, add them.

```
MQSeries  stream  tcp  nowait  mqm  /home/mqm/mqchl mqchl
```

In this example, we assume that there is a Shell Script file such as the following one in /home/mqm/mqchl:

```
Content of /home/mqm/mqchl
   -------------------------------------------
   #!/bin/sh
   export LANG=Ja_JP
   export LC_MESSAGES=Ja_JP.IBM-943
   exec /usr/lpp/mqm/bin/amqcrsta -m SERVER
   -------------------------------------------
```

c. Validate the inetd.conf setting:

```
> refresh -s inetd
```

9. Start running the channel on the PUBLISHER side. Open the respective command prompts or windows again, and run the following commands:

```
> runmqchl -m PUBLISHER -c PUBLISHER.TO.SERVER
> runmqchl -m PUBLISHER -c SERVER.TO.PUBLISHER
```

10. Open the respective prompts PUBLISH.lsp and SERVER.lsp in IDE. Edit PUBLISH.lss and SERVER.lss, and change the value of the constant QMGR_SERVER from “SIMPLE” to “PUBLISHER” and “SERVER”.

9.2.2.2 Client side

The client also should be set as explained here to run the sample program:

1. Register the UseMQLSX.nsf file in the Domino server or copy it into the Notes data directory (for example, NOTES\DATA) of the client that will do the running.
2. Register UseMQLSX.nsf in the work space. Specify the database name `UseMQLSX` in [**Open file data base**] on Notes. Then, select [**Add icon**]. It records the internal configuration of this sample.

9.2.3 Usage

This section explains how to run the UseMQLSX sample.

9.2.3.1 Server side

Before running, verify the following Uselsx statement located among the (Options) for MQCommon.lss, which is the one for the respective platforms to be run. If necessary, make one of them a comment:

For Windows NT, check:

```
Loading of Uselsx "mqlsx" MQLSX
```

For AIX, check:

```
Loading of Uselsx "/usr/lpp/mqm/mqlsx/lib/libmqlsx.a" MQLSX
```

Select one of following cases according to the environment that the sample program is testing:

- In case one queue manager is used from one ESB project SIMPLE.lsp, start running it after opening the project SIMPLE.lsp in IDE.
- In case individual queue managers are used on each machine, you have two options:
  - Start running them after opening the project PUBLISH.lsp on the machine where the queue manager PUBILSHER is running.
  - Start running them after opening the project SERVER.lsp on the machine where the queue manager SERVER is running.

Note

- When using MQLSX, you must specify the environment variable `GMQ_XLAT_PATH` correctly. See the document, which is attached the MQLSX component package, published by Lotus for details.

  For Windows NT, enter:

  ```
  GMQ_XLAT_PATH=,f:\mqm\MQLSX\conv
  ```

  For AIX, enter:

  ```
  GMQ_XLAT_PATH=/usr/lpp/mqm/mqlsx/conv
  ```

- When running on AIX, select the item [**Change/ display maximum data size of LSCube-system configuration-engine and parameter setting-engine**]. Set [**Maximum data size of engine**]. Then, restart the ESB Runtime.

9.2.3.2 Client side

Open the Notes database UseMQLSX.nsf. Run the sending and receiving of messages according to the MQLSX form usage sample.

Chapter 10. Deploying ESB applications

This chapter explains the deployment of ESB applications, which operate through ESB System Manager.

10.1 Outline

This section discusses the overall flow from development to deployment using ESB System Manager. It also focuses on the settings at the time of deployment and on more efficient deployment.

10.1.1 Overall flow from deployment to operation

The flow from development to deployment of an ESB application is shown in Figure 72.

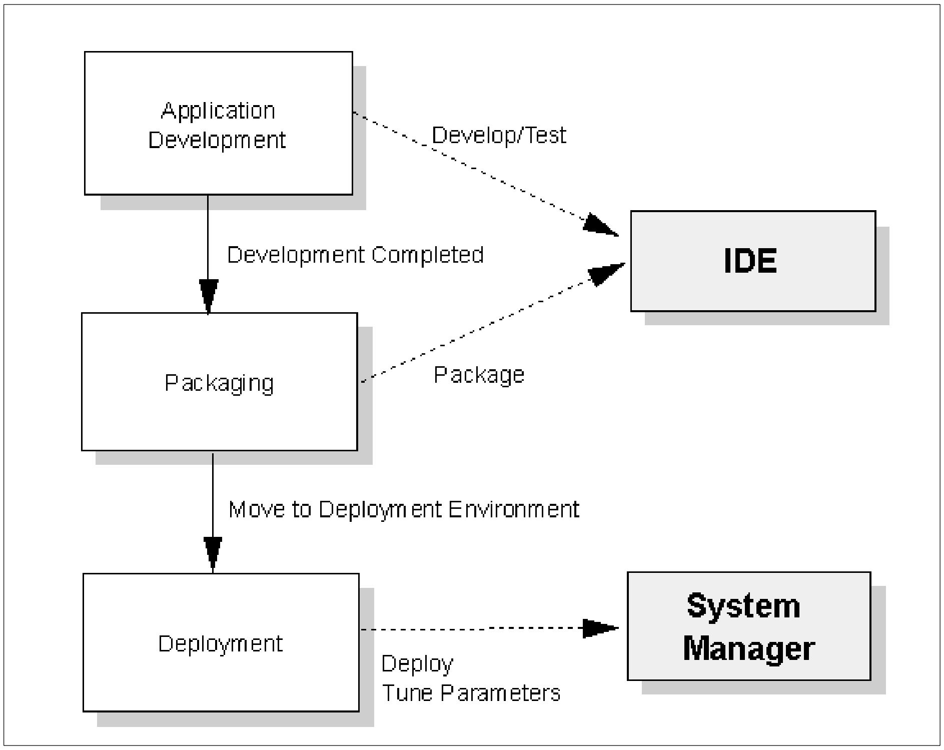


Figure 72. ESB development and deployment flow

The applications developed in ESB IDE are packaged by the developer using ESB IDE. Once the packaging is completed, the created package file (*.lpk) is placed in the deployment environment, which makes it possible to begin the deployment by the ESB System Manager. The ESB System Manager can change the various parameter settings of the respective package files other than package file deployment. The ESB is designed for deployment with optimum performance by setting the parameters appropriately.

10.1.2 Package files and projects

A package file is composed of a single project or multiple package files. When a package file is composed of multiple package files, it is referred to as a *group package file* to identify a single package file. Package files include project files where the information relating to projects is stored, object files that are inserted into projects (object files displayed on the IDE project browser), and object files which are compiled from the source code in the project. The LSX being inserted into projects and the object files using **Use** statements are not included in package files. Therefore, you must lay those out separately in the deployment environment. Figure 73 shows the difference between a package file and a group package file.

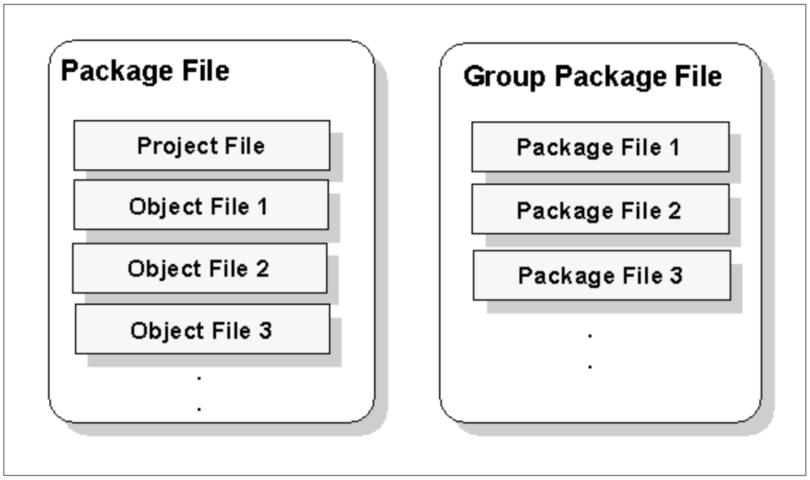


Figure 73. Package file and group package file

10.2 Project deployment flow

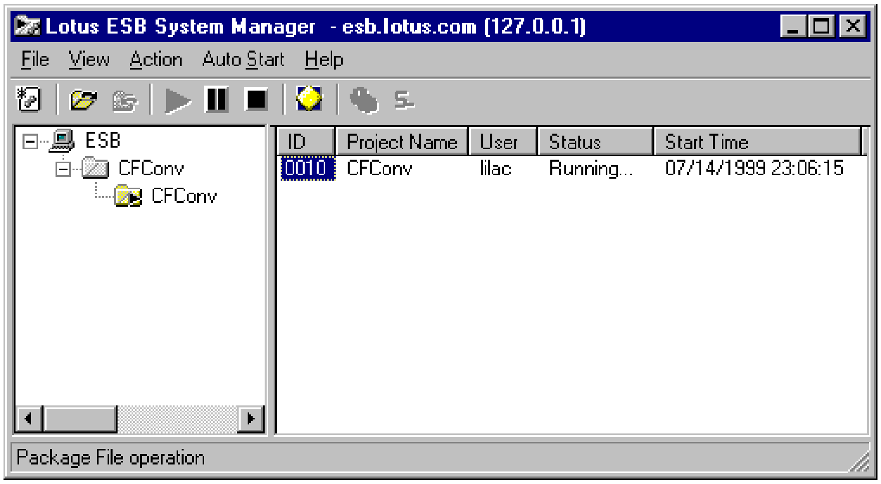
Let us look at the actual deployment flow using a package file created in IDE. First, start the System Manager to be used for the deployment. For the

Windows NT version, you can start from the Program Manager. For the AIX version, start `hpwisma` from the terminal window.

10.2.1 Starting the project

A project has to be started as described here:

1. After starting System Manager, open a package file from the path **File -> Open Package File**. The status in which a package file is opened implies that System Manager can operate the package file. An opened package file displayed in the Configuration panel can be manipulated for starting projects and setting the various project parameters.
2. Select the package in the Configuration panel.
3. Select **Action -> Start** or click the **Start** button ▶. The project starts to run. When multiple package files are opened, select **Action -> Start All** to start running all projects displayed on the configuration panel. The icon displayed in the Configuration panel changes for projects that are running. You can confirm that they are running. As shown in Figure 74, the user **lilac** is running the project **CFConv**.


Lotus ESB System Manager - esb.lotus.com (127.0.0.1)
File View Action Auto Start Help
ESB
CFConv
CFConv
Package File operation

ID	Project Name	User	Status	Start Time
0010	CFConv	lilac	Running...	07/14/1999 23:06:15

Figure 74. System Manager running a project

Hint

In the Windows NT version, when you specify the start of project from System Manager, the Runtime Monitor starts automatically and monitors the project running the status, the thread status, and so on.

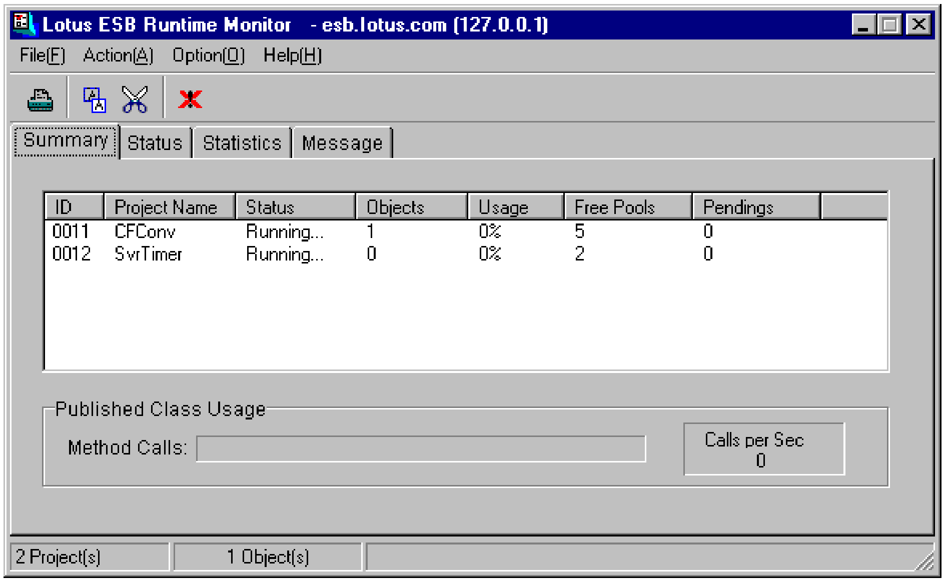
10.2.2 Managing the deployment conditions

At the deployment of a project, you are required to check the status of project that was run and the client connection in a timely manner. ESB provides the Runtime Monitor (integrated with System Manager in AIX) as a tool for the monitoring runtime. Try using the Runtime Monitor (System Manager in AIX) to obtain the various information and to manage the client:

1. Monitor the project deployment status:

   Complete the following procedure to display the summary panel:

   - For Windows NT, click the **Summary** tab on the NT Runtime Monitor as shown in Figure 75.
   - For AIX, select the **Summary** icon in the Monitor folder of System Manager.



ID	Project Name	Status	Objects	Usage	Free Pools	Pendings
0011	CFConv	Running...	1	0%	5	0
0012	SvrTimer	Running...	0	0%	2	0

Published Class Usage

Method Calls:

Calls per Sec
0

2 Project(s)

1 Object(s)

Figure 75. Runtime Monitor Summary panel

The status of each project is displayed by project basis in the Summary panel. The number of objects, the number of free pools, and the usage rate displayed can be considered as a rough standard for setting the appropriate pool size. Refer to 10.3.2, “Setting the pool size” on page 244, to learn more about setting the pool size.

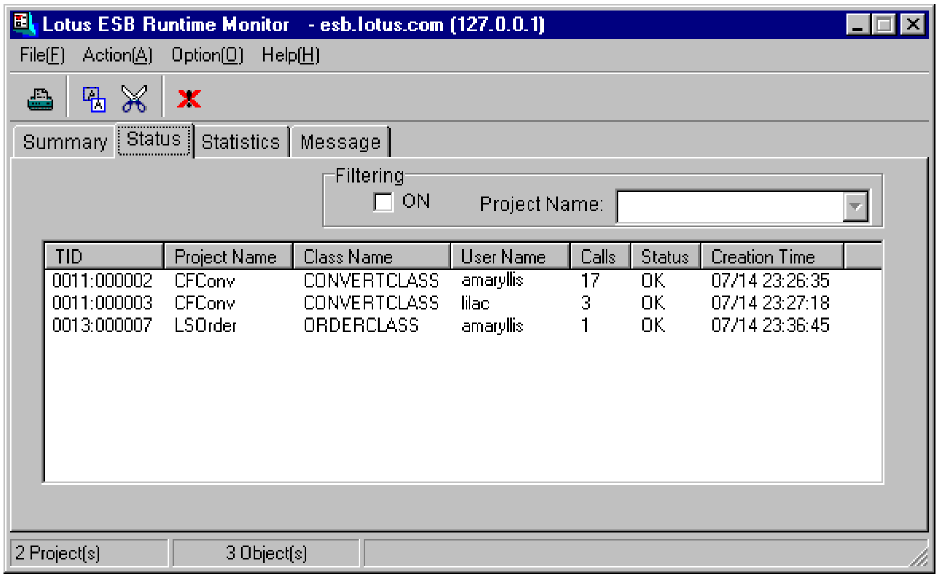
Note

The ID is a unique process number (engine ID) of ESB. The process number is not identical from the number assigned by the operating system, so be careful. This number is used to identify the message of the project.

2. Thread management.

   Complete the following procedure to display the Status panel:

   - For Windows NT, click the **Status** tab on the Runtime Monitor as shown in Figure 76.
   - For AIX, select the **Status** icon in the Monitor folder of System Manager.


Lotus ESB Runtime Monitor - esb.lotus.com (127.0.0.1)
File(F) Action(A) Option(O) Help(H)
Summary
Status
Statistics
Message
Filtering
□ ON
Project Name:

TID	Project Name	Class Name	User Name	Calls	Status	Creation Time
0011:000002	CFConv	CONVERTCLASS	amaryllis	17	OK	07/14 23:26:35
0011:000003	CFConv	CONVERTCLASS	lilac	3	OK	07/14 23:27:18
0013:000007	LSOrder	ORDERCLASS	amaryllis	1	OK	07/14 23:36:45

2 Project(s) 3 Object(s)

Figure 76. Runtime Monitor Status panel

The information is displayed on a client thread basis in the Status panel. You can monitor the project name to which it belongs, the Published class name, and the user name of the created the object. You should actually connect to the running Published class from the client and check the change of the status panel display.

In ESB, a client thread is created on the server when a client application creates a Published class object. In other words, monitoring each Published class object in the deployment environment means managing the basis of client thread.

To monitor whether the requests from clients are being processed normally, you need to check the status of the client thread corresponding to the respective Published class objects on the Status panel. You need to check the thread conditions by such means as test deployment. Plus, you need to assess whether the current pool setting is fit to gain adequate performance in the actual operation under business.

3. Display specific projects.

   A specific project can be displayed alone on the Status panel. While multiple projects are operating, select **Filtering Check on** and select a project name to display only the thread information created from that project. You can use the filtering function to display just the project you want to monitor.

4. Delete threads.

   Select a client thread on the Status panel. Then, select **Action -> Kill Thread** to delete a thread. Use this procedure if a thread cannot be deleted from the client during development or testing, or if a problem occurs during deployment and it becomes necessary to delete a specific thread. Select **Action -> Kill Thread (Forced)** only when a thread is not deleted by normal deletion. When forced deletion has been executed, you should restart the corresponding project as soon as possible.

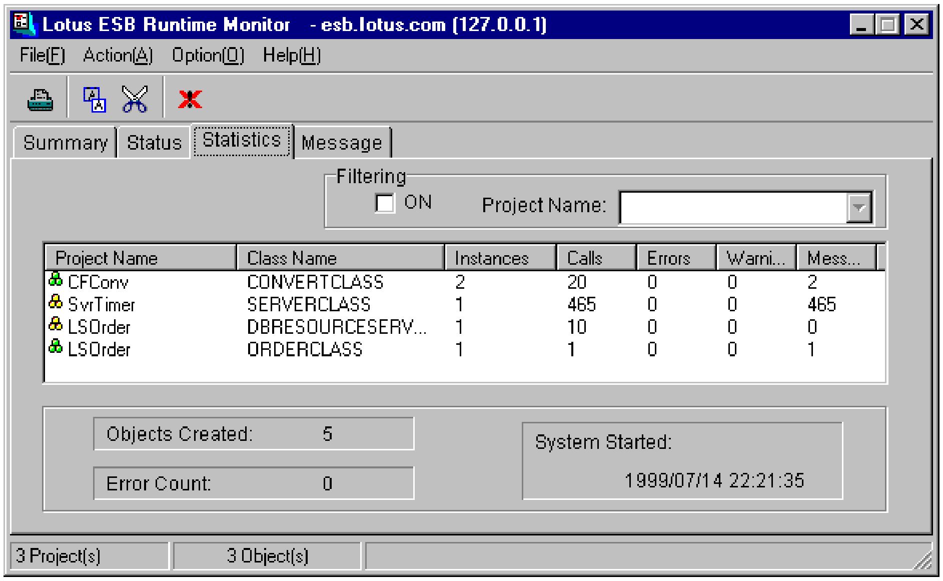
Hint

For Windows NT, you can also delete threads by right-clicking the corresponding client thread and selecting the **Kill Thread** pop-up menu.

5. Display class statistical information.

   Complete the following procedure to display the Statistics panel:

   - For Windows NT, click the **Statistics** tab on the Runtime Monitor as shown in Figure 77.
   - For AIX, select the **Statistics** icon in the Monitor folder in the System Manager.


Lotus ESB Runtime Monitor - esb.lotus.com (127.0.0.1)
File(F) Action(A) Option(O) Help(H)
Summary
Status
Statistics
Message
Filtering
□ ON
Project Name:

Project Name	Class Name	Instances	Calls	Errors	Warni...	Mess...
CFConv	CONVERTCLASS	2	20	0	0	2
SvrTimer	SERVERCLASS	1	465	0	0	465
LSOrder	DBRESOURCESERV...	1	10	0	0	0
LSOrder	ORDERCLASS	1	1	0	0	1

Objects Created: 5

Error Count: 0

System Started: 1999/07/14 22:21:35

3 Project(s) 3 Object(s)

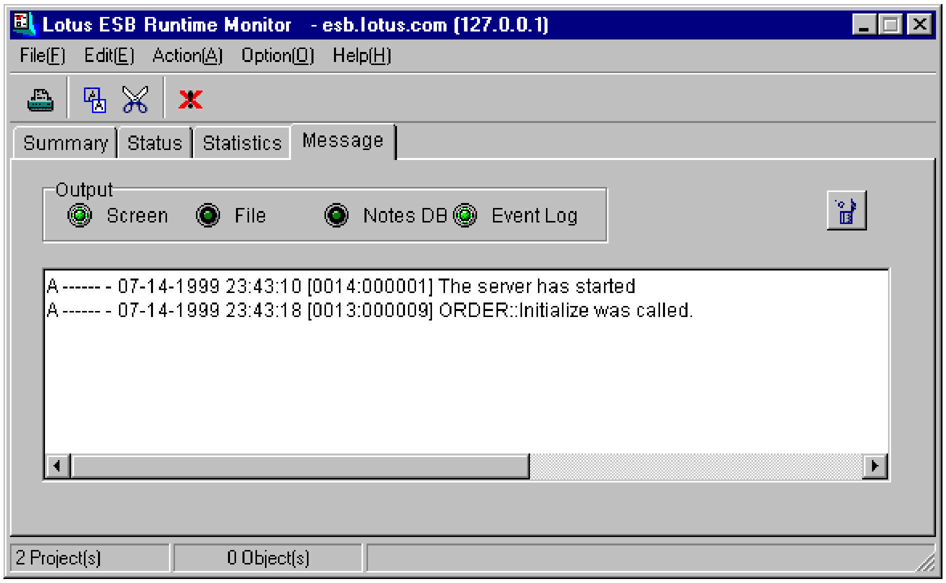
Figure 77. Runtime Monitor Statistics panel

The cumulative information is displayed on a per-class basis on the Statistics panel. It includes the statistics on the number of classes created for each project, on the number of method calls, and on the number of error, warning, or information messages. Because it allows such items as the number of objects created and the number of calls created per unit time of measurement, it is useful as a guideline for the required size of the main memory area. It is also useful as criteria for subdividing and integrating projects. When deploying a project, it is beneficial to check the degree that each project is being used, for example, for planning the resources for the deployment.

6. Set and display messages.

   Complete the following procedure to display the Message panel:

   - For Windows NT, click the **Message** tab on the Runtime Monitor as shown in Figure 78 on page 236.
   - For AIX, select the **Message** icon in the Monitor folder in the System Manager.


Lotus ESB Runtime Monitor - esb.lotus.com (127.0.0.1)
File(F) Edit(E) Action(A) Option(O) Help(H)
Summary
Status
Statistics
Message
Output
Screen
File
Notes DB
Event Log
2 Project(s)
0 Object(s)

```
A ------ - 07-14-1999 23:43:10 [0014:000001] The server has started
A ------ - 07-14-1999 23:43:18 [0013:000009] ORDER::Initialize was called.
```

Figure 78. Runtime Monitor Message panel

The display shown in Figure 78 shows the messages on the Message panel that are output by the project and the type of messages to be output. The message can be set to disable or enable to such output as a screen, file, NotesDB, or Event log. Monitoring messages is important for checking the deployment status. By appropriately setting messages, you can verify the operating status of the project. When you feel that there is something wrong in the project that is running, you should check the displayed messages.

Hint

Disabling unnecessary message output sometimes helps improve the overall performance of ESB applications. Unnecessary message outputs by PRINT statements in the application are particularly likely to adversely affect performance. You should also remove the settings for outputs to the Notes database or files unless processing is based on those outputs. Normally, you should specify only logs and screen outputs.

The following example shows the setting of the outputs to the log, with all of the messages pertaining to the system and the error messages pertaining to Runtime and the application.

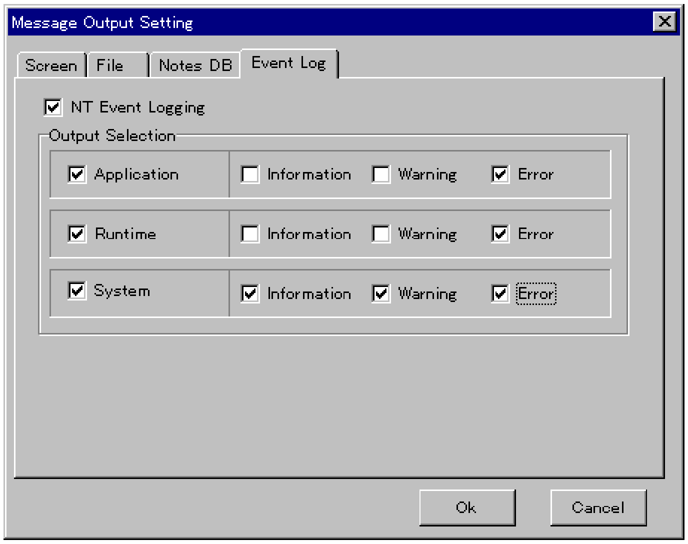
For Windows NT, when specifying the log to an output destination, it is written as an event log.

For AIX, when specifying the log to an output destination, it is written to the log provided as an ESB function.

a. Click the **Event** button (**Log** button for AIX) on the Message panel as shown in Figure 79.

b. Display the Event panel (Log panel for AIX).

c. Mark the check box that logs for NT events. Also, select the **Application** and **Runtime Error** check boxes and all of the **System** check boxes. Click the **OK** button.



☑ NT Event Logging

Output Selection

☑ Application   □ Information   □ Warning   ☑ Error

☑ Runtime   □ Information   □ Warning   ☑ Error

☑ System   ☑ Information   ☑ Warning   ☑ Error

Figure 79. ESB Message Output setting panel

d. Confirm the message that is output. Start and stop the project and repeat the connection from the client. Complete the following procedure to display the log:

- For Windows NT, start the Event View on Windows NT, and select **Log -> Application**.
- For AIX, select **SMIT**, and select **Applications -> ESB -> Log -> Show the Log**.

10.2.3 Stopping and starting projects

Stop the project first. Stopping a project refers to stopping the running alone, with the project placed in the main memory. At this time, you can resume the project immediately from the main memory. When you want to pause, resume a project under deployment, and select **Stop** without exiting.

The client process method for stopping a project is divided into three modes (*stopping modes*):

- **Warm**: The project stops when the client is disconnected. Specify stopping in Warm mode if you wish to safely stop in the deployment environment.
- **Cold** (default value): The project is stopped after the connected client is stopped. Specify the Cold mode when a forced stop is necessary during development tests or maintenance.
- **Force**: Forcibly stop the project regardless of the client condition. Force should not be used except for when the project cannot stop normally (could not stop even by Cold mode).

If the Stop or Start button is specified, the process for the connected client corresponds to the Stop mode specified in the ESB Project Properties of the machine, which is the platform on which ESB is running. Use the Project Properties dialog box to confirm or change the Stop mode:

At this point, stop a package in running mode, change the Stop mode to **Warm**, and then resume the package to run:

1. Select **Package** in the Configuration panel.
2. Select **Action -> Stop or Restart** or click the **Stop or Start** button ⏸. The project stops under holding in the main memory.
3. Select the **Machine** icon on the Configuration panel. Select **View -> Property** to display the Project Properties panel.
4. Select **Warm** mode on the Stop Mode panel. Then, click the **OK** button.
5. Select the stopped package in the Configuration panel.
6. Select **Action -> Stop or Restart**, or click the **Stop or Start** button ⏸.

10.2.4 Exiting a project

When you exit a project or when exiting a project has been specified, the project is removed from the main memory. This applies to the Stop mode specified on the Machine icon.

1. Select **Package** in the Configuration panel.
2. Select **Action -> Terminate**, or click the **Terminate** button ■.

10.2.5 Automatically starting a package

This section describes the autostart of a package when the operating system is started.

10.2.5.1 Starting automatically when the system starts

ESB can identify specific packages that automatically start when the machine starts such as for automatic deployment. Follow these steps:

1. Select **Package** in the Configuration panel.
2. Select **Auto Start -> Service**, or click the **Service** button ⚙.
3. When you restart the system, the registered package is started automatically.

   Figure 80 shows that CFConv is set as a service project.

> **Note**
>
> For the AIX version, automatic startup for the ESB system must be specified as well. Do it with SMIT on the AIX version.

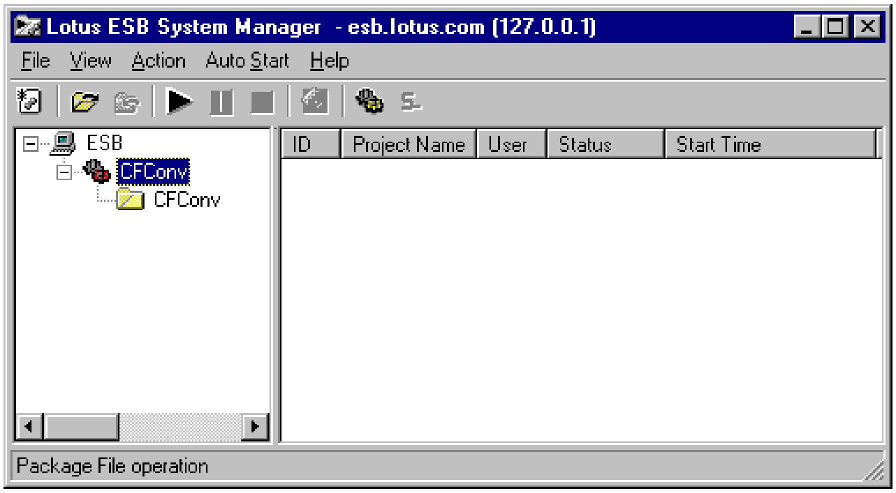


Figure 80. Service project on System Manager

10.2.5.2 Automatic startup upon user logon (NT version only)

In the Windows NT version of ESB Runtime, you may want to control startup so that the project starts automatically, when the user logs on, as described in the following cases:

- To separate the machine startup and the control of ESB
- For specific, preferred users to control startup
- When the ESB Engine Service cannot be started from Windows NT Service due to integration restrictions with other applications

ESB provides a way of registering a package in startup for such cases. Let's try registering a package for startup:

1. Select **Package** in the Configuration panel.
2. Select **Auto Start -> Startup**, or click the **Startup** button .
3. Log off, and then log on again with the user name you registered to automatically start up the specified package.

10.2.5.3 Automatically starting up multiple packages

Multiple projects are created when a large scale system is developed. To register these for automatic startup, create one consolidated group package file, and then register it. Now, let us create a group package and register it for automatic startup:

1. Select **File -> Create New Package File** to create an empty package file.
2. Open multiple package files to be used.
3. Drag and drop the **Package File** icon onto the group package file. A group package file containing multiple packages is created.
4. Select the group package you created. Then, click the **Service** button or the **Startup** button .

10.2.6 Starting the ESB Engine Service with a specific account

Note: This applies to Windows NT only.

For the Windows NT version of ESB Runtime installation, ESB Engine Service and the ESB Package Service are registered in the NT Service. ESB Engine Service is the default set for automatic startup. Leaving this setting "as is" allows it to run normally without any problems. In this case, ESB Engine Service starts with the ESB Service *user* specified when ESB was installed.

There may be cases where ESB cannot connect to the external resources with the account which ESB started, due to usage limitations of the application integrated with ESB, for example:

- Communication using the API of the PC3270 Emulator
- Communication using DB2 Connect on AS/400
- Using the Notes Class

In such cases, complete the following procedure to start the ESB Engine Service from the logged-on user:

1. Start **Service** on the Windows NT control panel. Then, select **Lotus ESB Engine Service** and click the **Stop** button.
2. Click the **Startup** button, select **Manual** in the group of Startup Type, and then click the **OK** button.
3. Click the **Close** button, and then terminate the Service window.
4. Right click the **Start** button, and select the **Open** menu. Open the **Start Menu** folder, and then switch to the **Program** and **Startup** folders.
5. Start Windows NT Explorer. Then, open the **Run Module** directory of Lotus ESB Runtime (C:\Program Files\Lotus ESB Runtime\bin).
6. Drag and drop the file **hpwarb2n.exe** from Windows NT Explorer to the Startup folder to create a shortcut.

When you complete these steps, the next time the ESB Engine Service starts is when the login is done by the user.

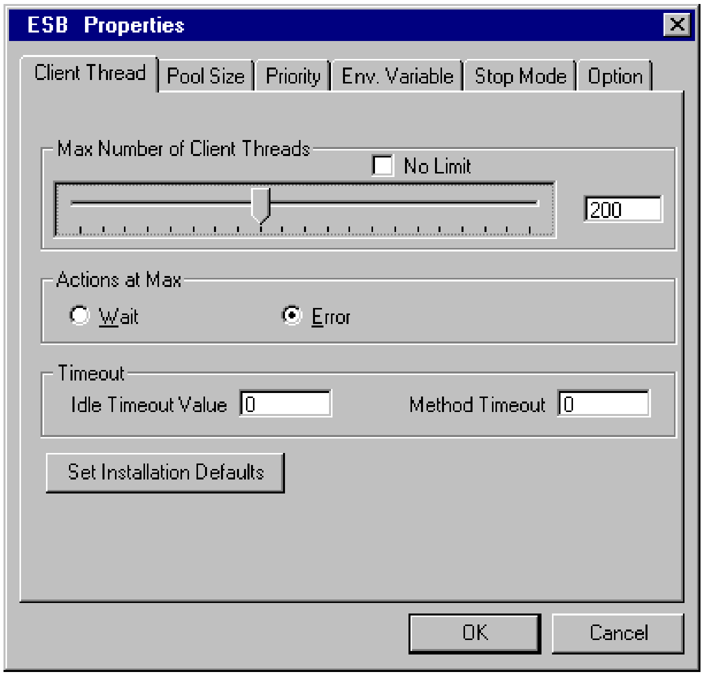
10.3 Setting properties

ESB has its own property settings to deploy the system conveniently.

10.3.1 Settings pertaining to client threads

It is necessary to run the project smoothly in deployment by automatically deleting objects (threads), which were discarded due to the exception of a network error and by managing the maximum number of connections. The settings pertaining to the client thread by adjusting the resources of the system for suitable settings in deployment are also required.

The settings pertaining to the client thread in the Project Properties are set on the Client Thread panel (Figure 81 on page 242).



Max Number of Client Threads

□ No Limit

200

Actions at Max

○ Wait ● Error

Timeout

Idle Timeout Value 0

Method Timeout 0

Figure 81. Project Property Client Thread panel

10.3.1.1 Max Number of Client Threads

The Max Number of Client Threads specifies the maximum number of client threads that can be created. This is the maximum number of Published class objects that can be created from a client at the same time. When “No Limit” is set, it creates client threads as much as the resources are available.

- When sufficient resources are available

  Set **No Limit**, if sufficient resources are available. In this scenario, there is a certain number of clients to be connected and no need to limit the number of connected clients.

- When there is a limit on the number of simultaneous connections

  Because machine resources are normally limited, it is required to prevent frequent swaps of the main memory for the operating system and avoid lowering the running speed under heavy load. By specifying the maximum number of threads, ESB limits the number of simultaneous connections and maintains the performance. In addition, **Actions at Max** can be specified in the settings, which causes the client to *wait* or *return an error* to the client when the maximum number of threads reaches the maximum limit.

In case “Wait” is specified, the server queues the requests. In other words, the client side shifts to the wait status in object creation (for example, CreateObject). After a Published class object is deleted during operation, a Published class object is created in the queued sequence.

When “Error” is specified, the client side produces an object creation error. For example, an automatic object cannot be created. You can attempt to reconnect by entering appropriate error handling (for example, the On Error statement) on the client side.

10.3.1.2 Idle Timeout

Idle Timeout can be set to delete unnecessary threads (Published class objects) automatically after a certain period of time when client/server communication is disabled due to such trouble as a network error.

“Idle Timeout Value” specifies the maximum period of time in which the created client thread can be alive and no method is called. For example, when this value is set to 60 seconds, if the client thread created by the CreateObject function is not called within 60 seconds after the creation, “Idle Timeout” is posted and the thread is deleted. If the initial value of “Idle Timeout Value” is 0, no timeout is generated.

You should set this value if you want to automatically delete the client thread which was not used due to a network error or trouble in the client computer, or when you want to prevent leaving the long term thread created by a certain client.

Hint

You must decide on the value to set for Idle Timeout according to the nature of the client threads (Published class objects) that are created. Because the lifetime for state-full threads is normally long, you should set it to a value of about 1,800 seconds (30 minutes). On the other hand, you should set it to a value of about 300 seconds (5 minutes) for stateless threads.

Note

The first digit of the timeout value set is rounded up to the next effective 10 second unit. For example, when 23 seconds is specified, it is treated internally as 30 seconds.

10.3.1.3 Method Timeout

If you can precisely predict the time until the method call is complete, you can set this value to automatically delete client objects (Published class objects), which can may indicate that a problem has occurred. You can use this setting to prevent unnecessary threads from being left in the main memory.

“Method Timeout” specifies, in seconds, the maximum period of time that a method is executed. Once the method is called, and the specified time elapsed before the method call completed for any reason, the client thread is automatically deleted. When the Default value is 0, no timeout occurs.

Hint

A normal method call does not take a long time. Consequently, you should set about 60 seconds as the timeout value.

Note

The first digit of the timeout value set is rounded up to the next effective 10 second unit. For example, when 23 seconds is specified, it is treated internally as 30 seconds.

10.3.2 Setting the pool size

Set the pool size when you want to improve the response time to object creation requests from clients. Set the following three pool size parameters:

- Initial pool size
- Additional pool size
- Maximum pool size

10.3.2.1 The thread pooling mechanism

When a request of the Published class object creation is received from a client, a corresponding client thread is created on the server. Creating threads imposes a certain cost, consequently. If possible, you should create several such threads prior to running a project and making the threads available, so the client can be run immediately. This may improve the response to client requests. By setting an appropriate pool size, you can improve the response time when running projects under the deployment environment.

Thread pooling operates as shown in Figure 82. The process is further described in the following series of events:

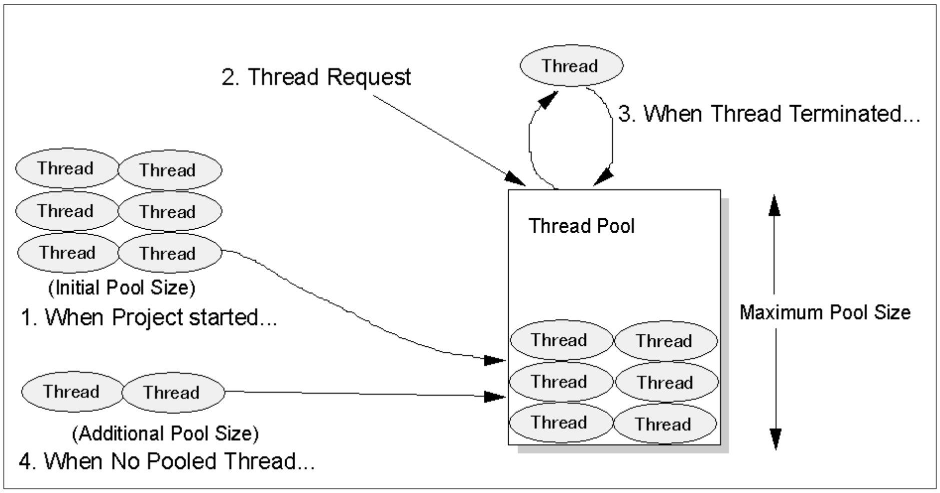
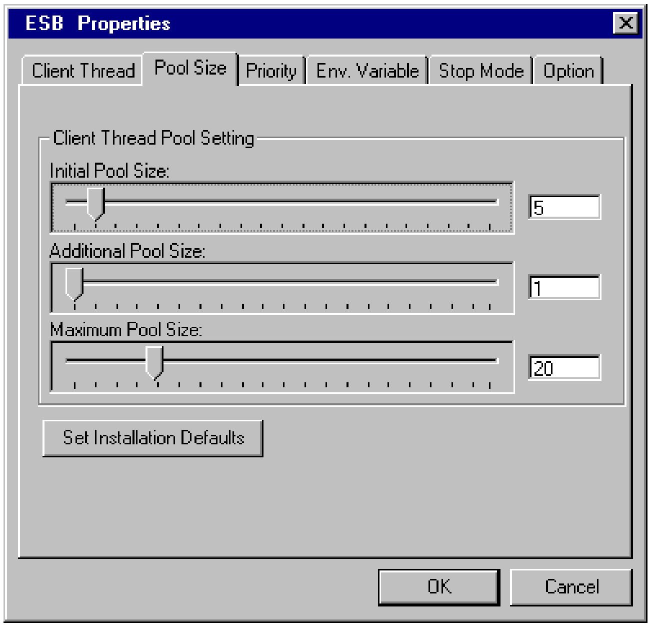


Figure 82. Thread pooling

1. When the project starts, the number of client threads specified in the Initial Pool Size are pooled in the thread pool. For example, if the Initial Pool Size is 6, when pooling starts, six threads are created and available for use.
2. When there is an object creation request from a client, a Thread Request arises to the thread pool and a client thread is obtained from the thread pool.
3. When the thread usage terminates, the thread is returned to the thread pool. However, when the thread pool reaches the Maximum Pool Size, it is discarded.
4. When there is an object creation request from a client while the thread pool is empty, it creates the number of threads specified in Additional Pool Size. When the first thread is created, it is immediately passed to the requester. There is no waiting for all the threads to be pooled. For example, when the Additional Pool Size is 2, the first thread is created and passed to the requester. Thereafter, another thread is created and stored in the thread pool.
5. The thread pool holds the number of threads specified by the Maximum Pool Size. When the Maximum Pool Size is exceeded, it creates a thread for one thread request and then returns it. For example, if the Maximum Pool Size is 20, a maximum of 20 threads are held in the pool. When this is exceeded, a thread is discarded even if it is returned.

10.3.2.2 Setting the pool size

Set the pool size on the Pool Size panel (Figure 83) in Project Properties of the System Manager.



Initial Pool Size: 5

Additional Pool Size: 1

Maximum Pool Size: 20

Figure 83. Pool Size panel in the Project Properties of System Manager

10.3.2.3 Initial Pool Size

“Initial Pool Size” specifies the number of threads automatically created at the time that the project starts running. If there are sufficient resources, you should set it for the projected maximum number of simultaneous client connections (the maximum value of the number of objects created simultaneously). This maximizes performance, because you can then allocate threads that can be run immediately for all the object creation requests from the clients. However, if you set a large value for the Initial Pool Size, it takes time to create the threads when the project starts running.

The pooled threads are loaded into the main memory to run them immediately. Consequently, if you set a large value for the Initial Pool Size, you should monitor that the setting does not impose an undue burden on the memory consumption in the system, while using system tools, and deploying efficiently.

10.3.2.4 Additional Pool Size

“Additional Pool Size” specifies the number of additional threads to be created when an object creation request is received from a client and the entire number of threads specified in the Initial Pool Size is being used.

For example, assume that the Initial Pool Size is set to 5 and the Additional Pool Size is set to 2. An additional object creation request comes from a client while all five threads are currently in use after they are created and passed on to the requester. It creates another one which it puts on standby. In this case, if there is again a subsequent request, it can immediately allocate a thread to the request.

To maintain a certain degree of performance even under the threads in the Initial Pool Size that have been consumed, you can immediately allocate an executable thread for more numerous requests by increasing the number in the Additional Pool Size.

10.3.2.5 Maximum Pool Size

“Maximum Pool Size” specifies the maximum number of threads to be pooled. When threads created in accordance with the Initial Pool Size and the Additional Pool Size have increased, it holds the maximum value that was created simultaneously until it reaches this value.

One of the criteria for determining the Maximum Pool Size value is the capacity of the main memory. In case the number of threads being pooled reaches the value of Maximum Pool Size, the memory area allocated to it is left until the project is exited. Therefore, we recommend that you do not set unnecessary large amounts for this value.

10.3.2.6 Pool size setting example

When the total number of clients connected simultaneously is 50 in normal conditions, you should set the following parameters as indicated here:

- Initial Pool Size … 10
- Additional Pool Size … 5
- Maximum Pool Size … 50

As a result, when the pool size exceeds 10, performance will not decline. Instead, it creates a pool with a total of five threads for one object creation request. In this case, it is presumed that the system has sufficient resources, even if 50 threads are permanently in memory.

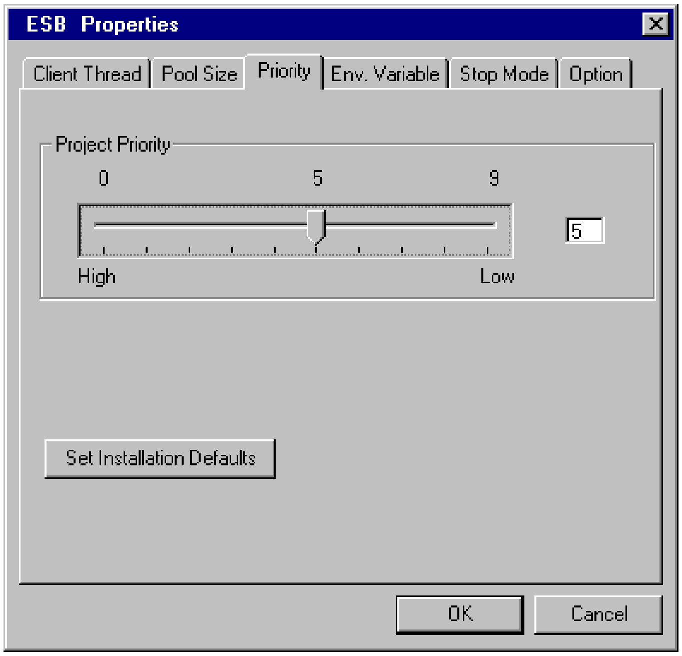
10.3.3 Setting project priority

Projects can be divided into several categories by objective as shown here:

- A project that has an object created by the interactive operation of client user.
- A project that has an object created by a request other than the interactive operation of client user.
- A project performing a periodic task on the server.

A project that has an object created by the interactive operation of a client user must be assigned a higher priority and get a shorter response than the other two cases. Conversely, projects such as periodically performed tasks are given a lower priority. Therefore, other tasks may be given preference over them. You set the project priority for this purpose.

Select the **Priority** panel (Figure 84) from the Project Properties of the System Manager to check and set the project priority.



ESB Properties

Project Priority: 5

Figure 84. Project Properties Priority panel in System Manager

In terms of project priority, “0” is high and “9” is low. The default value is “5”.

> **Hint**
>
> Set the project priority to 0 for projects that have an object relating to a user-interactive operation to be given a preferred response. Specify a priority of 9 for projects relating to background processing, such as periodic tasks.

10.3.4 System environment and project environment variables

When deploying a program whose development was completed in the IDE, it may be necessary to set elements, such as path names, to fit the deployment environment. If a program value is coded in the source program, which depends on the deployment environment, the freedom of deployment may decrease. The difficulty is significant for operating the same package files on a variety of environments. ESB provides system environment variables and project environment variables by projecting to resolve such problems.

Display the Env. Variable panel (Figure 85) from the Project Properties in System Manager to check and set the environment variables.

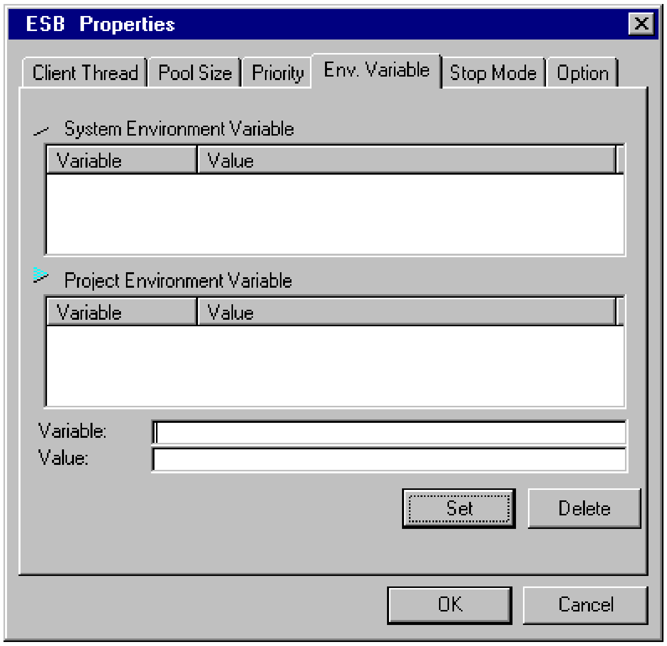


Figure 85. Project Properties Environment Variable panel in System Manager

10.3.4.1 System environment variables

System environment variables are set and enabled as operating system environment variables at the beginning of project. They are used in the following types of circumstances:

- Environment variables settings (path name and so on) of a system to be linked, such as DB2
- Environment variables settings required in LSX

10.3.4.2 Project environment variables

Project environment variables are used within a program, and are used when you want to use variables of different natures depending on the deployment environment. As a result, you can change the nature of the variables required in the deployment environment without changing the source code.

They are used in the following types of circumstances:

- Program specific user and group management
- Setting the host name and so on where the program communicates
- Setting the database to be used by the program and the user name to be used for login

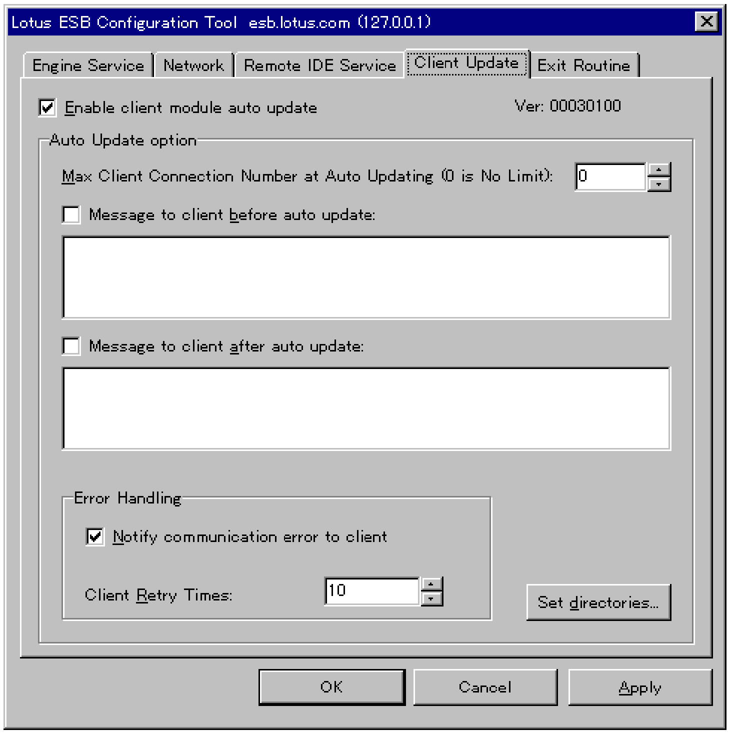
10.3.4.3 Initial setting of the environment variables

Set the initial values for these variables from the IDE. System Manager only permits you to change the values of variables that were defined previously. You cannot add or delete variables. Changed values are held within the project file, so they are effective even after you terminate the project.

10.3.5 Setting the client module automatic updating function

ESB provides a function for automatically updating the previously installed Client Enabler module. This enables the transmitting of the latest module from the server to the client as required. Data files used with the client module Automatic Updating Function are attached to such things as subsequent releases and correction modules for the ESB program and are automatically set when the installation program applies.

Stopping the Automatic Updating Function and setting the parameters of the Automatic Updating Function (for example, the number of retries when an automatic updating by a client has failed) can be changed in the Client Update panel (Figure 86) of the ESB Configuration Tool. This applies to Windows NT and in SMIT in the case of the AIX.



☑ Enable client module auto update

Ver: 00030100

Auto Update option

Max Client Connection Number at Auto Updating (0 is No Limit): 0

□ Message to client before auto update:

□ Message to client after auto update:

Error Handling

☑ Notify communication error to client

Client Retry Times: 10

Figure 86. Project Properties Client Update panel in System Manager

Appendix A. FAQs

This appendix summarizes the frequently asked questions (FAQs) of the LSCube Forum in IBM Japan. LSCube is the former product name of ESB and was marketed in IBM Japan since September 1997 as Version 1 (Windows NT), Version 2.0 (Windows NT), and Version 2.1 (AIX).

A.1 Creating ESB applications

- **Q**: How do I include other LotusScript files into the ESB program?

  A: There are two ways you can do this:

  – Add the LotusScript file to the Include folder on the IDE project browser.

  – Import the LotusScript file by specifying a %INCLUDE statement.

  Enter the %INCLUDE statement in the (Options) script. If no path name is specified in the file to be imported in the %INCLUDE statement, it searches under the current directory of the project or under the directory of ESB Runtime.

- **Q**: Can I include the Visual Basic source code into the ESB program?

  A: It is not formally supported. However, you can include source code files (.BAS) by selecting **Import Script**... on the File menu. Part of the code must be changed due to the differences of the language specification. The form module and the files saved in binary format cannot be read.

- **Q**: Which is compiled first: a file defined in the Include folder of ESB IDE, or a file defined by a %INCLUDE statement of the source file?

  A: LotusScript files described in the Include folder of ESB IDE are compiled first. Within each folder, they are compiled from the top downward in the order defined by the IDE. This order is also the same for LSX and object files.

- **Q**: Can character strings including a NULL character be sent and received between ESB clients and servers?

  A: In ESB, communication between clients and servers of string variables including NULL character (0x00 or Chr(0)) is not assured.

- **Q**: Can a file located in the hard disk of a computer where ESB Runtime is installed be referenced from ESB Runtime? When I use the FileCopy statement, it emits the message: `File cannot be found`.

  A: Include the path name in the file to be specified for the FileCopy statement. If the path name is not specified, the bin directory of ESB Runtime will be used as the current directory. You can use the ChDrive and ChDir statements to change the current directory.

  ```
  Example of changing the current directory:
  ChDrive "C"              ' setting of current drive
  ChDir "\TEMP"            ' setting of current directory
  Filecopy "DBAccess.log", "DBAccess.001"
  ```

- **Q**: I want to record the number of client connections to ESB Runtime at a certain point in time. Is this possible? For example, can I output the number of connections to ESB at five minutes intervals?

  A: ESB does not have such a function. However, it is possible in combination with the LSServer class and the SsTimer class. Specifically, LSServer keeps the number of client connections by the LSServer class. Periodically, SsTimer class outputs that number.

- **Q**: Can I access the LSServer class directly using the SvClink class or SsClink class from a client such as Notes or VB or from another ESB project?

  A: No. It is necessary to go through the Published class. That is, you define the Published class that wraps the LSServer class and access it indirectly through this Published class.

- **Q**: When I observe a sample, both the Variant type variable and the Object type variable are mixed when declaring reference variables for SvClink and SsClink objects. Which is better to use?

  A: The Variant type variable is used in LotusScript. In Microsoft Visual Basic, the Object type variable is used when creating an object using the SvClink class. There is no Object type in LotusScript.

- **Q**: What is the advantage of using the SsClink class compared to SvClink class?

  A: The SsClink class is superior to the SvClink class in the following ways:

  - Arrays of two or more dimensions can be used as the method’s arguments in the Lotus clients.
  - You can obtain more detailed error messages using RuntimeError events.

  Conversely, the SvClink class can be used, if OLE automation is supported even from an environment where LSX cannot be used (for example, Microsoft Visual Basic).

- **Q**: Can I send and receive user-defined type data between ESB client and the server?

  A: You cannot directly send and receive user-defined type data, but you can substitute them into Variant type arrays. Refer to Chapter 4, “Server application programming” on page 43, for details.

- **Q**: When there are variables in common with multiple projects, where and how can I declare them? Can I use the LSServer class?

  A: You cannot share variables between projects even using the LSServer class. Use the SsSharedStorage class to share variables in multiple projects.

- **Q**: Can I use the Published class of another project from a project running on the same ESB Runtime?

  A: Yes. You can create a Published class object using the SsClink class the same way as when creating it from a normal Notes client. However, you do not need to load SsClink LSX (Uselsx "*SsClink") because the SsClink class is loaded automatically.

  Due to the dependency established between those projects, you should remind it in the operation of system. For example, when calling the Published class of another project, the call destination project must be running. If there is a possibility of the call destination being stopped, enter an OnError statement in the program of the call source to set it up so that it displays an error on the console. Or, attempt to recreate it at fixed intervals until it recovers when an error occurs in the creation of a Published class object.

- **Q**: I am considering separating a single ESB project into multiple projects. What are the advantages and disadvantages of doing this?

  A: The advantage is that by separating a project, you can manipulate the operating time over the respective projects. You can also delimit the LSServer class.

  The disadvantage is described here. The memory usage volume increases due to the increased number of small projects. The program must also be changed. For example, because you cannot interchange the variables between projects by the LSServer class, the interchanging of variables using the LSServer class must be replaced by the SsSharedStorage class. Furthermore, because a Public class also cannot be used beyond the project interval, when you use a Public class of another project, you must change that Public class to a Published class.

- **Q**: Please indicate the size and range of the String type and Variant type storage area.

  A: They are defined as shown in the following tables in LotusScript. However, they are also limited by the scope that is defined by the concerned variable and the overall volume of the character string literal.

Table 26. Storage area size and range of data types

Data type	Storage area size	Range
String (variable length)	2 bytes per character	0 to 2GB
String (fixed length)	2 bytes per character + 2 bytes	2 to 64KB
Variant (character string)	2 bytes per character + 16 bytes + 2 bytes	0 to 64KB

Table 27. Maximum value of items

Item	Maximum value
Number of character strings	Depends on memory that can be used
Overall character string storage area	Depends on memory that can be used
Length of the character string literal	16,267 characters (32,000 bytes)
Length of the character string value	2GB

Table 28. Memory size for all data within a specified range

Module	Depends on memory size that can be used
Class	64KB
Procedure	32KB

- **Q**: Each function of the class and LSX used on the ESB must be developed with the condition of *thread-safe*. What does thread-safe mean?

  A: Thread-safe means that no problem is caused under a multi-thread operation.

  ESB Runtime server programs are run on multi-threads to efficiently process access from multiple clients. ESB programs described in LotusScript are automatically thread-safe. However, external modules, such as LSX, to be used from LotusScript must be created with a consciousness of their being thread-safe.

  When using an existing LSX, check the thread-safeness at the Lotus home page or the LSX vendor home page. Test it on ESB. Because multi-thread related problems may occur in the timing of access from multiple clients, the thread-safeness must be verified using multiple clients.

  When creating a new LSX, you generally implement thread safety using a C or C++ language functions. At such times, in addition to normal concerns, such as avoiding the use of static variables and exclusive control on the accessing of shared resources, you must implement thread safety specified within the LSX Toolkit.

- **Q**: Can I call an external module (DLL, or common library) developed using C or C++ from ESB?

  A: Yes. Refer to the DECLARE statement item of the language reference for the calling procedure.

A.1.1 Runtime errors

- **Q**: I am trying to create a Published class object for another project on the same Runtime from a certain ESB Runtime project. I declare Uselsx "*SsClink" for the (Options) script and describe the process normally done by the client. The following error is generated when I run it:

  ```
   Loading errro of the [0002:000000] USE or USELSX module: *SsClink
  (Module:SCRIPT5,Line:4)
  ```

  A: The Uselsx "*SsClink" is unnecessary within an ESB server program. Delete the Uselsx statement.

- **Q**: When I use DCOM from the client to connect to ESB Runtime, I cannot connect due to a 0x800706D3 error. Why did I get this error?

  A: 0x800706D3 is an error returned by the system (Windows), which is generated when the security level of the DCOM does not match. Check the following possibilities:

  - If DCOM was installed on a Windows 95 client, has the file copied by DCOM95 been overwritten by the old DLL file of Windows 95 or Internet Explorer 3.0? If this is the case, you should reinstall Client Enabler.
  - Has an error (LsaRegister - LogonProcess) been shown to the event viewer on the ESB server? If this is the case, you should install the RPC Configuration in the network service on the ESB server side.

- **Q:** The Windows version of Client Enabler reports an error number not included in the manual. Why? Also, what does this error number mean?

  A: The Windows version of Client Enabler sometimes returns error numbers for the Windows operating system. The main error numbers are converted to characteristic ESB error numbers, but all Windows operating system error numbers cannot be converted. Consequently, Windows error numbers are sometimes returned directly. For details about Windows error number and the operation, refer the Microsoft Win32 SDK.

- **Q:** The Windows version of Client Enabler returns a very large number as an error number. I checked the ESB manual and Microsoft's SDK, but cannot find it. What does this number mean?

A: This indicates a Windows operating system error number. Use the following procedure to locate it:

1. Note the error number as a hexadecimal.
2. If the hexadecimal error number begins with “8007”, segregate the following two bytes and convert it to a decimal:

   ```
   0x800706BA => 0x06BA => 1722
   ```

3. Use either of the values obtained in step 1 or step 2 to locate it.

A.1.2 Deploying an ESB application

- **Q:** I am considering using Microsoft Internet Explorer (IE) as an ESB client. What versions can be used as ESB clients?

  A: Use 3.01 or higher when using IE VBScript to create a client program. Use 4.01 or higher when using JavaScript (when using the HTTP communication function).

- **Q**: When I connect to ESB Runtime from VBScript of IE 4.01, the IE Security Warning dialog box is displayed. I do not want do display this dialog box, so what should I do?

  A: Perform the following procedure. Register the ESB Runtime site in the Intranet Zone or the Secure Site Zone of the four IE zones to lower the security setting. When this method is used, it does not lower the security setting for regular Internet access:

  1. Select **Display Internet Menu** in IE.
  2. Click the **Security** tab.
  3. Click **Intranet Zone** or **Secure Site Zone** from the list in the Zone box.
  4. Click **Add Site**.
  5. If you selected **Intranet Zone** in step 3, click the **Details** button.
  6. Enter the Web address (URL) of the ESB application in the **Add this Web site to the zone** box. Click the **Add** button.

- **Q**: Can a Published class object be created when the “Usage Rate” is 100% on the Summary panel of the Runtime Monitor (in the case of Windows NT) or of the Monitor Folder of System Manager (in the case of AIX)?

  A: Yes, it can be created.

When the number of Published class objects equivalent to the number of currently pooled threads is created, the usage rate becomes 100%. However, additional threads (Published class objects) can be created until the maximum number of client threads is reached. The value for the maximum number of client threads can be changed in System Manager. In addition, you can select either to return an error or to have it wait when the maximum value is reached.

- **Q**: Can a client program know why an object creation failed when the maximum number of clients is already created?

  A: Yes. By using System Manager, you can select either to make the client wait or to return an error when the maximum number of client threads has been reached. Therefore, you should first set it here to return an error. Subsequently, a Runtime error is notified and should then be handled using the OnError statement in the client.

- **Q**: For monthly data transfer from the Notes database to ESB Runtime, which method provides better performance: transferring data daily (30 times per month) or transferring data once a month?

  A: It is generally more advantageous to have a lower number of method calls on ESB Runtime, because the ESB Runtime and the Notes client is connected through the network logically. It is obviously clear that transferring month data one time can reduce the network overhead and provide better performance than transferring 30 times for daily data.

- **Q**: Although two network cards are used for ESB Runtime, is it true that ESB validates only one of them? Can it validate both network cards?

  A: It can only validate one of them. To insert and use more than one network card in ESB Runtime, you should specify the IP address that will communicate with ESB Runtime using the ESB Configuration Tool (for Windows) or SMIT (for AIX).

- **Q**: Please advise the port number used by DCOM.

  A: DCOM uses port number 135 and a dynamically assigned port number. The standard number for the dynamically assigned port number is from 1024 to 65535. Consult Microsoft's SDK to learn how to assign dynamically the port number.

- **Q**: Please advise the port number used by IIOP in ESB.

  A: Usually ESB uses port number 3003. Use the ESB Configuration Tool (for Windows) or SMIT (for AIX) to change this number. Other port numbers are used depending on the operating configuration of the application, but these have not been explicitly determined. ESB conforms to the standalone ORB specification of IBM Component Broker.

- **Q**: How much is the upper limit on the transmission packet size for IIOP connections?

  A: There is no upper limit on the data transmission volume. However, from the perspective of transmission speed and line quality, we recommend that you verify it on the actual operating environment and then adjust the transmission unit. You should also remember that the data size limitation on the client of Notes prior R5 has a restriction.

- **Q**: I am trying to develop both a client program and a server program on one computer. Must I install ESB Client Enabler after having installed ESB Runtime?

  A: No. ESB Client Enabler is automatically installed when you install ESB Runtime. There is no need to separately install ESB Client Enabler. When ESB Developer is installed, ESB Client Enabler is also installed automatically.

- **Q**: ESB supports both DCOM and CORBA/IIOP as the communication method between client and server. Which do you recommend?

  A: When DCOM is used, Windows NT's security can be used. However, DCOM is only supported in the Windows NT version. For this reason, we recommend using CORBA/IIOP when you consider supporting the different platforms.

- **Q**: Is it possible to automatically disconnect clients for which there is no fixed time access?

  A: Yes. Use the ESB Configuration Tool (for Windows) or SMIT (for AIX) to set the Idle Timeout. When the timeout value is exceeded, the Published class object corresponding to the client is deleted.

A.1.3 HTTP communication function

- **Q**: Must ESB Client Enabler be installed to use the HTTP support function of ESB to make a Web browser the client?

  A: No.

- **Q**: Is it necessary to install IBM WebSphere and ESB Runtime on the same machine to use the HTTP communication function of ESB?

  A: No.

- **Q**: The HTTP communication function of ESB Runtime is the prerequisite of the work with IBM WebSphere. Can the servlet provided by Domino R5 be used?

  A: No. The HTTP communication function of ESB Runtime uses the supplementary functions (for example, JSP) added by IBM WebSphere. For this reason, the current ESB Runtime does not support the Domino R5 servlet.

- **Q**: Is it necessary to develop a servlet in the Java language when using the HTTP communication function of ESB?

  A: The ESB servlet provided in the HTTP communication function can be used generically. Therefore, no development using Java is necessary.

A.1.4 Access to databases

- **Q**: I tried to access DB2 from ESB using CLI, but the connection failed. What should I do?

  A: Check the following possibilities:

  – Was DB2 started up correctly? Check the DB2 server.

  – Is the DB2 client configuration correct? Are the specified user ID and password correct? Try connecting to the database that you want to use from the DB2 command window. Be careful to use the correct upper and lower case letters in the user ID and password since it discriminates between them.

  – When using the AIX version of ESB Runtime, an initial setting is required prior to running the project. Refer to the description in Readme.En_US, and check that it is set correctly.

- **Q**: I tried accessing an Oracle database from ESB using OCI, but the connection failed. How can I overcome this?

  A: Check the following possibilities:

  – Were the Oracle service and listener started correctly? Check the Oracle server.

  – Do the Oracle server and client versions match? If they are different, check in the Oracle manual whether the client can connect to that server.

  – Is the Oracle client configuration correct? Are the specified service name, user ID and password correct? Try connecting to the database you want to use, using SQLPlus.

  – If you are using the AIX version of ESB Runtime, have the environment variables been set correctly? Check the following on the root user.

    - Has the Oracle installation directory been set to ORACLE_HOME?
    - Has the directory stored by the Oracle library been included in LIB_PATH?
    - Has the language environment been set to NLS_LANG?

- **Q**: Just as I was accessing the database from the Windows NT version of ESB Runtime using ODBC, I received the error message `[Microsoft] and [ODBC driver manager] datasource name and specified default driver cannot be found`, and the connection failed. How can I address this?

  A: The ODBC datasource must be registered in the system DSN because the NT version of ESB Runtime runs as an NT service. Consequently, you should check the following process:

  1. Click **32 Bit ODBC** from the control panel to start the ODC data source administrator.
  2. Click the **System DSN** tab. Check whether the data source you are trying to access is registered.
  3. If it is not registered, register it in the System DSN. You cannot access it, even if it is recorded in User DSN. You must register it again in System DSN.

- **Q**: I cannot connect from ESB to DB2/400 using Client Access ODBC. Why is this not possible?

A: A phenomenon has been reported where an error is occurred by the SQLConnect function when a connection is made through the ODBC using DB2/400 on the AS/400 system. The cause is thought to be attributed to trouble relating to usage of the driver for the DB2/400 from the program being operated from the Windows NT service. Try the following procedure for this:

1. Select the manual option of the ESB Engine Service in the Windows NT service.
2. Start **HPWARB2N.EXE** from the command prompt (or Startup folder).
3. Run the project from System Manager or the IDE as you do normally (see Chapter 9, “Accessing transaction systems” on page 209, for this procedure). It is essential that you log on to Windows NT because ESB operates on the user account.

- **Q**: The SQLGetDataStr function is defined as shown in the following example with include files lsdbcli.lss and lsdbodbc.lss provided by ESB:

```
Declare Function SQLGetDataStr Lib CLILIB Alias "SQLGetData" _
  (ByVal hstmt As Long, ByVal icol As Integer, _
   ByVal fCType As Integer, ByVal rgbValue As String, _
   ByVal cbValueMax As Long, pcbValue As Long) As Integer
```

The Target_Value of the fourth argument is defined in the pass by value, by the appending the keyword ByVal. Why is this?

A: For external C function calls, the character string argument is ordinarily passed by reference. That is, this keyword ByVAL does not imply that it is a pass by value. It implies that you must convert the concerned character string to an ASCII character string (ultimately, one containing 0x0).

A.1.5 ESB and Notes or Domino

- **Q**: Can I install ESB Runtime on a different computer than the Domino Server?

A: ESB Runtime does not rely on the Domino Server. Therefore, you can install it on either a different computer than Domino Server or on the same computer.

- **Q**: Are there cases where I must install the Domino Server and ESB Runtime on the same computer?

A: When you access a Notes database directly from ESB Runtime and when you set it so as to write the log in the Notes database from Runtime Monitor (for Windows NT) or from System Manager (for AIX), you must install Domino Server and ESB Runtime on the same computer.

- **Q**: Are Domino Server and Notes Client essential for a system configured by ESB?

  A: Domino Server and Notes Client are not essential, provided all the following conditions are met:

  – You do not access a Notes database directly using Uselsx "*Notes" from ESB Runtime.

  – You describe the ESB client program in a script language other than LotusScript.

  – You do not output the log to the Notes database from Runtime Monitor (for Windows NT) or from System Manager (for AIX).

- **Q**: I am considering directly accessing a Notes database using Uselsx "*Notes" from the LSSserver class of the ESB Server program, with the objective of exclusive control of Notes documents. What are some considerations when doing this?

  A: Use R4.6 or higher for both the Notes server and the Notes client. Notes classes from R4.6 are fully multi-thread supported.

- **Q**: What is necessary on ESB Runtime when using remote installation for the ESB client: Domino Server or Notes Client? Also, is it necessary that they operate constantly? Is it alright if a module is installed?

  A: To remotely install the ESB client, you just provide the Notes database (HPWCINST.NSF) for the remote installation. ESB Runtime is not particularly relevant. The setting is completed with the copying of the Notes database for remote installation into Domino Server. It is not necessary that ESB Runtime be operating, nor is it necessary to have Notes Client or Domino Server on ESB Runtime.

- **Q**: What should I do to check whether ESB Client Enabler has been introduced onto Notes Client using LotusScript or another method?

A: When you load an ESB client application including *Uselsx SsClink on a machine where ESB Client Enabler has not been installed, an error is occurred. This is how you can check whether ESB Client Enabler has been installed.

- **Q**: Please indicate the differences when using Notes Class on ESB and when using Notes Class on Notes and Domino.

  A: Only Notes Backend Class is supported on ESB. Also, when you use Notes Client, you must enter `USELSX "*notes"` in the (Option) section.

A.1.6 Linking with a mission-critical application using MQ

- **Q**: How do I create a server application to use ESB and MQ to automatically send Notes mail in response to requests from a host application?

  A: You create an LSServer class. Then define the method for receiving MQ messages sent from the host application and sending Notes mail. This method is run asynchronously using the SsTimer, which makes it possible to create the target server application. A detailed example of the usage of MQLSX in ESB is provided in Chapter 9, “Accessing transaction systems” on page 209.

A.1.7 ESB license

- **Q**: What kind of licenses does ESB have?

  A: The ESB product family includes:

  – **ESB Limited Runtime**

  This ESB Runtime has limitations, such as on the maximum number of client threads. It is intended primarily to be used during development and during test deployment prior to introduction of the main system. For more details about restriction, refer to the *ESB Users Guide*. It requires one license per node.

  – **ESB Standard Runtime**

  This is an ESB Runtime used during deployment. It requires one license per node, on which one to four CPUs are installed.

  – **ESB Enterprise Runtime**

  This is an ESB Runtime used during deployment. It requires one license per node, on which five to eight CPUs are installed.

- **ESB Developer**

  This is the ESB server application development tool. It cannot operate alone and requires a machine on which ESB Runtime is operating. One license per user is required.

- **ESB Client Enabler**

  This is the client module required for connection with ESB Runtime. No license is required. It is included with all products.

- **Q**: Is ESB Y2K (Year 2000 ready) compliant?

  A: Yes, it is compliant. Consult the Lotus home page at `http://www.lotus.com` for details.

Appendix B. Special notices

This publication is intended to help programmers to develop an ESB application server program. The information in this publication is not intended as the specification of any programming interfaces that are provided by Lotus Enterprise Solution Builder Release 3.0 for Domino. See the PUBLICATIONS section of the IBM Programming Announcement for Enterprise Solution Builder for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

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

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

CICS
CT
DB2
eNetwork
IBM
IMS
MQ
MQSeries
MVS/ESA
Netfinity
OS/2
RS/6000
SP
SP1
SupportPac
System/390
VisualAge
WebSphere
XT
400

The following terms are trademarks of the Lotus Development Corporation in the United States and/or other countries:

Lotus
Domino
Lotus Notes
Notes
NotesPump
Lotus Enterprise Integrator
LotusScript

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet

Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

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

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

Appendix C. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

C.1 IBM Redbooks publications

For information on ordering these ITSO publications see “How to get IBM Redbooks” on page 275.

- *Lotus Domino R5.0 Enterprise Integration: Architecture and Products*, SG24-5593
- *Lotus Domino Release 5.0: A Developer’s Handbook*, SG24-5331
- *Lotus Solution for Enterprise, Volume 2: Using DB2 in a Domino Environment*, SG24-4918
- *LotusScript for Visual Basic Programmers,* SG24-4856
- *Developing Web Applications Using Lotus Notes Designer for Domino 4.6*, SG24-2183

C.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at `http://www.redbooks.ibm.com/` for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

C.3 Other resources

For additional information, consult the following resources available from the Lotus Development Corporation:

- *ESB R3.0 User's Guide*, which is included in the ESB product package
- *Domino Connector Manual*, which is available with the Domino Connector product

C.4 Referenced Web sites

These Web sites are also relevant as further information sources:

- To learn more about Lotus products and to verify Lotus support issues, visit the Lotus home page at: `http://www.lotus.com`
- To access information and support regarding MA6D: MQSeries for AIX link LotusScript Extension, visit the Web site at: `http://www.software.ibm.com/ts/mqseries/txppacs/ma6d.html`
- To access information and support regarding MA7D: MQSeries for Windows 32-bit platforms link LotusScript Extension, visit the Web site at: `http://www.software.ibm.com/ts/mqseries/txppacs/ma7d.html`
- The latest information on MQSeries components, including MQLSX, is available at the Web site: `http://www.software.ibm.com/ts/mqseries/support/fixes`
- The Lotus Enterprise Integration Web site is at: `http://www.lotus.com/dominoei`
- A host of LotusScript tutorials and related publications can be purchased on the Web at: `http://www.amazon.coms`

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** http://www.redbooks.ibm.com/

  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

□ Invoice to customer number

□ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of abbreviations

CB	component broker
CLI	Call Level Interface
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed COM
DECS	Domino Enterprise Connection Services
ERP	Enterprise Resource Planning
ESB	Enterprise Solution Builder
HTTP	HyperText Transfer Protocol
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
ITSO	International Technical Support Organization
JSP	Java Server Page
JVM	Java Virtual Machine
LEI	Lotus Enterprise Integrator
LSX	Lotus Software eXtention
OCI	Oracle Call Interface
ODBC	Open Database Connectivity
OLE	object linking and embedding
ORB	object request broker
SMIT	System Management Interface Tool
URL	Uniform Resource Locator

Index

IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, **please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.**

- Use the online **Contact us** review redbook form found at http://www.redbooks.ibm.com/
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Document Number **Redbook Title**	SG24-5405-00 Developing e-business Applications Using Lotus Enterprise Solution Builder R3.0
Review	
What other subjects would you like to see IBM Redbooks address?	
Please rate your overall satisfaction:	O Very Good O Good O Average O Poor
Please identify yourself as belonging to one of the following groups:	O Customer O Business Partner O Solution Developer O IBM, Lotus or Tivoli Employee O None of the above
Your email address: The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.	 O Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.
Questions about IBM's privacy policy?	The following link explains how we protect your personal information. http://www.ibm.com/privacy/yourprivacy/

SG24-5405-00
Printed in the U.S.A.

Developing e-business Applications Using Lotus Enterprise Solution Builder R3.0

SG24-5405-00