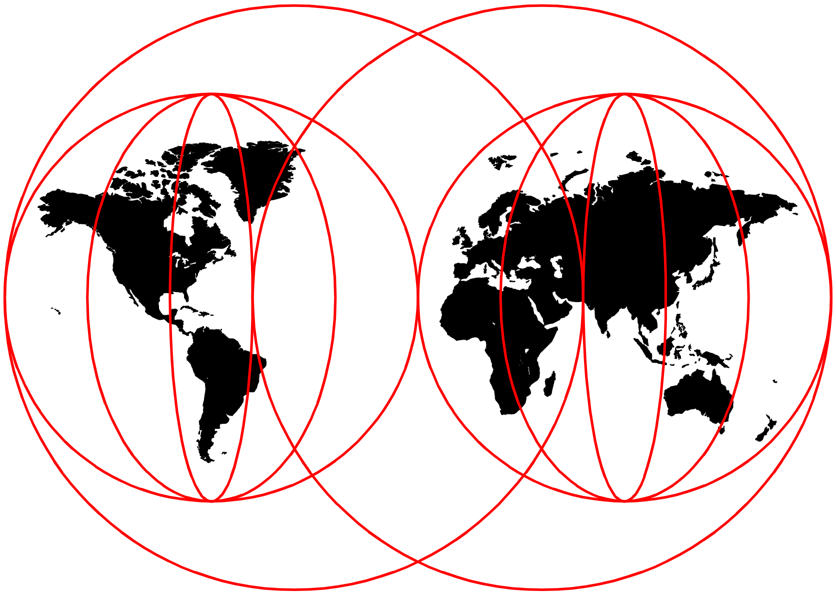
Connecting Domino to the Enterprise Using Java

Christophe Toulemonde, Justine Grose, Ari Pratiwi, Boyd Stratton



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5425-00

SG24-5425-00

International Technical Support Organization

Connecting Domino to the Enterprise Using Java

June 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix F, “Special Notices” on page 247.

First Edition (June 1999)

This edition applies to Release 4.6 and Release 5.0 of Domino

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

Contents

Figures

Tables

Preface

Organizations are making Java, Domino, and enterprise applications the foundations for their e-business strategies. Java is the Internet object-oriented programming language. Domino combines the open networking environment of the Internet standards and protocols with the powerful application facilities of Notes. Enterprise applications have proven their reliability in the processing of business transactions by supporting high-transaction volumes and secure, large, structured databases. Therefore e-business applications are Domino applications connected to enterprise services and developed in Java.

This redbook explains how to use Java to create Domino applications integrated with enterprise resources. It introduces the new CORBA support offered with Domino R5. It shows how to create applets, agents, and servlets that access DB2, CICS, and MQSeries resources. It covers the connection using:

- The IBM connectors such as the DB2 JDBC driver, the CICS Transaction Gateway, and the MQSeries Client for Java
- The Java libraries that support the new Lotus connectors to DB2, CICS, and MQSeries

This redbook also explains the use of Domino with WebSphere application server. It shows how to integrate an Enterprise JavaBean managed by WebSphere in a Domino application.

A full set of working samples illustrates the different types of Java applications and their connection to the enterprise resources.

This redbook will help managers and system architects to understand the Java support of Domino as well as its connection to the enterprise. It will help the developers to create a Domino application in Java that is integrated with enterprise resources.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization San Jose Center.

Christophe Toulemonde is a Senior ITSO Specialist for client/server and network computing at the Application Development and Data Management ITSO, San Jose Center. He writes extensively and teaches IBM classes worldwide on all areas of client/server and network computing. Before joining the ITSO, Christophe worked as a Technical Manager in an IBM subsidiary, Datablue, in France. You can reach him by e-mail at toulemon@us.ibm.com.

Justine Grose is a senior IT Specialist working for the EMEA Application Integration Middleware (AIM) Technical Sales group based in Hursley, England. She has worked in IBM for 10 years, with 4 years of experience in Transaction Systems and Lotus Domino. She holds a degree in Physics from the University of Bath. Her areas of expertise include MQSeries and its integration with Lotus Domino, in particular the MQLSX and MQEI. Justine wrote and teaches course MQ07 "MQSeries Connections to Domino" for IBM Education & Training and has created various other workshops in this area.You can reach Justine by e-mail at justine_grose@uk.ibm.com.

Ari Pratiwi is an e-business Sales Specialist in Indonesia. She has 2.5 years of experience in the Lotus Domino field. She has worked at IBM for 1.5 years in a software department and the last 1 year in an e-business department. She holds a degree in Information Technology from Bandung Institute of Technology, Indonesia. Her areas of expertise include Lotus Domino application programming, Lotus Domino system administrating and object-oriented programming. You can reach Ari by e-mail at apratiwi@id.ibm.com

Boyd Stratton is a Product Specialist at Lotus in the UK. Boyd has been working in the IT industry for 11 years. The last 5 of those years have been for Lotus development where he has been heavily involved with Notes/Domino application design, and Enterprise Integration. You can reach Boyd by e-mail at boyd_stratton@lotus.com.

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

Joaquin Picon
International Technical Support Organization, San Jose Center

John Akerley
International Technical Support Organization, San Jose Center

Martha Hoyt
Lotus Product Manager, Lotus Development

Michele Pennel
Lotus Product Manager, Lotus Development

Mary Peterson
Lotus Product Manager, Lotus Development

Peter Niblett
MQSeries Strategy, IBM Hursley

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 “ITSO Redbook Evaluation” on page 279 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

For Internet users	`http://www.redbooks.ibm.com`
For IBM Intranet users	`http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@us.ibm.com`

Part 1. e-business Components

The Web is changing every aspect of our lives, but no area is undergoing as rapid and significant a change as the way businesses operate. As businesses incorporate Internet technology into their core business processes they start to achieve real business value. Today, companies large and small are using the Web to communicate with their partners, to connect with their backend data systems, and to transact commerce. This is e-business — where the strength and reliability of traditional information technology meet the Internet.

This new Web +IT paradigm merges the standards, simplicity and connectivity of the Internet with the core processes that are the foundation of business. The new *killer apps* are interactive, transaction intensive, and let people do business in more meaningful ways.

In the following chapters, we set the IT scenery needed to create an e-business application:

- The framework, combining proven development and deployment methodologies and *battle-tested* products
- Java, the programming language and execution environment
- The enterprise resources, such as Domino, relational database management systems, online transaction processing systems, and messaging facilities.

Chapter 1. IBM Application Framework for e-business

IBM's Application Framework for e-business helps you achieve e-business critical mass quickly and safely, combining proven development and deployment methodologies and battle-tested products. Building e-business solutions is more effective when you have a blueprint.

The IBM framework is a multi-platform guide to transforming legacy systems into e-business capabilities. There is no need to scrap systems and rebuild because e-business applications are the *glue* between legacy data and the Internet. The framework allows you to expand your business by moving applications to new platforms and expanding or adding rich functionality to applications.

1.1 System Model

The IBM Application Framework for e-business provides a model for designing e-business solutions. The framework is based on an n-tier distributed environment where any number of tiers of application logic and business services are separated into components that communicate with each other across a network. In its most basic form, the framework can be depicted as a *logical* three-tier computing model, meaning that there is a logical, but not necessarily physical, separation of processes. This model is designed to support thin clients with high-function Web application and enterprise servers.

A prototypical three-tier architecture consists of:

- Tier 1:

  A client tier containing logic related to the presentation of information (for example, the graphical user interface) and requests to applications through a browser or Java applet.

- Tier 2:

  Web application servers containing the business logic and processes that control the reading and writing of data.

- Tier 3:

  Servers that provide the data storage and transaction applications used by the Web application server processes.

The application elements residing in these three logical tiers are connected through a set of industry-standard protocols, services, and software connectors.

1.2 Architecture

The application framework for e-business architecture provides a full range of services for developing and deploying e-business applications. Because it is based on industry standards, the framework has the ability to integrate multiple components provided by any vendor.

Figure 1 on page 4 illustrates the key elements of the framework.

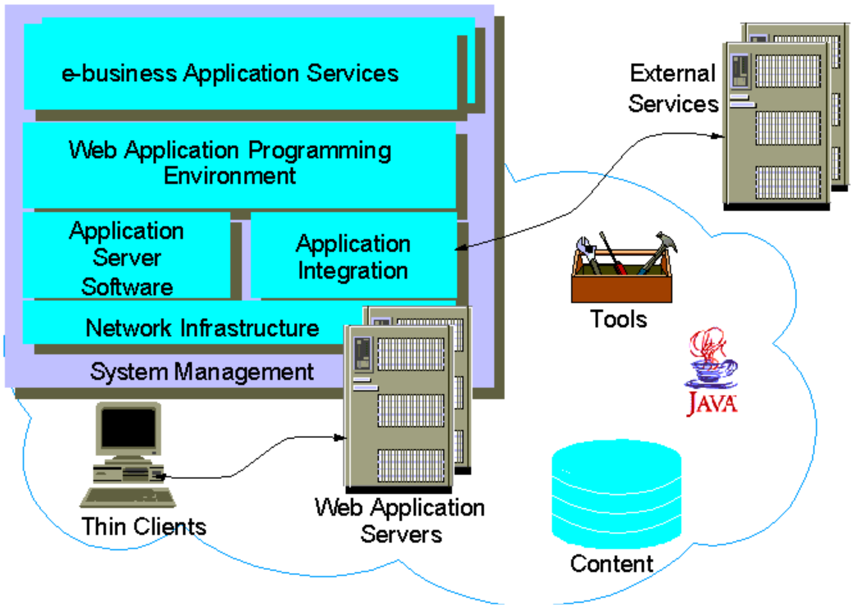


Figure 1. Architecture of IBM Application Framework for e-business

The framework architecture is composed of the following key elements:

- Clients based on a thin client, Web browser/Java applet model that enables universal access to framework applications, and on-demand delivery of application components.
- A network infrastructure that provides services such as TCP/IP, directory, and security whose capabilities can be accessed via open, standard interfaces and protocols.

- Application server software that provides a platform for e-business applications and includes an HTTP server, database and transaction services, mail, and groupware services.
- Application integration that provides access to existing data and applications.
- A Web application programming environment that provides the server-side servlet and Enterprise Java programming environment for creating dynamic and robust e-business applications.
- e-business application services that provide higher level application-specific functionality to facilitate the creation of e-business solutions.
- Systems management functions that accommodate the unique management requirements of network computing across all elements of the system, including users, applications, services, infrastructure, and hardware.
- Development tools to create, assemble, deploy, and manage applications.

1.2.1 Clients

Application framework for e-business clients are *thin clients*, meaning that little or no application logic is executed on the client and therefore relatively little software is required to be installed on the client. In this model, applications are managed on the server and dynamically downloaded *on-demand* to requesting clients. As such, the client portions of new applications should be implemented in HTML, Dynamic HTML, XML, and Java applets. The framework supports a broad range of fixed, mobile, and *Tier 0* clients such as personal data assistants, smartcards, digital cellphones from IBM and other industry leaders, based on industry initiatives such as the Network Computer Profile, and the Mobile Network Computer Reference Specification.

1.2.2 Network Infrastructure

The application framework for e-business network infrastructure provides a platform for the entire framework. It includes the following services, all based on open standards:

- TCP/IP and network services like DHCP, which dynamically assigns IP addresses as devices enter and leave the network.
- Security services based on public key technology that support user identification and authentication, access control, confidentiality, data integrity, and non-repudiation of transactions.
- Directory services that locate users, services, and resources in the network.

- Mobile services that provide access to valuable corporate data to the nomadic computing user.
- Client management services that support the setup, customization, and management of network computers, managed PCs, and in the future Tier 0 devices such as smartcards, digital cellphones, and so forth.
- File and print services that are accessed and managed by way of a standard Web browser interface.

1.2.3 Application Server Software

The application server software provides the core functionality required to develop and support the business logic of e-business applications running on the Web application server. It includes the following services:

- An HTTP server that coordinates, collects, and assembles Web pages composed from static and dynamic content and delivers them to framework clients.
- Mail and community services that provide e-mail messaging, calendaring and group scheduling, chat, and newsgroup discussions.
- Groupware services that provide a rich shared virtual workspace and support the coordination of business workflow processes.
- Database services that integrate the features and functions of an object-relational database with those of the Web application server.
- Transaction services that extend the Web application server by providing a highly available, robust, expandable and secure transaction application execution environment.

1.2.4 Application Integration

The application integration component of the framework allows disparate applications, potentially written in different programming languages and built on different architectures, to communicate with each other. The bulk of today's critical data and application (especially transaction) programs reside on and use existing enterprise systems.

Application integration allows Web clients and servers to work together with this data and these application programs, seamlessly linking the strength of the Internet with the strength of the enterprise. Three methods of integration are supported:

- Connectors are gateway software that provide linkage between the Web server and services that are reached through the use of application-specific protocols.

- Messaging services provide robust, asynchronous communication and message brokering facilities that support a publish/subscribe model of communication including message transformations.
- Managed object services enable object wrappering of existing application logic written in any language. As a result, existing application logic is extended to object-oriented environments.

1.2.5 Web Application Programming Environment

The Web application programming environment, based on Java servlets, Enterprise Java services and Enterprise JavaBean components, provides an environment for writing dynamic, transactional, secure business applications on Web application servers. Services are provided that promote separation of business and presentation logic enabling applications to dynamically adapt and tailor content based on user interests and client devices.

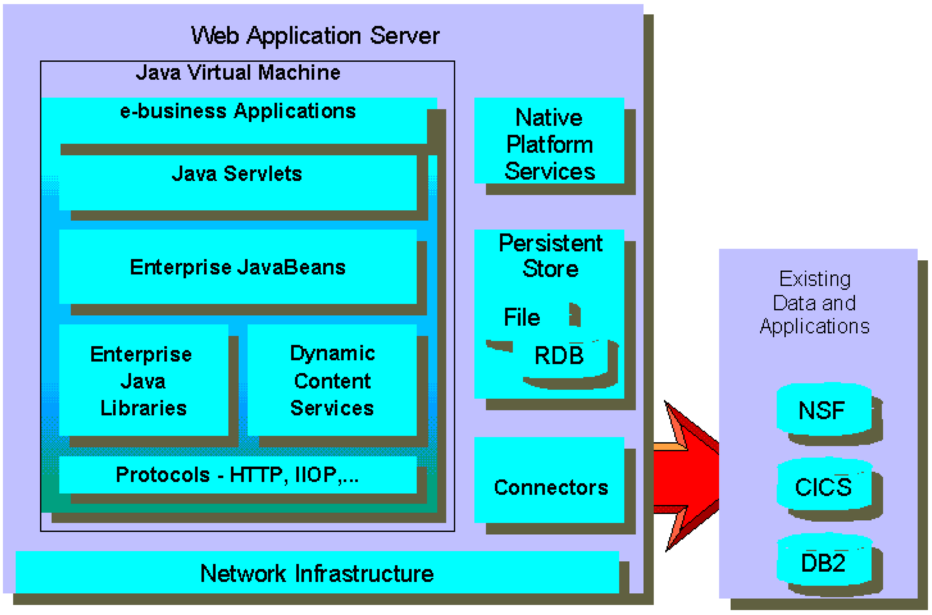


Figure 2. Application Framework for e-business Web Application Programming

1.2.6 e-business Application Services

The e-business application services are building blocks that facilitate the creation of e-business solutions. They are higher level application-oriented components that conform to the framework programming model. They build on and extend the underlying framework infrastructure and foundation services with functions required for specific types of applications, for example, e-commerce applications. As a result, e-business solutions can be developed faster with higher quality. Examples of framework e-business application services include payment services, catalog services, and order management services.

1.2.7 Systems Management

Within an enterprise, systems management services provide the core functionality that supports end-to-end management across networks, systems, middleware and applications. The application framework for e-business provides the tools and services that support management of the complete lifecycle of an application from installation and configuration, to the monitoring of its operational characteristics such as availability and security, to the controlled update of changes. Across multiple enterprises, the framework provides a collaborative management approach for establishing and following procedures to share information and coordinate problem resolution with business partners. This collaborative approach includes policy management, data repository, scheduling and report generation.

1.2.8 Development Tools

The application framework for e-business provides a broad range of tools to enable creation, deployment and management of e-business applications for Internet, extranet and intranet environments. It also supports integrating 3rd party tools into the development process. The framework supports the different skill sets involved in developing Web applications, providing tools that target specific skill sets, and facilitates collaboration among members of the development team.

Chapter 2. Java

Java is an object-oriented programming language and execution environment that offers significant new opportunities for software development, interoperability and portable execution. In this section we introduce the technology, explain the basics and position Java and Java technology.

Java was designed by Sun Microsystems to be small, portable, fast and safe, characteristics that are an essential part of Java's success, as they made Java an ideal language for the explosion in growth of the World-Wide Web and the Internet.

2.1 A Programming Language

Java is a high-level programming language that is simple to use. Java is object oriented, but without all the complications of other object-oriented languages such as C++ or Smalltalk. It has a single inheritance model, simple data types, and code that is organized into classes. These classes provide an excellent way to package functions.

Java has the following characteristics:

- Platform-independent

  You can write Java programs with an editor of your choice. The source code is plain ASCII code. You can transfer this source code to any system that can read ASCII code. Then you can compile this code using the Java compiler for that system. The compiler generates Java bytecode. The bytecode again can be transferred to any Java-enabled operating system to run it, or if it is an applet, to run it within a Java-enabled browser.

  Figure 3 on page 10 shows how Java programs are compiled and distributed. The bytecode generated by any Java compiler runs on any machine that supports Java.

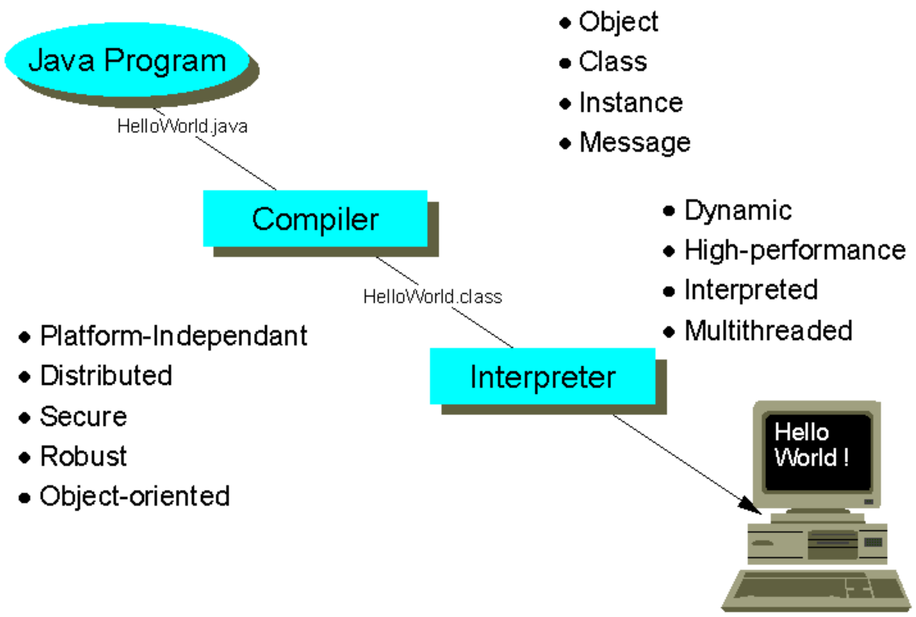


Figure 3. Java Compiler

- Distributed

  Java is inherently distributed. The Java class libraries contain routines for coping with TCP/IP protocols such as FTP and HTTP. Java programs can access URLs as easily as a file system.

  The user can download the Java bytecode of our program from the Internet and run it on his or her own system. That means that anyone with access to your Web server can load and run your applet with no prior installation needed on his or her machine. When an update to the program is required, you simply update the applet on your Web server and the user automatically receives the latest version the next time he or she accesses the applet.

  This significantly reduces the cost of program service and updates.

- Secure

  Java is intended to run in networked/distributed environments, and emphasis has been placed on security. Java programs cannot overrun their run-time stack, cannot corrupt memory outside of their process space, and when downloaded from the Internet, cannot even read or write local files.

- Robust

  Java performs early checking for possible problems, dynamic (run-time) checking, and eliminates situations that are error prone. Java uses a concept of references that eliminates the possibility of overwriting memory and corrupting data.

  The following features are what make it robust:

  - Java eliminates pointer manipulation, so that the memory usage is encapsulated in classes specifically built for that purpose.
  - Java maintains run-time integrity by ensuring that distribution and dynamic linking have not introduced errors into the code (in addition to type checking at compile time). The interpreter ensures that the bytecode has not been tampered with and that transmission errors have not modified the code.
  - Java eliminates the common problems of out-of-bounds array access attempts in C and C++. Java always catches accesses to invalid array elements. Some are caught at compile time and others at runtime, when computing index values.
  - Java supports multithreading by providing synchronization modifiers in the language. At the object level, threaded applications can inherit classes specifically created for that purpose. The priority of specific threads can be set by applications to suit specific needs, allowing unique modes of preemptive multitasking.

- Object-oriented

  Like Smalltalk and C++, Java is an object-oriented programming language. However, it is not a hybrid like C++. Java uses the concepts of classes and objects, instances, interfaces, methods, single inheritance, encapsulation, and polymorphism.

  With Java you have many classes for the main functions you need. So it is easy to start writing your first application or applet. The following packages are included in Java:

  - Language package (*java.lang*)

    This package provides the elementary classes for strings, arrays and elementary data types.

  - Utility package (*java.util)*

This package includes classes for the support of handling vectors, stacks, hash tables, encoding and decoding.

- I/O package (*java.io)*

  This package includes classes for standard input and output, as well as file I/O.

- Applet package (*java.applet)*

  This package provides support to interact with the browser.

- Abstract Window Toolkit (AWT) package (*java.awt*)

  This package was used mainly by us to build the GUI (graphical user interface). It provides support to control the visual aspects of your application or applet. Objects such as buttons, scroll bars, text fields, lists and fonts are available in this class.

- Network package (*java.net*)

  For communication with other applications, this package provides the basic support to communicate with peer programs over the network, as well as standard protocols such as TCP, FTP and URL access.

- JDBC package (*java.sql*)

  This package provides support to access relational DBMS.

2.2 Java Virtual Machine

At the core of the Java concept and implementation is the Java Virtual Machine (JVM). This is a complete software microprocessor with its own instruction set and operation (op) codes. The JVM provides automatic memory management, garbage collection and other functions for the programmer.

The IBM JVM, as are most other JVMs, is implemented by licensed source code from Sun Microsystems. The source code is provided in C and Java languages and is highly portable. It has been already ported by IBM to many platforms: IBM AIX, OS/2, OS/400, OS/390 and Microsoft Windows 3.11.

The JVM is the *essence* of Java. That is, the JVM provides the machine independence which is the most significant advantage of Java. While the Java Virtual Machine is not unique and there have been other software microprocessors over the past 20 years, it is the first and only one to achieve wide-scale acceptance. This is primarily a result of Sun Microsystems making the source code for the Virtual Machine available under license, thus making it much quicker to implement via the source code than implement from scratch working from a reference document.

2.3 Remote Method Invocation (RMI)

RMI is SUN's standard protocol for communication between Java objects residing on different computers. RMI provides a way for client and server applications to invoke methods across a distributed network of clients and servers running the JVM. RMI is supported in JVMs of 1.1 and higher. You can invoke methods on the remote RMI object like you would on a local Java object.

The following paragraph has been taken from this Web site:

`http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-protocol.doc.html`

RMI makes use of two other protocols for its on-the-wire format: Java Object Serialization and HTTP. The Object Serialization protocol is used to marshal call and return data. The HTTP protocol is used to “POST” a remote method invocation and obtain return data when circumstances warrant. It is cross-platform but not cross-language.

Ideally, the Java application server should be a multithreaded application and connectors are used to communicate with the backend systems.

Figure 4 on page 14 shows an example of an RMI client/server application.

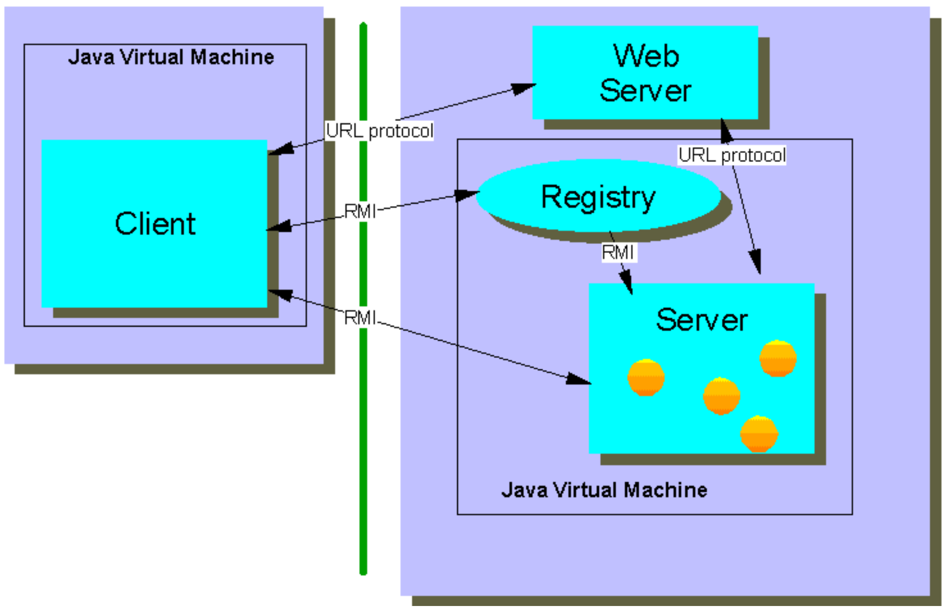


Figure 4. Remote Method Invocation

RMI allows Java objects running on different virtual machines to communicate with one another. Remote objects implement a remote interface that defines which methods a client can invoke. Clients can send any message defined in this interface to the remote object, and the Java RMI handles the routing of the message sent under the covers. To the client, the object appears local. This type of message sending is often referred to as *by-reference*.

In addition to remote references, you can also pass copies of an object from one machine to another. This is known as *by-value*. Passing an object by-value requires that the object be converted to a byte stream before it is passed and then converted from the byte stream back to an object after it is received. In Java this is known as object serialization.

The *java.rmi.Naming* class provides a naming service registry that maps a name to a remote object. Each entry in the registry has a reference to a remote object and a name associated with the reference. Java clients get the references to remote objects by performing a registry lookup, using the object's name.

2.4 Development

In this section we explain the different components required to develop Java programs.

2.4.1 Java Development Kit

If you want to develop Java applets or applications, you need to install the Java Development Kit (JDK) on your machine.

Each release of the JDK contains the following: the Java compiler, JVM, Java class libraries, Java applet viewer, Java debugger, and other tools. JDK Version 2.1 is the latest set of Java technologies made available to licensed developers by Sun Microsystems. However, in this redbook we use JDK 1.1.6 since our tools and drivers only support this version.

JDK 1.1.6 is available on the Sun Web site for Windows platforms, and on the IBM Java Web site for IBM platforms (AIX, OS/2, OS/390 (UNIX Services), OS/400, and VM/ESA).

2.4.2 Java Servlet Development Kit

The Java Servlet Development Kit (JSDK) contains a simple servlet engine for developing and testing servlets, the *javax.servlet* package sources, and API documentation. JSDK Version 2.0 is available on the Sun Web site for Windows platforms.

2.4.3 Java Database Connectivity

Java Database Connectivity (JDBC) is an object interface that allows Java applications, applets, servlets, and agents to retrieve and manipulate data in database management systems using SQL. The interface allows a single application to connect to many different types of databases through a standard protocol. JDBC handles details such as connecting to a database, fetching query results, committing or rolling back transactions, and converting SQL types to and from Java program variables. JDBC is implemented as a driver manager and multiple drivers. Each driver links the application to a specific type of database.

JDBC was first introduced in JDK 1.1. The JDBC classes and interfaces are part of the *java.sql* package. The major components of JDBC are the JDBC driver manager and the underlying drivers. JDBC uses the driver manager to handle finding and connecting to a driver. A JDBC data source consists of the data the user application wants to access and its associated parameters.

Each JDBC driver processes JDBC method invocations, sends SQL statements to a specific data source, and returns results to the application.

JDBC drivers generally fit into one of four types:

- The JDBC-ODBC bridge provides JDBC access via ODBC drivers. NotesSQL (the Domino/Notes ODBC driver) may be used with the JDBC-ODBC bridge.
- A native-API partly-Java driver converts JDBC calls into calls on the client API for the DBMS in question. This style of driver requires that some binary code be loaded on the client machine. Domino Driver for JDBC is a Type 2 driver.
- A net-protocol all-Java driver translates JDBC calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its all-Java clients to many different databases. This is the most flexible Java alternative.
- A native-protocol all-Java driver converts JDBC calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Internet access.

Figure 5 on page 17 shows all the types of JDBC drivers.

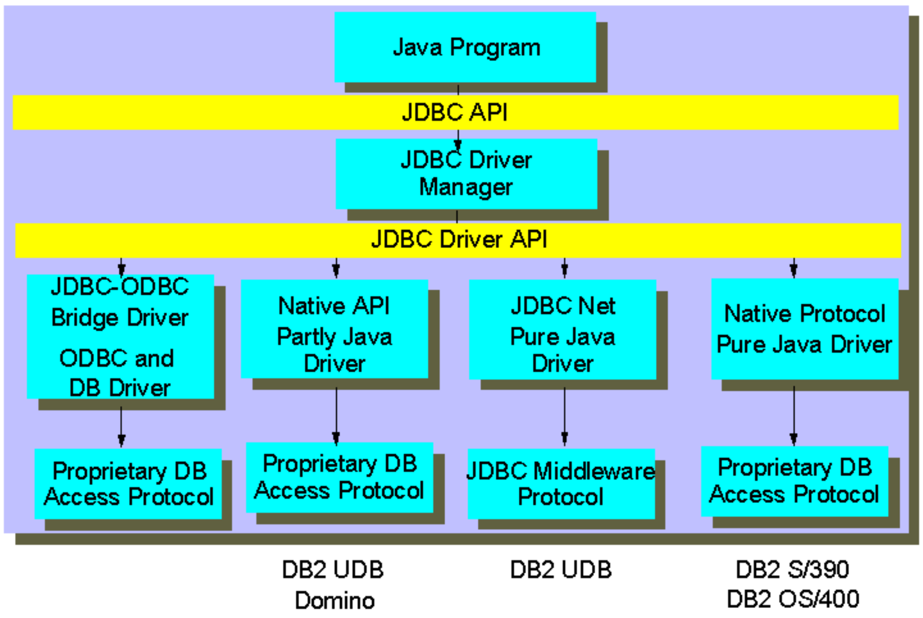

Java Program
JDBC API
JDBC Driver Manager
JDBC Driver API
JDBC-ODBC Bridge Driver
ODBC and DB Driver
Native API Partly Java Driver
JDBC Net Pure Java Driver
Native Protocol Pure Java Driver
Proprietary DB Access Protocol
Proprietary DB Access Protocol
JDBC Middleware Protocol
Proprietary DB Access Protocol
DB2 UDB Domino
DB2 UDB
DB2 S/390 DB2 OS/400

Figure 5. JDBC Driver Types

IBM for DB2 and Lotus for Domino have built JDBC-based products. Other companies, such as Informix, Merant (previously Intersolv), Oracle, Sybase, Symantec, or WebLogic have endorsed the JDBC database access API.

2.5 Application Architecture

In this section we explain the different types of Java programs and components.

2.5.1 Java Applets

A Java applet is a small application program that is downloaded to and executed on a Web browser or network computer. A Java applet typically performs the type of operations that client code would perform in a client/server architecture. It edits input, controls the screen, and communicates transactions to a server, which in turn performs the data or database operations.

Applets are created with dedicated Java development tools such as VisualAge from IBM, Symantec's Visual Cafe, and Microsoft's Visual J++.

These tools often assist the developer by allowing most of the applet to be created visually, and then automatically generating the Java source code.

Figure 6 on page 18 shows a Java applet executing on a Web browser.

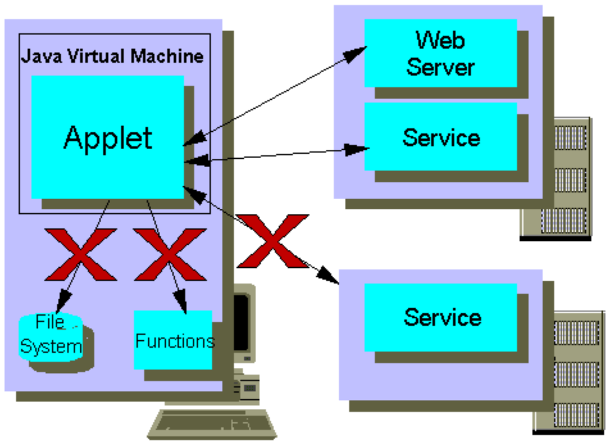


Figure 6. Java Applet

Applets have certain restrictions imposed on them, mainly for security reasons. Applets have no access to the workstation they run on to prevent malicious acts to the operating system by rogue applets. However using security technologies such as Netscape's Object-Signing, and Microsoft's Authenticode, it is possible to grant higher levels of access to applets, and be safe in the knowledge that the applet is from a trusted party.

Another important security restriction to be aware of is that applets can only communicate back across the network with the server from which they originated.

The downloading of applets should not have a significant performance impact on response time because the applets are typically not very large. In fact, applets, by performing processing on the browser or network computer, can improve the overall browser performance by eliminating iterations with the Web server. Note that, just as images are cached in Web browsers, applets are cached, thereby minimizing the frequency of applet downloading. A current performance consideration is the iterative compiling of the Java bytecode at the time of execution. This consideration, however, is rapidly being addressed by the industry and is losing its importance.

2.5.2 Java Applications

A Java application is a program written in Java that executes locally on a computer. It allows programming operations in addition to those used in applets which can make the code platform dependent. It can access local files, create and accept general network connections, and call native C or C++ functions in machine-specific libraries.

2.5.3 Java Servlets

A Java servlet is a protocol and platform-independent server-side software component, written in Java. Servlets run on a Web server machine inside a Java-enabled server, that is, a server that can start the JVM to support the use of Java servlets. They dynamically extend the capabilities of the server because they provide services over the Web, using the request-response paradigm.

From a high-level perspective, the servlet process flow would be:

1. The client sends a request to the server.
2. The server sends the request information to the servlet.
3. The servlet builds a response and passes it to the server. The response is dynamically built, and the contents of the response usually depend on the client's request.
4. The server sends the response back to the client.

Figure 7 on page 19 shows the servlet process flow.

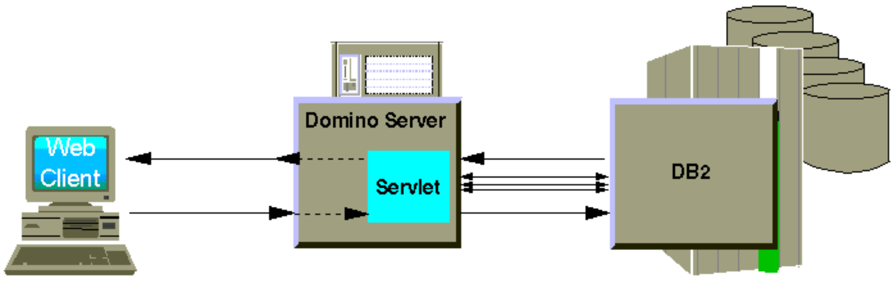


Figure 7. Servlet Process Flow

Servlets look like ordinary Java programs. The servlets import particular Java packages that belong to the Java servlet API. Servlets can be loaded when the Web server starts, and therefore be resident in memory waiting to be called. Because servlets are object bytecodes that can be dynamically loaded off the Web, we could say that servlets are to the server what applets are to the client. However, servlets run inside servers, so they do not need a graphical user interface (GUI). In this sense servlets are also called *faceless objects*.

2.5.4 JavaBeans

According to its inventors at JavaSoft, “A JavaBean is a reusable software component that can be manipulated visually using a builder tool.”

The JavaSoft definition allows for a broad range of components that can be thought of as beans. Beans can be visual components, such as buttons or entry fields, or even an entire spreadsheet application. Beans can also be non-visual components, encapsulating business tasks or entities such as processing employee paychecks, a bank account, or even an entire credit rating component. Non-visual beans still have a visual representation, such as an icon and/or name, to allow visual manipulation. While this visual representation may not appear to the user of an application, non-visual beans are depicted on-screen so developers can work with them.

Beans can only be manipulated and reused if they are built in a standardized way. To build beans, JavaSoft provides the JavaBeans API, an architecture that defines a software component model for Java. The JavaBeans architecture delivers four key benefits:

- Support for a range of component granularity as beans may come in different shapes and sizes.
- Portability as the API is platform neutral. A bean, especially non-visual components, developed under Windows, for example, should behave the same whether it is run under OS/2 Warp, UNIX, or even OS/390.
- Uniform, high-quality API as, ideally, every platform that supports Java will support the entire JavaBeans API.
- Simplicity as the API is simple, universal and compact, easy to learn and begin to use.

The Beans API defines the distinguishing characteristics of a bean, such as how they look and feel.

2.5.5 Java Server Pages

Java Server Pages (JSP) enable server-side Java and JavaBeans to be integrated into HTML Web pages. JSP is analogous to how Active Server Pages (ASP) enable the integration of server-side ActiveX into HTML Web pages. Without JSP, using servlets to run server-side Java or JavaBeans and

HTML can be difficult to write and maintain, because both are mixed together inside the servlet. JSP makes it possible to keep server-side Java separate from the voluminous HTML used for the Web browser GUI. Here's how WebSphere does it:

- JSP defines HTML tags that enable HTML pages to call server-side JavaBeans.
- JSP defines a Java Servlet API that enables a servlet to call JSP HTML pages.

For instance, when a servlet receives a request (for example, a button click) from a Web browser client, it calls one or more JavaBeans. Once the response is formulated using a variety of database, transaction, and/or business logic JavaBeans:

- Servlets can pass the response to Web browsers by invoking a JSP HTML page
  - Without JSP, servlets need to format an HTML Web page dynamically inside the servlet.
  - Hand coded and/or generated HTML inside servlets is hard to maintain or enhance over time.
  - Competitive Web application servers generate non-JSP HTML and combine it into large servlets.
- Accessing JavaBeans directly from JSP HTML pages is as easy as calling a Java method or property
  - JSP allows the Java developer to focus on writing the data, transactional, or business logic Java.
  - JSP allows the GUI developer to focus on writing or generating HTML.

2.5.6 Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a specification of the server-side component architecture for the Java platform. It defines the EJB component architecture and the interface between the EJB server (see 8.2.2, “Enterprise JavaBeans Server” on page 84 for more information) and the EJB components.

The EJB architecture has the following characteristics:

- Transactional

  EJB transactions use a subset of the Java Transaction Service (JTS) API for programmatically starting and stopping transactions. JTS supports distributed transactions that can span multiple databases on multiple systems.

- Portable

  EJB components may run on any EJB server on any operating system.

- Multi-tier

  You can partition and deploy your application onto three or more interacting tiers, the client providing the presentation logic, the application server providing the business logic, and the data server providing the business data.

- Distributed

  EJBs are intended to live on one machine and be invoked remotely from another machine.

- Scalable

  The following is a list of just a few product families which support or will support a container for EJBs:

  - TP monitors such as BEA Tuxedo or IBM TX Series
  - Component Transaction servers such as IBM Component Broker Connectors (CB-Connector) or Microsoft Transaction Server
  - CORBA platform, such as Inprise VisiBroker/ITS, or IBM Component Broker Connector
  - Web platform such as IBM WebSphere Application Server
  - Database management systems, such as IBM DB2, Oracle, or Sybase

- Secure

  EJB architecture is built on the standard Java security services supported in the JDK 1.1.x. Java security supports authentication and authorization services to restrict access to secure objects and methods.

- Protocol neutral

  EJBs are based on industry-standard protocols such as TCP/IP, IIOP, JRMP, HTTP, and even DCOM.

Figure 8 on page 23 illustrates the core elements defined by the EJB specification.

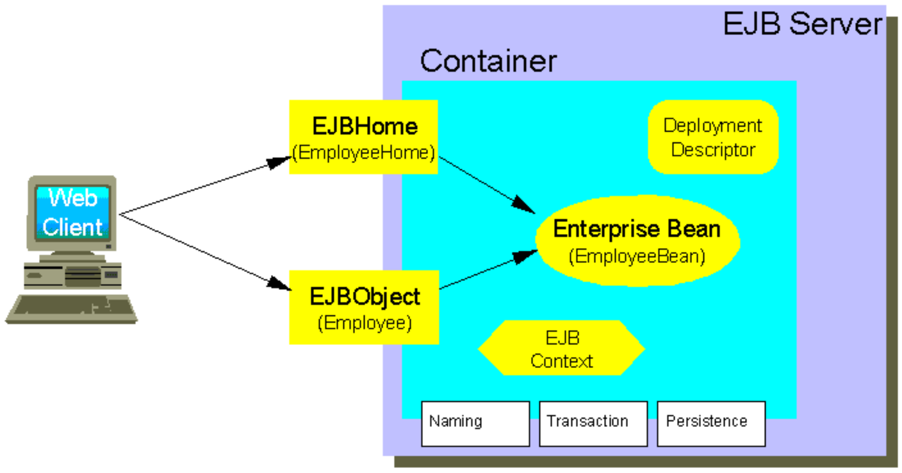


Figure 8. EJB Core Elements

How EJB Works

The EJB server provides an environment that supports the execution of applications developed using Enterprise JavaBeans technology. It manages and coordinates the allocation of resources to the application.

The EJB server must provide one or more EJB containers, which provide homes for enterprise beans. An EJB container manages the enterprise beans contained within it. For each enterprise bean, the container is responsible for registering the object, providing a remote interface for the object, managing the active state for the object, and coordinating distributed transactions. Optionally, the container can also manage all persistent data within object.

Any number of EJB classes can be installed in a single EJB container. A particular class of enterprise bean is assigned to one and only one EJB container, but a container may not necessary represent a physical location. The physical manifestation of an EJB container is not defined in the Enterprise JavaBeans specification. An EJB container could be implemented as physical entity, such as multithread process within an EJB server. It also could be implemented as a logical entity that can be replicated and distributed across any number of systems and processes.

Enterprise beans are deployed in an EJB container within an EJB server (as illustrated in Figure 9 on page 24).

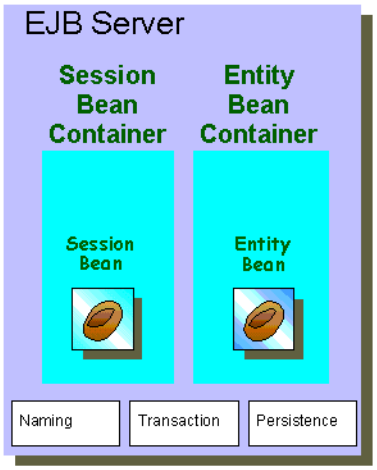


Figure 9. EJB Server and Container

The EJB container acts as a liaison between the client and the enterprise bean. At deployment time, the container automatically generates an EJB Home interface to represent the enterprise bean class and an EJB Object interface for each enterprise bean instance. The EJB Home (as a bean identifier) interface is accessible through the Java naming and directory interface (JNDI), it identifies the enterprise bean class, and is used to create, find, and remove enterprise bean instances. The EJB Object (as a client view) interface provides access to the business methods within the bean based on deployment descriptor settings. All client requests directed at the EJB Home or EJB Object interfaces are intercepted by the EJB container to insert lifecycle, transaction, state, security, and persistence rules on all operations.

EJB Components

The Enterprise JavaBeans specification defines four entities:

- Server
- Container
- Client
- Bean

EJB Server

EJB components—the beans—are server-side components written in Java and contain only the business logic. The system-level services such as transaction management, threading, and persistence are managed for the bean by the EJB server. The EJB server manages the various elements making up an EJB environment. It manages the EJB containers and provides the required support services such as transaction management, persistence, and client access using a JNDI-accessible naming space used by the client to locate the enterprise beans.

EJB Container
The EJB Container is a facility that manages one or more EJB classes and their instances. It is designed to handle EJB lifecycle, transaction, and security management. It makes the required services available to the component.

Client
The client's view of the EJB is provided by two interfaces:

- The Home Interface, which provides methods for creating, destroying and locating enterprise beans in the container.

  A client can use the JNDI to look up the name of a particular EJB class in the name space on the server using JNDI. The Home interface allows the client to create or remove an EJB object.

- The Remote Interface, which defines the business methods offered by an enterprise bean class.

Bean
EJB classes and instances—in short, beans—implement three categories of methods:

- Methods for creating, locating, and accessing instances of the bean, corresponding to those in its home interface
- Methods supporting the business logic, corresponding to those in its remote interface
- Methods for interacting with the container

  (As these are not intended for client access, they are hidden.)

You can develop two types of beans:

- Session bean

  These represent a conversation with a client, and as such, a logical extension of the client program running on the server. Stateless beans don't maintain their state across method calls. They are different than stateful beans, which maintain client-specific session information.

- Entity bean

  The most common use of entity beans is to represent persistent data that is maintained either directly in a database or accessed by way of a backend application as objects. Persistence can be container-managed—the entity bean data is automatically maintained by the container using a mechanism of its choosing. Persistence can also be bean-managed—the bean is entirely responsible for storing and retrieving its instance data.

Entity objects are transactional and they are recoverable following a system crash.

Deployment descriptor

A deployment descriptor provides information that is used by the container when the bean is deployed. A deployment descriptor contains the transaction and security attributes, the EJB environment properties, the names of the EJB class, its Home and Remote interfaces.

EJB-jar file

This file is an EJB's package, and is usually distributed to every machine that needs the EJB's functions. The package can be a collection of enterprise beans, or a complete application system. The EJB-jar file contains information outlining the contents of the file, the enterprise bean class files, a DeploymentDescriptor which tells the EJB container how to manage and control the enterprise bean and, optionally, the environment properties files.

2.5.7 How Servlets and Enterprise JavaBeans Differ

Servlets are the server-side equivalent of applets. They differ from applets in that they have no user interface and do not use either the Abstract Window Toolkit (AWT) or Swing. They are intended as the Java replacements for the Common Gateway Interface (CGI) programs.

Most Web servers support the use of CGI programs to connect to external programs. CGI programs can return both dynamic and static HTML documents. They are typically used to gather and return information from forms and databases. The main advantage of Java servlets is their platform independence and the security that they provide. They should be seen as extensions of your Web server.

Enterprise JavaBeans depend for security on the servlets that are provided with Enterprise Server for Java. Their advantage lies in their persistence, component architecture, and reusability.

EJBs may also be more simple to program than servlets as many functions are managed outside the bean. Functions such as distribution, transaction, security, and persistence are generated during the deployment phase of the bean.

The main point here is that a given application may use a combination of servlets, JSPs, and EJBs.

Chapter 3. CORBA

Common Object Request Broker Architecture (CORBA) is the standard distributed object architecture developed by the Object Management Group consortium (OMG). Since the founding of the OMG, its mission has been to define open standards in software development so that objects written by different vendors in different languages, running on any platform, could interoperate in a distributed environment.

3.1 Implementation

The design of CORBA is based on the OMG Object Model, which defines common object semantics in order to have the same external visible characteristics between objects.

A CORBA implementation has four discrete elements:

- The Object Request Broker (ORB), through which objects intercommunicate.
- Object Services, which define system-level services that are added to the ORB. Examples of these services are naming, security, persistence, and transaction.
- Common Facilities, which define application-level services. Examples of these services are components, compound documents, and other vertical facilities.
- Application Objects, which attempt to capture the real-world behavior of things such as bank accounts, customers, and even airplanes.

Figure 10 on page 29 depicts the relationships among these components.

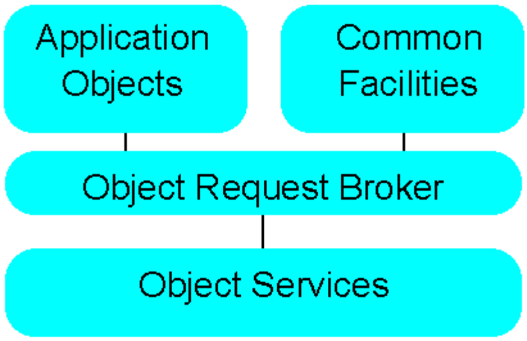


Figure 10. CORBA Components

3.1.1 Object Request Broker

The ORB is the piece of middleware that establishes the client/server relationship between objects. It intercepts any requests that the client makes, is responsible for finding an object that can implement that request, passes the required parameters, invokes the method, and returns the result. The client does not need to be aware of the physical location of the object, its programming language, its operating system, or any other specific information regarding its whereabouts. The ORB shields that information from the client.

Some of the responsibilities of an ORB are:

- Locating and instantiating objects on remote machines
- Marshalling method parameters in a consistent way between any combination of programming languages and operating systems
- Invoking methods on remote objects, using static invocation (compile-time resolution) or dynamic invocation (run-time resolution)
- Routing callback methods from the server to the appropriate client objects

The ORB provides all these services, and more, transparently. All you, as a developer, have to do is provide the appropriate initialization parameters to the ORB, and the rest is handled seamlessly by the ORB implementation.

3.1.2 From the Client to the Server

CORBA is a standard for a distributed application in which computers remotely invoke methods on objects residing on other computers. CORBA allows interconnection of objects and applications regardless of language, location or computer architecture.

We can summarize these concepts as follows:

- CORBA objects can be located anywhere on a network.
- CORBA objects can interoperate with objects on other platforms.
- CORBA objects can be written in any programming language for which there is a mapping from the OMG interface definition language (IDL) to that language. (Mappings currently specified include Java, C++, C, Smalltalk, COBOL and Ada.)

Figure 11 on page 31 shows the architecture of a CORBA-compliant application invoking multiple objects on multiple computers.

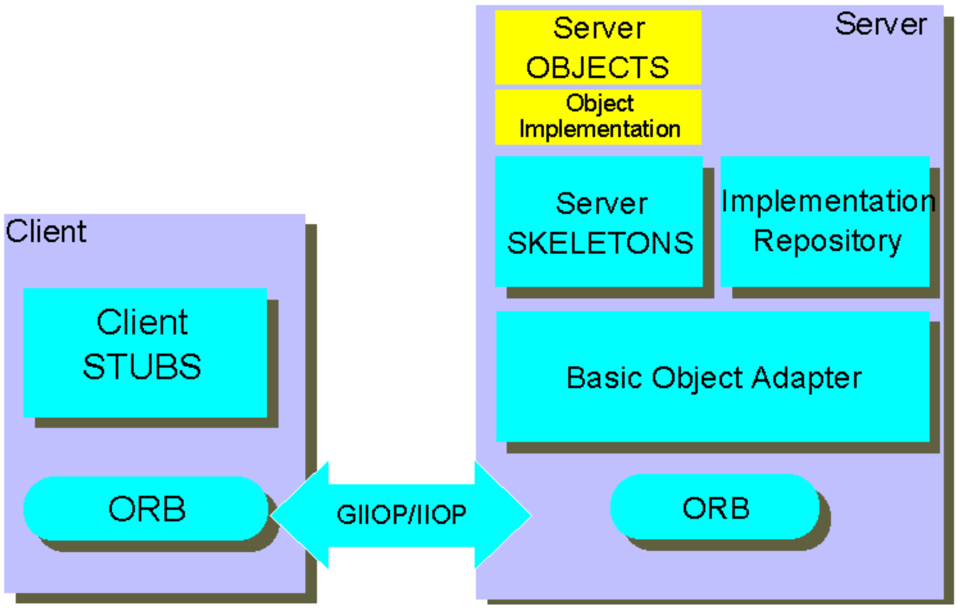


Figure 11. CORBA Architecture

Using CORBA, you can design a distributed system where various components (user interface, business logic, database access, and so on) are packaged in separate programs running on different machines. Each component communicates with the other only through its published interface, and can therefore be maintained separately.

CORBA Client Components

When a CORBA-compliant object wants to use a method located in another object, it calls the stub, that is, the code contained in said stub files. The stub then uses the local ORB, sends the request and receives the response. The object does not need to know anything about the ORB or the actual location of the server. All CORBA clients and servers communicate through ORBs. JDK 1.2 contains an ORB that allows you to write CORBA-compliant Java programs.

From the client's perspective, these local stub classes are the same as the actual implementations. Internally, these stub classes act as a local proxy for the remote server objects and define how clients invoke corresponding remote services on servers. Once a successful request has been made to instantiate a remote server object, a reference ID to that object is returned. Future method requests on that remote object are sent with the reference ID to the remote server and executed on the remote object via the CORBA server components with data being returned if required. This is seamless to the programmer.

The client stubs are created by first defining your server interfaces using the CORBA IDL language. The CORBA IDL defines the type of objects, their attributes, the methods they export, and the method parameters. The CORBA IDL is language neutral and totally declarative; that is, it contains no implementation details. There must be one IDL definition per interface. You can think of an interface as a class definition without the implementation. The IDL file is pre-compiled into the language of the client and the server by a CORBA pre-compiler.

The stub then uses the local ORB, sends the request and receives the response. The object does not need to know anything about the ORB or the actual location of the server. All CORBA clients and servers communicate through ORBs. The ORB is the transportation bus for CORBA object requests to and from remote objects. The bus uses IIOP as the transport protocol between ORBs. (IIOP is described in 3.3, “Internet Inter-ORB Protocol” on page 33.)

Other CORBA components loaded down to the client include Client Side Objects (CSO). These classes feature caching, helper and holder classes, a binary compatibility layer, and the ability to run on other protocols (for example, XML and DCOM). These are transparent to the developer. Helper classes contain methods that manipulate IDL types. A helper Java class is defined for each IDL type and interface. Holder classes provide the parameter passing modes that Java does not.

CORBA Server Component

On retrieving IIOP requests, the server ORB uses the Basic Object Adapter (BOA) in combination with the Implementation Repository to pass parameters and method requests to the required server object via the server stubs. The server stubs communicate with the actual object implementation.

- Basic Object Adapter

  The BOA is a run-time core ORB communication service for instantiating server objects, passing requests, and assigning object IDs (object references).

- Server IDL Stub

  The server stub provides interfaces to each service provided by the server. On the server side, the stub is called a skeleton. It plays the symmetric role of the stub: it gathers the request's parameters and sends back the response.

- Implementation Repository

  This is a run-time repository of object information such as the classes that a server supports, which objects are instantiated, and their IDs.

3.2 CORBA Services

A CORBA system provides many services to control the environment and to access system resources such as networks, databases, security, and so on.

The main services offered by a CORBA system are:

Naming Service	It provides the naming resolution ability, that is, the binding between the object and a name which identifies it.
Event Service	It is responsible for the event notifications (in synchronous or asynchronous mode) between objects.
Life Cycle Service	It defines the conventions for creating, copying, moving and deleting objects locally and remotely.
Persistent Object Service	It defines a set of interfaces to support object storage management, for example database interfacing.
Transaction Service	It provides the specification for running objects in a multiple transaction environment, implements the Unit of Work (UOW) concept, as well as object commit and rollback.
Security Service	It provides a complete framework for distributed object security. It supports authentication, access control lists, confidentiality, and non-repudiation.

3.3 Internet Inter-ORB Protocol

The standard network protocol that CORBA uses for inter-object communication across networks is Internet Inter-ORB Protocol (IIOP). IIOP is actually the TCP/IP implementation of a more general protocol, the General Inter-ORB Protocol (GIOP) that can work on any kind of connection-oriented transport protocol. GIOP specifies a set of message formats and common data representations for communication between ORBs. It is designed to work over any connection-oriented transport protocol.

IIOP specifies how IIOP messages are exchanged over a TCP/IP network. IIOP is basically TCP/IP with some CORBA-defined message exchanges that serve as a common backbone protocol. It defines an architecturally neutral format for representing data, which tries to minimize the processing time devoted to message exchange. IIOP is a more advanced protocol than HTTP or RMI, as it supports *services*, like transaction services and messaging services.

The IIOP architecture is quite simple. A client and a server can exchange messages of seven different types, as follows (note that the originator of the message is given between parentheses as Client, Server or Both):

Request (Client)	Clients send requests to a server to invoke remote objects. Requests contain the desired operation and all the parameters.
Reply (Server)	If a request specifies that it expects a response, the server sends a reply message.
LocateRequest (Client)	Some requests are potentially large because they contain a large amount of data (for example, a high-definition image sent to a storage server). Before sending a large Request message, the client might send a LocateRequest message to check if the server is accepting these requests, or if the object to be invoked is now located on another server.
LocateReply (Server)	The server's answer to a LocateRequest message. It can inform the client that the requested object is indeed on the server, that it has moved to another one, or that it is unknown.
CancelRequest (Client)	A client may want to inform a server that it no longer expects an answer to a given pending Request or LocateRequest, so the server can quit processing that request. This is roughly the equivalent of the Stop button of your browser.
CloseConnection (Server)	The server is closing the connection, and all pending Request and LocateRequest messages are lost.

MessageError (Both)	A reply informing the other party that its message was incorrectly formatted.
Fragment (Both)	This type of message is used to split messages into several parts that are either very large or have sizes that cannot be determined beforehand.

Chapter 4. Domino

Domino is a line of server software that supports your organization's messaging and Web application needs. Domino helps you improve your effectiveness by integrating all communication, collaboration and business application needs. Domino servers are based on a single architecture, so you can choose the one that meets your current needs knowing it has the flexibility and power to grow when you do.

Built on an open, unified architecture, Domino delivers secure communication, collaboration and business applications. Domino R5 Servers set a new standard for rich Internet messaging, ease of administration, integration with backend systems and reliability.

Domino Server Family

The Domino Server Family allows you to quickly and easily start with what you need today — whether that is messaging or applications — and extend your Domino infrastructure investment whenever you are ready. The Domino server family is comprised of three core servers:

- Domino Mail Server

  Domino Mail Server combines full support for the latest Internet mail standards with Domino's industry-leading messaging capabilities, all in one manageable and reliable infrastructure. Its integrated, cross-platform services include Web access, group scheduling, collaborative workspaces, and newsgroups, all of which are accessible from a Web browser or other standards-based client.

  Domino Mail Server is used for messaging only. Customers who want to deploy their own applications on the Domino server should consider Domino Application Server or Domino Enterprise Server.

- Domino Application Server

  Domino Application Server is an open, secure platform optimized to deliver collaborative Web applications that integrate your enterprise systems with rapidly changing business processes.

  Domino Application Server combines integrated messaging and application serving. It delivers best-of-breed messaging as well as an open secure Web application platform. The server easily integrates backend systems with front-end business processes.

- Domino Enterprise Server

Domino Enterprise Server delivers all the functionality of Domino Mail and Application Servers, reinforced with clustering for the high availability and reliability required by mission-critical applications.

Clients for Domino R5.0

Previous versions of Lotus Domino had one, all-purpose client that would be used by users, administrators, and application developers. With Lotus Domino R4.6, a special client for developers called Lotus Notes Designer for Domino was introduced.

As a result of the strong focus on ease-of-use in the design of Lotus Domino R5.0, three individual clients are now available. They are:

- Notes R5: the user's client

  Notes R5 is state-of-the-art e-mail, calendaring, group scheduling, Web access and information management, all integrated in an easy-to-use and customizable environment.

- Domino Administrator R5: the administrator's client

  Domino Administrator is a new, integrated administration control panel that provides simple, yet flexible administration for your Domino environment.

- Domino Designer R5: the developer's client

  Domino Designer R5 is an integrated development environment. It enables developers to rapidly build secure Web applications that incorporate enterprise data and streamline business processes.

Most of the functionality in Lotus Domino can also be accessed from Web browsers.

In the following, we concentrate the description on the Domino Application Server used for our tests.

4.1 Domino Application Server

Domino Application Server R5 allows you to integrate easily your Domino applications and enterprise systems. It leverages current information assets with built-in connection services for live access to relational databases, transaction systems and ERP applications.

It is optimized for collaboration and provides comprehensive application services like workflow and messaging, so you can easily build and manage integrated, collaborative solutions.

You can deploy and maintain the applications with its integrated development tools, standards support and unmatched server-to-server replication. But Domino is also open as you can use your favorite HTML authoring tools, Java IDEs and scripting tools to create Domino applications.

Domino Application Server is an open, secure platform optimized to support rapid delivery of collaborative Web applications that integrate your enterprise systems with dynamic business processes. Domino Enterprise Connection Services (DECS) provides rapid connectivity to enterprise data using a visual mapping interface.

Figure 12 on page 39 shows the different services of a Domino application server available to Web applications.

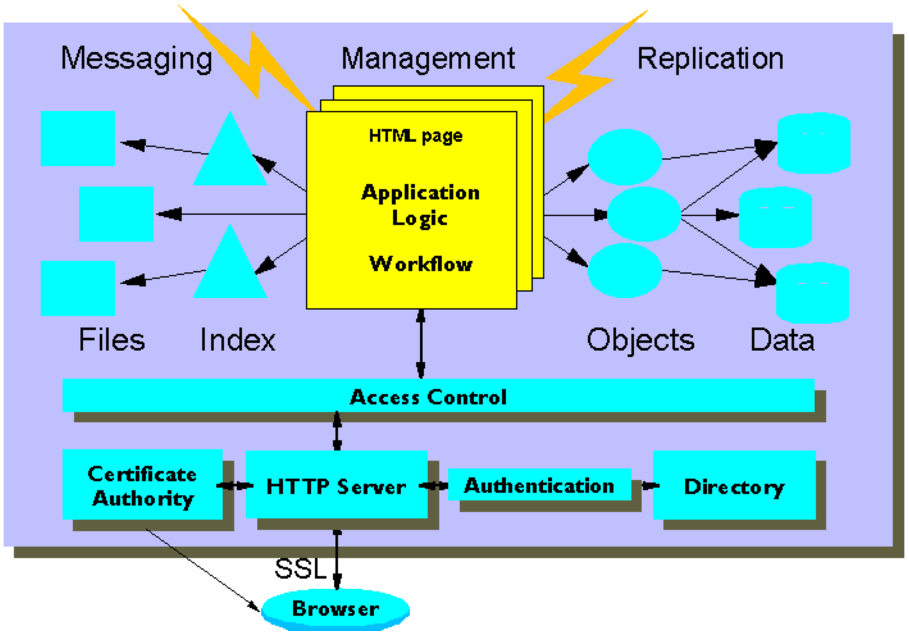


Figure 12. Domino Application Services

CORBA/IIOP support lets you integrate Domino with your application architecture. This support extends Domino application services to Web clients allowing you to serve Lotus Notes clients and Web browsers with the same application.

With its comprehensive development environment, the Domino Application Server lets you move beyond static Web sites to create high-value business solutions that include workflow, content management and highly flexible security. With Domino, you can easily create self-service applications like e-commerce and customer care, and connect them to backend systems.

The flexible security of Domino allows you to personalize access to data and applications based on individual and group roles. It is also extended to HTML files and other data, for pervasive security no matter how or where Web content is stored.

The Domino R5 HTTP engine delivers outstanding performance and Java servlet support.

Domino Application Server offers the following services:

- Object store

  Documents in a Domino database can contain any number of objects and data types, including text, rich text, numerical data, structured data, images, graphics, sound, video, file attachments, embedded objects, and Java and ActiveX applets. The object store also lets your Domino applications dynamically present information based on variables such as user identity, user preferences, user input, and time.

- Search engine

  A built-in full text search engine makes it easy to index and search documents stored in Domino and files in the file system.

- Security

  Integrated X.509 support lets you register new users with Notes and/or X.509 certificates. S/MIME support ensures message integrity for all client types, including SSL V3 for IIOP and LDAP clients. Authentication via trusted third-party directories reduces complexity and duplication of information.

- Directory

  The directory supports a multi-enterprise infrastructure of any size and integrates with other directories via full support for LDAP V3, the open standard for directory access. Its extensible schema allow you to store any information you choose.

- Workflow

  With Domino workflow support, you can define processes to route and track documents, to coordinate activities both within and beyond your organization.

- Messaging

An advanced client/server messaging system with built-in calendaring and scheduling enables individuals and groups to send and share information easily. Message transfer agents (MTAs) seamlessly extend the system to Simple Mail Transfer Protocol (SMTP)/Multipurpose Internet Mail Extension (MIME), X.400, and cc:Mail messaging environments. The Domino messaging service provides a single server supporting a variety of mail clients: Post Office Protocol V3 (POP3), Internet Message Access Protocol V4 (IMAP4), Message Application Programming Interface (MAPI), and Lotus Notes clients.

- Development environment

  Domino Designer is optimized to work with Domino, and features a complete set of visual tools for rapid development and deployment of secure, e-business solutions. It supports your favorite tools for HTML authoring, Java development, and scripting.

- Domino objects

  Domino offers a collection of software objects that expose Domino functionality to several language bindings including Java, JavaScript LotusScript, and, due in 1999, OLE and COM. This allows you to switch programming languages without having to learn new ways to program for Domino.

- Domino Enterprise Connection Services (DECS)

  Domino Application Server includes DECS, for live access to enterprise systems. DECS supports a wide range of enterprise systems, including DB2, Oracle, Sybase, ODBC, EDA/SQL, SAP, PeopleSoft, JD Edwards, Oracle Applications, MQSeries, CICS, and more. Without programming, DECS allows you to create Web applications that access or update enterprise data in real-time, via persistent, parallel, pooled connections.

Domino Application Server delivers reliability and manageability with:

- Transactional logging for Domino databases
- Backup support and APIs to allow tight integration with third-party backup tools on all Domino platforms, including NT, UNIX, AS/400 and S/390
- High availability services such as online indexing and database compaction, fast server restart and more
- Remote server management options
- Centralized control of Notes desktops
- Mail Server capabilities

4.2 Domino and Enterprise Integration

Lotus provides services for connecting Domino R5 to relational database management systems, transaction processing systems, enterprise resource planning applications, and unstructured data. These services present the user with a common interface to the many enterprise systems. As shown in Figure 13 on page 43, the basic services are as follows:

- Domino Enterprise Connection Services: provides a real-time forms-based interface to enterprise data. DECS is an add-in task bundled with the Domino server. DECS is available since Domino R4.6.3.
- Lotus Enterprise Integrator (LEI): provides scheduled and event-driven high-speed data transfer and synchronization capabilities between Domino and enterprise systems. LEI is a separate product.
- Lotus Connector LotusScript Extension (LC LSX): provides LotusScript access to enterprise systems. The LC LSX is bundled with Domino R4.6.3 and above, and LEI 3.0.
- Lotus Connector Java classes provide Java access to enterprise systems using the Lotus Connectors. The LC Java classes are available on the Lotus Enterprise Integration Web site, and are bundled with LEI 3.0.
- Lotus Connector API: provides C/C++ access to enterprise systems. The LC API is available on the Lotus Enterprise Integration Web site.

Underlying the above services are individual connectors for the supported enterprise systems.

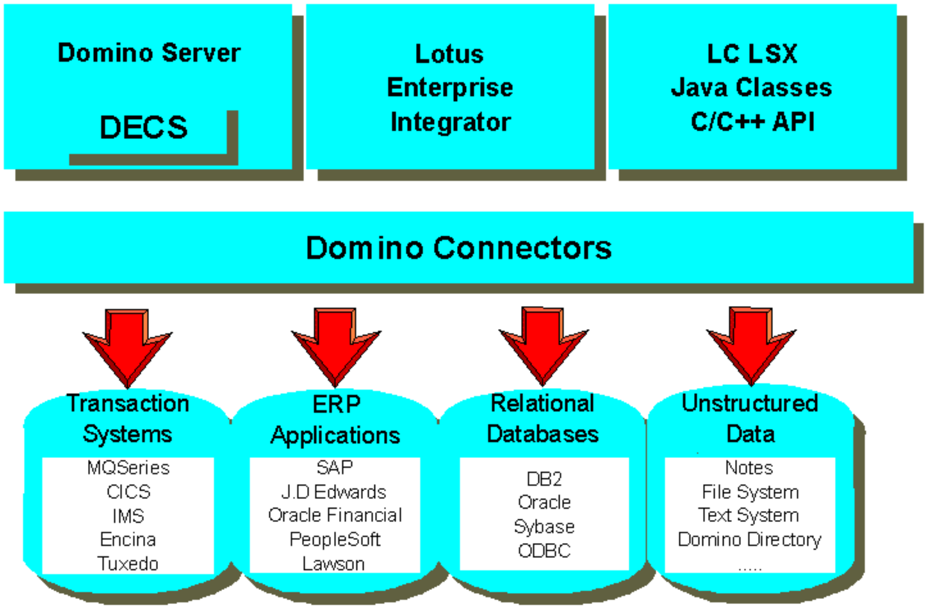


Figure 13. Domino Enterprise Integration Solutions

For current users, LEI is an upgrade of NotesPump and connectors are analogous to links. DECS is functionally similar to the RealTime Notes activity in LEI and NotesPump. DECS and LEI are supported by Domino R4.6.3 and above, as well as R5.

For LotusScript and Java programmers, the LC LSX and LC Java classes provide a common interface to all enterprise systems. You can still use the ODBC (LS:DO), DB2, MQSeries, and SAP R/3 LSXs available with Domino R4 and compatible with R5.

Lotus Connectors

Lotus connectors (LC) exist to permit access to external data sources from Domino for relational database management systems, enterprise resource planning systems, transaction processing systems, directory services, and other services. The connectors are outlined below.

- Relational Database Management Systems (DBMS)

  Listed below are the relational database management systems connectors. These connectors are bundled with DECS and LEI.

  - DB2
  - ODBC

    - Oracle
    - Sybase
    - EDA/SQL

- Enterprise Resource Planning Applications

  Listed below are the enterprise resource planning (ERP) connectors. These connectors are all separate products.

    - J.D. Edwards One World
    - Oracle Financial Applications
    - PeopleSoft
    - SAP R/3
    - Lawson Enterprise/400

- Transaction Processing Systems

  Listed below are the online transaction processing (OLTP) system connectors. These connectors are all separate products.

    - CICS
    - IMS
    - MQSeries
    - Transarc Encina TXSeries
    - BEA Tuxedo

- Directory Services

  Listed below are the directory services connectors. These connectors are bundled with DECS and LEI.

    - Domino Directory
    - Lightweight Directory Access Protocol (LDAP)
    - Novell Directory Services (NDS)

- Other Services

  Listed below are additional connectors. These connectors are bundled with DECS and LEI.

    - File System
    - Notes
    - Text

4.3 DECS

Domino R5 also includes the Domino Enterprise Connection Services (DECS) for building live connections between Domino pages and forms, to data from relational databases. To set up the connections, you simply use the DECS template application to identify your forms and fields that will contain external source data, and to define the real-time connection settings. You can set up native connections for DB2, Oracle, Sybase, EDA/SQL, and more. A Domino server add-in task passes the real-time connection instructions to the Domino Extension Manager, which monitors the server for your user-initiated events. When events are intercepted (such as opening a form), the extension manager transfers the appropriate query to the external source, which performs the query on behalf of the end user. Results are presented to the user in real-time, as if the data were stored natively in Domino. No coding is required when using DECS.

4.4 Lotus Enterprise Integrator

Lotus Enterprise Integrator (LEI) performs data transfer, data synchronization, and other activities between data sources. A data source can be Domino or any LC-supported enterprise system. Data activities occur on a scheduled or event-driven basis and are capable of high-volume, high-speed transfers.

LEI 3.0 can be administered through a Domino R4.6 server or later, and can move data from any Domino or Notes server. LEI 3.0 is an upgrade of NotesPump 2.5a.

4.5 Domino and Java

Domino R5 is a complete Web application server which fully supports the Java environment. Domino applications can be written in Java as you can call into the Domino object interface from a Java program. Domino also supports CORBA to build distributed Domino applications. Finally, Domino supports JDBC calls to allow Java programs access to Domino data.

4.5.1 Java Language

Domino R5 supports many Internet models of programming, so you can choose your favorite language when designing Web applications—whether that language is JavaScript, Java, HTML 4.0, or LotusScript.

Java and JavaScript support are now both available within Designer. The Domino Designer R5 includes a Java editor and Java Virtual Machine (JVM) for developing applications. This allows you to create and compile Java agents, and edit all scripts and formulas, all from within Designer.

With native support for JavaScript and HTML in the Notes client, you can now design applications that run the same on the Web as they do within Notes. In addition, Domino R5 allows you to use third-party design tools, such as NetObjects Fusion, NetObjects ScriptBuilder, and IBM VisualAge for Java.

From a Java program coded as an application, a Domino agent, an applet, or a servlet, you can call into the Domino object interface. In Domino R5, you need to import the *lotus.domino* package. The lotus.domino package has the same content as the R4.6 *lotus.notes* package, and supports the classes, methods, and enhancements of the new release.

The lotus.domino classes allow you to access named databases (*lotus.notes.Database* class), views and folders (*lotus.notes.View* class), and documents (*lotus.notes.Document* class) within a database, and items within a document (*lotus.notes.Item* and *lotus.notes.RichTextItem* classes). The session class (*lotus.notes.Session* class) is the root of the Domino object hierarchy, providing access to the other Domino objects, and represents the Domino environment of your Java program.

4.5.2 CORBA Support

Domino R5 unveils support for CORBA, so you can build robust, distributed applications. The ORB technology and Java allow you to create client applications that are dynamically loaded from the server with transparent access to the server-side Domino objects. While Notes client applications have been able to access Domino objects for quite some time, CORBA and IIOP support in Domino R5 expands this access to Web clients. Your primary access to this ORB is through Java applets or applications. For example, you can place a custom Java applet on a form and have that applet access objects in either the Notes client or a Web browser. For the Notes client, you're actually using the local Java interfaces. For the browser, you're using the CORBA-remote objects; that is, the applet uses IIOP to connect back to the ORB on the server.

CORBA allows you to create client-side objects that *talk* IIOP across the wire to the server-side ORB, which is hard-wired to Domino’s backend classes for better performance. The main purpose is to offload the server by projecting its services to the client. This means, for example, that a browser application can now execute locally with minimal interaction between itself and the Domino server. For instance, you can interact with your server-based mail locally, without involving the server in each user transaction.

With support for CORBA and IIOP, Domino now allows you to create client/server Web applications that take advantage of the Domino objects and application services. In addition, you can now access backend relational databases for enhanced data integration using DECS. In previous releases, when you designed a Web application, the Domino objects classes—formerly known as the remote backend classes, or the Notes Object Interfaces (NOI)—allowed you to access data that was not on display in the browser. These backend classes were LotusScript or Java objects. In R5, these objects are available to the browser using CORBA (see Figure 14 on page 47).

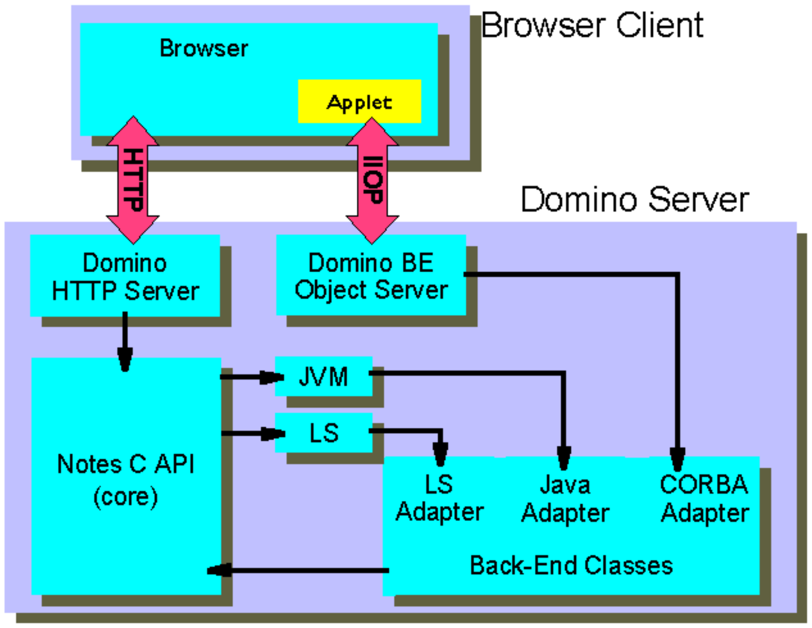


Figure 14. Domino CORBA Architecture

What this means is that now your Web application is similar to your Notes application in terms of programmability. In a Notes application, you could always manipulate data that was on display, as well as data in other databases. In a Web application, you had to wait for a user to open or save a Web page to access this same data. CORBA allows you to access the data without the user opening or saving the Web document; instead, you can use Java or JavaScript.

You can also embed a CORBA applet in a document or a form using the same procedure as for any other applet. You can use a browser to view embedded CORBA applets on a Domino server. It is no longer necessary to set alternate HTML. A CORBA property box setting tells Domino to provide the HTML source that the applet needs to make an IIOP connection back to the server.

4.5.3 CORBA Implementation

Domino R5 uses CORBA-to-Java programs on remote clients such as applets in browsers, and stand-alone Java applications to access the Domino objects on the Domino server. From an implementation standpoint, a remote client instantiates and references Domino objects as if they were resident on the client. In fact, these objects are instantiated at the Domino server. When the client is referencing these objects it is actually communicating with the objects on the server. This is seamless to the programmer (see Figure 15 on page 48).

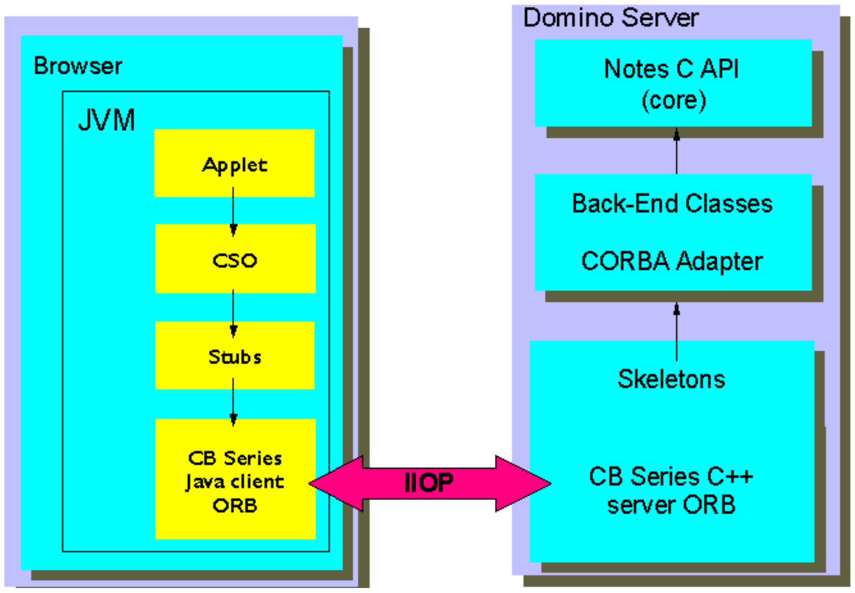


Figure 15. Domino CORBA Implementation

Java Client IDL

There is one IDL definition file per Domino object C++ class that is exposed to CORBA. IDL files are published for developers to create their own stubs. The Java Client IDL stubs are contained in the *lotus.domino* package in the *NCSO.jar* file. This jar file also contains the Java Client ORB classes.

The jar file is automatically loaded down to a browser client if it is designated as a CORBA applet via the Properties box.

Domino R5 Java Client ORB

The Domino R5 Java Client ORB is essentially a complete Server ORB but has been stripped down and compressed, and also provided with enhanced security allowing clients to use SSL to create authenticated sessions with the server.

The Java Client ORB classes are contained in the *NCSO.jar* file. Currently, the ORB is instantiated once per *getSession()* method invocation. Two applets could have more than one instantiation of the ORB per HTML page. Technologies like the InfoBus could fix this situation by sharing the same session object reference.

Domino R5 Server ORB Implementation

The Server ORB is the *broker* which receives requests from objects to access other objects. It functions as a sophisticated router which passes the requested information to the requested object. It also passes information back to the requester, as necessary.

The R5 ORB allows Domino objects to load and respond to client IIOP requests. Its primary use is for doing client-side processing in Domino Web applications. The R5 ORB is a modified version of IBM's ORB. A significant portion of the original IBM ORB has been stripped out, including the interface repository. Improvements have also been made to address issues such as scalability. Other customer-created ORBs may also run on the Domino server.

On Windows NT, this ORB is released as a DLL. The two things Lotus eliminated from the ORB (by hard-wiring) are the Location Service and the Name Service. Other than that it is a standard CORBA object server.

This Server ORB process can be loaded at server startup by placing it in the *notes.ini* file:

```
ServerTasks =<other tasks>,http,diiop
```

The Server ORB listens to IIOP requests on a different port than HTTP. This port can be changed via the server document.

4.5.4 Domino JDBC Driver

The Lotus Domino Driver for JDBC allows Java programmers to use any JDBC standard-enabled application tool to access Lotus Domino databases as easily as any relational database.

The Domino driver for JDBC makes Domino databases *look* like other relational backend sources to the SQL tool or application interface by producing result sets that mirror the relational model. An application can also perform a *SQL Join* of data from Domino with data from a relational database such as Oracle, Sybase, or DB2 (see Figure 16 on page 50).

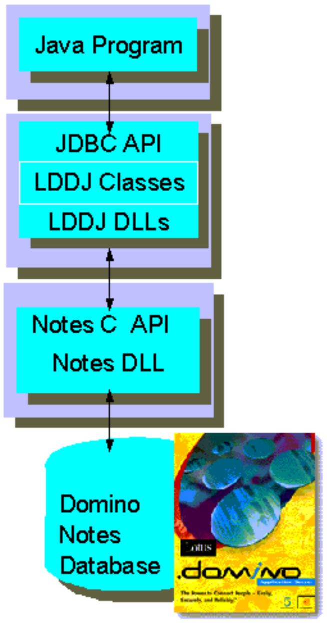


Figure 16. Domino Driver for JDBC

JDBC provides Java programmers with a uniform interface to a wide range of relational databases, and provides a common base on which higher level tools and interfaces can be built. JDBC is now a standard part of Java.

Lotus Domino Driver for JDBC is another step in the Lotus commitment to open standards and accessibility to Domino databases and services so that customers can integrate Domino into any corporate system regardless of their enterprise configuration or tools selection.

Product features include:

- Java standard access to Domino 4.5x and 4.6x databases
- Compatible with Netscape Communicator 4.05
- Compatible with Microsoft Internet Explorer 4.01 with service pack 1
- Includes signed and unsigned versions for maximum flexibility when creating Java applets which use JDBC
- Year 2000 ready, so correctly reads and interprets dates in 2-digit (mm/dd/yy) or 4-digit (mm/dd/yyyy) format

- Tested with IBM's Visual Age for Java, IBM's WebSphere, Lotus eSuite DevPack, Borland's JBuilder, and Symantec's Visual Cafe Web application development tools
- Multithreaded for Web use

Classes

The Domino driver for JDBC implements JDBC interfaces as the following main classes:

- *lotus.jdbc.domino.DominoDriver* class

  Domino implements the java.sql.Driver interface in its DominoDriver class.
- *lotus.jdbc.domino.DominoConnection* class

  A connection represents a session with the Domino driver for JDBC. You need a connection to execute SQL statements and get the results.
- *lotus.jdbc.domino.DominoStatement* and *lotus.jdbc.domino.DominoPreparedStatement* classes

  You use a statement object to execute an SQL statement and obtain a result set.

  A prepared statement is a compiled SQL statement that may handle parameters.
- *lotus.jdbc.domino.DominoResultSet* class

  Using the ResultSet object, you have access to a table of data generated by executing an SQL statement. The table rows are retrieved in sequence, but columns within a row can be accessed in any order.
- *lotus.jdbc.domino.DominoResultSetMetaData* class

  You use this class to find out the types and properties of the columns in a result set.
- *lotus.jdbc.domino.DominoDatabaseMetaData* class

  The DatabaseMetaData class provides information about the database as a whole.

Universal Relation

The Domino driver for JDBC recognizes both Domino forms and views as tables. Table 1 on page 52 shows how SQL components map to Domino components.

Table 1. SQL and Domino Mapping

SQL Component	Domino Component	Comments
Table	Form, View, or Universal Relation	Domino forms and views are recognized as tables. However, a Domino database contains only one real table, referred to as the Universal Relation. This table has the same name as the database.
Column	Form Field or View Column	Avoid the use of column names that are JDBC/ODBC or SQL reserved words or that contain characters other than alphabetics, numerics, dollar sign ($), or underscore.
Index	View	All sorted columns refer directly to fields in a single form.
View	View	Except for private views, all Domino views are reported as SQL views.

In addition to forms and views, the Domino database contains a table that has the same name as the database. This table is called the *Universal Relation*. The Universal Relation contains all fields defined in all forms in the Domino database. The Universal Relation is the only true table in a Domino database. As a result, SQL tables created with the Domino driver for JDBC behave more like SQL views than traditional relational database tables.

For example, with the JDBC driver, you can create a Domino form with the CREATE TABLE statement. However, the DROP TABLE statement deletes the Domino form, but it does not delete any data from the database. Using DROP TABLE with the Domino driver for JDBC is like deleting an SQL view. The data remains in the database. You can view the data through other forms or views that use the same field names, or by referencing the Universal Relation table.

4.5.5 Lotus Connector Java Classes

The Lotus connector Java classes provide programmatic access to enterprise data through a common set of classes. The programmer uses a single model no matter the enterprise source. Programming provides more control and additional capabilities over DECS and LEI.

The classes use the same connectors as DECS and LEI to access the enterprise data. During the residency we used an evaluation version of the Lotus connector Java classes and the final product may change.

Classes

The Lotus Connector classes are imported in the Java program using the following statement:

```
import lotus.lcjava.*;
```

The LC classes become available:

- *lotus.lcjava.LCSession* class

  You must create an LCSession object before using any other LC Java facilities. The LCSession object contains global state information and error information.

- *lotus.lcjava.LCConnection* class

  The LCConnection class represents an instance of a connector to provide access to an enterprise system. Multiple connections can be allocated to a single connector.

- *lotus.lcjava.LCFieldList* class

  The LCField class is the primary class for manipulating data through a connection. It binds fields together with names and an implied order.

- *lotus.lcjava.LCStream* class

  The LCStream class is a general class for text and binary data.

- *lotus.lcjava.LCCurrency* class

  The LCCurrency class represents fixed point data.

- *lotus.lcjava.LCDatetime* class

  The LCDatetime represents date and time data.

- *lotus.lcjava.LCDatetimeParts* class

  The LCDatetimeParts class supports LCDatetime and consists of public variables to access date and time data.

Chapter 5. Database Management Systems

The database management system (DBMS) is a software system that controls the creation, organization, and modification of a database and access to the data stored within it.

Business-critical applications rely on the data managed by a DBMS. Many enterprises are recognizing the value of their data, utilizing it in data warehousing solutions, data mining, and decision support systems.

Relational DBMSs (RDBMS) are distinguished by their structure, which is relational rather than hierarchical.

Relational DBMS applications can:

- Capture, manage and share an organization's structured data.
- Implement immediate access or updates to the data source.
- Control concurrent operations by using locking and isolation levels that ensure database integrity.
- Often require network connections to support functions like locking and journaling by the transaction systems.
- Determine the components used to build network infrastructure. Sharing applications externally may require external users to conform to a pre-determined set of network specifications.
- Have rigorously defined on-storage physical and logical database schema, requiring designers to translate business terms into highly structured RDBMS domains and entities.

An example of an RDBMS is IBM DB2.

5.1 DB2

The DB2 family offers open, industrial-strength database management for business intelligence, transaction processing, and a broad range of applications for all types of businesses. It is the backbone database server in many of the world's largest corporations, handling over 7.7 billion transactions worldwide every day.

DB2 is a relational DBMS that enables you to store, query, update, insert, or delete data in a database from local or remote client applications. It facilitates all database administration tasks such as configuring, backing up and recovering data, managing directories, scheduling jobs, and managing media.

The product family spans operating systems such as AIX, HP-UX, SCO OpenServer, SINIX, Sun Solaris, OS/2, OS/400, OS/390 and Windows (95 and NT).

In DB2 terminology a database is a collection of tables, or a collection of table spaces and index spaces. This is different from the Domino concept of a database, which is a collection of documents and design elements used in the creation and display of the documents.

Typically, the (data access) interfaces to DB2 are:

- Structured Query Language (SQL)
- Call Level Interface (CLI) or ODBC
- REXX
- Query products

5.2 DB2 and Java

In this section, we explain the Java support in DB2.

Java support in DB2 consists of three pieces:

- Client side: Java Database Connectivity (JDBC)
- Server side: Java user-defined functions (UDF) and stored procedures
- Support for the Embedded SQL for Java API (SQLJ)

5.2.1 DB2 JDBC Support

On the client side, DB2 uses JDBC. JDBC is a Java API for client access to a relational database such as DB2. DB2 Universal Database (UDB) Version 5 includes support for the JDBC API, as distributed with Java Development Kit 1.1. DB2 Version 2.1.2 includes support for an earlier version of the JDBC API running under JDK 1.0.2. We refer to these as *client-side* Java support.

DB2 provides two distinct JDBC implementations (as shown in Figure 17 on page 57).

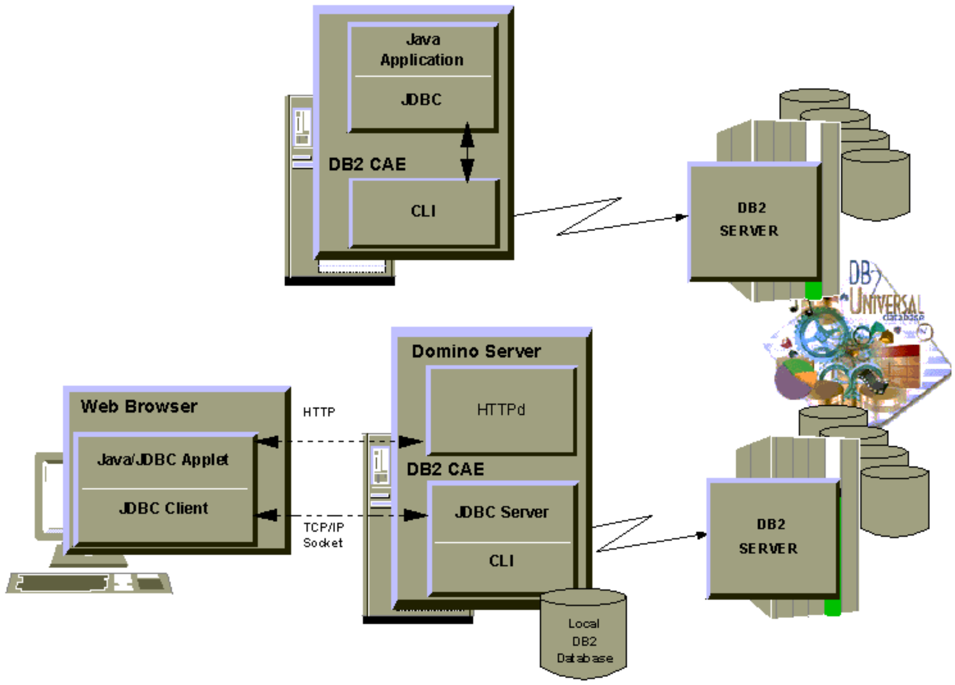


Figure 17. DB2 JDBC Implementation

The DB2 JDBC implementations are:

- Application JDBC driver

  Using this driver, you can build Java applications that rely on the DB2 CAE to connect to DB2.

- Applet JDBC driver

  Using this driver, you can build Java applets that do not require any DB2 component code on the client.

Each uses a distinct JDBC implementation: the application or applet JDBC driver. These JDBC drivers are implemented as wrappers to the DB2 implementation of CLI, the X/Open Call Level Interface; the latter is much like the Microsoft ODBC API. Just like CLI, JDBC is a *dynamic* SQL interface, where SQL statements in transactions are all evaluated on-the-fly. There are no prep or bind steps needed to run a JDBC program.

On the other hand, JDBC presents a convenient object-oriented version of CLI which makes a JDBC program's structure resemble classical embedded SQL programs.

5.2.2 Java User-Defined Functions and Stored Procedures

On the server side, DB2 allows developers to extend the database server with Java user-defined functions and stored procedures. User-defined functions allow the SQL query language to be extended with new scalar and table functions. We refer to this as *server-side* Java support.

Scalar user-defined functions may be used in an SQL expression to compute a complex function of several values in a given row. Table functions (new in DB2 UDB 5) may be used in the FROM clause in a query, so tables may be created on-the-fly from a user program, a row at a time. For example, DB2 extenders generally interface to DB2 as UDFs called in queries. Java UDFs enable developers to create their own extenders in Java.

Stored procedures are parts of a database application that are executed at the database server instead of the client. They are called as subroutines from the client, and may perform SQL transactions or other work. This work does not pay penalties for network delay or poor client performance. Java stored procedures allow Java applications to be factored into pieces, some of which run at the client and some at the server.

5.2.3 SQLJ

DB2 SQLJ allows you to create, build, and run embedded SQL for Java applications, applets, and stored procedures. These contain static SQL and use embedded SQL statements that are bound to a DB2 database. SQLJ also supports calling user-defined functions (UDFs).

SQLJ consists of a set of programming extensions that define the interaction between SQL and Java. It contains a set of clauses that extend Java programs to include static SQL constructs. An SQLJ translator is a utility that transforms those SQLJ clauses into standard Java code that can access the database through a call interface. The output of an SQLJ translator is a generated Java source program that can then be compiled by any Java compiler. Java programs containing embedded SQL can be subjected to static analysis of SQL statements for the purposes of syntax checking, type checking and schema validation.

SQLJ supports only static SQL constructs as it relies upon JDBC for support of dynamic SQL, and does not attempt to replicate the features of JDBC.

5.3 Domino and DB2

Figure 18 on page 59 shows the different ways a Notes or Domino application can access DB2 data.

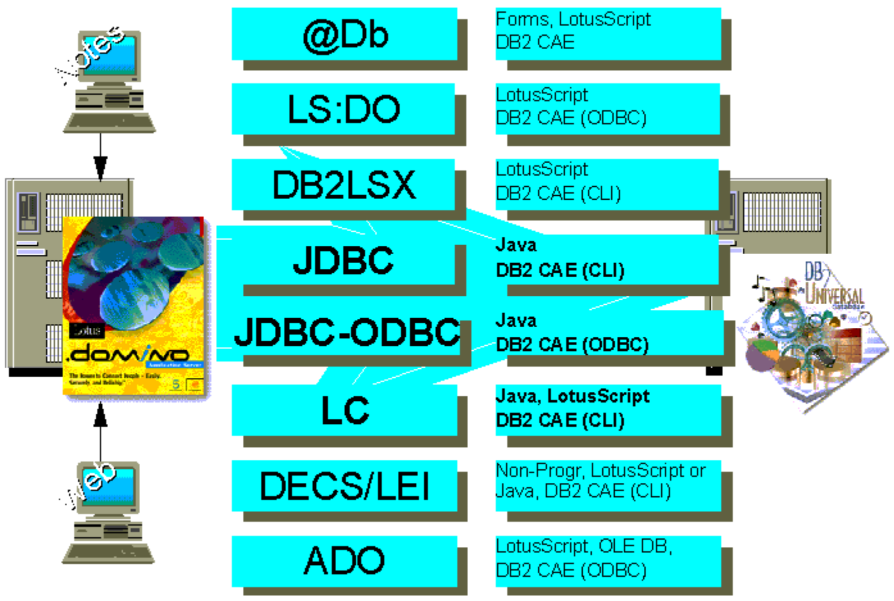


Figure 18. Notes/Domino Access to DB2

A variety of integration techniques and products are available to leverage the data storage and manipulation power of DB2 and the messaging and groupware capabilities of Domino. They fall within the following categories:

- Native programmatic Domino access to DB2 from a LotusScript program
  - @DbFunctions
  - LotusScript Data Option (LS:DO)
  - DB2 LotusScript extension (DB2LSX)
  - Lotus Connector LotusScript extension (LC LSX) and the Lotus Connector for DB2
- Native programmatic Domino access to DB2 from a Java program using
  - DB2 JDBC applet or application drivers
  - Domino JDBC-ODBC driver
  - Lotus Connector Java classes and the Lotus Connector for DB2
- Native non-programmatic Domino access to DB2
  - Domino Enterprise Connection Services (DECS)

    - Lotus Enterprise Integrator Realtime Notes activity
- Server-to-server high-volume data transfer
    - Lotus Enterprise Integrator (formerly called NotesPump)
- Domino access to DB2 data through a transaction system
    - MQSeries and CICS Connections for Domino
- Domino on a Windows platform
    - ActiveX Data Object (ADO)

Chapter 6. Online Transaction Processing

Online Transaction Processing (OLTP) systems provide system-level services such as rollback, backup and recovery facilities, and logging and auditing functions.

Transaction processing applications can rely on the OLTP system to provide these system-level services, together with resource coordination.

6.1 CICS

IBM’s Customer Information Control System (CICS) has been available for 30 years. Found at the heart of large online networks, CICS opens the door to application compatibility with platforms like IBM's AIX, OS/2, OS/400, OS/390 and VSE, as well non-IBM environments such as Windows NT, HP, Digital and Sun. This gives you an easy route to client/server computing, and the software brings your system the ability to develop and implement applications on whichever platforms make the most business sense.

The business applications running in a CICS system consist of a set of CICS transactions, which are often executed by many users at the same time. A CICS transaction consists of one or more CICS programs. CICS provides functions that allow users to concurrently execute those transactions and ensures the consistency and integrity of data that those transactions access.

Distributed program link (DPL) enables a CICS application program to link to a program on a remote CICS system. The linked-to program executes and returns control to the calling program. It can be thought of as a type of remote procedure call (RPC).

The CICS communication area (COMMAREA) is the data area that can be passed to CICS programs when the programs are called by another program. The calling program could be a CICS program using the DPL API or the ECI API. The invoked program receives the data as a parameter. The program must contain a definition of a data area to allow access to the passed data.

The External Call Interface (ECI) is a remote call from a workstation's application to a CICS program on a server. ECI enables a non-CICS client application to call a CICS application synchronously (that is, the calling program waits for a response from the linked-to program) or asynchronously (that is, the two programs continue to execute independently) as a subroutine. The client application communicates with the CICS server program, using the COMMAREA, and the CICS client software. At the CICS server, the ECI looks like a DPL from a partner CICS system.

CICS clients also support the External Presentation Interface (EPI) API. The CICS EPI enables existing CICS applications to send and receive 3270 data streams (for example, a CICS BMS transaction) to and from the client application as though it were conversing with a 3270 terminal. The client application captures this data and processes it as desired.

6.2 CICS and Java

The CICS Transaction Gateway Version 3 incorporates, in a single integrated product, all components required to link Java applets, applications, and servlets, into any CICS server. The contents of the CICS Transaction Gateway include (see Figure 19 on page 62):

- A Java gateway application
- A CICS Universal Client
- A CICS Java class library
- The *TerminalServlet* servlet
- A set of EPI Java beans

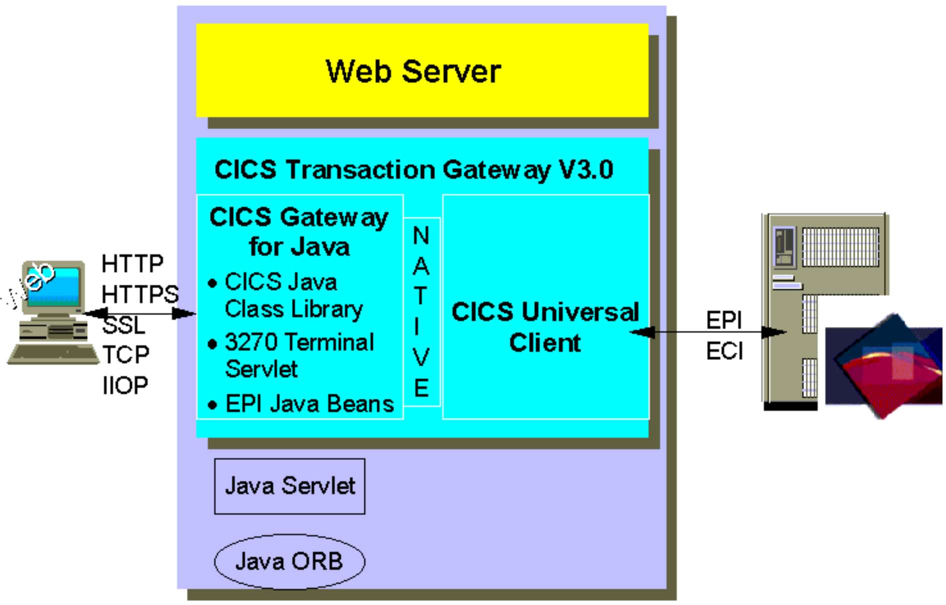


Figure 19. CICS Java Support

6.2.1 CICS Gateway for Java

The CICS Gateway for Java is a Java gateway application that is usually resident (for security reasons) on a Web server workstation. It communicates with CICS applications running in CICS servers through the ECI or EPI interfaces provided by the CICS Universal Clients. This Java application was previously available in the IBM CICS Gateway for Java.

6.2.2 CICS Universal Client

The CICS Universal Client provides the ECI and EPI interfaces, as well as terminal emulation function. The ECI interface enables a non-CICS client application to call a CICS program synchronously or asynchronously as a subroutine. The EPI interface enables a non-CICS client application to act as a logical 3270 terminal and so control a CICS 3270 application. The CICS Universal Clients allow communication with CICS servers over the NetBIOS, TCP/IP, and APPC protocols, depending on the platform.

6.2.3 CICS Java Class Library

The CICS Java class library includes classes that provide an application programming interface (API), and are used to communicate between the Java Gateway application and a Java application (applet or servlet). The class JavaGateway is used to establish communication with the gateway process, and uses Java's sockets protocol. The *ECIRequest* class is used to specify the ECI calls that are sent to the gateway. The *EPIRequest* class is used to specify the EPI calls that are sent to the gateway. These Java classes were previously available in the IBM CICS Gateway for Java product.

(In addition to the former CICS Gateway for Java, the CICS Transaction Gateway also contains the functionality of the CICS Internet Gateway.)

The gateway implementation combines the strengths of Java and CICS's client/server function to provide installations with a unique set of Internet functions for business-critical Internet applications:

- Automatic version control
- Security/integrity
- Protocol support/connectivity
- Performance
- Portability
- Application development

The multi-threaded gateway and CICS clients can communicate asynchronously with CICS servers either on the same processor or across TCP/IP or SNA communication links.

The gateway can be used in various configurations. Multiple gateway processors can be used in parallel against the same CICS servers, or a single gateway processor can connect to multiple CICS server systems, for example as might be contained in an RS/6000 Scalable POWERParallel (SP) processor complex.

The gateway classes are:

- JGateConnection

  Used to establish communication with the long running gateway process using Java's sockets protocol.

- ECIRequest

  Used to specify the CICS ECI call which is sent to the gateway. The gateway channels the ECI call through a CICS client to the desired CICS server applications, manages the many communication links to the connected browser or network computers, and controls asynchronous conversations to the CICS server systems.

- EPIRequest

  Used to specify the CICS EPI call which is sent to the gateway. The gateway channels the EPI call through a CICS client to the desired CICS server applications as for ECIRequest.

- CicsCpRequest

  Queries the code page of the CICS client.

6.2.4 TerminalServlet Servlet

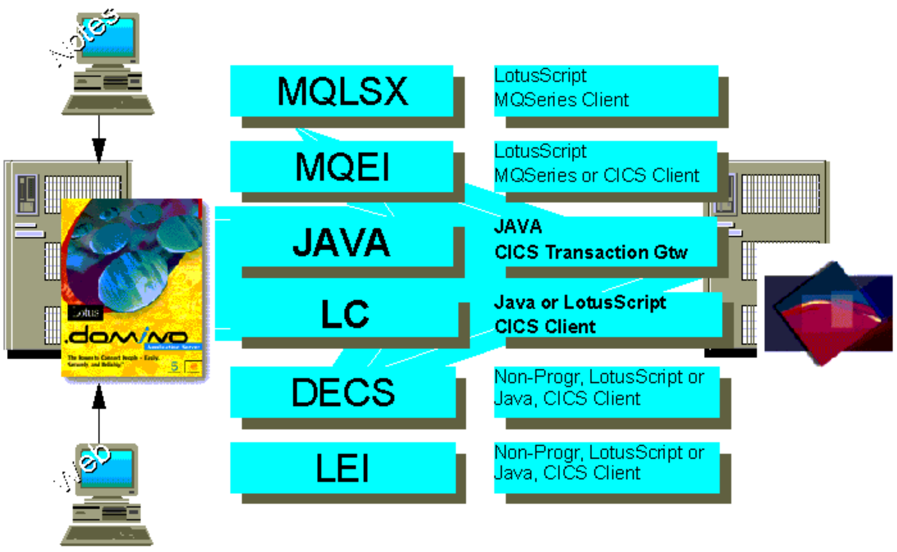
The TerminalServlet allows you to use a Web browser as an emulator for a 3270 CICS application running on a CICS server. The TerminalServlet can be used with a Web server or a servlet engine that provides support equivalent to JSDK Version 1.1 or later.

6.2.5 Set of EPI Java Beans

The set of EPI Java beans allows you to create Java front-ends for existing CICS 3270 applications, without any additional programming.

6.3 Domino and CICS

Figure 20 on page 65 shows the different ways a Notes or Domino application can access CICS transactions.



Notes
Web
Lotus
domino
MQLSX
LotusScript MQSeries Client
MQEI
LotusScript MQSeries or CICS Client
JAVA
JAVA CICS Transaction Gtw
LC
Java or LotusScript CICS Client
DECS
Non-Progr, LotusScript or Java, CICS Client
LEI
Non-Progr, LotusScript or Java, CICS Client

Figure 20. Notes/Domino Access to CICS

A variety of integration techniques and products are available to leverage the data storage and manipulation power of CICS and the messaging and groupware capabilities of Domino. They fall within the following categories:

- Native programmatic Domino access to CICS from a LotusScript program
  - MQSeries Enterprise Integrator (MQEI)
  - MQSeries LotusScript extension (MQLSX)
  - Lotus Connector LotusScript extension (LC LSX) and the Lotus Connector for CICS (not available yet)
- Native programmatic Domino access to CICS from a Java program
  - CICS Transaction Gateway
  - Lotus Connector Java classes and the Lotus Connector for CICS (not available yet)
- Native non-programmatic Domino access to CICS (not available yet)
  - Domino Enterprise Connection Services (DECS)
  - Lotus Enterprise Integrator (formerly called NotesPump)

Chapter 7. Messaging Middleware

A networked company today can do business anytime, anywhere, using advanced technology such as electronic commerce to get ahead of those who still conduct business the old fashioned way. But integration can be a nightmare.

In the past, a business application would probably have been run at a single site, on a single computer with software from a single vendor. Today, networked applications are distributed across different locations. Often customers, suppliers and business partners will be part of integrated business processes, which must support users with a variety of software and hardware which may not be compatible.

This is where messaging middleware comes in. It can helps you integrate your business, giving you a simple way to solve the complex problem of reliable transfer of information between applications in a fast-changing distributed computing environment.

An example of messaging middleware is IBM's award winning MQSeries. MQSeries allows you to take whatever applications, databases or system components you need, and simply integrate them all into a single coherent business information system.

7.1 MQSeries

MQSeries is IBM’s robust messaging middleware product family. It enables businesses to develop applications based on asynchronous messaging principles. An application can put information into a message and pass it to another application through the use of a queue as illustrated in Figure 21 on page 68.

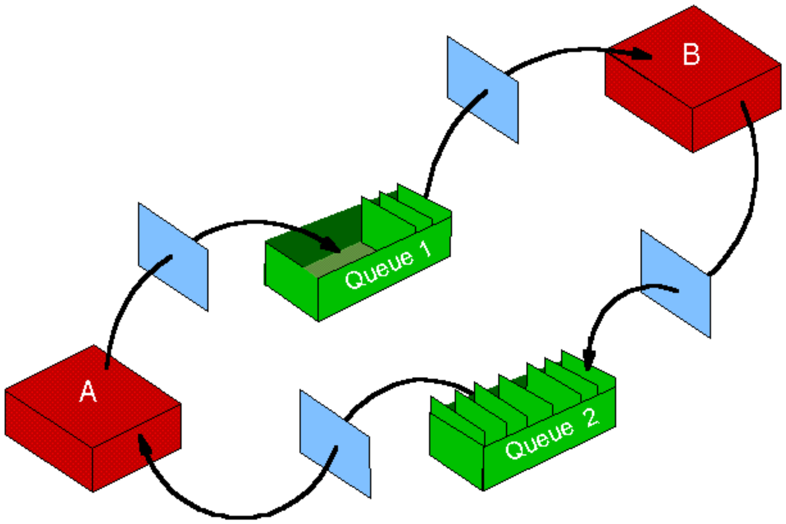


Figure 21. Messaging and Queueing

MQSeries provides:

- A single, multi-platform API
- Assured message delivery
- Faster application development
- Time independent processing
- Application parallelism

7.1.1 A Single, Multi-Platform API

MQSeries has a simple, consistent programming API across a wide range of platforms and networks protocols. Since its initial release in 1993, MQSeries has increased its platform coverage to support over 30 platforms and a range of network protocols. Figure 22 on page 69 shows the platforms for which MQSeries is available. Some platforms only support MQSeries clients, while other platforms offer an MQSeries queue manager and a client, or the possibility to attach a client.

The MQSeries API, the message queue interface (MQI), has a rich set of features, such as triggering, priorities, data conversion and dynamic queues. These features give application designers the flexibility to use MQ in a wide variety of solutions.

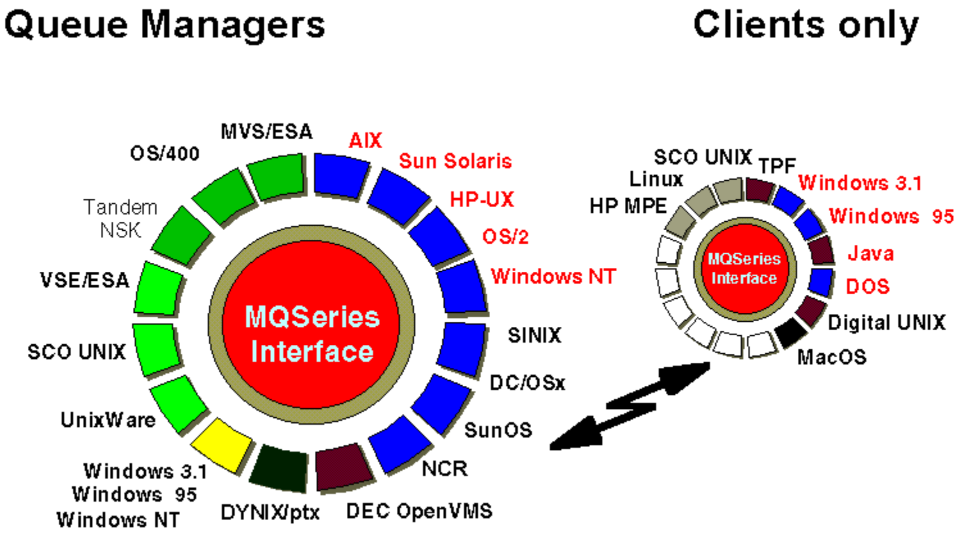


Figure 22. MQSeries Platform Coverage

MQSeries clients only provide access to the MQI and must be *attached* to a queue manager to use the queueing service.

7.1.2 Assured Message Delivery

MQSeries is robust middleware. Middleware is essentially the application enabling layer between your application and the operating system. Importantly, MQSeries assures the delivery of messages between applications. This delivery is assured to be a once and once only delivery.

The basic elements of MQSeries work together to pass messages between applications and assure delivery. These elements are:

- The message queue interface (MQI) that enables application programs to use the facilities provided by MQSeries.
- The queue manager which manages resources, such as queues, and access to them.
- The message channel agent (MCA) which ensures the delivery of messages for queues on other queue managers.

For application A to pass information into application B (as illustrated in Figure 21 on page 68), application A needs to use the MQI to PUT a message. Application A only requires the queue name that it should use; it does not need to know where application B resides. The queue manager will then resolve the queue name. If not a local queue on that queue manager, the queue manager will store the message (on a transmission queue) and the message channel agent (or channel) then becomes responsible for moving the message to the target queue manager, and the intended queue. The initial queue manager retains a copy of the message until the channel has safely delivered it. Receiving application B can GET the message from its local queue as soon as it is ready to process it.

The MQSeries product provides a command interface for MQSeries object definition. Queues, channels and other MQSeries resources must, in general, be defined to the queue manager before they can be used. MQSeries object names are case sensitive and must be unique within one queue manager and object type. Queue manager names must be unique within a network of queue managers.

Each MQSeries message consists of a message descriptor (MQMD) whose contents are architected by MQSeries, and application data whose contents are entirely application-dependent.

7.1.3 Faster Application Development

In an open-systems environment, communications coding is complex and difficult to manage. MQSeries handles any networking complexities and leaves you to concentrate on the business logic of your application. It enables business applications to exchange information across different operating system platforms in a way that is straightforward and easy to implement. Shielding you from the complexities of network communications speeds up application development, and can reduces development costs by up to 40%.

7.1.4 Time Independent Processing

Synchronous application communications, such as Remote Procedure Call (RPC) only work if the target application is available when called. The requesting application must wait for the reply before proceeding.

MQSeries is based on asynchronous processing and is therefore time independent processing. That is to say, while MQSeries can provide synchronous communications between applications, it is not reliant on the network or target application being available immediately. It delivers the message when the network and/or target application becomes available.

Asynchronous processing does not have to mean slow processing. With MQSeries you can still achieve high performance and throughput. Messages can be placed on queues and retrieved from queues instantaneously.

7.1.5 Application Parallelism

MQSeries's use of messages and queues enables parts of a business application to be handled in parallel. For example, task A might generate three inquiries to be processed by independent tasks B, C and D, as shown in Figure 23 on page 71, with the replies being processed by a further task E. Splitting an application into tasks this way and being able to process some tasks in parallel means that your response time may be faster than conventional direct communication.

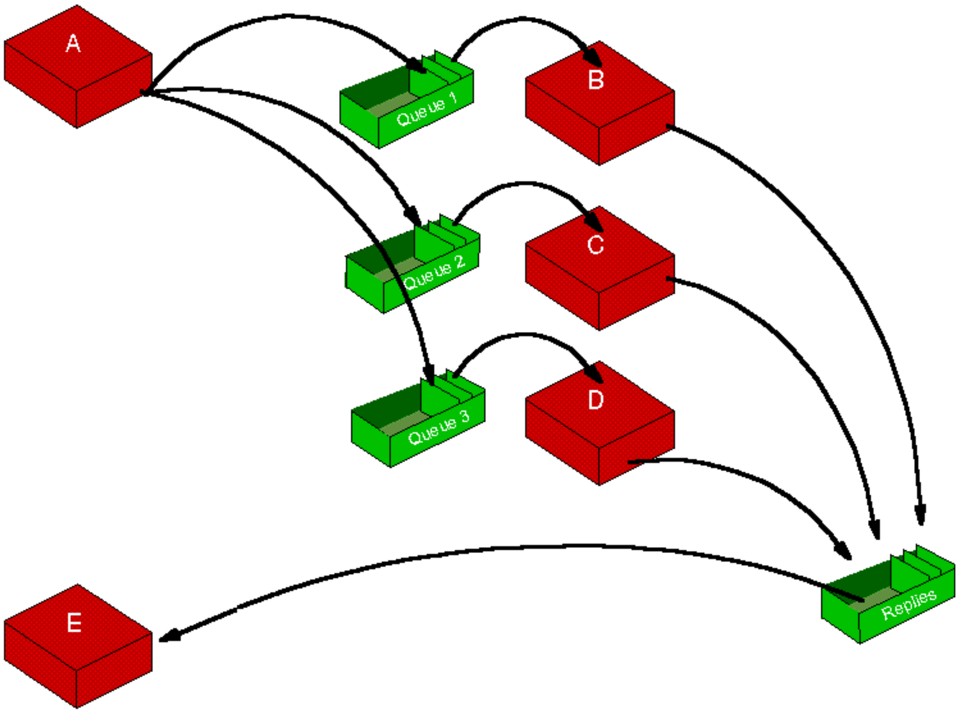


Figure 23. Processing Requests in Parallel

The MQSeries product family is based on a simple concept; however, you should refer to the MQSeries manuals for detailed information on how to implement MQSeries in your environment and how to utilize the MQI in your applications. Most MQSeries manuals can be viewed on the Web at: `http://www.software.ibm.com/ts/mqseries/library/manuals`

MQSeries provides various additional tools. In this redbook, we make use of the MQSeries Client for Java and the MQSeries Trigger Monitor for Lotus Notes agents.

7.2 MQSeries and Java

MQSeries Version 5 provides two different types of Java support:

- MQSeries Client for Java

  Enables Java applets and applications to use MQSeries applications through a Web browser or applet viewer. It is written entirely in Java and provides you the same messaging APIs as MQSeries's COBOL and C interfaces but with a Java class library countenance.

- MQSeries Bindings for Java

  Enable Java applications to connect directly to an MQSeries queue manager.

Both products include the MQSeries for Java classes (see Figure 24 on page 72).

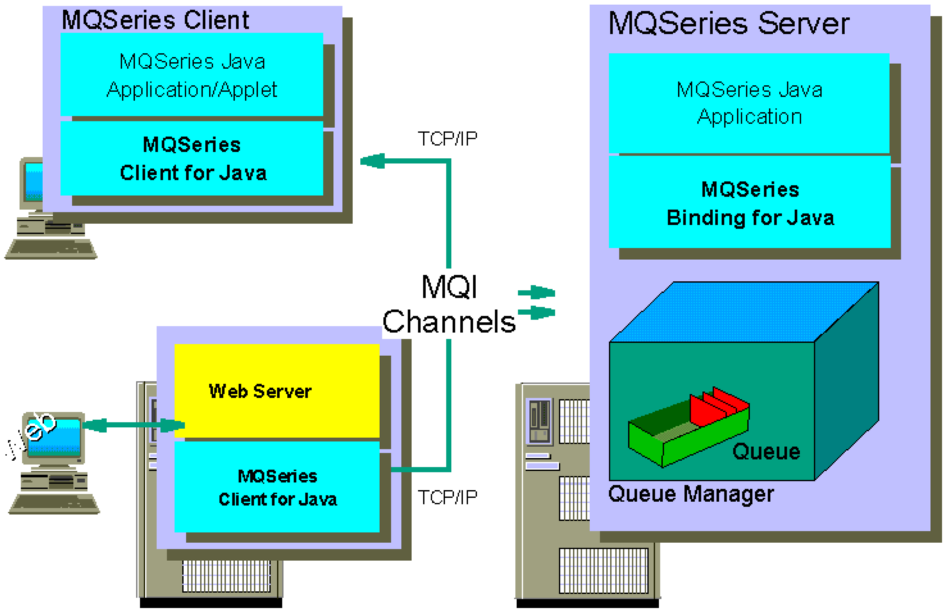


Figure 24. MQSeries Java Support

MQSeries is the tool of choice for heterogeneous platforms containing a mixture of Java and other operating environments. Using MQSeries Version 5 and its Java client is a matter of setting up MQSeries channels, establishing queues and adding a few Java statements to your programs. For more details refer to the MQSeries for Java documentation.

7.2.1 MQSeries Client for Java

MQSeries Client for Java is an MQSeries client written in the Java programming language for communicating through TCP/IP. It enables Web browsers and Java applets to issue calls and queries to MQSeries giving access to mainframe and legacy applications over the Internet without the need for any other MQSeries code on the client machine.

The MQSeries Client for Java enables application developers to exploit the power of the Java programming language to create applets and applications which can run on any platform that supports the Java run-time environment. These factors combine to dramatically reduce the development time for multi-platform MQSeries applications and future enhancements to applets are automatically picked up by end users when the applet code is downloaded.

The MQSeries Client for Java is shipped as part of the V5 MQSeries Client on AIX, HP-UX, OS/2, Sun Solaris and Windows NT. These MQSeries clients can be obtained with the MQSeries product or downloaded from the MQSeries Web site. The MQSeries Client for Java obtained in this way, provides MQSeries V5 function and is capable of running with JDK V1.1.1 or later. The MQSeries Client for Java can also be obtained for other platforms, such as Windows 95, from the MQSeries Web site as SupportPac MA83. The MQSeries Client for Java supplied with this SupportPac provides MQSeries V2 function and runs with JDK V1.02.

7.2.2 MQSeries Bindings for Java

The MQSeries Bindings for Java enable you to write MQSeries applications using the Java programming language. These applications communicate directly with MQSeries queue managers to provide a high-productivity, high-performance development option.

The MQSeries Bindings for Java also enable application developers to exploit the power of the Java programming language to create applications which can run on any platform that supports the Java run-time environment. These factors combine to dramatically reduce the development time for multi-platform MQSeries applications.

The MQSeries Bindings for Java provide the same programming interface as the MQSeries Client for Java, but they use Java native methods to call directly into the existing queue manager API rather than communicating through an MQSeries server connection channel. This provides better performance for Java MQSeries applications than the equivalent function written to use the MQSeries Client for Java. Unlike the MQSeries Client for Java, applications written using the bindings for Java cannot be downloaded as applets and they cannot be run inside an applet viewer or Web browser. However, as the MQSeries Client for Java, and the MQSeries Bindings for Java share a common programming interface, application code can be quickly and easily modified to run in either environment.

To develop applets or applications that use the MQSeries Bindings for Java, you also need to have JDK 1.1.1 (or later) on your machine.

7.2.3 MQSeries Java Classes

MQSeries for Java contains the following classes all prefixed with *com.ibm.mq* (or for the MQSeries Bindings for Java with *com.ibm.mqbind*). All these classes extend the java.lang.Object class, unless otherwise indicated:

- MQChannelDefinition

  Used to pass information concerning the connection to the queue manager to the send, receive and security exits.

- MQChannelExit

  Defines context information passed to the send, receive and security exits when they are invoked. The exitResponse data member should be set by the exit to indicate what action the MQSeries Client for Java should take next.

- MQEnvironment

  Contains static data members which control the environment in which an MQQueueManager object (and its corresponding connection to MQ) is constructed. You should always set the values in the MQEnvironment class before constructing an MQQueueManager instance.

- MQException

  Extends java.lang.Exception. Thrown whenever an MQ error occurs. You can change the java.io.PrintStream to indicate which exceptions are logged by setting the value of MQException.log. The default value is System.err. Constants beginning MQCC_ are MQSeries completion codes, and constants beginning MQRC_ are MQSeries reason codes. TMQGetMessageOptions

  Contains options that control the behavior of MQQueue.get.

- MQManagedObject

  Superclass for MQQueueManager, MQQueue and MQProcess provides the ability to inquire and set attributes for the following objects:

  - MQDistributionList

  Created through the MQDistributionList constructor or through the accessDistributionList method for MQQueueManager. A distribution list represents a set of open queues to which a message can be sent using a single call to the put() method. This class can only be used with MQSeries V5.

  - MQProcess

    Provides inquire operations for MQ processes.

  - MQQueueManager

    Used to create a connection to the named queue manager.

  - MQQueue

    Provides inquire, set, put and get operations for MQ queues. The inquire and set capabilities are inherited from MQ.MQManagedObject.

- MQMessage

  Represents both the message descriptor and the data for an MQ message. There are a group of readXXX methods for reading data from a message, and a group of writeXXX data for writing data into a message. The format of numbers and strings used by these read and write methods can be controlled by the encoding and characterSet data members. The remaining data members contain control information that accompanies the application message data when a message travels between sending and receiving applications. Implements java.io.DataInput, java.io.DataOutput.

- MQMessageTracker

  Inherited by MQDistributionListItem, where it is used to tailor message parameters for a given destination in a distribution list.

  - MQDistributionListItem

    Represents a single item (queue) within a distribution list.

- MQPutMessageOptions

  Contains options that control the behavior of MQQueue.put. This class contains no get version or set version methods. When used with distribution lists, a Version 2 structure (for MQSeries V5) is automatically used.

The following Java interfaces also extend the java.lang.Object class:

- MQC

  The MQC interface defines all the constants used by the MQSeries Java programming interface. To refer to one of these constants from within your programs, simply prefix the constant name with MQC.

- MQReceiveExit (MQSeries Client for Java only)

  Allows you to examine and possibly alter the data received from the queue manager by the MQSeries Client for Java. To provide your own receive exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.receiveExit variable to it before constructing your MQQueueManager object.

- MQSecurityExit (MQSeries Client for Java only)

  Allows you to customize the security flows that occur when an attempt is made to connect to a queue manager. To provide your own security exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.securityExit variable to it before constructing your MQQueueManager object.

- MQSendExit (MQSeries Client for Java only)

  Allows you to examine and possibly alter the data sent to the queue manager by the MQSeries Client for Java. To provide your own send exit, define a class that implements this interface. Create a new instance of your class and assign the MQEnvironment.sendExit variable to it before constructing your MQQueueManager object.

7.2.4 MQSeries Trigger Monitor for Lotus Notes Agents

Some MQSeries applications that service queues run continuously, so they are always available to retrieve messages that arrive on the queues. When the number and frequency of messages arriving on the queues is unpredictable it may not be desirable to have the application continuously running. In this case, applications could be consuming system resources even when there are no messages to retrieve.

An alternative is to use *triggering*. This means that your application runs only when it has a message or batch of messages to process, depending on how you set the MQSeries application queue attributes.

The MQSeries Trigger Monitor for Lotus Notes agents is a modified version of the standard MQSeries trigger monitor, with support for triggering Lotus Notes agents in addition to other MQSeries application types. It can be used to trigger any Notes agent, including those written using the MQSeries link LotusScript Extension (MQLSX), the MQSeries Enterprise Integrator (MQEI), or in our case the MQSeries Client for Java.

The MQSeries Trigger Monitor for Lotus Notes agents is available from the MQSeries Web site as a SupportPac: MA3L for OS/2 and MA7E for Windows

NT. (It is hoped that versions for AIX, HP-UX and Sun Solaris will also be made available.)

The trigger monitor is a continuously-running program that uses only a small amount of system resources. In order to use it you need three things:

1. An application program queue known to your Notes agent
2. An initiation queue
3. A process definition containing details of the Notes agent

Example MQ definitions are shown in “From the Enterprise to Domino Using MQSeries” on page 176.

You start the trigger monitor from the command line with the following syntax:

```
runmqtnm [-m QueueManagerName] [-q InitiationQueueName]
runmqtnc [-m QueueManagerName] [-q InitiationQueueName]
```

where the runmqtnm command expects an MQSeries queue manager to be installed on the same machine, while runmqtnc makes use of an MQSeries client (not the MQSeries Client for Java). If no parameters are entered, the default queue manager and queue SYSTEM.DEFAULT.INITIATION.QUEUE are used.

7.3 Domino and MQSeries

Figure 25 on page 78 shows the different ways a Notes or Domino application can access MQSeries applications.

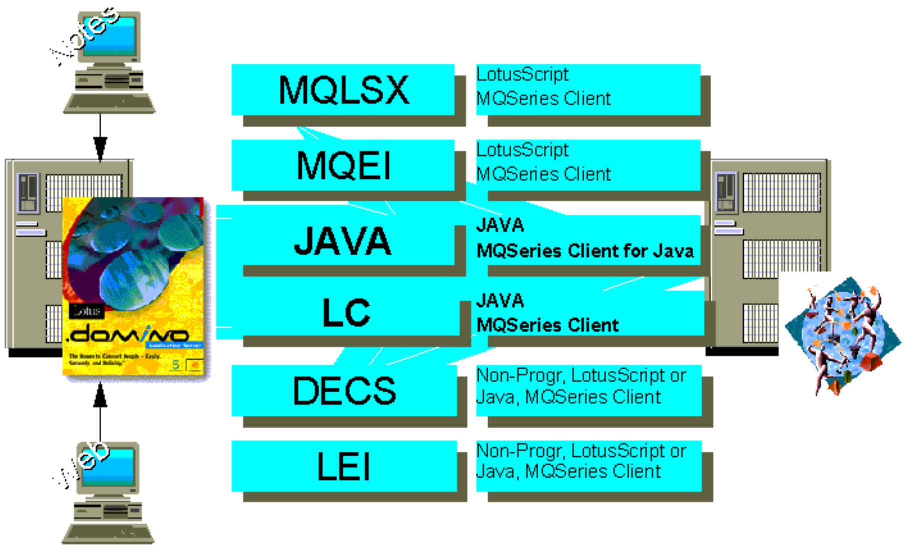


Figure 25. Notes/Domino Access to MQSeries

A variety of integration techniques and products are available to leverage the data storage and manipulation power of MQSeries and the messaging and groupware capabilities of Domino. They fall within the following categories:

- Native programmatic Domino access to MQSeries from a LotusScript program
  - MQSeries Enterprise Integrator (MQEI)
  - MQSeries LotusScript extension (MQLSX)
  - Lotus Connector LotusScript extension (LC LSX) and the Lotus Connector for MQSeries (not available yet)
- Native programmatic Domino access to MQSeries from a Java program
  - MQSeries Client for Java
  - MQSeries Bindings for Java
  - Lotus Connector Java classes and the Lotus Connector for MQSeries (not available yet)
- Native non-programmatic Domino access to MQSeries (not available yet)
  - Domino Enterprise Connection Services (DECS)
  - Lotus Enterprise Integrator (formerly called NotesPump)

Chapter 8. WebSphere

As the use of the Internet and intranets has increased, there has been a rapid progression from the publishing of static HTML pages to running applications over the Web. The trend now is towards the convergence of business processes, that is the integration of applications.

The focus for requirements can be split into three areas:

1. Infrastructure services
   - Component services
   - Web services
2. Application services
   - Component consumption
   - Provider/consumer of external services

     Security - Messaging - Directory - and so on...
   - Legacy and enterprise connectivity
   - Productive development tools
3. Advanced services
   - Workflow
   - Business rules
   - Mobile services
   - Repository

8.1 WebSphere Product Family

The IBM WebSphere software product line is a set of software products that help customers build and manage high-performance Web sites to ease the transition from simple Web publishing to advanced e-business Web applications.

WebSphere is a core element of IBM's e-business strategy. It brings together three markets:

- Web computing, with its rapid development, rapid deployment and scalability
- Java computing, with its object-oriented, portable applications
- Enterprise computing, with its transactional integrity and control

The WebSphere family, along with Lotus Domino, represents the industry's most complete range of Web application server environments that support business applications from simple Web publishing through enterprise-scale transaction processing.

The WebSphere family integrates the Web server, transaction processing, Web commerce and distributed component technologies of IBM's WebSphere Application Server.

The product line also includes WebSphere Studio, an integrated set of Web development tools and WebSphere Performance Pack, Web facilities management software that supports rapid growth of high-volume Web sites.

Figure 26 on page 80 shows the different components of the WebSphere family.

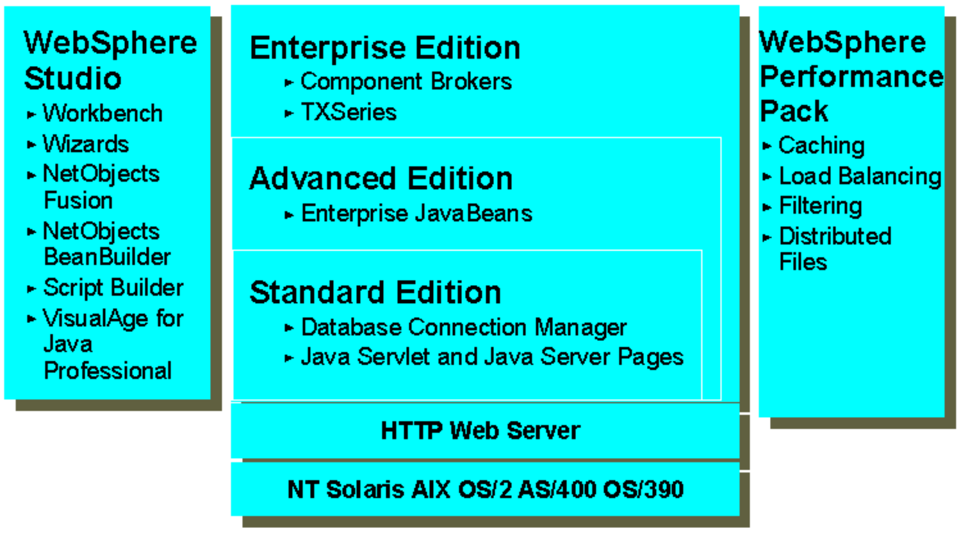


Figure 26. WebSphere Family

In the following we describe each component of the family:

8.1.1 WebSphere Application Server

WebSphere Application Server (WAS) lets you achieve your write once, use everywhere goal for servlet development. The product consists of a Java-based servlet engine that is independent of both your Web server and its underlying operating system.

WAS offers a choice of server plug-ins that are compatible with the most popular server APIs. The supported Web servers are:

- Lotus Domino R5
- Lotus Domino Go Webserver
- Netscape Enterprise Server
- Netscape FastTrack Server
- Microsoft Internet Information Server
- Apache Server

WAS runs on multiple platforms: AIX, OS/2, OS/390, Sun Solaris and Windows NT.

In addition to the servlet engine and plug-ins, WAS provides:

- Implementation of the JavaSoft Java Servlet API, plus extensions of and additions to the API.
- Sample applications that demonstrate the basic classes and the extensions.
- The Application Server Manager, a graphical interface that makes it easy to set options for loading local and remote servlets, to set initialization parameters, to specify servlet aliases, to create servlet chains and filters, to monitor resources used by WAS, to monitor loaded servlets and active servlet sessions, to log servlet messages, and to perform other servlet management tasks.

8.1.2 WebSphere Studio

WebSphere Studio combines graphical development wizards with tools for Web site design and Java development. These wizards and tools simplify and speed the application development process, and include:

- Web Development Workbench - a Web site project organizer and launch platform.
- Servlet generation wizards - for building Java servlets to access JDBC databases and JavaBean components.
- VisualAge for Java, Professional Edition V2.0 - IBM's award-winning Java application development environment for building Java applications, applets, servlets and JavaBean components.
- NetObjects Fusion V3.0 - allows Web site developers to design and produce an entire Web site, including individual pages and all links. It

features automated site building, automatic link management, remote database access, and design and publishing capabilities.

- NetObjects BeanBuilder V1.0 - the visual authoring tool for combining JavaBeans and Java applets, BeanBuilder allows individuals overseeing the content of online business processes to create more compelling, highly interactive Web sites with revolutionary ease-of-use. NetObjects ScriptBuilder V3.0 - combines a text-based script editor and development tools for creating and editing HTML, script and Java server pages.

8.1.3 WebSphere Performance Pack

WebSphere Performance Pack is built up of three main components, which permit you to reduce Web server congestion, increase content availability and improve Web server general performance:

1. File sharing

   The file sharing component is an enterprise file system that enables cooperating hosts (clients and servers) to efficiently share file system resources across both local area networks and wide area networks. It provides non-disruptive real-time replication of information across multiple servers, which guarantees data consistency, availability, global stability and administrative efficiency, which are required by large distributed Web sites or by Web sites with volatile content, which require considerable administrative effort to maintain such things as content links, URLs, and file I/O mapping.

2. Caching and filtering

   The caching and filtering component is a caching proxy server that provides highly scalable caching and filtering functions associated with receiving requests and serving URLs. With tunable caching capable of supporting high cache hit rates, this component can reduce bandwidth costs and provide more consistent rapid customer response times.

3. Load balancing

   The load balancing component is a server that is able to dynamically monitor and balance TCP servers and applications in real time. The main advantage of the load balancing component is that it allows heavily accessed Web sites to increase capacity, since multiple TCP servers can be dynamically linked into a single entity that appears in the network as a single logical server.

8.2 WebSphere Application Server

WAS is a Java-based application environment for building, deploying and managing Internet and intranet Web applications. This complete set of products expands to fit your Web application server needs, ranging from the simple to the advanced to the enterprise level.

WAS provides a secure and scalable run-time environment server, with additional SSL-based security and performance features, that supports Java servlets and JavaServer page scripting.

Three editions of WAS are available to meet your needs:

- Standard Edition combines the control and portability of server-side business applications with the performance and manageability of Java technologies to offer a comprehensive Java-based Web application platform. It enables powerful interactions with enterprise databases and transaction systems.

  Standard Edition provides you with an open, standards-based, Web server deployment platform and includes Web site development and management tools to help accelerate the process of moving to e-business. It provides support for Java servlets and JSPs.

  Standard Edition focuses on a servlet manager, high-performance database connections, and application services for session and state management.

- Advanced Edition, which builds off the Standard Edition, introduces a key element for the purpose of our book: server capabilities for applications built to Sun's Enterprise JavaBean specification. Deploying and managing JavaBean components provides a stronger CORBA implementation that maps to portable Java technologies.

  Advanced Edition provides enhanced support for scaling your Web site into a secure, transactional e-business application site. This edition connects Web applications to existing databases and host-based transaction systems, and offers sophisticated tools to simplify distributed component-based application development.

- Enterprise Edition, which enhances the Advanced Edition, offers a robust solution to grow e-business applications into mission-critical enterprise environments. It combines TXSeries, IBM's world-class transactional application environment (providing the CICS and Encina products), with the full distributed object and business process integration capabilities of Component Broker.

WAS's Java environment supports multiple Web servers. This allows you to choose between the native Domino, Apache, Microsoft IIS or Netscape Web servers. Using Domino R5 you can also substitute the native application server function (servlet engine) in Domino with WAS Standard Edition 2.0.

8.2.1 Servlet Runtime Environment

The Servlet Runtime Environment provides a fast engine for running server-side Java servlets and JavaBeans. Java servlets can be used to coordinate I/O between server-side JavaBeans and Web browser HTML clients. Servlets facilitate layering and separating the presentation logic, business logic, and backend data access logic. Server-side JavaBeans are used for business logic and access to backend databases, transactions and existing applications.

Within the middle-tier and the WAS environment, the focus here is on the servlet engine, which is Java-based. The servlet runtime provides the Sun/JavaSoft APIs for the Java servlet environment, including the servlet life cycle: init, service, destroy. Servlets can be preloaded, so that when a client request comes in, a servlet is loaded and waiting to act on it. Servlets send and receive most of their data through output and input streams. These streams are supplied each time a servlet is invoked using the service callback.

The servlet manager creates instances of the servlets, deploys them, manages their execution, and provides tracing and monitoring facilities for them. Servlets themselves handle HTTP requests, maintain an HTTP session with the client, produce presentation logic via HTML, stream and non-transactional business logic. Servlets can also call componentized functions or routines built as JavaBeans. These beans can be called to connect to or interface with remote systems of various types and formats using different native APIs.

8.2.2 Enterprise JavaBeans Server

WAS provides an Enterprise JavaBeans Server (EJS) for the EJBs, a run-time engine with high-performance database and transaction connections, and similar application services. It provides a base for the execution of EJBs.

The server is the outermost *container* of the various elements making up an EJB environment. It offers the following services:

- Container run-time environment

EJS is able to support EJB containers providing client access, multi-threading, and memory management.

- Access to a distributed transaction system

  In WAS Version 2.0, EJS implements a distributed transaction service.

- Access to a data store

  EJS uses DB2 UDB to support the persistent storage of entity beans.

- Access to a naming service

  A JNDI-accessible naming space is used by clients to locate the enterprise beans. A container registers the home interface with a JNDI-compliant naming service in order for clients to gain access to the home interface and through it to the beans.

- Provide operation resources

  EJS provides operation resources such as processes and execution threads, memory, networking facilities, etc. to the containers and the elements within.

Figure 27 on page 85 shows the architecture of the Enterprise Java Server.

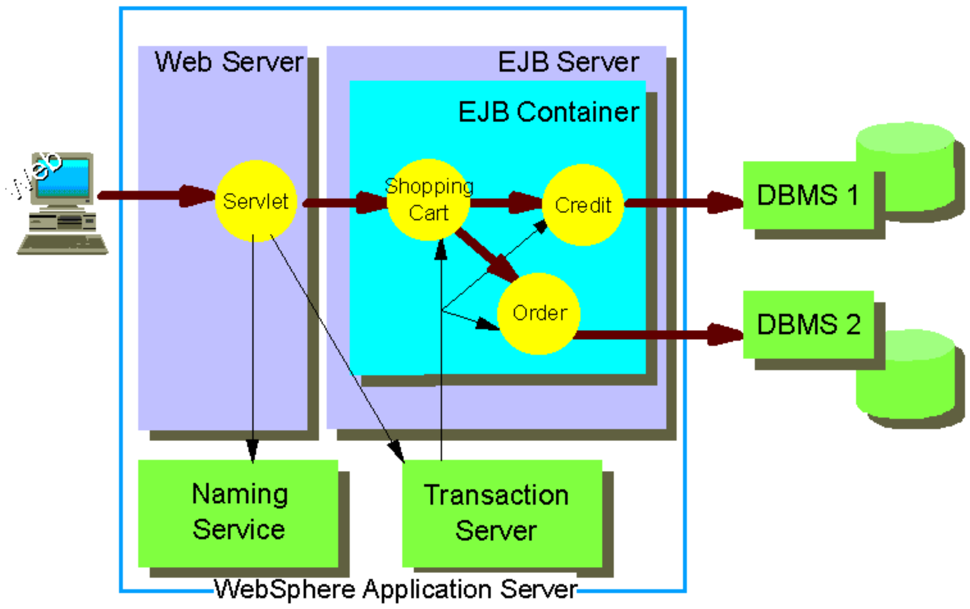


Figure 27. Enterprise Java Server

This server handles running the business logic, which ensures transactional integrity. It is a managed object framework and deals with the life-cycle and persistence issues of the enterprise beans it manages.

Clients communicate with the EJB application using a variety of protocols such as RMI. Browsers can invoke the EJB application through a servlet running on the HTTP server. The browser communicates with the servlet using HTTP, and the servlet communicates with the EJB application using RMI.

The Java platform provides some APIs that enable the EJB to access enterprise services and data resources, such as:

- Java Transaction Services (JTS), for invoking transaction services
- Java Naming and Directory Interface (JNDI), for accessing naming and directory services
- Java DataBase Connectivity(JDBC) and Java SQL (JSQL), for accessing data in existing databases through a common interface
- RMI, for creating remote interfaces to distributed computing on the Java platform

8.3 Domino and WebSphere

Effective Web strategies require both collaboration and transaction capabilities. Together, Domino and WebSphere offer you all the capabilities needed to build, run, and manage high value e-business applications:

- Domino provides a powerful environment for developing and running collaborative applications (WWW, intranet or extranet), messaging and enterprise calendaring and scheduling.

  As a Web application server, Domino is optimized to manage work and information flow, and to facilitate electronic relationships through focused business collaboration. Domino also provides a highly productive rapid application development environment.

- WebSphere is a line of Web application servers with graduated capabilities optimized to manage the execution of distributed transactions and components, and to meet the performance and throughput needs of applications ranging from simple Web sites to sophisticated enterprise systems.

  WebSphere provides an ideal foundation for building, deploying and managing Java-based transactional Web applications. In its different versions, it provides:

  - The foundation of Web server deployment, site development and management
  - Security, transaction capability, connection to existing databases and host transaction systems and simplifies distributed component-based application development
  - Full distributed object and integration capabilities required of a transaction system

The positioning of Domino and WebSphere is illustrated in Figure 28 on page 88.

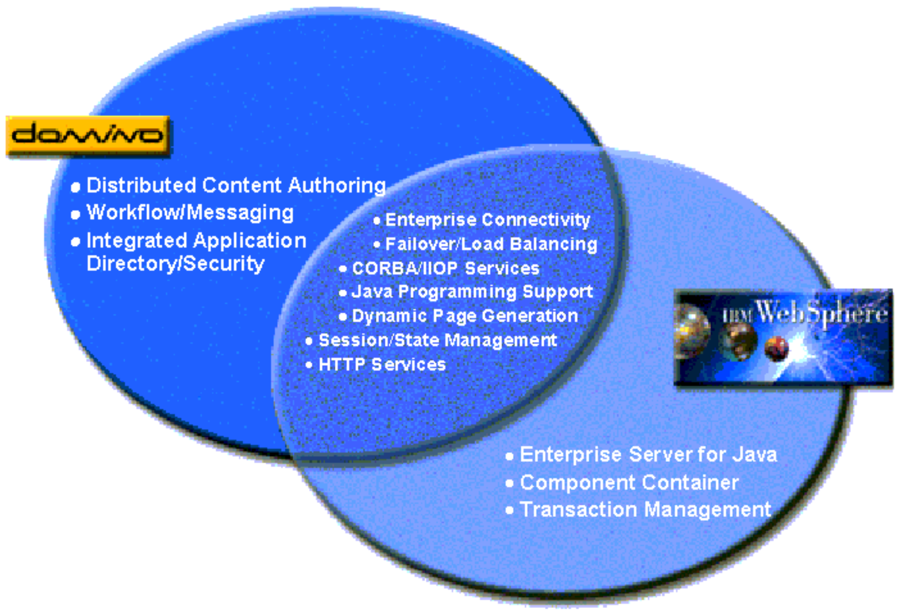

domino
• Distributed Content Authoring
• Workflow/Messaging
• Integrated Application Directory/Security
• Enterprise Connectivity
• Failover/Load Balancing
• CORBA/IIOP Services
• Java Programming Support
• Dynamic Page Generation
• Session/State Management
• HTTP Services
IBM WebSphere
• Enterprise Server for Java
• Component Container
• Transaction Management

Figure 28. Domino and WebSphere Positioning

In many Web applications, which require both collaboration and transaction capabilities, deploying both Domino and WebSphere is the appropriate solution. Domino provides the rich set of application services that WebSphere application servers can use, and WebSphere provides the robust transaction management that Domino applications can use.

Domino complements WebSphere by offering a set of services which expand the power and interactive nature of a Web site through workflow and collaboration functionality. Lotus Domino has a focus on collaboration and interaction through broad business applications which can rely on workflow capabilities, whereas WebSphere's focus is on transaction processing and serving up Java bean applets. Together they form a powerful solution.

Domino and WebSphere can be easily linked to comprise a complete system capable of many tasks from global workgroup collaboration and interactive multilingual Web application development, to transactional systems based on Java components. Connector software and joint ship vehicles for Domino and WebSphere demonstrate how they work together:

- Domino R5 incorporates the Standard Edition of WebSphere as part of the Domino run-time environment.

- VisualAge for Java includes Domino Java class descriptions that WebSphere can use to invoke Domino object services.
- Domino allows Java agents to invoke EJBs managed by WebSphere Advanced Edition.

Lotus will also offer a connector to integrate Domino with the transaction management services of WebSphere Enterprise Edition.

8.3.1 Advantages of Using Domino with WebSphere

WebSphere is a Java-based execution and management platform. The use of WebSphere with Domino enhances the intersection of Domino's collaboration and e-mail strengths with IBM's transactional products.

The IBM WebSphere Application Server provides better performance than other server function extensions, because it runs as an in-process plug-in with the server. It fully supports the latest session-tracking APIs, including session clustering (sharing of HTTP sessions among several Web servers). It also provides a JDBC-based user profile class that can be used in conjunction with sessions to further personalize your Web site. And it includes CORBA support, a configuration interface, robust security features and more.

EJB Support

WAS contains an EJB manager. EJBs provide a component architecture for multitiered, distributed Java applications. The EJB infrastructure provides transactional and system services for the application components, making distributed, client/server applications easier to develop, deploy, manage and maintain.

JDBC Connection Management

A JDBC connection manager is provided as part of the servlet engine in WAS. It provides a bean interface and maintains a pool of active JDBC connections to a database, hence improving performance and throughput.

Enhanced Servlet Support

Servlet queues in WAS help to ensure the reliability, security and throughput of the server.

8.3.2 Recommendations of Use

Lotus Domino and IBM WebSphere Application Server are both robust Web application servers that meet different customer needs. Lotus Domino is ideal for customers who are focusing on collaborative/workflow solutions and want to build intranet and extranet Web applications that integrate their business processes with those IT systems. For example, a new supplier suggestion system might be built using Lotus Domino that allows suppliers, through an extranet, to send in documents with their suggestions, which are then processed in a workflow system to generate cost savings for the company.

The IBM WebSphere Application Server is ideal for organizations building more transactional Web applications using Java servlets, Enterprise JavaBeans and Java-based connectors. WebSphere Application Server is a deployment and management platform that enables companies to upgrade from a publishing-based Web presence on HTTP servers to e-business solutions. For example, WebSphere Application Server is the foundation on which the next release of IBM Net.Commerce will provide e-commerce capabilities that allow customers to set up electronic storefronts with a more personalized, dynamic selling environment.

Using WebSphere Application Server, customers can build highly scalable dynamic front-ends to legacy applications. Requirements that are driven more by the desire to integrate your content and data with your business processes, imply you should consider the enterprise integration, collaborative, messaging, workflow, and process management capabilities of Domino.

The IBM/Lotus e-business strategy leads to a preferred option of using Domino with WAS. Domino will handle HTML requests and workflow applications. WAS will provide the Web server and deal with all servlets and Java applications. The direction is also to utilize VisualAge for Java to create/modify Java programs, as it provides a superior development and testing environment (even for Domino agents) than Domino R5.

Ultimately, the result should be that users see better scalability and reliability in their applications, by combining the strengths of these products.

Part 2. Installation and Setup

In this part, we explain how to install and set up the different tools that support the development and execution of Domino applications that connect to the Enterprise using Java.

Chapter 9. Java

In this chapter we explain the installation and settings needed to support Java and JDBC. We concentrate on the installation and configuration for the Windows NT platform that we used during our tests.

Java Development Kit
If you want to develop Java applets or applications, you need to install JDK Version 1.1 or higher on the server. JDK Version 1.1 is available on the Sun Web site for Windows platforms, and on the IBM Java Web site for IBM platforms (AIX, OS/2, OS/390 (UNIX Services), OS/400, and VM/ESA).

Java Servlet Development Kit
The Java Servlet Development Kit (JSDK) contains a simple servlet engine for developing and testing servlets, the javax.servlet package sources, and API documentation. JSDK Version 2.0 is available on the Sun Web site for Windows platforms.

Java Database Connectivity
The JDBC API is a SUN standard data access interface. This API provides Java programmers with universal access to a wide range of relational databases. JDBC is included in JDK 1.1.

Environment Variables
To run Java programs, the system must be able to access the required Java class files. Table 2 on page 93 gives the different values that must be added to the CLASSPATH environment variable if you need the corresponding Java package.

Table 2. CLASSPATH Environment Variable Values

Directory	**Java Package**
. (dot)	Local package and class files
c:\domino\notes.jar	Domino classes
c:\domino\NCSO.jar	Domino CORBA classes
c:\domino\JdbcSqllib\JdbcDomino.jar	Domino Driver for JDBC classes
c:\lotus\domino\lcjava.zip	Lotus Connector Java classes
c:\jdk1.1.7\lib\classes.zip	JDK 1.1.7 classes
c:\jsdk2.0\lib\jsdk.jar	Java servlet classes
c:\sqllib\java\db2java.zip	DB2 classes

Directory	Java Package
c:\ibm\ctg\classes\ctgclient.jar	Java CICS application classes
c:\ibm\ctg\classes\ctgserver.jar	Java CICS local gateway classes
c:\mqm\java\lib; c:\mqm\tools\javaclnt\samples\En_US	MQSeries client for Java
c:\mqm\java\lib; c:\mqm\tools\mqbind\samples\En_US	MQSeries binding for Java classes

Modify the path to fit your own installation.

Chapter 10. Domino

In this chapter, we explain the installation and setup of the Domino environment for both releases of Domino as the Java support has changed between the releases.

10.1 Java Applet

Before you can add a Java applet to a Domino or a Notes application, you must set up your workstation to enable Java applets (Advanced option of the user preferences). If you want to link to applets on the Web, make sure your Location document specifies a valid Web proxy.

10.2 Release 4.6

In this section we explain how to set up and configure Domino R4.6 to support the Java environment.

10.2.1 Installation

You may have to modify the Lotus Notes initialization file (*notes.ini*) to support the tools.

HTTP Server
To load the Domino HTTP Web server task automatically, add the following line to notes.ini:

```
ServerTask=<any other tasks>,HTTP
```

When you restart the server, it loads the HTTP Web server task. The following line appears on the server console:

```
01/01/98 11:23:36 AM HTTP Web Server started
```

JavaUserClasses
Domino needs to be able to find all classes of a server agent or servlet, not just the agent or servlet itself. The act of compiling the agent does not pull in the outside packages; they are just referenced by the class. There is no concept of static linking in Java because every class is dynamically linked at runtime. When the JVM in Domino or Notes loads the agent class, it also resolves and tries to load other classes that the agent class uses. If it cannot find a class that it needs, you get the *ClassNotFoundException*. The JVM in Domino looks for classes in three places:

- JavaUserClasses

If you have a JavaUserClasses setting in *notes.ini*, this list is searched for any needed classes. This list is similar in format to the more familiar CLASSPATH environment variable. It can contain directories, jar files, and/or zip files.

- Notes and Java core classes

  Next, the various Notes and Java core class archives are searched, such as:

  - Notes Java classes (*Notes.jar*)
  - Domino servlet classes (*icsclass.jar*)
  - Core Java classes (*rt.jar* and *i18n.jar*)

- Attached with the agent

  These are any classes, resources, or whatever else you decide to include when you define the agent. The JVM runs through the list of attachments and looks for the needed class. If any of the attachments are jar or zip files, they are searched internally for the required class or resource.

If after looking in these three places the JVM cannot find the class or resource, you get an appropriate error message.

Classpath

To develop Java servlets, you have to make the Domino servlets classes available to your Java IDE. If you are using the JDK, add servlet classes to the CLASSPATH:

```
CLASSPATH=<other path>;c:\notes\icsclass.jar
```

In Java applications the CLASSPATH environment variable is set properly to find the package. To minimize outside interference and sources of strange errors, Domino does not use CLASSPATH; rather it constructs its own internal class path, using the *JavaUserClasses* setting and its own core classes.

Agent Scheduling

When using scheduled agents, add the following lines:

```
AMgr_DocUpdateAgentMinInterval=1
AMgr_DocUpdateEventDelay=1
```

The agent manager responds to events as soon as possible.

You may have to enable locally scheduled agents in the *UserPreferences* section under File/Tools.

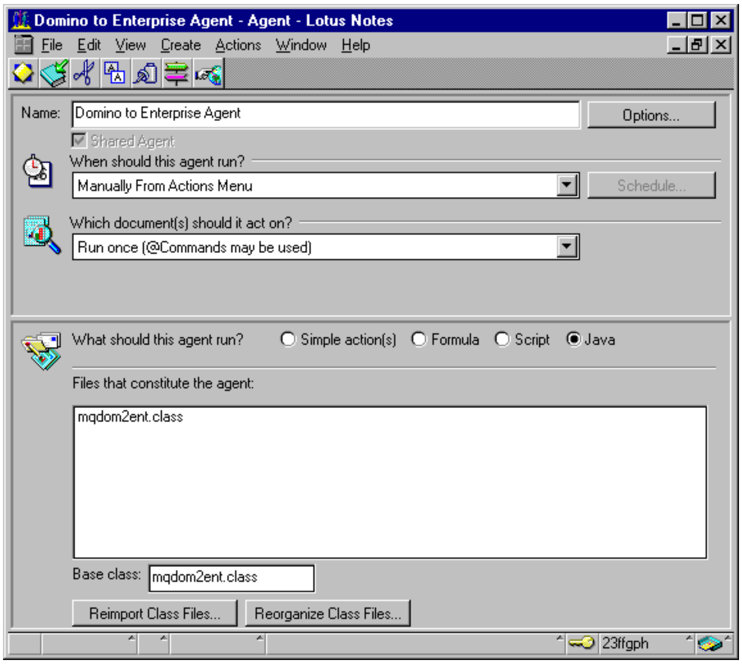
10.2.2 Java Agent Support

Domino R4.6 does not have an integrated development environment (IDE) that supports Java.The creation of a Java agent must, therefore, be done outside of Domino. This can mean using javac (to compile code written using a text editor)

```
javac class_name.java
```

or a visual tool such as VisualAge for Java (refer to the redbook *Using VisualAge for Java to Develop Domino Applications,* SG24-5424 for more details).

Domino R4.6 does provide support for Java and it is possible to select Java as the language of your agent (as illustrated in Figure 29 on page 97), and to import the class or classes that constitute your agent.


Domino to Enterprise Agent - Agent - Lotus Notes
File Edit View Create Actions Window Help
Options...
Schedule...
Reimport Class Files...
Reorganize Class Files...
23ffgph

Name: Domino to Enterprise Agent

Shared Agent

When should this agent run? Manually From Actions Menu

Which document(s) should it act on? Run once (@Commands may be used)

What should this agent run? ○ Simple action(s) ○ Formula ○ Script [✓ ● Java]

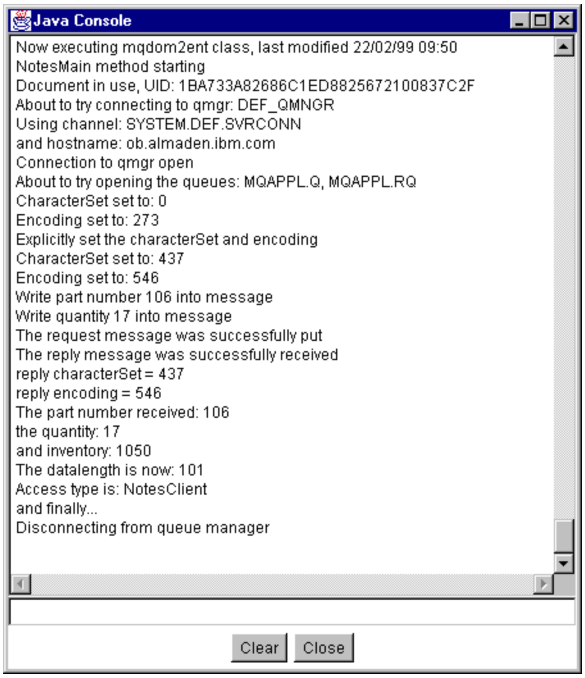
Files that constitute the agent:

mqdom2ent.class

Base class: mqdom2ent.class

Figure 29. Domino R4.6 Integrated Development Environment

If you run the agent from the Notes client, output goes to the Java debug console. To see the console, choose File - Tools - Show Java Debug Console. Output from scheduled agents goes to the Notes log. This is useful for debugging your agent as it displays any *print* statements (as illustrated in Figure 30 on page 98), but is not a complete debugger.



```
Now executing mqdom2ent class, last modified 22/02/99 09:50
NotesMain method starting
Document in use, UID: 1BA733A82686C1ED8825672100837C2F
About to try connecting to qmgr: DEF_QMNGR
Using channel: SYSTEM.DEF.SVRCONN
and hostname: ob.almaden.ibm.com
Connection to qmgr open
About to try opening the queues: MQAPPL.Q, MQAPPL.RQ
CharacterSet set to: 0
Encoding set to: 273
Explicitly set the characterSet and encoding
CharacterSet set to: 437
Encoding set to: 546
Write part number 106 into message
Write quantity 17 into message
The request message was successfully put
The reply message was successfully received
reply characterSet = 437
reply encoding = 546
The part number received: 106
the quantity: 17
and inventory: 1050
The datalength is now: 101
Access type is: NotesClient
and finally...
Disconnecting from queue manager
```

Figure 30. Java Console Started from Domino 4.6

You can run the agent from a browser by putting the following @command in an action hotspot, button ("Web access: Use JavaScript when generating pages" must be in effect), WebQueryOpen event, or WebQuerySave event:

```
@Command([ToolsRunMacro]; "My agent name")
```

You must use the PrintWriter object to print to the browser. However, print output is discarded for a WebQueryOpen event or if "Web access: Use JavaScript when generating pages" is in effect.

However, besides this there is little in terms of development support for Java agents in Domino 4.6.

Remember, if you modify your Java code, you will need to reimport the compiled class into the agent. Select the class in the Agent screen (refer to Figure 29 on page 97) and click the **Reimport Class Files...** button to display the screen shown in Figure 31 on page 99.

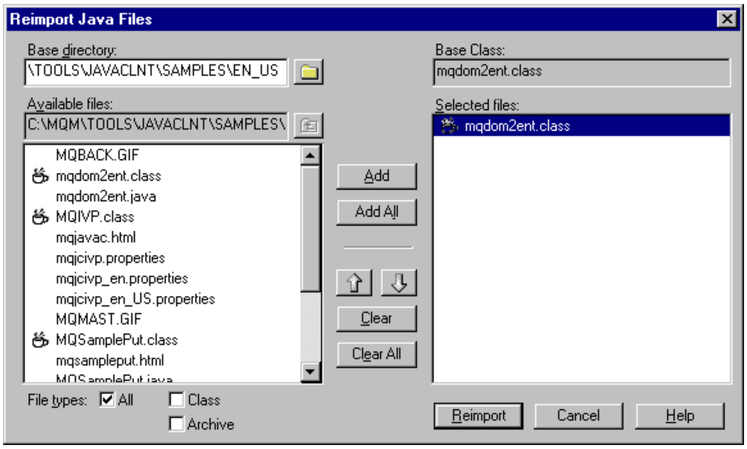


Figure 31. Reimporting Java Classes

Do not be too hasty following a reimport when running your agent on a Domino Server invoked from the Web. You need to restart your HTTP Web server before the change takes affect using the load command:

```
> tell http q
02/25/99 04:45:34 PM HTTP Web Server shutdown
> l http
02/25/99 04:45:46 PM HTTP Web Server shutdown
```

The practice of using the reimport function is recommended, although it does not seem to be necessary if running only on a Notes client. A simple way to ensure you are running the code you think you should be running is to include a *print* statement specifying the level of the agent at the beginning of your Java code.

10.2.3 Java Servlet Support

To use the servlet manager in Domino R4.6, you have to:

1. Load the Domino Web HTTP server task.

   This can be done manually by starting the HTTP task on the server using the following command:

   ```
   load http
   ```

   or by starting the HTTP task automatically each time the Domino server is started.

2. Enable the Java servlet support

In Domino R4.6, support for servlets is disabled by default. To enable servlet support, add the following line:

```
DominoEnableJavaServlets=1
```

When you restart the server, it loads the JVM and locates the ServletManager Java class, adding the *icsclass.jar* file to the CLASSPATH environment variable automatically. As the servlet support is loading, the following three lines appear on the server console:

```
01/01/98 11:23:36 AM HTTP Web Server started
01/01/98 11:23:38 AM JVM: Java Virtual Machine initialized
01/01/98 11:23:38 AM Java Servlet Manager initialized
```

3. Install servlets.

   To install a servlet, whether one you wrote or a pre-built one, you create a Servlet subdirectory in the Domino server data directory and copy the compiled servlets into that directory. Add a setting for JavaUserClasses to include a path for the directory that contains the servlets, as well as the directories that may contain any additional classes used by the servlet:

   ```
   JavaUserClasses=c:\notes\data\domino\servlet;c:\sqllib\java\db2java.zip
   ```

4. Register the servlet.

   To access a servlet from the Web, it must be registered in a Domino servlet configuration file, *servlet.cnf*, in the Domino data directory on the server. This file holds details of the servlets to load, whether to load at startup time, initialization parameters and URL mappings for your servlet.

   The following example registers the *NoParamServlet* servlet that requires no startup parameters and is started on demand:

   ```
   # Servlet NoParamServlet Registration
   Servlet NoParamServlet {
   }
   # URL to trigger the Servlet
   Service NoParamServlet /Servlet/MyFirstServlet
   ```

   In this example the *NoParamServlet* servlet can be triggered using the following URL:

   ```
   http://MyCompany.com/Servlet/MyFirstServlet
   ```

   The following example registers the *TwoParamServlet* servlet that requires two startup parameters (Username and CaseSensitive) and is loaded automatically when the servlet manager starts:

   ```
   #Servlet TwoParamServlet Registration
   Servlet TwoParamServlet {
   Username=GenericUser
   ```

```
CaseSensitive=No
GO_LOAD_STARTUP=Yes
}
# URL to trigger the Servlet
Service TwoParamServlet /Servlet/FindCustomer
```

In this example the *TwoParamServlet* servlet can be triggered using the following URL:

```
http://MyCompany.com/Servlet/FindCustomer
```

10.3 Release 5.0

In this section we explain how to set up and configure Domino R5 to support the Java environment.

10.3.1 HTTP and DIIOP Tasks

To load the Domino HTTP Web server task and the CORBA task automatically, add the following line in the Lotus Notes initialization file (*notes.ini*):

```
ServerTask=<any other tasks>,HTTP,DIIOP
```

When you restart the server, it loads both tasks. The following lines appear on the server console:

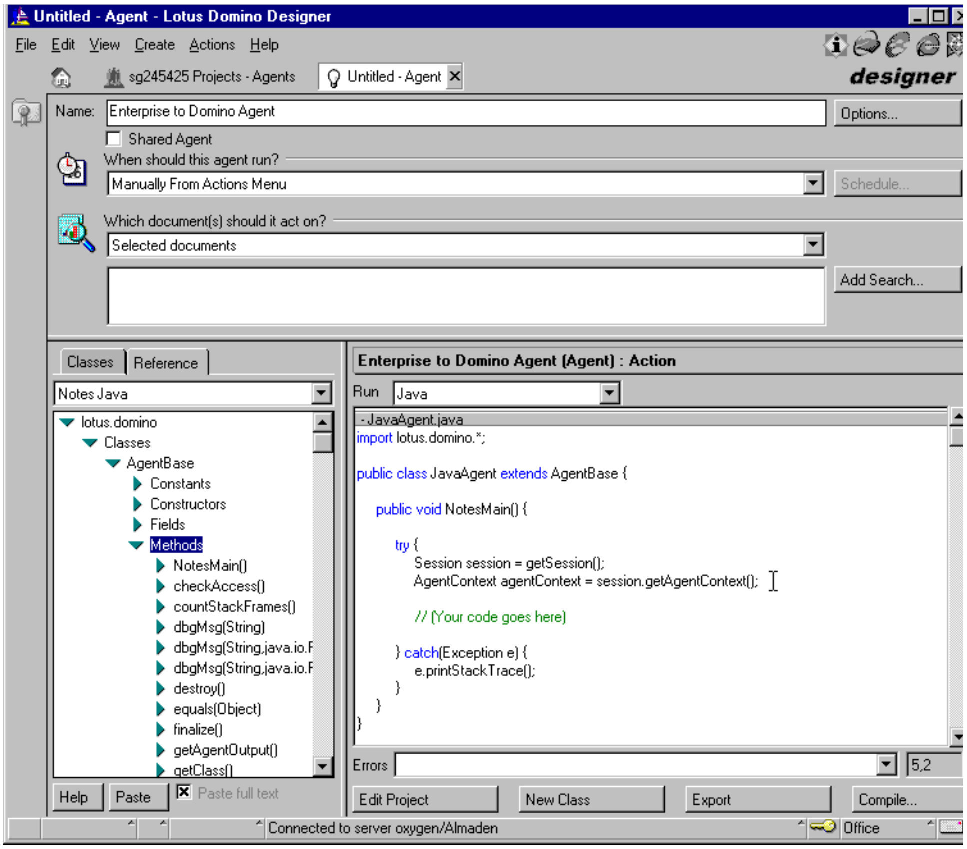
```
03/21/99 11:23:38 AM DIIOP Server started on oxygen.almaden.ibm.com
03/21/99 11:23:36 AM HTTP Web Server started
```

When Domino is running, you can start the tasks with the load console command.

10.3.2 Java Agent Support

In Domino R5 there is an IDE for Java. This allows you to create a Java agent within Domino in the same way as you would create a LotusScript agent. Alternatively, Java classes can still be imported as for Domino R4.6.

The IDE automatically includes statements in your new agent when you select to create it in Java, as shown in Figure 32 on page 102.



```
import lotus.domino.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {

        try {
            Session session = getSession();
            AgentContext agentContext = session.getAgentContext();

            // (Your code goes here)

        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 32. Domino R5 Java Integrated Development Environment

To develop a Java agent, you can use, apart from a simple text editor and the java compiler shipped with the JDK, the Domino R5 designer or a visual Java development tool such as VisualAge for Java.

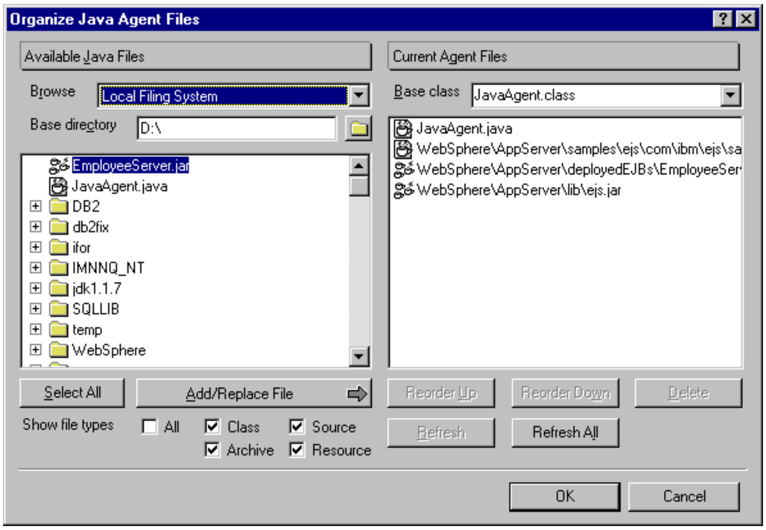
The Domino R5 designer allows you to display and paste Notes and Java constants, constructors and methods into your agent code. It helps you to create agents only. To create servlets you need to create your code externally.

The Java debugging environment is rudimentary in Domino as you use print method to the standard output to control the code. Domino offers also a Java console that can be used to debug foreground agents. For background agents, you have to use the Domino server console.

If you want a full environment to create and debug Java agents, you should use an IDE such as VisualAge for Java that allows you to display and paste Notes and Java constants, constructors and methods into any Java program, that is: agent, applet, or servlet. VisualAge for Java, with its connection to the AgentRunner, is a good choice for complex agent coding. The completed code can then be cut and paste into a Java agent, or the class used in an *Imported Java* agent.

10.3.3 JavaUserClasses

The JavaUserClasses statement in the notes.ini file is not strictly required in R5, if you are using Java code in your agent, although it can make creating agents (especially those using EJBs) easier. When using Java code in your agent, it is possible to **Edit Project** and add supporting Java classes to the agent (as illustrated in Figure 33 on page 103 and discussed in more detail in “Domino Agent” on page 205).


Organize Java Agent Files

Available Java Files

Browse: Local Filing System

Base directory: D:\

EmployeeServer.jar
JavaAgent.java
DB2
db2fix
ifor
IMNNQ_NT
jdk1.1.7
SQLLIB
temp
WebSphere

Current Agent Files

Base class: JavaAgent.class

JavaAgent.java
WebSphere\AppServer\samples\ejs\com\ibm\ejs\sa
WebSphere\AppServer\deployedEJBs\EmployeeSer
WebSphere\AppServer\lib\ejs.jar

Show file types: [ ] All [✓] Class [✓] Source [✓] Archive [✓] Resource

Figure 33. Importing Java Support Classes into the Project

If, however, you are using imported Java code, it would appear that all supporting Java classes still need to be listed in the JavaUserClasses statement.

The need or otherwise for a JavaUserClasses statement is only really significant if you:

- Distribute the Domino application between multiple servers, in which case you will need to remember to update the notes.ini file on each server.

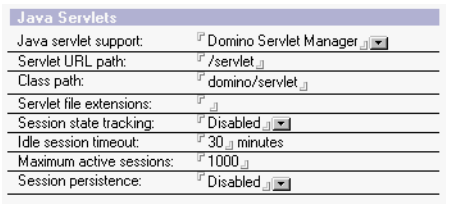
- Use a model where the agent code will actually be executed on Notes client machines. In this case, the need for a JavaUserClasses statement in each user's notes.ini file would not be desirable.

Remember also that with R5, you may have two notes.ini files: one in the *c:\lotus\domino\notes.ini* directory if you installed the server, and one in the *c:\lotus\notes\notes.ini* directory if you installed a client.

10.3.4 Servlet Manager

With Domino R5.0 the process for enabling the servlet manager is much simpler, although there are more configuration options. Most parameters for controlling the servlet manager are now in the Domino directory.

To set up the servlet configuration, open the server document in the Domino directory, select the **Internet Protocols - Domino Web Engine** tab. Figure 34 on page 104 shows the servlet configuration parameters section of the Domino server document.



Java Servlets

Java servlet support:	Domino Servlet Manager
Servlet URL path:	/servlet
Class path:	domino/servlet
Servlet file extensions:	
Session state tracking:	Disabled
Idle session timeout:	30 minutes
Maximum active sessions:	1000
Session persistence:	Disabled

Figure 34. Java Servlet Configuration

To support servlets in Domino R5, you have to:

1. Enable the servlet manager.

   Select one of the following options in the Java servlet support field of the servlet configuration parameters panel:

   Domino Servlet Manager
   : The Domino HTTP task loads both the JVM and the native servlet manager.

   Third Party Servlet Support
   : The HTTP task loads the JVM, but not the Domino servlet manager. This allows the use of third-party servlet managers such as IBM's WebSphere Application Server. In 12.2.2, “Servlet Manager” on page 125, we explain how to use WebSphere Application Server as the servlet manager.

In this panel, you can also specify the path in a URL that signals Domino that the URL refers to a servlet (*Servlet URL path*), the list of paths which the servlet manager class loader searches to find servlets and their dependent classes (*Class path*), and the list of URL file extensions that signal Domino that a URL refers to a servlet (*Servlet file extensions*). Refer to the Domino Designer help file for additional information.

2. Set the servlet properties

   Special properties for individual servlets can be specified in a text file called *servlets.properties* located in the Domino data directory. The following properties can be specified:

   - Alias
   - Initialization arguments
   - URL extension mapping
   - Load at servlet manager startup

   The following example registers the *TwoParamServlet* servlet that requires two startup parameters (Username and CaseSensitive) and is loaded automatically when the servlet manager starts:

   ```
   #Servlet TwoParamServlet Registration
   Servlet.FindCustomer.code=TwoParamServlet
   servlet TwoParamServlet.initArgs=Username=GenericUser,CaseSensitive=No
   servlet.startup=TwoParamServlet
   ```

3. Configure additional parameters

   You can configure any of the other servlet configuration parameters in the server document. Refer to the Domino Designer help file for more information.

10.3.5 Designer Setup

You need to install Domino Designer R5 to compile a Java program that uses the *lotus.domino* package.

The notes.ini initialization file must contain the following line:

```
ALLOW_NOTES_PACKAGE_APPLETS=1
```

If you are developing stand-alone Java applications, you have to include the following CLASSPATH environment variable:

```
CLASSPATH=<other path>;.;c:\notes\domino\java\ncso.jar;c:\notes\Notes.jar
```

Notes.jar contains the high-level *lotus.domino* package, the *lotus.domino.local* package for local calls, and the R4.6 *lotus.notes* package for compatibility. *NCSO.jar* contains the high-level *lotus.domino* package, and the *lotus.domino.corba* package for remote calls.

10.3.6 Run-Time Requirements

A machine running a Java application that makes local Domino calls must contain Domino R5 (Client, Designer, or Server) and must include Notes.jar in the CLASSPATH.

A machine running a Java application that makes remote Domino calls need not contain Domino R5, but must contain NCSO.jar and must include it in the CLASSPATH.

A machine running a Domino R5 agent that makes Domino calls must include Notes.jar in the CLASSPATH.

A machine running an applet that makes Domino calls needs no Domino software or CLASSPATH assignments.

10.4 Domino Object Classes

In Domino R4.6, a set of Java classes that accessed the Notes Object Interface (NOI) was introduced into the Notes client and Domino server. Based on the NOI classes, Domino R5 contains an enhanced set of Java classes to support the Domino object classes.

Domino agents as well as Java applications whether at the client or the server can be written in Java and utilize these Java Notes classes to access Domino information.

These are the same backend objects accessible through LotusScript and OLE. In Java, the Session class is the root class of the Java package. The Domino object classes do not include the front-end classes rooted at NotesUIWorkstation. Figure 35 on page 107 shows the Java classes to access backend objects in Domino R4.6.

In Domino R5 only, remote Java programs can access the Domino object classes on the Domino server using CORBA/IIOP network communication mechanisms.

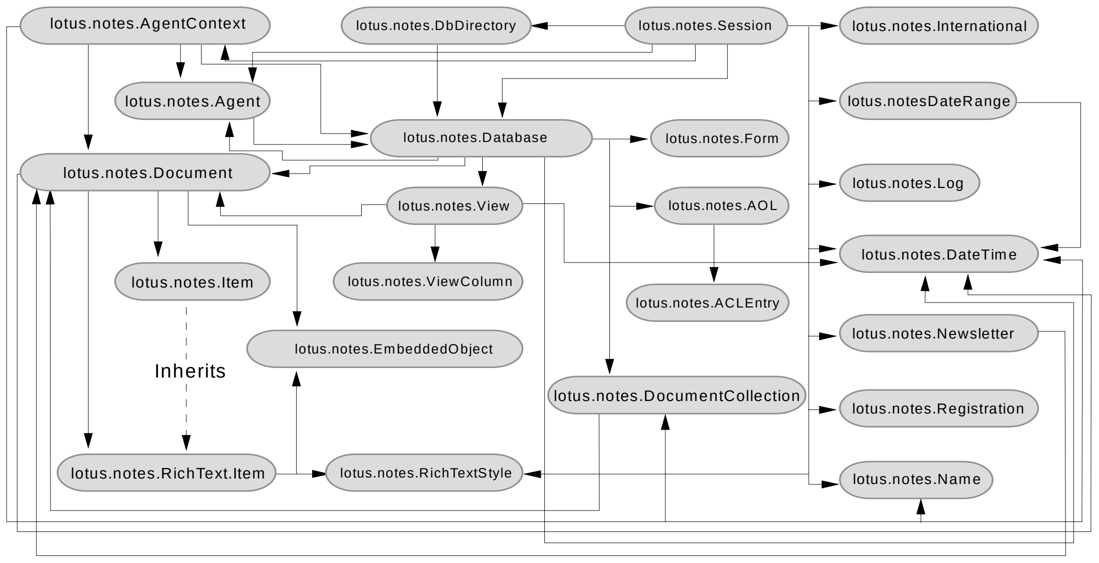


Figure 35. Java Classes for Domino R4.6

If you are using Domino R5, you also can implement CORBA applications. We give the description of the CORBA implementation in Chapter 21, “Domino with CORBA” on page 227.

The Domino Java classes are provided in the following packages:

- Domino R4.6 notes.jar file contains all R4.6 lotus.notes classes.
- Domino R5 notes.jar file contains:
  - R4.6 *lotus.notes* classes. This package allows Domino R4.6 agents and applications to run unchanged.
  - R5 *lotus.domino* classes. This package improves the functions of the existing classes and provides new classes.
- Domino R5 NCSO.jar file contains the CORBA implementation of the *lotus.domino* classes. It is used by remote Java applications and applets only.

You have to import the correct package in your program. The notes.jar archive file containing the package must be in the CLASSPATH of your machine. The CLASSPATH environment variable should point to the Domino Java classes archive file as in the following example:

```
CLASSPATH=<any other path/archive>;c:\notes\notes.jar
```

With Domino R4.6 and for local Domino R5 access, to run a Java program that uses the Notes classes, the Domino server or the Notes client must be installed on the machine for both compilation and execution. The PATH environment variable must include the Notes directory. For example:

```
set PATH=c:\notes;%path%
```

Domino Java classes can be used in applications and agents:

- Java applications can import the lotus.notes package and use the backend classes provided that Domino/Notes is installed on the machine both for compilation and running.
- Domino/Notes agents can be simple actions, formulas, LotusScript programs, and Java programs. In Domino R4.6, Java programs must be written outside of Notes and imported because the R4.6 IDE does not support Java.

In R4.6, the Java classes cannot be used in applets for reasons of accessibility and security. The lotus.notes package makes calls into Domino/Notes DLLs. A browser could not load the DLLs (security restriction) even if they happened to be on the browser's machine. If you want to use the Java classes in applets, then Domino R5 should be used.

Java programs cannot be attached to front-end objects. For example, a Java program cannot be a form event in the manner of a LotusScript program.

10.5 Domino Driver for JDBC

The Domino driver for JDBC is a Type2 JDBC driver for accessing Domino databases; that is, it converts JDBC calls into Domino calls using Domino API.

The Domino driver for JDBC is available on the following Intel platforms:

- Microsoft Windows 95 and 98
- Microsoft Windows NT 4.0 or higher

To install the Domino driver for JDBC, you first need to install either Domino Designer or Lotus Notes. Notes database files can reside on a server. You do not need to have local copies of these files, but must have at least reader access to them through Notes.

Domino driver for JDBC also requires the Java Virtual Machine 1.1.

After the installation, you need to check the setting of the CLASSPATH environment variable. It should point to the Domino driver for JDBC classes file (*JdbcDomino.jar*) as in the following example:

```
CLASSPATH=c:\notes\JdbcSql\lib\JdbcDomino.jar;
```

The native code needed by the Domino driver for JDBC is contained in three DLLs: *JdbcDomino.dll, JdbcDriver.dll* and *JdbcRniDomino.dll*.

You can download the Domino driver for JDBC from the Lotus developer Web site at:

`http://www.lotus-developer.com`

10.6 Java Classes for Lotus Connectors

The Java classes for Lotus Connectors (LC Java classes) extend use of the Lotus Connectors to Java. The programming model is independent of individual connectors. This eliminates the need to learn each system, but allows experienced users to access specific system features.

Through the LC Java classes, Java agents, applications, and applets can retrieve and act upon data from enterprise systems.

Software Requirements

Obtain the LC Java software and documentation from `www.eicentral.lotus.com` the Lotus Enterprise Integration Web site. It also comes bundled with LEI.

Make the following adjustments to the CLASSPATH and PATH environment variables:

- CLASSPATH

  Add the file lcjava.zip. This file is typically installed in your Domino, LEI Java, or program directory.

- PATH

  Add the directory to which LC Java was installed. This is typically your Domino or LEI program directory.

For example:

```
set CLASSPATH=%CLASSPATH%;c:\notes\domino\java\lcjava.zip
set PATH=%PATH%;c:\notes
```

If you are developing Java agents, create a Domino environment variable named JavaUserClasses and add the file lcjava.zip. For example:

```
JavaUserClasses=c:\notes\domino\java\lcjava.zip
```

Put the following import statement in your program:

```
import lotus.lcjava.*;
```

Chapter 11. Enterprise Resources

In our project we used three enterprise systems that our examples connect to:

- DB2 Universal Database Version 5.2
- CICS OS/2
- MQSeries Version 5

11.1 DB2

In this section we present the installation and configuration required for DB2 Universal Database Version 5.2 for Windows NT (DB2 UDB).

The Domino application can access DB2 data either locally on the DB2 server or remotely using DB2 CAE or DB2 Connect.

- Local access

  The Domino server (or in specific cases, the Lotus Notes client) accesses the DB2 database using the DB2 server installed on the same machine. In this configuration, the operating system must support both Domino (or Lotus Notes) and the DB2 server, for example, OS/2, Windows NT, AIX, HP-UX, OS/400, and OS/390. No network protocol configuration is needed.

  If you are developing and running applets that use the DB2 JDBC Applet driver to connect to DB2, you have to install DB2 UDB on the same machine. In that case, the applet connects to the DB2 JDBC applet server which runs on the same machine from which the applet was downloaded.

- Using DB2 CAE

  The Domino server (or the Lotus Notes client) accesses the DB2 server running DB2 UDB on an Intel or UNIX platform. In this configuration you must install DB2 CAE on the Domino server (or the Lotus Notes client) and configure communication to the DB2 server, using a compatible protocol.

- Using DB2 Connect

  The Domino server (or the Lotus Notes client) accesses the DB2 server on AS/400 or S/390 using DRDA. Two configurations are possible:

  - With DB2 Connect Personal Edition installed on the Domino server (or the Lotus Notes client), you can access the DB2 database. You have to configure communication to the DB2 server using a compatible protocol.

- With DB2 Connect Enterprise Edition installed on a gateway, you can access the DB2 database. You must install DB2 CAE on the Domino server (or the Lotus Notes client) and configure communication to the gateway using a compatible protocol. On the gateway, you also have to configure communication to the DB2 server using a compatible protocol.

JDBC Support

To write DB2 JDBC applets or agents for use in a Domino application, you have to install DB2 (server or client) on the same machine as your Domino server.

DB2 UDB Version 5 includes support for the JDBC API, as distributed with JDK 1.1. JDBC is available as a no-charge feature to all DB2 for OS/390 Version 5 customers. The AS/400 Toolbox for Java, a library of Java classes that give Java programs easy access to AS/400 data and resources, includes a JDBC driver to access DB2/400 databases.

With DB2 UDB Version 5, you can use DB2 JDBC support to run the following types of Java programs:

- Java applications, which rely on DB2 CAE to connect to DB2
- Java applets, which do not require any other DB2 component code on the client

Java can also be used on the server to write user-defined functions, stored procedures, and table functions.

PATH and CLASSPATH

After installing DB2, the PATH environment variable is updated to reflect the DB2 directory, such as:

```
PATH=<other paths>;c:\sqllib\bin ;
```

To be able to develop Java programs that use the DB2 Java classes, you have to update the CLASSPATH environment variable to include the jar files supplied by DB2.

```
CLASSPATH=<other paths>;.;c:\sqllib\java\db2java.zip
```

DB2 Java Application Support

To test a Java application, you can just start it from the desktop or command line, like any other application. The DB2 JDBC driver handles the JDBC API calls from your application and uses DB2 CAE to communicate the requests to the server and receive the results.

DB2 Java Applet Support

Because Java applets are delivered over the Web, you treat them a bit differently from Java applications.

To run your applet, you need a Java-enabled Web browser on the client machine. When you load your HTML page, the applet tag downloads the Java applet to your machine, which then downloads the Java class files, including the com.ibm.db2.java.sql and com.ibm.db2.jdbc.net classes and DB2's JDBC driver. When your applet calls the JDBC API to connect to DB2, the JDBC driver establishes separate communications with the DB2 database through the JDBC applet server residing on the DB2 server.

To run your applets, follow these steps:

1. Start the DB2 JDBC applet server on your Web server by entering:

   ```
   db2jstrt portno
   ```

   where *portno* is the number of the unused TCP/IP port that you specified in the applet file.

2. On your client system, start your Web browser and load the HTML file that embeds your applet.

11.2 CICS Transaction Gateway

In this section we present the installation, configuration required for the CICS Transaction Gateway for Windows NT.

More complex scenarios for installing and configuring the CICS Transaction Gateway and CICS Universal Client are described in the redbook, *Revealed! CICS Transaction Gateway with More CICS Clients Unmasked*, SG24-5277.

JDK

A prerequisite for running the CICS Transaction Gateway is a supported level of JDK. On the Windows NT platform, the CICS Transaction Gateway requires JDK 1.1.6 or later, with the JIT update. Check that your version of the JDK is supported by the CICS Transaction Gateway.

PATH and CLASSPATH

After installation, the PATH environment variable is updated to reflect the CICS Transaction Gateway directory, such as:

```
PATH=<other paths>;c:\IBM\CTG\BIN ;
```

To be able to develop Java programs that use the CICS Transaction Gateway classes, you have to update the CLASSPATH environment variable to include the jar files supplied by the CICS Transaction Gateway. In addition to the *ctgclient.jar* file, the *ctgserver.jar* file is required on the CLASSPATH environment variable if you want to use the local CICS Transaction Gateway during your testing.

Servlets also use the local CICS Transaction Gateway. Therefore, you must specify the *ctgserver.jar* file on the CLASSPATH used by the Web server:

```
CLASSPATH=<other paths>;
         c:\IBM\CTG\classes\ctgclient.jar;
         c:\IBM\CTG\classes\ctgserver.jar
```

Web Server

CICS Transaction Gateway needs the support of a Web server such as Domino, Lotus Domino Go Webserver, or Microsoft IIS.

If you are going to run Java servlets, you will need a Web server or a servlet engine that provides servlet support equivalent to JSDK Version 1.1 or later, such as:

- Lotus Domino R4.6 or R5
- IBM WebSphere Version 1.0 or later

CICS Universal Client V3

CICS Transaction Gateway incorporates CICS Universal Clients Version 3. On Windows NT, the CICS Universal Client can communicate with CICS servers using NetBIOS, TCP/IP, APPC, TCP62 (APPC communication over a TCP/IP network).

CICS Initialization File

The client initialization file contains configuration information used to inform the CICS Transaction Gateway of the servers it can connect to and the necessary communication protocols. Here is the *cicscli.ini* file we used in our environment to connect our Domino server to the CICS server:

```
;* IBM CICS Universal Client - Initialization File *
; Client section
Client = *
   MaxServers = 1
   MaxRequests = 256
   MaxBufferSize = 32
   LogFile = CICSCLI.LOG
   TraceFile = CICSCLI.TRC
   DumpFile = CICSCLI.DMP
   DumpMemSize = 16
; Server section
```

```
Server = CICSOS2
   Description = TCP/IP Server
   Protocol = TCPIP
   NetName = samoa.almaden.ibm.com
   Port = 1435
   UpperCaseSecurity = N

; Driver section
Driver = TCPIP
   DriverName = CCLWNTIP
```

11.3 MQSeries

In this section we present the installation and configuration required for the MQSeries Client for Java and the MQSeries Bindings for Java on the WIndows NT platform.

11.3.1 Installation

We installed both MQSeries Client for Java and MQSeries Bindings for Java.

Prerequisites

Both products require JDK 1.1.1 or higher installed on the machine.

To run MQSeries Client for Java applets (for example the installation verification program) inside a Web browser, you need a browser that can run Java 1.1.1 applets.

PATH and CLASSPATH

After the installation, we set the CLASSPATH environment variable to the following values:

- For MQSeries Client for Java:

  ```
  CLASSPATH=<paths>;c:\mqm\java\lib;c:\mqm\tools\javaclnt\samples\En_US;
  ```

- For MQSeries Bindings for Java:

  ```
  CLASSPATH=<paths>;c:\mqm\java\lib;c:\mqm\tools\mqbind\samples\En_US;
  ```

Verification

To verify our MQSeries Client for Java installation, we used the sample Java applet (*mqjavac.html*), included with the MQSeries Client for Java.

To verify our MQSeries Bindings for Java installation, we used the verification program (*MQIVP*) provided with the MQSeries Bindings for Java.

Web Server Configuration

If you install the MQSeries Client for Java on the Domino server you can download and run MQSeries Client applications on machines which do not have the MQSeries Client for Java installed locally.

To make the MQSeries Client for Java files accessible to Domino, you need to set up Domino configuration to point to the directory where the client is installed.

MQSeries Bindings for Java and WebSphere

If you plan to use the MQSeries Bindings for Java with WebSphere on Windows NT and you are running MQSeries for Windows NT V5.0, a useful hint for better performance is to ensure that you have PTF U200091 applied to MQSeries.

The particular APAR you require is IC20925. This APAR adds function to improve the performance of MQSeries for Windows NT V5.0 in a Windows NT domain environment, particularly when MQSeries is started as a service via scmmqm.

WebSphere on NT has a useful management console, which only works if the servlet service is set to run as the system account in the services control panel. The system account cannot be added into the mqm group as other users would be, but this APAR helps MQ find a definition for a user ID of SYSTEM on the local machine.

To obtain the fix go to:

```
http://www.software.ibm.com/ts/mqseries/support/summary/wnt.html
```

Chapter 12. WebSphere

In this chapter we explain the installation and setup of the WebSphere environment.

12.1 Installation

WebSphere complements Domino in two areas:

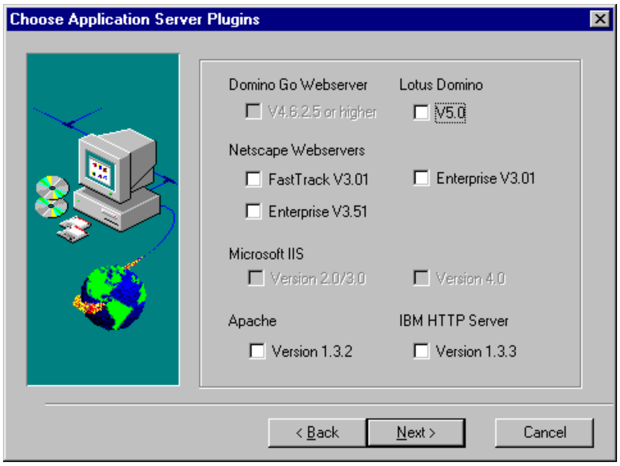
- Servlet manager
- Enterprise JavaBeans server

If you want to replace the Domino servlet manager with the WebSphere servlet manager, you have to install WebSphere on the same machine as the Domino server. In that case, WebSphere uses the Domino HTTP server as its Web server.

If you want to use the EJB support of WebSphere only, then you can install WebSphere on another machine. In that case, the machine running WebSphere will also need an HTTP server that can be one of the following:

- Lotus Domino R5
- Lotus Domino Go Webserver
- Netscape Enterprise Server
- Netscape FastTrack Server
- Microsoft Internet Information Server
- Apache Server

We installed WebSphere on the same machine as Domino, because we wanted to test EJB and servlet support. During the installation of WebSphere we selected Domino R5 as our Web server (see Figure 36 on page 118).


Choose Application Server Plugins

Domino Go Webserver
- [ ] V4.6.2.5 or higher

Lotus Domino
- [ ] V5.0

Netscape Webservers
- [ ] FastTrack V3.01
- [ ] Enterprise V3.01
- [ ] Enterprise V3.51

Microsoft IIS
- [ ] Version 2.0/3.0
- [ ] Version 4.0

Apache
- [ ] Version 1.3.2

IBM HTTP Server
- [ ] Version 1.3.3

Figure 36. WebSphere Application Server Plugins Selection Panel

12.2 Application Server Manager

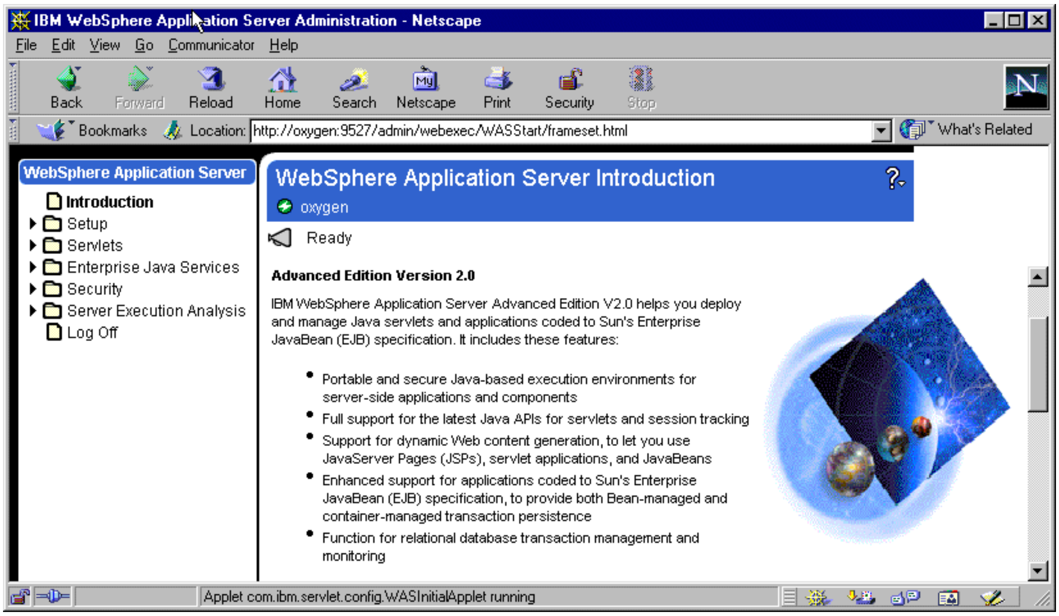
After the WebSphere installation has completed, you have to configure it in order to deploy the EJB or to use its servlet manager. To configure WebSphere, you use the Application Server Manager.

To start the Application Server Manager, enter this URL from your Web browser:

```
http://your.server.name:9527/
```

The manager starts and displays the login page. We use *admin* as the login user ID and password.

Figure 37 on page 119 shows the main menu of the Application Server Manager.


IBM WebSphere Application Server Administration - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://oxygen:9527/admin/webexec/WASStart/frameset.html
What's Related
WebSphere Application Server
Introduction
Setup
Servlets
Enterprise Java Services
Security
Server Execution Analysis
Log Off
WebSphere Application Server Introduction
oxygen
Ready
Applet com.ibm.servlet.config.WASInitialApplet running

Advanced Edition Version 2.0

IBM WebSphere Application Server Advanced Edition V2.0 helps you deploy and manage Java servlets and applications coded to Sun's Enterprise JavaBean (EJB) specification. It includes these features:

- Portable and secure Java-based execution environments for server-side applications and components
- Full support for the latest Java APIs for servlets and session tracking
- Support for dynamic Web content generation, to let you use JavaServer Pages (JSPs), servlet applications, and JavaBeans
- Enhanced support for applications coded to Sun's Enterprise JavaBean (EJB) specification, to provide both Bean-managed and container-managed transaction persistence
- Function for relational database transaction management and monitoring

Figure 37. WebSphere Application Server Administration

The navigation area on the left-hand side of the Application Server Manager enables you to:

- Customize settings for a variety of Application Server components such as:
  - Administrator user ID and password
  - Connection pools to the data server
  - Directory server to Application Server, connecting Application Server security and your existing directory server security
  - Java compiler settings
- Configure servlets and set up aliasing and filtering to:
  - Define configuration information and initialization parameters for individual servlets
  - Specify servlet aliases
- Use the Enterprise Java Services (EJS) to:
  - Enable EJB support
  - Define containers in which to deploy your EJB's JAR files
  - Deploy JAR files holding EJBs
- Establish and maintain security by defining users, groups, resources, and access control lists

- Collect and monitor Application Server, connection, and servlet data

We first verify that the basic JVM settings are correctly initialized. In particular, the WAS CLASSPATH should point to the Domino and DB2 jar files as we are not using the system CLASSPATH. Select **Setup - Java Engine** and verify that the Application Server CLASSPATH has the following setting:

```
c:\JDK11~1.7B\lib\classes.zip;c:\WEBSPH~1\APPSER~1\classes;c:\WEBSPH~1\APP
SER~1\web\classes;c:\Lotus\Domino\notes.jar;c:\sqllib\java\db2java.zip
```

12.2.1 Enterprise JavaBeans

In our test, we used the *Employee* Enterprise JavaBean, a sample EJB provided by WAS. In the following we explain how we deploy and test the bean using WAS before accessing it from a Domino agent or servlet.

Configuring DB2

The *Employee* EJB sample supplied uses the SAMPLE database provided by DB2 UDB V5.2. You have to configure DB2 by following these steps:

- Start DB2 server.

  To start a DB2 server, open a DB2 command window from the DB2 menu and enter the commands:

  ```
  db2start
  db2admin start
  ```

  Alternatively, open the Services window from your Control panel and start the DB2 - DB2 and DB2 - DB2DAS00 processes.

- Create the SAMPLE database.

  To create the SAMPLE database, open the DB2 menu and select First Steps to create the SAMPLE database. It is important to create this particular database this way to ensure that it has the entries that are used by the Phone sample.

  Alternatively, you can create the SAMPLE database using the following command in a DB2 command window:

  ```
  db2sampl
  ```

- Create an alias for the employee table.

  When you select the First Steps application or run the *db2sampl* command, DB2 creates sample tables using a schema of the ID of the user connected on the Windows NT workstation. You may have to create an alias to use db2admin required by the Employee bean.

To check the schema, open a DB2 command line processor window from the DB2 menu and enter the following commands:

```
db2 => connect to sample

   Database Connection Information
   Database server        = DB2/NT 5.2.0
   SQL authorization ID   = CHRISTO
   Local database alias   = SAMPLE

db2 => list tables

Table/View    Schema    Type   Creation time
------------- --------- ------ -----------------------
DEPARTMENT    CHRISTO   T      1999-03-18-18.42.30.136001
EMPLOYEE      CHRISTO   T      1999-03-18-18.42.30.807001
STAFF         CHRISTO   T      1999-03-18-18.42.29.415001
......
```

You create an alias using the following command:

```
db2 => CREATE ALIAS DB2ADMIN.EMPLOYEE FOR CHRISTO.EMPLOYEE
DB20000I The SQL command completed successfully
```

Getting Started with Application Server

Check that DB2—the DB2 - DB2 and DB2 - DB2DAS00 processes— is running, using the Services window from your Control panel.

Start your Domino Web server. If WAS has been installed with Lotus Domino R5.0 as Application Server plug-in (see 12.1, “Installation” on page 117), then WebSphere is started automatically. The Location Server Daemon, the Persistent Name Service and the EJS runtime are started in addition to your Web server.

Now you are ready to start the Application Server Manager by going to the URL:

```
http://your.server.name:9527/
```

and logging in using *admin* for user and password.

There are two changes that you should check to verify your configuration:

1. Check that the *db2java.zip* file has been added to your server's CLASSPATH.
   - From the Application Server Manager, select **Setup** - **Java Engine**.

- Select the Paths tab and add the following to your Application Server CLASSPATH: c:\sqllib\java\db2java.zip or the equivalent if it is not already there.

2. Check that the Database Container Authentication is correct:
    - From the Application Server Manager initial screen, select **Enterprise Java Services - Containers**.
    - For the *defaultEntityContainer* check that the Container User ID and password are correct for your database. By default DB2 UDB is set up with *db2admin* for user ID and password.

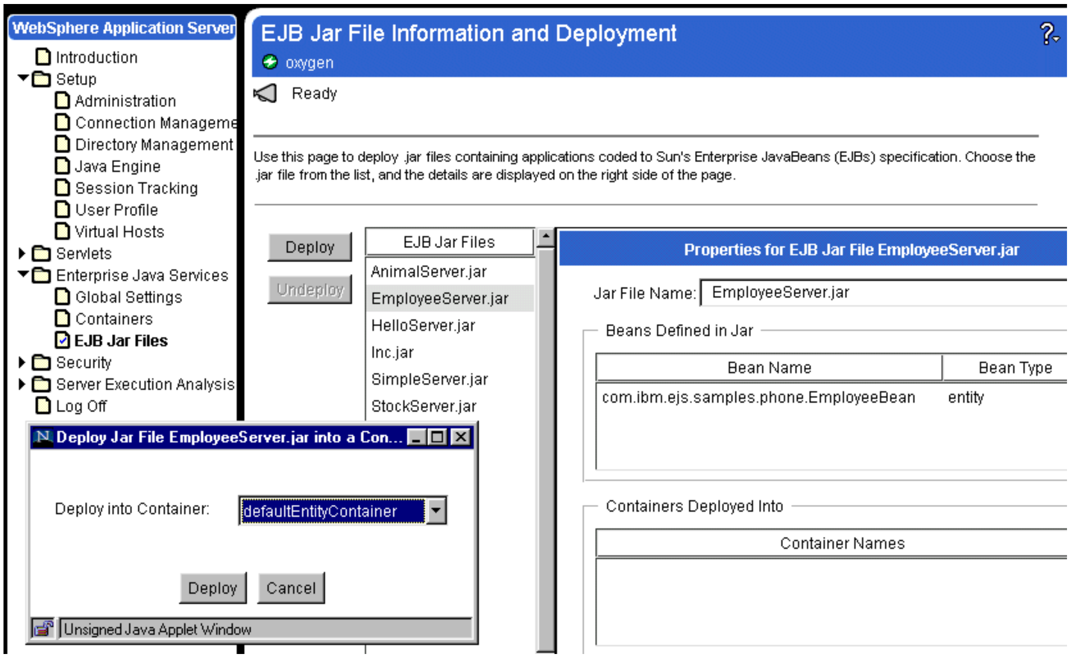
Managing Containers

WebSphere Application Server is installed with two containers. From Application Server Manager select **Enterprise Java Services - Containers**. This shows two containers that are already defined and also the contents of each container. The *Employee* file is deployed in the defaultEntityContainer. The second container is called defaultSessionContainer and is empty. If you are not going to use this container you should remove it. In general you should have no empty containers as this may cause problems.

Deploying EJBs

From the menu select **Enterprise Java Services - EJB JAR files**. The samples are ready to be deployed (AnimalServer, EmployeeServer, HelloServer, SimpleServer, StockServer, and Inc.). If an EJB is already deployed, the name of the container is shown in the bottom panel.

Figure 38 on page 123 shows how to deploy the *EmployeeServer* JAR file. You have to highlight the JAR file, select **Deploy,** and from the Deploy Jar File into a Container panel choose the relevant container. The *EmployeeBean* is an entity bean and has to be deployed in the *defaultEntityContainer*. Select Deploy.



EJB Jar File Information and Deployment

oxygen

Ready

Use this page to deploy .jar files containing applications coded to Sun's Enterprise JavaBeans (EJBs) specification. Choose the .jar file from the list, and the details are displayed on the right side of the page.

Properties for EJB Jar File EmployeeServer.jar

Jar File Name: EmployeeServer.jar

Beans Defined in Jar

Bean Name	Bean Type
com.ibm.ejs.samples.phone.EmployeeBean	entity

Containers Deployed Into

Container Names

Deploy Jar File EmployeeServer.jar into a Con...

Deploy into Container: defaultEntityContainer

Figure 38. Deploying an EJB

If the EJB is already deployed, the dialog box shows three options:

Regenerate	Is used for JAR files that have never been deployed; it creates the stubs, skeletons, and database tables necessary in a deployed EJB.
Redeploy Existing	Is used for any predeployed samples; it uses the existing stubs and skeletons contained in the JAR file and creates the necessary database tables.
Cancel	Allows you to revert to the previous screen.

To deploy the other samples, highlight the JAR file, select a container for your file, and press the Deploy button. To help you decide which type of container to deploy your EJB in, the type of bean, session or entity, is listed on the right side of the panel next to each bean. If it has only one type of bean, deploy that bean into the relevant container. If your JAR file has both entity beans and session beans you must deploy it into both.

You have to stop and restart your Domino server for this to take effect.

Running the Client

WAS provides a simple Web-based phone book application based on the existing DB2 table of employee information in the SAMPLE database. The phone book sample is based on the Employee EJB. It illustrates the use of container-managed persistence entity beans and how they can be mapped onto existing DB2 tables by modifying the code generated by the stand-alone deployment tool. It also shows how entity beans can be accessed directly by a servlet and JSP.

The phone book sample is based on the Employee Enterprise JavaBean, a container-managed persistence entity bean that is mapped onto the EMPLOYEE table included as part of the SAMPLE database distributed with the DB2 product. The EMPLOYEE table contains over a dozen different columns, of which only six are used by the Employee bean. The phone book sample shows how the SQL statements in the persister class generated by the stand-alone deployment tool can be modified in a straightforward manner to accommodate the actual data contained in an existing DB2 table.

The phone book sample also shows how entity beans can be accessed through a client application written using the servlet and JSP capabilities of WAS, allowing user access to the application through a Web browser without requiring any Java code or infrastructure on the user's system.

To run the phone book sample application you can enter the following URL from your browser:

```
http://your.server.name/servlet/PhoneBook
```

The Web page that appears allows you to run any of the Web-based Application Server samples:

- To run the phone example, select the **Find a Phone number** link.
- Next, a Web page appears containing a form prompting you to enter the last name of the person you wish to look up. You can enter either a complete last name or just the first few letters of the name. The case of the letters entered is not important. Some examples of names contained in the sample DB2 EMPLOYEE table are "Brown", "Smith" and "Lee".
- Once you have entered the name, select the button next to the Entry field. After a delay while the matching Employee beans are retrieved, the Web page will be redisplayed with a table of the names and phone numbers of the employees whose last name matches the name you entered.
- If you wish, you can enter a different last name and click the form button to request a new name lookup. You can leave the phone book application at any time by selecting a different URL or closing your Web browser.

12.2.2 Servlet Manager

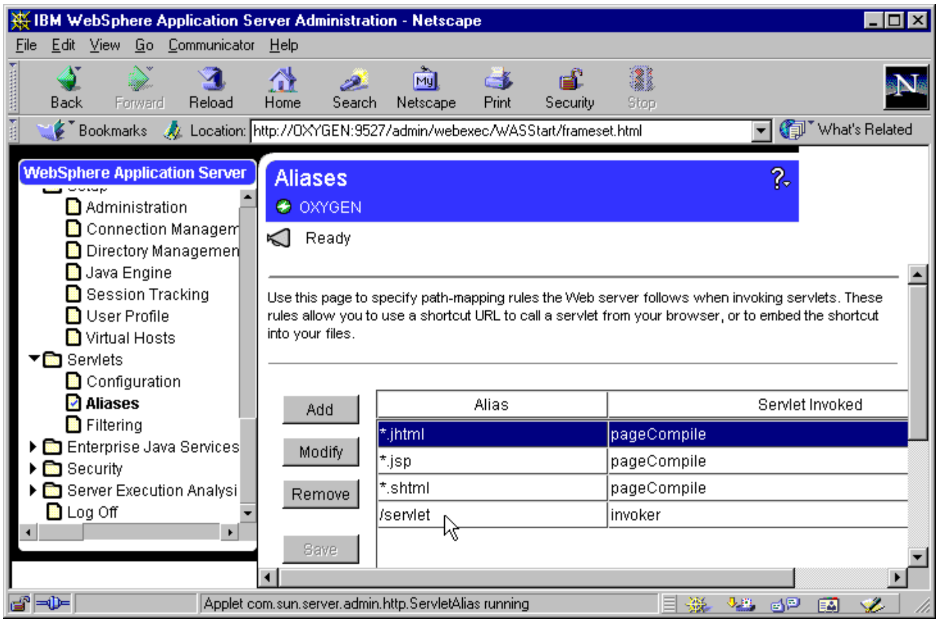
Domino offers the ability to use a third-party's servlet manager instead of the native Domino R5 servlet manager. This is useful if your third-party servlet manager offers you more functionality than the one integral to Domino R5.

We used WAS as the third-party servlet manager. In our tests we used build 165 of Domino R5 along with WAS V2.01. Both of these were installed on the same Microsoft NT server. Selecting this option during the WebSphere installation modifies the *http.cnf* file of the Domino server adding a service handler for URLs.

```
################################################################################
Service       /*.jhtml     e:\WebSphere\AppServer\plugins\nt\go46.dll:service_exit
Service       /*.shtml     e:\WebSphere\AppServer\plugins\nt\go46.dll:service_exit
Service       /*.jsp       e:\WebSphere\AppServer\plugins\nt\go46.dll:service_exit
Service       /servlet/*   e:\WebSphere\AppServer\plugins\nt\go46.dll:service_exit


ServerInit  e:\WebSphere\AppServer\plugins\nt\go46.dll:init_exit
            e:\WebSphere\AppServer\properties\bootstrap.properties
ServerTerm  e:\WebSphere\AppServer\plugins\nt\go46.dll:term_exit
Pass            /IBMWebAS/samples/*          e:\WebSphere\AppServer\samples\*
Pass            /IBMWebAS/*                  e:\WebSphere\AppServer\web\*
```

By default WAS is started automatically when the HTTP server is started. However, you can stop and start the WebSphere servlet service using the Service icon of the control panel.

As shown in Figure 39 on page 126, we used the WAS Administration application to check the invoker parameter under the Servlet Aliases section. We checked that the parameter matched the service */servlet* defined in the httpd.cnf file of the Domino server.


IBM WebSphere Application Server Administration - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://OXYGEN:9527/admin/webexec/WASStart/frameset.html
What's Related
WebSphere Application Server
Administration
Connection Managem
Directory Managemen
Java Engine
Session Tracking
User Profile
Virtual Hosts
Servlets
Configuration
Aliases
Filtering
Enterprise Java Services
Security
Server Execution Analysi
Log Off
Aliases
OXYGEN
Ready
Add
Modify
Remove
Save

Use this page to specify path-mapping rules the Web server follows when invoking servlets. These rules allow you to use a shortcut URL to call a servlet from your browser, or to embed the shortcut into your files.

Alias	Servlet Invoked
*.jhtml	pageCompile
*.jsp	pageCompile
*.shtml	pageCompile
/servlet	invoker

Applet com.sun.server.admin.http.ServletAlias running

Figure 39. WebSphere Administration - Servlet Aliases

Once completed, WAS is ready to run your servlets. If you installed WAS on the C: drive, the default servlet root directory is:

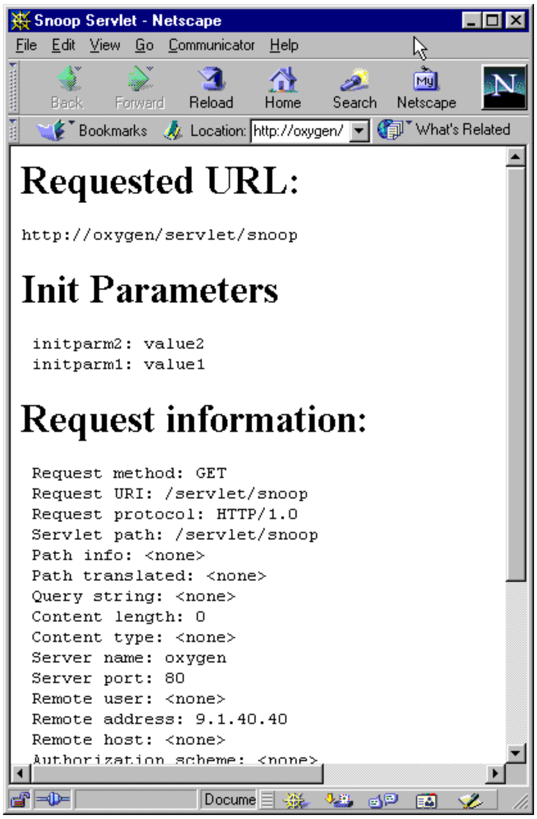
```
c:\WebSphere\AppServer\servlets.
```

In this directory you should place the .class or .jar files that make up your servlets. Remember to create subdirectories beneath this directory to mirror the package hierarchy your servlets are in.

To test the installation, WAS provides the SnoopServlet servlet. In your Web browser, enter the following URL:

```
http://your.server.name/servlet/snoop
```

Figure 40 on page 127 shows an example of the output of the SnoopServlet servlet.


Snoop Servlet - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape
Bookmarks Location: http://oxygen/ What's Related
Docume

Requested URL:

```
http://oxygen/servlet/snoop
```

Init Parameters

```
 initparm2: value2
 initparm1: value1
```

Request information:

```
 Request method: GET
 Request URI: /servlet/snoop
 Request protocol: HTTP/1.0
 Servlet path: /servlet/snoop
 Path info: <none>
 Path translated: <none>
 Query string: <none>
 Content length: 0
 Content type: <none>
 Server name: oxygen
 Server port: 80
 Remote user: <none>
 Remote address: 9.1.40.40
 Remote host: <none>
 Authorization scheme: <none>
```

Figure 40. Checking WAS Servlet Manager

Part 3. Connecting Domino

Lotus first provided Java support in Domino R4.6. Domino Java classes parallel the LotusScript backend classes. You can use these classes from any Java program, within the Domino Designer environment or outside of it, as long as Domino R4.6 or later is installed on the machine.

Java programs can take various forms. Our discussions in Part 3 will focus on:

- Applets

  Java programs designed to execute as part of a Web page. The possibilities for using an applet with Domino to integrate to the enterprise are very limited (limited to Domino R5 with CORBA) and no example code is provided.

- Servlets

  Java programs designed to execute on a Web server. We use an example that calls a CICS transaction from a Web browser, with Domino as our Web server.

- Applications

  Stand-alone, Web-independent Java programs. We use an example that connects to a DB2 database, independent from Domino, and then uses the DB2 data to update a Domino application.

- Agents

  These are design elements of a Domino application that can execute Java code. In our example we connect to an MQSeries application from a Notes client or a Web browser.

You will notice that we give only one example of each Java program type and that each talks to a different enterprise resource. It would be wrong of us to imply that the enterprise resource being targeted was irrelevant; the method you use to set up the connection to the enterprise resource and the way that information is passed between Domino and the enterprise resource will differ between DBMSs, OLTP systems, and messaging systems. However, the details of connecting to the enterprise resource will not be affected by whether you choose to use a servlet rather than a Java agent.

Our selection of an enterprise resource in our Java program examples does not represent a recommendation for that enterprise type. However, we do give some comparisons and recommendations for Java program types in

Chapter 17, “A Comparison” on page 189. The source code for our examples is given in the appendixes.

A factor that will affect the connection to the enterprise resource is your choice of tool. The tools available to Java programs include:

- For database access:
  - Domino Java classes (where Domino is the data store)
  - JDBC
  - Lotus Connectors using the Lotus Connector Java class library, for example:
    - Lotus Connector for DB2
    - Lotus Connector for EDA/SQL
    - Lotus Connector for ODBC
    - Lotus Connector for Oracle
    - Lotus Connector for Sybase
- For OLTP access:
  - CICS Transaction Gateway
  - IMS Client for Java
  - Lotus Connectors using the Lotus Connector Java class library, for example:
    - Lotus Connector for CICS
    - Lotus Connector for IMS
    - Lotus Connector for BEA Tuxedo
- For application access through messaging middleware:
  - MQSeries Client for Java
  - MQSeries Bindings for Java
  - Lotus Connectors using the Lotus Connector Java class library:
    - Lotus Connector for MQSeries
- For access to enterprise resource planning (ERP) packages:
  - Lotus Connectors using the Lotus Connector Java class library, for example:
    - Lotus Connector for J.D. Edwards
    - Lotus Connector for Oracle Financials
    - Lotus Connector for SAP
    - Lotus Connector for PeopleSoft

For information about the tools listed here, refer to the following URLs:

```
http://java.sun.com/products/jdbc/
http://www.software.ibm.com/webservers/connectors/
http://www.eicentral.lotus.com/eibu_knowbase.nsf/
```

In Part 3, we give examples of using a few of these tools while discussing the standard Java support provided within Domino R4.6 and R5. We use the term standard, in order to distinguish between the support provided by Domino itself and those enhancements that can be gained with the addition of WebSphere or CORBA (see Part 4, “Domino and WebSphere” on page 195).

Java programs can perform the same tasks as LotusScript programs. For those of you who are used to developing applications in LotusScript, limited comparisons between Java and LotusScript can be found in Appendix A, “Use of Java versus LotusScript” on page 215.

Chapter 13. Applets

Applets are small Java programs that run within a Web browser client. Since Java is a full-featured programming language, the inclusion of Java applets on a Web page can provide very rich and sophisticated Web-based applications to the Web-user.

13.1 Domino Applet Support

Any Domino server above Version 4.5 will host Java applets, but there are some important distinctions between what is available to you in Domino 4.6 and Release 5.

Table 3 on page 133 summarizes the applet support in Domino.

Table 3. Domino Applet Support

Domino Server	Client Type	Applet Support?	Applet access to Domino server object store?
Domino R4.6	Web browser	Yes	Not directly, URL access only
	Lotus Notes Client Release 4.6x	Yes	Not directly, URL access only
Domino R5	Web browser	Yes	CORBA
	Lotus Notes Client Release 5	Yes	Native local access to Domino objects

With Domino 4.5 and above, applets running in a remote client, be that a Web browser or Notes client, have limited access to the services offered by a Domino server. Services such as creating documents, sending e-mail, and searching databases are not available to the applet. One workaround to this that offers some degree of access to the power of Domino servers, is to have your applet make URL queries against a Domino server (through the java.net.URL class). The returned HTML can then be processed and used within your applet. This approach whilst possible, is cumbersome.

For true interaction with a Domino Server from a remote Java applet, Domino Release 5 is the greatly preferred solution. Release 5 supports CORBA, the remote object technology. CORBA allows remote applets to make calls to remote server objects as if they were *there* on the Web browser workstation.

The ability to fully access the Domino server object model from a Web browser with Release 5, opens up many possibilities for Domino-based Web applications.

For more information on the CORBA implementation in Domino Release 5, refer to 4.5.2, “CORBA Support” on page 46.

13.2 Structure of an Applet

In this section we explain the structure of a Domino Java applet. A simple applet has security restrictions. It cannot write or read the local file system (that includes loading dynamic link libraries from the local file system), or run any program on the local machine. An applet can neither communicate with, nor access, any server other than the one on which, and from which, it was originally stored and loaded.

We also introduce how to create an applet that can access the DOM of Domino R5. This applet can perform Domino tasks, such as opening a session, retrieving information from a database and accessing Domino documents.

13.2.1 Domino R4.6 Java Applet

All Java applets must derive from, or be a subclass of, a class called *Applet*. This class is located in a package called *java.applet*. This class provides the appropriate structure that all Java applets must follow. And specifically, it has unique methods in it that specify an interface that the Web browser assumes to be there.

The key methods that you must implement or override are:

init() method
The init() method is executed to set up the graphical user interface. This method is called when the applet is first loaded or reloaded. The initialization might include reading and parsing any parameters to the applet, setting up an initial state, or loading images or fonts.

start() method
The *start()* method is executed every time the Web page is shown. This method can be called many times during the life-cycle of the applet and also every time the applet is stopped.

stop() method
The *stop()* method is executed when the Web page is no longer shown.

destroy() method
The *destroy()* method is executed when the applet is no longer needed and can be discarded. This method enables the applet to clean up after itself just before it is freed or the browser exits.

13.2.2 Domino R5 Applet

In Domino R5 you can write an applet that uses CORBA to access Domino classes, thereby allowing it to access named databases, views, documents, and other Domino backend objects.

The Domino server hosts the applet and downloads the applet to the browser when requested.

To develop an applet that can access Domino backend objects, you extend the *AppletBase* class and put the functional code in the following methods: *notesAppletInit()*, *notesAppletStart()*, *notesAppletStop(), notesAppletDestroy().* These methods are called by the standard applet methods *init(), start(), stop()* and destroy*(),* as implemented by the *AppletBase* class.

As this class already implements the standard applet methods *init(), start(), stop()* and *destroy()*, you cannot create or override them in your applet. Instead these final methods call the following methods: *notesAppletInit()*, *notesAppletStart()*, *notesAppletStop()*, *notesAppletDestroy()*. *AppletBase* is new with Domino 5.0 and the *lotus.domino* package. These methods are also contained in the AppletBase class and are not final. If you want to implement the functionality that you would normally in the traditional Applet methods, you must override the respective *notesAppletXXX()* method in your Applet class.

When you develop a Domino R5 applet, you do not have to distinguish between local and remote access in your code. AppletBase makes local calls if the Applet is running through the Notes client and remote CORBA calls if it is running through a browser.

To access a Session object within your applet you use the *getSession()* inherited instance method of the *AppletBase* class. The *getSession()* method call will also instantiate and initialize the client-side ORB as well as request a remote Session object reference from the Domino server. Each applet in an HTML page that invokes the *getSession()* method will instantiate another client ORB. As this situation has to be avoided, methods for inter-applet communication such as the InfoBus technology may be used to overcome this problem.

13.3 Writing a DB2 and Domino Applet

The following example explains how to develop an applet that:

- Uses CORBA to access a database on a Domino server
- Lists all the documents of the database using a view
- Allows the selection of one document key in the list
- Allows access to a DB2 database using the selected key to display additional information

Only the important part of the code is shown here. For a complete listing of the applet, refer to Appendix A, “Applet Example” on page 219.

Import Statements

We first load all the required packages needed to create our applet:

- Java package to write Java code
- Java awt to support graphical user interface
- Java JDBC classes to support JDBC access
- DB2 Java classes to access DB2 using the DB2 JDBC Applet driver
- Domino package to use Domino backend classes

The following import statements perform the task:

```
import java.applet.*;
import java.awt.*;
import lotus.domino.*;
import COM.ibm.db2.*;
import java.sql.* ;
```

To access the Domino server from the applet and use its backend classes you have to import the *lotus.domino* package into your code. The applet uses the CORBA-enabled Java classes to access the remote Domino R5 server. To support remote calls, the Domino server must be running the HTTP and DIIOP server tasks.

Db2DominoApplet

Our class starts by extending the *AppletBase* class, and defining a few global variables for the class.

```
public class Db2DominoApplet extends AppletBase implements
java.awt.event.ItemListener {
   //Intitialization for Domino
   Session s;
   Database db;
   DocumentCollection dcol;
   String dbname = "SG245425T";
   String server = "oxygen";
   String viewname = "Employee\\Last Name";
   String user = "Administrator";
   String pwd = "password";
   //Initialization for DB2
   Connection con;
   Statement stmt;
   ResultSet rs;
   ResultSetMetaData rsmd = null;
   String name;
   String sqlinit = "SELECT LASTNAME, FIRSTNME,SALARY, BONUS, COMM
                     FROM christo.employee where empno ='";
   String sqlend = "'";
   String userdb2 = "db2admin" ;
   String pwddb2 = "db2admin" ;
```

notesAppletInit Method

The *notesAppletInit()* method is called once when the Web server loads the applet. In this method, we initialize all fields of the user interface and connect to Domino (*connectDomino* method) and DB2 (*connectDomino* method). If the connections are successful, we access Domino to get the information (*accessDominoDatabase* method).

```
public void notesAppletInit() {
   super.notesAppletInit();
   try {
      setName("Db2DominoApplet");
      setLayout(null);
      setSize(364, 416);
      add(getLabel1(), getLabel1().getName());
      add(getEmployeeNumberList(), getEmployeeNumberList().getName());
      ...
      connectDomino() ;
      connectDB2() ;
      accessDominoDatabase() ;

   } catch (java.lang.Throwable ivjExc) {
      handleException(ivjExc);
   }
}
```

connectDomino Method

The Session class is the root of the Domino backend object containment hierarchy. Depending on the type of Java program, use the following methods to create a session object:

- For applications making local calls:

  NotesFactory.createSession()

- For applications making remote calls,

  NotesFactory.createSession(String host)
  or
  NotesFactory.createSession(String host, String user, String pwd)

- For agents:

  AgentBase.getSession()

- For applets:

  ```
  AppletBase.openSession()
  or
  AppletBase.openSession(String user, String pwd) and
  AppletBase.closeSession(Session session)
  ```

The user and password parameters of *NotesFactory.createSession(String host, String user, String pwd)* and *AppletBase.openSession(String user, String pwd)* must be a user name and Internet password in the Domino Directory on the server being accessed. If a name and password are not specified, anonymous access must be permitted by the server.

```
public void connectDomino() {
   try {
      s = this.openSession("Administrator","password");
      if ( s==null) { // not able to make the connection
         System.out.println("Unable to create a session with the server") ;
         return ;
      }
      System.out.println("Connected on server :" + s.getServerName());
      System.out.println("for User  :" + s.getCommonUserName());
   } catch (NotesException e) {
      System.out.println("Error in Connecting Domino Server");
      e.printStackTrace();
   }
```

accessDominoDatabase

Once the applet has established the connection to the Domino database, we can use various methods to access the database, create a collection of documents, access the documents in a sorted order using a view, and finally access a specified document and its items.

The *getDatabase* method of the *Session* class accesses a specified database. You provide the name of the server, or *null* for local, and the name of the database. The *getView* method of the Database class accesses a specified view. The *get---Document* methods of the View class allow you to access a specific document in the view (replace --- with First, Last, Next, Nth). The *getItemValue*--- methods of the Document class allow you to access a specific field of the document (replace --- with Integer, String, Double).

```
public void accessDominoDatabase() {
   try {

      // Access Database
      db = s.getDatabase(s.getServerName(), dbname);
      //List All Documents
      dcol = db.getAllDocuments();
      System.out.println("Database \"" + dbname + "\" has "
                       + dcol.getCount() + " documents");
      //Get the Document through a view
      View view = db.getView(viewname);
      Document doc = view.getFirstDocument();
      // Document doc;
      String [] val = new String[dcol.getCount()];
      //String[] val;
      for (int i = 1; i <= dcol.getCount(); i++) {
         val[i] = doc.getItemValueString("EMPNO");
         getEmployeeNumberList().add(val[i]);
         doc = view.getNextDocument(doc);
      }
   } catch (NotesException e) {
      System.out.println("Error in Access Database");
      e.printStackTrace();
   }
}
```

connectDB2

We used JDBC to connect to the DB2 database. To load the DB2 applet JDBC driver, we used the *forName* method that creates an instance of the driver and registers it with the JDBC driver manager. We established a connection using the *getConnection* method of the DriverManager class, passing the URL. A DB2 applet JDBC driver URL always begins with *jdbc* as protocol, *db2* as subprotocol, followed by the name of the DB2 server, the

TCP/IP port number where the DB2 applet JDBC server is listening, and the name of the database.

```
public void connectDB2() {
   try {

      String port = "999";
      Class.forName("COM.ibm.db2.jdbc.net.DB2Driver"); // .newInstance();
      // construct the URL ( sample is the database name )
      String url = "jdbc:db2://" + server + ":" + port + "/sample";
      // connect to database with userid and password
      con = DriverManager.getConnection(url, userdb2, pwddb2);
      System.out.println("Connection DB2 OK");
   } catch (Exception e) {
      System.out.println("Error in Connecting DB2 Server") ;
      e.printStackTrace();
   }
}
```

accessDB2Database

When the user selects an employee number displayed on the applet, a query is triggered to fetch additional data from the DB2 database.

We created a SQL statement containing the query. We executed the query using this SQL statement. This query generates a result set. The result set is read to get the additional data to the applet fields. As the query generated a result set containing only one row, we used the *next()* method only once to access that row.

```
public void accessDB2Database(String empno) {
   try {
      String query ;
      stmt = con.createStatement() ;
      query = sqlinit + empno + sqlend ;
      rs = stmt.executeQuery(query) ;
   } catch (Exception e) {
      System.out.println("Error in AccessDB2Database") ;
      e.printStackTrace();
   }
}

public void employeeNumberList_ItemStateChanged(java.awt.event.ItemEvent
event, String empno) {
   try {
      if (event.getStateChange() != java.awt.event.ItemEvent.SELECTED) {
         return;
   }
```

```
      accessDB2Database(empno);
      if (!rs.next()) {
         System.out.println("No Corresponding row");
      } else {
         getLastNameTextField().setText(rs.getString(1));
         getFirstNameTextField().setText(rs.getString(2));
         getSalaryTextField().setText(rs.getString(3));
         getBonusTextField().setText(rs.getString(4));
         getCommTextField().setText(rs.getString(5));
      }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
```

13.4 Downloading and Displaying an Applet

The applet is downloaded as bytecode which is interpreted and executed by the JVM of the Web browser.

The applet can edit screen input, generate screen output, and communicate back to the computer from which it was downloaded. Multiple applets can execute concurrently.

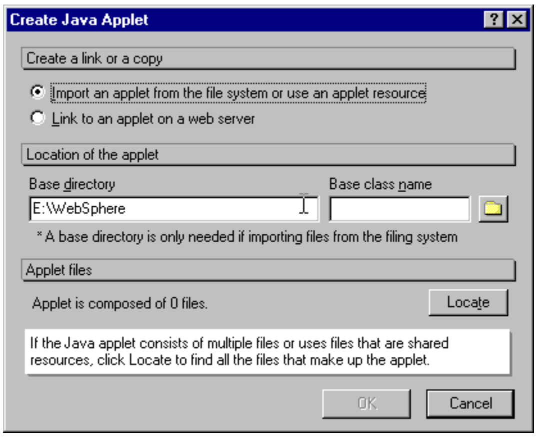
The downloading of applets should not have a significant performance impact on response time because the applets are typically not very large. In fact, applets, by performing processing on the browser or network computer, can improve the overall browser performance by eliminating iterations with the Web server. Note that, just as images are cached in Web browsers, applets are cached, thereby minimizing the frequency of applet downloading. A current performance consideration is the iterative compiling of the Java bytecode at the time of execution. This consideration, however, is rapidly being addressed by the industry and is losing its importance.

Applets can be embedded in a Domino form or in an HTML page. In the following we describe how to reference an applet in both environments.

13.4.1 From a Domino Form

In a Domino form, you can create a Java applet in a rich text field of a Domino form. You can use the following options (see Figure 41 on page 142):

- Using a link to an existing applet on a Web server
- Importing the applet into the Domino form



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: E:\WebSphere

Base class name:

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

Applet files

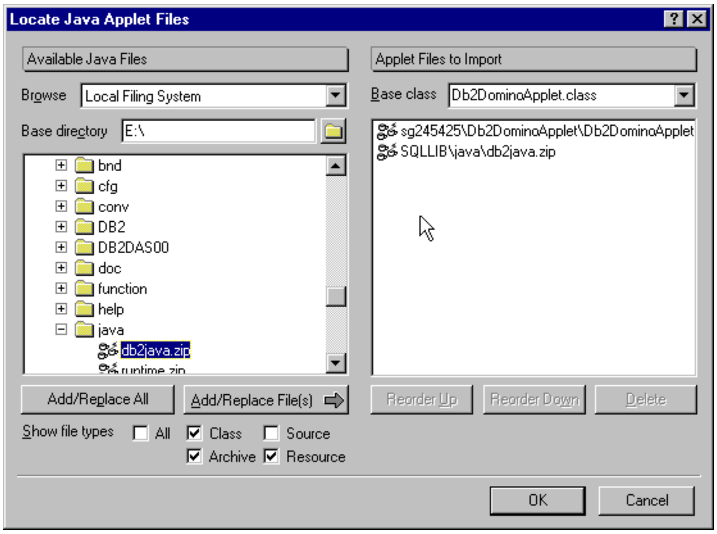
Applet is composed of 0 files.

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 41. Create Java Applet

If you choose to link to an applet on a Web server, you have to specify the base URL and the applet class name. The NCSO jar and cab files must be located in the HTML directory on the Domino server.

If you choose the import option, you have to specify the applet class name and its directory. You must import all applet related classes into Domino (see Figure 42 on page 142).



Locate Java Applet Files

Available Java Files

Browse: Local Filing System

Base directory: E:\

bnd
cfg
conv
DB2
DB2DAS00
doc
function
help
java
db2java.zip
runtime.zip

Applet Files to Import

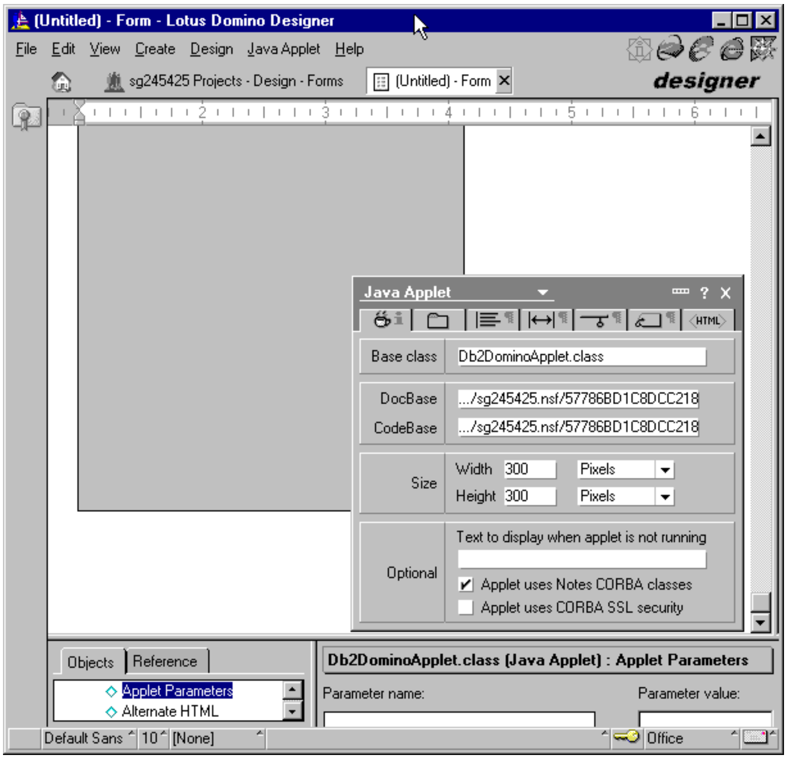
Base class: Db2DominoApplet.class

sg245425\Db2DominoApplet\Db2DominoApplet
SQLLIB\java\db2java.zip

Show file types: [ ] All [✓] Class [ ] Source [✓] Archive [✓] Resource

Figure 42. Locate Java Applet Files

After creating the applet, you modify the applet parameters such as its width and height using its properties box (see Figure 43 on page 143).


(Untitled) - Form - Lotus Domino Designer
File Edit View Create Design Java Applet Help
sg245425 Projects - Design - Forms
(Untitled) - Form
designer
Java Applet
Objects
Reference
Applet Parameters
Alternate HTML
Default Sans 10 [None]
Office

Java Applet

Base class	Db2DominoApplet.class
DocBase	.../sg245425.nsf/57786BD1C8DCC218
CodeBase	.../sg245425.nsf/57786BD1C8DCC218
Size	Width 300 Pixels; Height 300 Pixels
Optional	Text to display when applet is not running:
	[✓] Applet uses Notes CORBA classes
	[ ] Applet uses CORBA SSL security

Db2DominoApplet.class (Java Applet) : Applet Parameters

Parameter name: Parameter value:

Figure 43. Applet Parameters

In the applet parameter properties box, you can also specify if the applet uses the Domino CORBA classes. When this option is selected, Domino automatically includes the NCSO.jar file and adds applet parameters to the applet HTML tag before serving the document to the Web browser:

```
<APPLET WIDTH="300" HEIGHT="500"
      CODEBASE="/sg245425.nsf/cb77cc059215d0488825673c006c4a9d/$FILE"
      CODE="itso.sg245425.applet.Db2DominoApplet.class"
      ARCHIVE="Db2DominoApplet.jar,db2java.zip,NCSOC.jar">
<PARAM NAME="NOI_IOR"
VALUE="IOR:01ffffff2900000049444c3a6c6f7475732f646f6.....
......ffffff00000000">
<PARAM NAME="NOI_COOKIE_URL" VALUE="/sg245425.nsf?GetOrbCookie">
</APPLET>
```

The NOI_IOR parameter is the object reference to the DOM server. This is the object on which you call the getSession().

The NOI_COOKIE parameter ensures single user login; that is, the CORBA applet is not challenged for a user name and password again. The cookie comes from the server when the client logs on.

13.4.2 From an HTML Page

Applets begin execution by loading the HTML page that *contains* them. Either the appletviewer or a Web browser is required to run applets. A browser detects an applet by processing an <applet> HTML tag. The <applet> tag instructs the Web browser to load the applet in a particular location on the Web page. When this Web page is displayed, the Java code that makes up the applet is fetched from your Web server, and then run in your client as part of the Web page.

Applets are invoked through the use of the applet HTML parameter:

```
<applet code="ReptApplt.class" width=325 height=275 archive="db2java.zip">
</applet>
```

with:

```
code=represent the name of the applet. Typically, it would be some
          filename.class.
width=represents the width in pixels within your web page where the Java
          applet executes and displays its information.
height= represents the height in pixels within your web page where the Java
          applet executes and displays its information.
archive=represent the name of the archive file to download with the applet
(refer to the Java documentation for others parameters)
```

13.5 Java Applets and Enterprise Integration

Generally, applets are used to provide the user interface (UI) in a Web-based application, rather than actually doing the work of hosting a conversation with say a CICS or DB2 machine. The main reason for this is that Java applets must be kept fairly small to ensure acceptable performance. Since applets must be downloaded and run each time they are used, keeping them small is desirable.

However, a signed applet is able to make calls to external libraries on the client workstation. For example, you can develop an applet that uses the DB2 CAE to fetch data from the local DB2 databases.

To access the enterprise directly the applet must be authenticated. For security reasons applets are not able to make calls to other programs running on the workstation, unless they are signed and authenticated as coming from a trusted supplier. Without this trust it is impossible to make a call from an applet to, for example, the DB2 clients that you have running on your users' machines. To provide the authentication services required to implement this type of architecture you need to adopt one of the security models provided by the respective Web browser companies such as Netscape's Object-Signing and Microsoft's Authenticode.

It is also important to note that the applet itself is unable to communicate with another machine other than the server it originated from, but if the applet can access external libraries (that is, it is trusted to do so), the external library could establish a connection with an enterprise system and pass the results back to the applet. In this scenario there would, of course, be the need for external library software (to gain access to the enterprise) to be installed on each Web browser workstation.

In general, this *two-tier* approach of applet-to-enterprise is practical for smaller projects within the enterprise. When the requirement is for many hundreds of users to access complex enterprise systems then other services are required as part of your overall solution. Features like connection management, load balancing and transaction management are often mandatory, but providing these features from an applet is difficult and cumbersome. It is more pragmatic for features of this nature to be provided from a centralized point, rather than the decentralized nature of having applets talk to your enterprise systems directly. This *three-tier* approach has the middle tier performing the interaction with the enterprise system on behalf of the applets. For example, you may have your applets talking to a number of servlets on your Domino server, and the servlets in turn perform the actual communication with the enterprise system.

Chapter 14. Java Applications

A Java application is a program written in Java. To create the application, you can use any Java development environment such as IBM VisualAge for Java. If you don't own a Java development environment, you can install the Java Development Kit (JDK). The JDK provides a compiler you can use to compile all kinds of Java programs. It also provides an interpreter you can use to run Java applications.

To run the Java application, you need to install the JVM for your hardware platform. Figure 44 on page 147 depicts a Java program, such as an application or applet, that's running on the JVM supporting the Java API. As the figure shows, the Java API and Virtual Machine insulates the Java program from hardware dependencies.

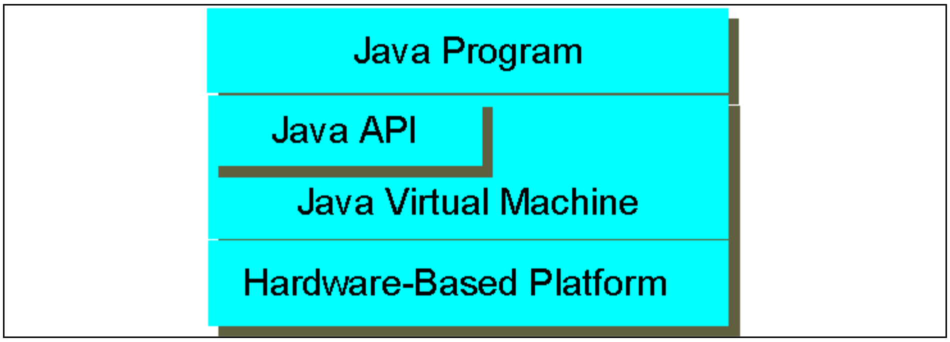


Figure 44. Java Application Environment

As a platform-independent environment, Java can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time bytecode compilers can bring Java's performance close to that of native code without threatening portability.

As with any application, a Java application can connect to Domino as well as the enterprise DBMS or OLTP. Unlike Java applets or Java servlets, which are Internet-based applications, Java applications can be used in a local architecture that does not require Internet or intranet connections.

To illustrate how to code a Java application, we have created two applications that access a Domino environment. The first uses the Domino driver for JDBC and manipulates Domino data, while the second uses the Domino Java classes directly.

14.1 Domino Driver for JDBC Application

This section describes how to write a Java application that uses the Domino driver for JDBC to access Domino data. Writing a JDBC application is similar to writing an ODBC application, but in Java.

Only the important part of the code is shown here. For a complete listing of the application, refer to Appendix C.1, “Domino JDBC Driver” on page 227.

Import Statements

We first load all the required packages needed to create the application:

- Java package, to write Java code
- JDBC package, to write a JDBC application
- The Domino driver for JDBC package

```
import java.util.*;
import java.sql.*;
import lotus.jdbc.domino.*;
```

DominoJDBC Sample

Our DominoJDBCSample defines the entry point of this Java application. The main method is defined as a class method (*static*), can be called by any object (*public*), and does not return any values (*void*).

```
public class DominoJDBCSample {
   public static void main ( String[] args ) {
```

Loading the Domino Driver for JDBC

The *forName* method creates an instance of the driver and registers it with the JDBC driver manager.

```
Class.forName("lotus.jdbc.domino.DominoDriver");
```

Establishing a Connection

To establish a connection, we use the *getConnection* method of the DriverManager class, passing the URL. A Domino driver for JDBC URL always begins with *jdbc* as protocol, *domino* as subprotocol, followed by the name of the Domino or Notes database. For example:

- To open the database *DomDemo.nsf* in the Domino data directory of the current machine:

  ```
  Connection con ;

  String connStr = "jdbc:domino/DomDemo.nsf"
  con = DriverManager.getConnection(connSrt,"","");
  ```

- To open the database *DomDemo.nsf* in the Domino data directory of the Domino server named *Oxygen:*

  ```
  String connStr = "jdbc:domino/DomDemo.nsf/Oxygen"
  con = DriverManager.getConnection(connSrt,"","");
  ```

The second and third parameters—the user ID and password in many JDBC systems—of the *getConnection* method are not needed in the Domino driver for JDBC.

Creating and Executing an SQL Statement

Use the *createStatement* method of the connection object to create an SQL statement. Use the *executeQuery* method of the statement object to execute SELECT statements that produce a result set. Use the *executeUpdate* method of the statement object to execute INSERT, UPDATE, and DELETE statements and any data definition language statement such as CREATE TABLE.

Domino is more flexible about names than SQL. When naming a form or view, Domino allows many special characters that are not part of the SQL syntax, for example the backslash identifying a hierarchical view (Product\By Name) or a space. This syntax is supported by the Domino driver for JDBC. Enter the name between double quotes.

Views in Domino/Notes databases list documents in a specific order. Avoid selecting from a table based on a view and then specifying a different sort order. When you specify a different sort order on an existing view, Domino driver for JDBC creates a temporary table on your workstation and re-sorts the documents. Creating a large temporary table and sorting the documents in that table will take a long time.

```
Statement stmt ;
ResultSet rs ;
String sql = "SELECT * FROM \"Employee\\Last Name\" ;

//Create Statement
stmt = con.createStatement();

//Execute Statement
rs = stmt.executeQuery(sql)
```

Manipulating Data

Once you have obtained a result set, you can get the metadata, information about the types and properties of a column in the result set. As the query generated the result set, we used the *next()* method in a loop to read the entire row using the getObject() method.

```
ResultSet rs ;
ResultSetMetaData rsmd = null ;

rsmd = rs.getMetaData() ;
//# of columns in the result set
colcount = rsmd.getColumnCount();
//Name of column i
colName(i) = rsmd.getColumnName(i) ;
//loop through the result set
while (rs.next()) {
   for (int i = 1; i <= colCount; i++) {
      Object obj = rs.getObject(i);
      boolean nl = rs.wasNull();
      if (nl)
         printCol(len[i], "null");
      else
         printCol(len[i], obj.toString());
      }
   System.out.println();
}
```

Closing Statements

When you have finished using the objects, you have to close them using the *close* method of each object.

```
rs.close();
stmt.close();
```

Closing the Connection

When you have finished with the application, you need to close the connection using the *close* method of the connection object.

```
con.close();
```

14.1.1 Security

When you create a connection to the Domino server and try to access the Domino data through JDBC, Domino looks for the ID file referenced in the *notes.ini* file of the machine running the Java application. It prompts you for the password.

This is the same security that is in place with any Notes C API application. This is why Domino does not support the sending of a user name and password through the API; doing so would breach Domino security. To run an application unattended without ever receiving a password prompt, you must use a non-password-protected ID. You can remove password protection from your ID by clearing it (File - Tools - User ID - Clear Password), unless your

Domino administrator required a password to be used when your ID was created. In that case, you won't be able to clear it.

If you run an applet over a network that uses the Domino driver for JDBC, security is important. One approach to security is to allow applets to run only within the browser's sandbox. Another approach to security is to sign the driver's archive files (jar or *Java archive* files for Netscape Navigator/Communicator, and cab or *cabinet* files for Microsoft Internet Explorer). The process of signing the driver identifies the creator of the signed files. The user is then prompted to choose to allow this signed file access to restricted operations like loading native Domino files. Because the Domino driver for JDBC needs to communicate with Domino, it must go outside the sandbox. You may create your own signed archive files, or use those provided with the Domino driver for JDBC.

14.2 Local Access to the Domino Object Classes

This section describe how to write a Java application that uses Notes backend objects and access Domino data. Domino R4.6 users as well as Domino R5 users may implement such applications. As local calls access run-time code on the local machine, you must install Domino server or Notes client on the local machine.

The Notes classes for Java are similar to the LotusScript classes.

Only the important part of the code is shown here. For a complete listing of the application, refer to Appendix C.2, “Domino Objects Classes” on page 229.

In general, creating a Java stand-alone application program which is importing Notes classes should include the following:

Import Statements

We first load the Domino Java package needed to create the application.

```
import lotus.domino.*;
```

Main Statement

A Java application that uses the Notes objects must use the *lotus.notes.NotesThread* class, which extends *java.lang.Thread*. It contains some special per-thread housekeeping code.

There are three ways to organize the application using the Notes classes:

- Extend the NotesThread class and use its runNotes method as the entry point to the functional code.

  To execute the method, the main method creates a new object of the class it is in (the class that extends NotesThread) and calls the start method. The start method is overloaded in lotus.notes.NotesThread to call runNotes.

```
public class DominoTestDirectNotesThread extends NotesThread {
   // Starts the application.
public static void main(java.lang.String[] args) {
   try {
      DominoTestDirectNotesThread thread1 = new
DominoTestDirectNotesThread();
      thread1.start();
      thread1.join();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
}
public  void runNotes() {
// Entry Point Method
.....
}
```

- Implement the Runnable interface and use the run method of the java.lang.Thread class as the entry point to the functional code.

  To execute the method, the main method creates a new object of the class it is in (the class that implements Runnable), creates a new NotesThread object passing to it the Runnable object, and calls the start method.

```
public class DominoTestDirectRunnable implements Runnable {
   // Starts the application.
public static void main(java.lang.String[] args) {
   try {
      DominoTestDirectRunnable thread1 = new DominoTestDirectRunnable();
      NotesThread nt = new NotesThread((Runnable) thread1);
      nt.start();
      nt.join();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
}
public  void run() {
// Entry Point Method
......
}
```

- Call the static sinitThread and stermThread methods of the NotesThread class to explicitly start and terminate a thread. If you do use these methods, be sure to call stermThread exactly one time for each call to sinitThread.

```
public class DominoTestDirect {
   // Starts the application.
public static void main (java.lang.String argv[]) {
   try {
      NotesThread.sinitThread() ;
      Session s = Session.newInstance
      ......
   } catch (NotesException e) {
      e.printStackTrace();
   }
finally
{
   NotesThread.stermThread() ;
}
}
```

Session, Database, View Methods

Depending on the organization of the application, the entry point of the Java application can be:

- *runNotes()* when extending NotesThread
- *run()* when implementing the Runnable interface
- *main()* when calling the static NotesThread methods.

```
public  void runNotes() {
   String dbname = "SG245425T";
   String servername = "china" ;
   String viewname = "Employee\\Last Name" ;
   try {
```

A session object is necessary to do anything with the Notes classes. In applications, the *newInstance* static method of the *Session* class creates the necessary object.

```
      System.out.println("Session" );
      Session s = Session.newInstance() ;
```

The *getDatabase* method of the *Session* class accesses a specified database. You provide the name of the server, or *null* for local, and the name of the database. For local databases, the name is relative to the Notes data directory unless you supply a full path name. The suffix default is *nsf*.

```
Database db = s.getDatabase(servername, dbname) ;
System.out.print("Database "+ dbname + "has been last modified on" );
System.out.println(db.getLastModified()) ;
```

The *getView* method of the Database class accesses a specified view.

```
View view = db.getView(viewname) ;
Vector columns = view.getColumns();
```

The *get---Document* methods of the View class allow you to access a specific document in the view (replace --- with First, Last, Next, Nth).

```
System.out.println("View " + viewname + "contains the following documents" );
Document doc = view.getFirstDocument();
.....
```

The getItemValue--- methods of the Document class allow you to access a specific field of the document (replace --- with Integer, String, Double).

```
while (doc != null) {
System.out.println("Last Name: " + doc.getItemValueString("LASTNAME") );
...
System.out.println("Salary : " + doc.getItemValueInteger("SALARY"));
doc = view.getNextDocument(doc);
}
```

14.2.1 Security

To enable Java applications to use Notes classes, Domino will look for the ID file referenced in the notes.ini file on the machine where the application is running. You have to enter the password if this ID file is password-protected.

14.3 Local Access Using the Lotus Connectors

This section describes how to write a Java application that uses the Lotus Connector Java classes to access Domino data. As local calls access run-time code on the local machine, you must install Domino server or Notes client on the local machine, as well as the Lotus Connector Java classes.

The Lotus Connector Java classes are similar in functionality to the Lotus Connector LotusScript extension.

Only the important part of the code is shown here. For a complete listing of the application, refer to Appendix C.3, “Lotus Connector” on page 231.

This application accesses a Domino database and selects documents using a selection criteria.

Import Statements

We need to load the Lotus Connector Java package to create the application.

```
import lotus.lcjava.*;
```

Main Statement

Our Java application is standard. We don't have any special requirement.

```
public static void main(java.lang.String[] args) {
   LCSession session = null;
   LCConnection connection = null;
   try {
```

LCSession and LCConnection

You must create an LCSession object before using any other LC Java facilities. The LCSession object contains global state information.

The LCConnection class represents an instance of a Lotus connector, providing access capabilities to the enterprise system. Multiple connections can be allocated to a single connector.

The LCConnection constructor identifies the enterprise system by name (first parameter) or session token (second parameter). The connection method establishes a connection. Before establishing the connection, you must set the properties that apply to the connection. In our case, we set the server name and the database name. We also set the metadata and the Notes form name, since the selection formula does not include any information about the type of data accessed.

```
// Create LC session
      session = new LCSession(0);
      System.out.println("Session ready");
      // Create LC Connection to Notes
      connection = new LCConnection("notes", 0);
      System.out.println("Connection to Notes OK");
      connection.setPropertyJavaString(LCTOKEN.SERVER, "oxygen/Almaden");
      connection.setPropertyJavaString(LCTOKEN.DATABASE,
                                    "sg245425\\sg245425T.nsf");
      connection.setPropertyJavaString(LCTOKEN.METADATA, "Employee");
      connection.connection();
```

Field Lists

We now set up the field lists for the query, the result set returned by the query, and the data fetched from the result set.

```
//Set the data field lists
      LCFieldlist keyList = new LCFieldlist(1, 0);
      LCFieldlist resultList = new LCFieldlist(1, 0);
      LCFieldlist fetchList = new LCFieldlist(1, 0);
```

Key Field List

We want to select only the documents of employees belonging to department D11.

```
//Set up the key
      LCField keyField = new LCField();
// the key column
      keyList.append("WORKDEPT", LCTYPE.TEXT, keyField);
      keyField.setFlags(LCFIELDF.KEY);
// the key value
      LCStream Workdept = new LCStream("D11");
      keyField.setStream(1, Workdept);
```

Result Set Field List

We now set up the result set field list. In this list we specify which fields have to be returned by the connector. *LASTNAME*, *FIRSTNME*, *SALARY*, and *BONUS* are the fields of the Domino document. *lastName*, *firstName*, *salary*, and *bonus* are the Lotus connector fields.

```
      LCField lastName = new LCField();
      LCField firstName = new LCField();
      LCField salary = new LCField();
      LCField bonus = new LCField();
      resultList.append("LASTNAME", LCTYPE.TEXT, lastName);
      resultList.append("FIRSTNME", LCTYPE.TEXT, firstName);
      resultList.append("SALARY", LCTYPE.FLOAT, salary);
      resultList.append("BONUS", LCTYPE.FLOAT, bonus);
```

Get the Result Set

We are now ready to execute the query to get the result set. We use the select method of the connection object. If available from the source, *returnedRows* is set to the number of rows returned.

```
      connection.setPropertyJavaString(LCTOKEN.FIELD_LIST,
                     "LASTNAME,FIRSTNME,SALARY,BONUS");
      Integer returnedRows = new Integer(0);
      connection.select(keyList, 1, resultList, returnedRows);
```

Manipulate the Result Set

We can now manipulate the result set, entering a loop to display all the data fetched from the Notes database.

```
// Set up and fetch the result
      LCField lastNameF = new LCField();
      LCField firstNameF = new LCField();
      LCField salaryF = new LCField();
      LCField bonusF = new LCField();
      fetchList.append("LASTNAME", LCTYPE.TEXT, lastNameF);
      fetchList.append("FIRSTNME", LCTYPE.TEXT, firstNameF);
      fetchList.append("SALARY", LCTYPE.FLOAT, salaryF);
      fetchList.append ("BONUS", LCTYPE.FLOAT, bonusF) ;
      int rc = 0;
      rc = connection.fetch(fetchList, 1, 1);
      while (rc != LCFAIL.END_OF_DATA) {
         System.out.println("Name: " + lastNameF.toJavaString());
         System.out.println("First Name: " + firstNameF.toJavaString());
         System.out.println("Salary: " + salaryF.toJavaString());
         System.out.println( "Bonus: " + bonusF.toJavaString()) ;
         rc = connection.fetch(fetchList, 1, 1);
      }
```

Error Handling

The LCSession constructor and many other LC methods throw an LCException, which extends java.lang.Exception. LCException contains the method getLCErrorCode to get an integer error code specific to LC. The LCSession method getStatusText returns the message associated with an error.

```
catch (LCException e) {
      int err = e.getLCErrorCode();
      System.out.println(err) ;
      String errmsg = session.getStatusText(err);
      System.out.println(errmsg);

   }
```

14.4 JDBC, DOM, or LC to Domino Data?

To access Domino data, you can use one of the following methods:

- Domino driver for JDBC
- Domino object classes
- Lotus Connector for Notes database

14.4.1 Domino Driver for JDBC or Domino Object Classes?

Table 4 on page 158 summarizes the differences between Domino driver for JDBC and Domino object classes.

Table 4. Domino Driver for JDBC and Domino Object Classes

Description	Domino driver for JDBC	Domino Object Classes
Local Access	Data	Data and Applications
Remote Access	Data	Data and Applications
Access method to Domino data	SQL and JDBC	Domino object classes and CORBA
Software installed on the server	Domino Driver for JDBC	None
Classes to import	lotus.jdbc.domino.*	lotus.notes.* in R4.6 lotus.domino.* in R5.0

If you are familiar with developing JDBC applications, you can easily develop applications that access Domino data using the Domino driver for JDBC. The architecture of an application accessing Domino data is identical to an application accessing any relational DBMS using JDBC. Although Domino is not a relational database, you can access and manipulate the Domino data using the SQL.

The Domino object classes allow you to access Domino data and Domino objects such as applications, agents, forms,... With Domino R5, the application can run on a remote machine and access the Domino object classes using CORBA. Using Domino object classes, Domino data is just another Domino object. Domino object classes are based on the same model as the Domino LotusScript classes. If you know how to develop Domino applications using LotusScript, you will find the Domino object classes architecture very similar. The Java class names are the same as for LotusScript with the Notes prefix omitted.

14.4.2 Lotus Connectors

One advantage of using the Lotus Connector Java classes is the portability of the code for every source supported.

In our sample, we created the Lotus Connector application to access a Notes database and its *Employee* documents. This application can be easily changed to support a DB2 database and its *Employee* table.

Assuming you have a DB2 table that contains the same data as the Notes database, you only have to change the setting of the connection object (differences are **shown like this**) to change the data source from Notes to DB2:

```
// Create LC Connection to DB2
      connection = new LCConnection("db2", 0);
      System.out.println("Connection to Db2 OK");
      connection.setPropertyJavaString(LCTOKEN.DATABASE, "sample");
      connection.setPropertyJavaString(LCTOKEN.METADATA,
                                       "db2admin.employee");
      connection.setPropertyJavaString(LCTOKEN.USERID, "db2admin");
      connection.setPropertyJavaString(LCTOKEN.PASSWORD, "db2admin");
      connection.connection();
```

In our sample, we also had to change the format of the numeric fields (SALARY and BONUS), because they were defined as FLOAT in Notes and NUMERIC in DB2.

Chapter 15. Agents

Agents are discrete, embedded programs that perform a specific task in a database. They are not tied to a specific Domino view or form. This agent concept is almost unique to Lotus Domino as an application server and its Lotus Notes clients.

An agent requires three things:

1. Instructions about when to run
2. Instructions about which documents to process
3. A program that processes the selected documents

An agent can be invoked in various ways:

- Manually from a Notes client
- Triggered by a selected event, such as the arrival of new mail or the creation/modification of a document
- Triggered from a Web browser, although some functions supported by the Notes client are not valid from a Web browser
- Programmatically from another agent or application
- Scheduled to run periodically

The options for selecting which document(s) to process include:

- All documents in the database
- New documents
- Changed documents
- Documents that meet search criteria

Agents in Domino can be:

- Simple actions
- Formulas
- LotusScript programs
- Java programs

The level of control provided by agents makes them an attractive way of programming. Agents have a couple of other advantages over servlets and applications: security and transportability.

Domino agents are automatically signed by the person who created or last modified them. Personal agents are only available for modification and execution by that person. Shared agents can be invoked by anyone with access to the database and modified by those with designer access or greater. Domino agents run with a particular user’s identity, whether run on the Domino server or the Notes client. Agents that execute on the Notes client (whether in the foreground or background) run with the identity and, therefore, the access authority of the user. The agent signer’s access authority is used for background agents running on the server.

By default, a shared background agent that runs from the Web uses the access rights of the agent signer to determine if the agent can run and how much access it has to the server's file system. Each Web user who is allowed to run the agent should be registered in the Public Address Book (known as the Domino Directory in Domino R5) and be named in the ACL and the server document with the appropriate rights. Agents invoked by a Web user, who has not been authenticated, run with the special identity of Anonymous, if this access has been permitted on the server. To add more security to a Web agent, force Domino to check the access rights of registered Web users.

Agents are a design element of a database, just like any form or view in that database. This means that they are transported with the database whenever it is moved, copied or replicated. Thus, Java agents are an integral part of a Domino application.

Agents have advantages, but you should not forget that Java servlets and applications can be written to run in isolation. Java agents, on the other hand, must run with Domino.

Agents written in Java can communicate with enterprise applications. We use MQSeries to illustrate our discussion of agents, as this allows us to show you a method of communicating from the enterprise into Domino.

In the following sections, we explain the structure of an agent and how to trigger it. We give two examples of agents that connect to and from the enterprise using MQSeries.

15.1 Structure of an Agent

Domino agents have full access to the standard Java run-time services. They can access other Java libraries (with the appropriate import statement, for example, JDBC or the MQSeries Java classes), URLs and network sockets.

An agent extends the *AgentBase* class, which extends the *NotesThread* class. The class that contains the agent code must be public.

The entry point for any Domino agent in Java must be a method called *NotesMain()*. Other methods can be called from *NotesMain()*, but *NotesMain()* is where the execution of functional code starts.

For output to browsers, assign a *PrintWriter* object using the *getAgentOutput* method of the AgentBase class. If the agent is never run from a browser, you can use System.out as usual although the PrintWriter object also works.

So, with these pieces of information, the starting point for any Java agent is going to be along the following lines:

```
import lotus.notes.*;           // Include Notes package
import java.io.PrintWriter;
public class class_name extends AgentBase {
public void NotesMain() {
   try {
      PrintWriter pw = getAgentOutput();
      String username = s.getUserName();
      pw.println("Your Notes user name is " + username + ".");
   } catch(NotesException e) {
      e.printStackTrace();
      }
   }
}
```

Domino agents written in Java can be multithreaded, making use of the java.lang.Thread class. This gives agents written in Java an advantage over those written in LotusScript.

There are no software prerequisites, for invoking an agent from a Web browser. The agent will always run on the Domino server and it is the Domino server that must meet any software prerequisites.

15.2 Triggering an Agent

Table 5 lists the available methods to trigger an agent.

Table 5. Agent Triggers

Trigger	Details
URL	?OpenForm
Open page/form event	WebQueryOpen (R4.6 and R5) $$QueryOpenAgent (R4.5)
Save page/form event	WebQuerySave (R4.6 and R5) $$QuerySaveAgent (R4.5)
Schedule	Hourly Daily Weekly Monthly
Database event	When New Mail has Arrived Document Created or Modified Document Pasted
Manually	Action Menu Agent list @Command

Agents run either on the Notes client or on the Domino server.

If your agent runs on the Notes client, you may have to install the enterprise Java library, for example the MQSeries Client for Java, on the Notes client. For example, an agent executes locally on the Notes client if you trigger the agent manually from an Action menu or if the application invokes it by use of an @Command([ToolsRunMacro;""]) hidden in a form.

Alternatively, agents can run on the Domino server. In that configuration, you have to install the enterprise Java library on the server. For example, you can trigger an agent with database events such as *If Documents Have Been Created or Modified*. Saving a new or modified document into the server-based application causes the Domino Agent Manager to schedule the execution of the agent on the server. In the agent runs on the server, there is no additional software prerequisite for the Notes client machine.

Another option to consider is the *runOnServer* method of the Agent class. Using this method, a Notes client application can trigger the execution of the agent on the remote server that hosts the agent's parent database. The database must be on a remote server. No parameters can be passed with, or returned by, the method. It is, therefore, not always possible to use the *runOnServer* method and certainly not a sensible option with a Web browser. However, from a Notes client, it may be the right option, as it has the advantage of executing the agent on the Domino server without requiring the intervention of the agent manager.

The method returns an integer indication of the completion status and you code it in the following way:

```
Agent agent = db.getAgent("agent_name");
int status = agent.runOnServer();
```

Java code cannot be used in form events. If you wish to use this Java method, you must put the code into a locally executed agent. To avoid creating this additional agent, you can use the equivalent RunOnServer method of the LotusScript Agent class in a form event.

15.3 From Domino to the Enterprise Using MQSeries

MQSeries lets applications, such as Domino applications, use message queuing to participate in message-driven processing.

In this example we use the MQSeries Client for Java (as discussed in 3.8.2, “The MQSeries Client for Java” on page 35). We connect from a Domino application to an MQSeries application with a Java agent. Our target MQSeries application is written in C and runs on a Windows NT machine, but could be any MQ-enabled application, running on any supported platform.

Our agent extracts the requested part number from the Domino document, together with the required quantity. This document is created from a Notes client or from a Web browser. The target application, DominoToMQAppl, provides the inventory level for the specified part number and indicates whether our required quantity can be supplied from stock or not. If we were writing a complete Domino application, we would need to extract other information about the customer. We have chosen to ignore this requirement, as it would probably be processed as a different part of the workflow in a complete application.

15.3.1 Setup

Queue Manager Setup

Our queue manager must be configured to accept incoming connection requests from the MQSeries clients. This requires the following on our Windows NT target machine:

- Define a server connection channel using the following procedure. We can use the default channel, SYSTEM.DEF.SVRCONN, or using the *runmqsc* program, we can define a new specific channel. A specific channel would need to be defined along the following lines:

  ```
  DEF CHL("JAVA.CHANNEL") CHLTYPE(SVRCONN) TRPTYPE(TCP) +
  MCAUSER(" ") DESCRIPTION("Channel for MQSeries Client for Java")
  ```

  (There is no need to set the MQSERVER environment variable on the client machine, as the MQSeries Client for Java uses the MQEnvironment object to store this information.)

- Start a listener program with the following command:

  ```
  runmqlsr -t tcp [-m QMgr_name] [-p 1414]
  ```

- The queue manager must also have definitions for the queues we use. In our case, that means two local queue definitions:

  ```
  DEFINE QLOCAL('MQAPPL.Q') REPLACE +
        DESCR('Request queue for MQ application')
  DEFINE QLOCAL('MQAPPL.RQ') REPLACE +
        DESCR('Reply queue for MQ application')
  ```

Domino Setup

In order to access the MQSeries Java class library successfully we must add its path to the JavaUserClasses statement in the notes.ini file, together with the path of the directory we will use to develop our code:

```
JavaUserClasses=...;c:\mqm\java\lib;c:\mqm\tools\javaclnt\samples\En_us
```

Agent Invocation

In this example, our agent is invoked in the following ways:

- A Notes client user creates a new document from our form, or modifies an existing document, and clicks the Submit button:

  ```
  @Command([ViewRefreshFields]);
  @Command([FileSave]);
  @Command([ToolsRunMacro];"agent_name");
  ```

- A Web browser user creates a new document from our form, or modifies an existing document, and clicks the Submit button:

  The document is saved into Domino and the agent named in the $$QuerySaveAgent field on the form is invoked.

This requires the MQSeries Client for Java on both the Domino server machine and the Notes client machine. To avoid installing the MQSeries Client for Java on the Notes client we could use the *If Documents Have Been Created or Modified* run option of the agent and have our Submit button simply save the document to the Domino server. We are then dependent on the agent manager to schedule the agent and the agent itself needs to use the *DocumentCollection* class to select appropriate documents for processing.

Every program needs a method of reporting errors and passing other information to a user. Running from a Notes client, System.out prints to the Java console. When our agent is invoked from a Web browser, any System.out printouts appear in the Notes log (as they do for scheduled agents). In order to display output to Web browsers, we need to assign a PrintWriter object.

15.3.2 Writing the Agent

In the following section we explain how we developed the agent. Only the important part of the code is shown here. For a complete listing of the agent, refer to Appendix D.1, “Domino Agent to the Enterprise Using MQSeries” on page 233.

Import Statements

We first load all the required packages needed to create our servlet:

- Java package, to write Java code
- PrintWriter package, to print to the browser
- Domino package, to use Domino backend classes
- MQSeries Java package, to connect to MQSeries

The following import statements perform the task:

```
import java.lang.*;
import java.io.PrintWriter;
import lotus.notes.*; // Include Notes package
import com.ibm.mq.*; // Include MQ package
```

MqDom2Ent Agent

Our class starts by extending the AgentBase class, and defining a few global variables for the class:

```
public class Mqdom2ent extends AgentBase {
   // Variables used by MQSeries
   String hostname = "ob.almaden.ibm.com"; // hostname to connect to
   String channel = "SYSTEM.DEF.SVRCONN"; // channel name used by client
   int iPort = 1414; // default port 1414
   String qManager = "DEF_QMNGR"; // queue manager to connect to
   String requestQueue = "MQAPPL.Q"; // queue to put request to
   String replyQueue = "MQAPPL.RQ"; // queue to get reply from
   MQQueueManager qMgr; // queue manager object
   MQQueue requestQ; // Request queue object
   MQQueue replyQ; // Reply queue object
```

NotesMain Method

The NotesMain method is the entry point to the functional code of the agent:

```
public void NotesMain() {
   try {
```

Notes Initialization

We first need to initialize a session with the Domino server. The Session class is the root of the Domino backend object containment hierarchy. We used the *getSession()* method of AgentBase as we are writing an agent.

AgentContext represents the agent environment, since the current program is running as an agent.

We assigned a java.io.PrintWriter object with the *getAgentOutput()* method of AgentBase, and write using the *println()* method of the PrintWriter object. This method works for output to Notes clients and Web browsers from agents.

For foreground agents, System.out and System.err output goes to the Java debug console. For locally scheduled agents, System.out and System.err output goes to the Domino log:

```
Session session = getSession();
AgentContext ac = session.getAgentContext();
Database db = ac.getCurrentDatabase();
Document doc = ac.getDocumentContext();
System.out.println("Document in use, UID: " + doc.getUniversalID());
PrintWriter pw = getAgentOutput();
```

Extract from the Domino Document

As this agent is triggered from an action on an opened document, we can extract required information from the document:

```
String partNumber = doc.getItemValueString("I_PARTNO");
String quantity = doc.getItemValueString("I_QUANTITY");
```

MQSeries Initialization

Before we can connect to a queue manager we must establish the *properties* of our MQEnvironment object. As Java does not have properties, these are referred to as static data members. The MQEnvironment object holds information equivalent to an MQSERVER environment variable setting and more:

```
MQEnvironment.channel  = "SYSTEM.DEF.SVRCONN";
MQEnvironment.hostname = "ob";
MQEnvironment.port = 1414;
```

A channel name must be specified. If the hostname and port are not set they default to *localhost* and port 1414, but no protocol specifier is defined as the Java client always communicates using TCP/IP. Other class members that can be set include user ID, password and CCSID.

Connecting to the Queue Manager

The creation of the MQQueueManager object and connection to it is then a straightforward step:

```
qMgr = new MQQueueManager(qManager);
if (qMgr.isOpen) {
   System.out.println("Connection to qmgr open");
}
```

Once our connection to the queue manager is established we can open queues, set put message options, put a message(s) to a queue(s), set get message options, get a message(s), close queues and disconnect from the queue manager, as we would with any other MQSeries program.

Opening the Queues

After setting options on the queues, we can open them:

```
int openOptionsOut = MQC.MQOO_OUTPUT;
int openOptionsIn = MQC.MQOO_INPUT_SHARED;
requestQ = qMgr.accessQueue(requestQueue,
                     openOptionsOut,
                     null, // default q manager
                     null, // no dynamic q name
                     null); // no alternate user id
replyQ = qMgr.accessQueue(replyQueue,
                     openOptionsIn,
                     null, // default q manager
                     null, // no dynamic q name
                     null); // no alternate user id
```

Creating the MQ Request Message Object

We are responsible for building the message we wish to put. We must build it in the correct order and use the appropriate data types. We should also set any required parameters of the MQMD. If, in our example, we need to take the requested part number from the document as character text and the required quantity as a short and put them in that order into our request message, we can code it in the following way:

```
MQMessage requestMsg = new MQMessage();
```

```
requestMsg.characterSet = 437;
requestMsg.encoding = 546;
requestMsg.writeString(partNumber);
if (quantity == null)
   requestMsg.writeShort(0);
else
   requestMsg.writeShort(Short.parseShort(quantity));
requestMsg.messageType = MQC.MQMT_REQUEST;
requestMsg.format = MQC.MQFMT_STRING;
requestMsg.replyToQueueName = replyQueue;
requestMsg.replyToQueueManagerName = qManager;
```

Putting the Message on the Request Queue

To put the message on the queue, we first set the message options accepting the default values and then use the *put()* method of the request queue object:

```
MQPutMessageOptions pmo = new MQPutMessageOptions();
requestQ.put(requestMsg, pmo);
```

Receiving a Message from the Reply Queue

To receive a message from the reply queue, we first define a buffer where to receive the message:

```
MQMessage replyMsg = new MQMessage();
```

To ensure that we only retrieve the message specifically intended to our request, we use the correlation ID:

```
replyMsg.correlationId = requestMsg.messageId;
```

We have to set the options for receiving the message such as how long to wait for the reply message:

```
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.waitInterval = 1000;
gmo.options = MQC.MQGMO_WAIT;
```

We can read the reply message from the queue and move the received information back to the document:

```
replyQ.get(replyMsg, gmo, 300);
String replyPartNumber = replyMsg.readString(3);
String replyQuantity = Short.toString(replyMsg.readShort());
String replyInventory = Integer.toString(replyMsg.readInt());
if (replyMsg.getDataLength() != 0) {
   String replyComments =
      replyMsg.readString(replyMsg.getDataLength());
   doc.replaceItemValue("I_COMMENTS", replyComments.trim());
   }
   doc.replaceItemValue("I_PARTNO", replyPartNumber);
   doc.replaceItemValue("I_QUANTITY", replyQuantity);
   doc.replaceItemValue("I_INVENTORY", replyInventory);
```

Saving and Displaying the Updated Document

Once the document has been updated, we can save it and display it back to the Web user:

```
doc.save(true, true);
String accessType = doc.getItemValueString("AccessType");
System.out.println("Access type is: " + accessType);
if (accessType.equals("WebClient")) {
   pw.println("[/" + db.getFileName() + "/All+Documents/"
               + doc.getNoteID() + "?OpenDocument]");
   }
```

Finally Block

To conclude our example, we use the *finally* block to close any open queues and disconnect from the queue manager:

```
finally {
   try {
      if (requestQ.isOpen)
         requestQ.close();
      if (replyQ.isOpen)
         replyQ.close();
      if (qMgr.isOpen) {
         System.out.println("Disconnecting from queue manager");
         qMgr.disconnect();
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

15.3.3 Remarks and Comments

In this section, we give you some remarks and comments on the development of an MQSeries Java program. They highlight some techniques that you may find interesting to use in your own application

Building the Message

We are responsible for building the message we wish to put. We must build it in the correct order and use the appropriate data types. We should also set any required parameters of the MQMD. If, in our example, we need to take the requested part number from the document as character text and the required quantity as a short and put them in that order into our request message, we can code it in the following way:

```
MQMessage requestMsg = new MQMessage();
requestMsg.writeString(partNumber);
requestMsg.writeShort(Short.parseShort(quantity));

requestMsg.messageType = MQC.MQMT_REQUEST;
requestMsg.format = MQC.MQFMT_STRING;
requestMsg.replyToQueueName = replyQueue;
requestMsg.replyToQueueManagerName = qManager;
```

Both the read and write methods support a range of data types. The range is wider than the equivalent LotusScript Read and Write methods provided in the MQMessage class of the MQLSX or the EIMessage class of the MQEI. The supported data types include:

- Byte

  A single byte or an array of bytes can be read from or written into the message buffer. This includes support for packed decimals.

- Boolean

  A boolean can be read from or written into the message buffer.

- Character

  A Unicode character or a sequence of Unicode characters (such as a string) can be read from or written into the message buffer.

- Float

  A float or double can be read from or written into the message buffer. The encoding data member determines the behavior of these methods.

- Integer

  An integer, either short or long, can be read from or written into the message buffer. The encoding data member determines the behavior of these methods.

- Object

  An object can be read from or written into the message buffer.

- String

  A string can be read from or written into the message buffer. The codeset identified by the characterSet data member determines whether conversion is required.

- UTF

UTF is an encoding of Unicode characters, and more generally the universal character set (UCS), and is used for transmission and storage. UTF stands for UCS transformation format.

CharacterSet and Encoding

The characterSet data member of the MQMessage class specifies the coded character set identifier of character data in the application message data.The behavior of the read and write methods using character data is altered accordingly. The default is to use the coded character set of the queue manager, that is to set the encoding to MQC.MQCCSI_Q_MGR.

Changing the value of the CCSID used, affects the way that the queue manager you connect to translates information in the MQSeries headers. If you change the client's CCSID to be the same as that of the queue manager to which you are connecting you can gain a performance benefit at the queue manager since it no longer attempts to translate the message headers. The default value for a Java client is "819" (ISO-8859-2).

We have mentioned another way of setting the characterSet used by the client, the MQEnvironment.CCSID data member. It is supposed to override the code page, but is not always effective. It can also leave the MQMD.characterSet field set to zero, so the server (and data conversion exit) has no way of finding out what the code page actually was. This is a bug, and should hopefully be fixed in MQSeries V5.1. Until then, we recommend that you not rely on using the MQEnvironment.CCSID.

Using the default CCSID can get you in trouble if you have clients with different CCSIDs all connecting to the same target. We declare the characterSet explicitly in our code to avoid any such problems.

The encoding data member specifies the representation used for numeric values in the application message data; this applies to binary, packed decimal and floating point data. The behavior of the read and write methods for these numeric formats is altered accordingly. The default is MQC.MQENC_NATIVE, which is made up of individual settings for binary integers, packed-decimal integers and floating-point integers.

The following encodings are defined for binary integers:

- MQC.MQENC_INTEGER_NORMAL

  This defines big-endian integers and by default is used by Java.

- MQC.MQENC_INTEGER_REVERSED

  This defines little-endian integers and is typically the default for other PC-based programs.

If no encoding is explicitly set the default value for a Java client is "273".

The characterSet of "819" and encoding of "273" are the natural values for Java, and the client doesn't really know what the server is. We found that when our part number and quantity were passed to the NT target with these default settings, the characters were handled without problem, but the short, for example a required quantity of "45" was seen as "002D" rather than "2D00". This is a result of a big-endian, little-endian mismatch.

To correct this problem, we set characterSet and encoding for our message prior to writing the message contents:

```
MQMessage requestMsg = new MQMessage();
requestMsg.characterSet = 437;
requestMsg.encoding = 546;
requestMsg.writeString(partNumber);
requestMsg.writeShort(Short.parseShort(quantity));
```

where characterSet of "437" and encoding of "546" are the values typically used by both our client NT and our target NT environments (and those that would automatically be used by a LotusScript agent running in the same Domino machine as our Java agent). Our encoding could also be set to MQC.MQENC_INTEGER_REVERSED. This would appear in the MQMD as an encoding of "2".

Reading the Reply

While your target application may well use the MQGMO_CONVERT option set to perform a get with convert, we do not recommend using the equivalent MQC.MQGMO_CONVERT in your Java programs. There is usually no need, as conversion is done by the read methods.

(The use of the MQGMO_CONVERT option is generally recommended in MQSeries. Using this option ensures that you only convert once, in case you pass through an intermediate server, and gives you the option to convert only the messages you want converted. (On MVS, you need MQSeries V1.1.4 or later to use the MQGMO_CONVERT option.))

Reading the contents of a reply message is similar to writing the contents of a request message, in that we must know where in the message the required information is and in what format. Using the read methods we can read in information from the message buffer. The setDataOffset(int offset), seek (int pos) and skipBytes(int n) methods allow us to move past information that isn’t required.

The target application required a 3-character string containing the part number and a short (2 bytes) containing the quantity as input. It returns those same two pieces of information at the beginning of the reply message, followed by an integer containing the inventory level (4 bytes) and a 101 character string containing comments. (There is a difference here between Java data types and those used by LotusScript. In LotusScript a long is defined as 4 bytes and an integer is only 2 bytes. This and other differences between Java and LotusScript are given in Appendix A, “Use of Java versus LotusScript” on page 215.) Our code, to read in the information and update the document was along the following lines:

```
String replyPartNumber = replyMsg.readString(3);
String replyQuantity = Short.toString(replyMsg.readShort());
String replyInventory = Integer.toString(replyMsg.readInt());
String replyComments = replyMsg.readString(replyMsg.getDataLength());

doc.replaceItemValue("I_PARTNO", replyPartNumber);
doc.replaceItemValue("I_QUANTITY", replyQuantity);
doc.replaceItemValue("I_INVENTORY", replyInventory);
doc.replaceItemValue("I_COMMENTS", replyComments.trim());
```

Publishing the Results

Java cannot be attached to front-end objects, as there is no equivalent to the LotusScript NotesUIDocument class. Saving the document allows Notes users to see the updated document when they refresh their view, but we cannot reload or refresh the document for them automatically using Java. For our Web browser user, we can however, publish the document back to them. So, once we have updated fields in our document, we add code along the following lines:

```
doc.save(true, true);

String accessType = doc.getItemValueString("AccessType");
System.out.println ("Access type is: " + accessType);
   if (accessType.equals("WebClient"))
   {
   pw.println ("[/"
               + db.getFileName()
               + "/All+Documents/"
               + doc.getNoteID()
               + "?OpenDocument]");
}
```

AccessType is a hidden field on our form containing the following code:

```
IsWebClient:=@IsMember("$$WebClient";@UserRoles);
@If(IsWebClient;"WebClient";"NotesClient")
```

Exception Handling

We catch any exceptions, looking for specific exceptions from MQSeries or Domino, before handling all other exceptions. An MQSeries class throws an MQException. We can check for a specific completionCode or reasonCode and identify the exceptionSource:

```
catch (MQException mqex) {
      System.out.println("An MQ error occurred:"+
         " Completion code " + mqex.completionCode +
         " Reason code " + mqex.reasonCode +
         " from: " + mqex.exceptionSource);
      if (mqex.reasonCode == 2085) {
         System.out.println("The explanation is: Unknown object name.");
      } else {
         if (mqex.reasonCode == 2033) {
            System.out.println("The explanation is: No message
available.");
         } else {
            System.out.println("No specific explanation available for this
error, please refer to the MQ Application Programming Reference");
         }
      }
```

15.4 From the Enterprise to Domino Using MQSeries

A Domino agent can be initiated in a number of ways (see Table 5 on page 164). Normally, however, an agent cannot be set to run as a result of an action taking place elsewhere. So, if we wish to make updates in a Domino application as a result of unsolicited information coming from an enterprise application, what can we do?

In this example we use the MQSeries Trigger Monitor for Lotus Notes agents (as discussed in 7.2.4, “MQSeries Trigger Monitor for Lotus Notes Agents” on page 76), to start our Java agent from outside of Domino. Using this modified trigger monitor program together with the MQSeries Client for Java, allows us to take information from an MQSeries message, sent from the enterprise application, and use it to create or update the relevant Domino document(s) in the appropriate Domino application(s).

We use *MQToDominoAppl*, a modified version of the previous example’s target MQSeries application. The *MQToDominoAppl* application is used to represent an enterprise application that notifies Domino of particular events. In our case, that event would be a change in inventory level for a part number. We can mimic this event-triggered notification by sending a request (PUT only) to MQToDominoAppl, which in turn sends a response containing the part number (3 characters), its inventory level (an integer) and an optional comment (101 characters) to the replyToQueue specified in the request message.

Additional MQSeries Definitions

In this example, the output from the application is sent to a triggered queue with the following associated MQSeries definitions:

```
DEFINE QLOCAL('MQAPPL.TQ') REPLACE +
          DESCR('Enterprise to Domino Queue') +
          BOQNAME('NOTES.BACKOUT.REQUEUEQ') +
          INITQ('MQAPPL.NOTES.AGENT.INITQ') +
          PROCESS('MQAPPL.NOTES.AGENT.PROCESS') +
          TRIGGER +
          TRIGTYPE(FIRST) +
          TRIGDATA('')

DEFINE QLOCAL('MQAPPL.NOTES.AGENT.INITQ') REPLACE +
          DESCR('Sample Triggered Notes Agent Initiation Queue')

DEFINE QLOCAL('NOTES.BACKOUT.REQUEUEQ') REPLACE

DEFINE PROCESS('MQAPPL.NOTES.AGENT.PROCESS') REPLACE +
          APPLTYPE(22) +
          APPLICID('mqjava.nsf Enterprise to Domino Agent') +
          USERDATA('')
```

In the process definition, the application type of "22" indicates that the application to be started is a Domino agent. (The curious setting of "22" is used to represent Domino agents, while other valid settings include CICS, OS2 and WINDOWSNT.).

The application identifier parameter specifies both the name of the agent database (*mqjava.nsf*) and the name of the agent (*Enterprise to Domino Agent*). The agent name must be exact, including any spaces. We would recommend cutting and pasting the agent name from Domino to your definitions file in order to ensure that it is correct.

The user data parameter can be used by the agent, but if not required can be left blank (single quotes with space between) or omitted from the definition.

Running the Trigger Monitor

In addition to creating these definitions for our queue manager, we must start the trigger monitor to detect the arrival of messages. The trigger monitor must run local to our Domino environment in order to start the agent. So, for our configuration we require an MQSeries client on our local machine to provide the client connection to the queue manager. To tell the system which channel to use and how to find it, we use the MQSERVER environment variable that defines the name of the channel, the communication method to be used, and the MQSeries server name. In our environment:

```
SET MQSERVER=SYSTEM.DEF.SVRCONN/TCP/ob
```

If more than one channel is required in your environment, for example, you need to connect to more than one queue manager, you should make use of the MQCHLLIB and MQCHLTAB environment variables to point to the client channel definition table.

We start the client version of the trigger monitor, specifying the initialization queue we defined (and if not the default queue manager, the queue manager):

```
runmqtnc [-m QMgr_name] -q MQAPPL.NOTES.AGENT.INITQ
```

Agent Setup

Our agent is created as a shared agent, with the run option of "Manually From Actions Menu" and "Run once (@Commands may be used)" specified. The code in our agent does not differ greatly from that in our first example. However, we need only open one queue (for input) and perform a GET with no proceeding PUT. (The full code from this example is given in Appendix D.2, “The Enterprise to a Domino Agent Using MQSeries” on page 237.) We also need to select the document to act on in a different way.

In our first example, the agent was called from a specific document and this document was then updated with the results of the inquiry. When we communicate only from the enterprise to Domino, the correct document or documents to create/update may vary depending on the content of the message. The selection of the document or documents can be done in several ways, for example, assuming the database is the same database as contains the agent:

- If you will always be using the exact same document:

  ```
  Document doc = db.getDocumentByUNID("UniversalId");
  ```

- If the document always exists in a particular view and is the first or only document in that view:

  ```
  View view = db.getView("View_name");
  Document doc = view.getFirstDocument();
  ```

- If there is a suitable key displayed in a sorted column of the view that can be used to select the correct document:

  ```
  String partnum = myMsg.readString(3);
  ```

```
Document doc = view.getDocumentByKey(partnum);
```

- If instead we are selecting multiple documents we use the DocumentCollection class to build a collection of documents that we can then update.

In our example, we have chosen to update a summary table in one document to show the inventory level for all part numbers.

CharacterSet, Encoding and Byte Alignment

The characterSet and encoding of the message we receive are set by the queue manager. In our case, these values are set to "437" and "546" respectively. We don’t have to set any values, although we could choose to explicitly set our encoding to "546" or MQC.MQENC_INTEGER_REVERSED ("2"). Setting the encoding to MQC.MQENC_INTEGER_NORMAL ("1") prior to reading the integer would result in a very different number being displayed in our document. Naturally, the setting of the encoding and characterSet are dependent on the nature of the sending enterprise resource.

Another factor controlled by the sending enterprise resource is the packing or byte alignment of the data. When we modified the original C program for use as the "enterprise sender" application in this example, we compiled it with different options by accident. When we then tried to receive the integer as the second piece of information in the message, we consistently saw a very different number in the document from that expected.

An inventory level of "5" was being displayed as "1280". Our integer is being returned in little endian encoding from the enterprise, that is as "05 00 00 00", but we were reading "00 05 00 00". This was not an encoding problem.The getMessageLength and getDataLength methods of the MQMessage object showed us that our 108-byte message structure was actually being received as a 112-byte message. We were in fact ignoring the last byte of our integer and reading a spurious byte at the front. Our C compiler had byte aligned the message structure. We may never have seen this problem if our part number had been set as 4 characters rather than 3!

Basically, you need to understand the structure and format of all information being exchanged. To resolve the problem we have two options:

1. Change the C compiler options on the "enterprise sender" program.

   This would be an easy option for our example, but might not be acceptable to an organization that chooses to byte align for other applications in their enterprise.

2. Reflect the aligned structure in the agent.

We can do this by reading in a dummy byte between the part number and the inventory level, or by using one of the mechanisms for moving past unrequired message data.

Starting the Agent

Once the document(s) is selected, the remaining operations are the same as in our first example. The difference here being that our agent is started by the arrival of a message on a specific queue. This might be due to an enterprise event as in our example here. Alternatively, you could choose to use this mechanism for processing replies generated by a Domino request. That is to say, rather than having an agent that sends a request message and waits a given period of time for a reply from the enterprise application, you could instead have two agents: one is started by the creation/update of a document, takes information from the document, sends the request message and finishes; the second is started by the arrival of the reply message and updates the document.

Chapter 16. Servlets

Servlets are Java-written programs that run on your Web server. These programs when triggered, usually through a URL, perform a task for you, for example fetch some data from an enterprise system, and typically return with a page of HTML. Parameters can be passed to your servlet through the URL.

The servlet technology created by Sun, is an extension to the standard Java language and you need to download the JSDK, from the JavaSoft Web site (`http://java.sun.com`), if you want to run or create your own servlets.

16.1 Domino Java Servlet Manager

Servlets are controlled by the Domino Java Servlet Manager which is part of the HTTP server task.

Using servlets with Domino has a number of advantages that may make them appropriate for your particular solution. They run on the Domino server which places no requirement on the Web browser to support Java. Servlets can be loaded and initialized when the Web server loads. Once loaded they are held in memory, running in the server's JVM process, and subsequently are triggered extremely quickly. This gives you a performance benefit over CGI programs or Domino agents which must be reloaded.

Support for Java servlets has been available as part of Domino from Version 4.6. The configuration details for setting up the servlet manager supplied with Domino differ a fair amount between R4.6.x and R5. The steps required to set up the servlet manager are given below, and generally things are a little easier with Domino R5.

We explain how to configure the servlet manager in “Java Servlet Support” on page 99 for Domino R4.6, and in “Servlet Manager” on page 104 for Domino R5. Domino R5 Java servlet manager is based on Sun's JSDK 2.0.

In addition, Domino R5 also supports the use of a third party servlet manager. We provide details of how to use the servlet manager that is supplied with IBM's WebSphere product instead of the default Domino servlet manager in “Servlet Manager” on page 125.

Loading/Unloading Servlets

To unload a servlet from memory, you must stop and restart the HTTP task on the server, using the following commands:

```
Tell http quit
```

```
load http
```

The *restart* command forces a refresh of settings in the server document that relate to the servlet manager, or HTTP task, but it doesn’t destroy already loaded servlets. It’s important to remember this when your developing your servlet, to ensure your using the servlet you have just modified, and not one that is still cached in memory.

16.2 Structure of a Servlet

To write a servlet, you extend the Servlet class and define Java methods for establishing and managing connections. Servlets offer valuable features such as thread-safe code, automatic memory management, and built-in networking support.

HTTP Servlet Class

The *javax.servlet.http* package provides interfaces and classes for writing http servlets. The *HttpServlet* class () contains the *init()*, *destroy()*, and *service()* methods. The *init()* and *destroy()* methods are inherited. You can override the class methods you need to create your own servlet.

init() Method

The *init()* method executes only once when the server loads the servlet. You can configure the server to load the servlet when the server starts or when a client first accesses the servlet. The *init()* is not repeated regardless of how many clients access the servlet. The default *init()* method is usually adequate but can be overridden typically to manage servlet-wide resources. For example, you might write a custom *init()* to load GIF images only one time, improving the performance of servlets that return GIF images and have multiple client requests. Another example is initializing a database connection. This method will not be called again. The *init()* method is guaranteed to complete before the service() method is called.

destroy() Method

The *destroy()* method executes only once when the server stops and unloads the servlet. Typically, servlets are stopped as part of the process of bringing the server down. The default *destroy()* method is usually adequate, but can be overridden, typically to manage servlet-wide resources. For example, if a servlet accumulates statistics while it is running, you might write a *destroy()* method that saves the statistics to a file when the servlet is unloaded. Another example is closing a database connection. When the server unloads a servlet, the *destroy()* method is called after all *service()* method calls complete or after a specified time interval. Where threads have been spawned from within the *service()* method, and the threads have long-running operations, those threads may be outstanding when the *destroy()* method is called. Because this is undesirable, make sure those threads are ended or completed when the *destroy()* method is called.

service() Method

The *service()* method is the heart of the servlet. Unlike *init()* and *destroy()*, it is invoked for each client request. In HttpServlet, the *service()* method already exists.The default service function invokes the *do* function corresponding to the method of the HTTP request. For example, if the HTTP request method is GET, *doGet()* is called by default. A servlet should override the *do* functions for the HTTP methods that the servlet supports. Because the *HttpServlet.service()* method checks whether the request method calls the appropriate handler method, it is not necessary to override the *service()* method. Only override the appropriate *do* method.

16.3 Writing a CICS Java Program

At the simplest level, the flow of program control needed to write a simple CICS Transaction Gateway Java-client program is as follows:

1. The Java program creates and opens an instance of an *com.ibm.ctg.client.JavaGateway* object.

   The default JavaGateway constructor creates a blank JavaGateway object. You must then set the correct properties in this object using the relevant set methods. The JavaGateway is then opened by calling the open method.

2. The Java program creates an instance of one of the gateway request classes containing the request that it wishes to make, that is:

   - A *com.ibm.ctg.client.ECIRequest* is created for an ECI request.
   - A *com.ibm.ctg.client.EPIRequest* is created for an EPI request.
   - A *com.ibm.ctg.client.CicsCpRequest* is created for querying the code page of the CICS Universal Client it is connected through.

3. The Java program then flows the request to the CICS Transaction Gateway using the flow method of the JavaGateway object.

4. The Java program checks the return code of the flow operation to see whether the request was successful.

5. The program continues to create request objects and flow them through the JavaGateway object, as appropriate.

6. The Java program then closes the JavaGateway object.

16.4 CICS Connected Servlet

In the ExampleServletV2 servlet, data is drawn from both a Domino database and a CICS transaction and presented back to the user in the browser.

The servlet takes the employee number parameter (*Empno*) from the URL used to trigger the servlet. The parameter is used to fetch the related document from a Notes database, by using it as a key in a view. The document is then displayed in a frame on the screen.

The servlet then uses the Surname field returned from the Notes document as a parameter to a CICS program to fetch further information about the person, including address and credit rating data and this information is displayed in the remaining frame.

16.4.1 Development Environment Setup

The code makes use of IBM's CICS Java Gateway, which is now included as part of the CICS Transaction Gateway. The JAR files that make up the CICS Java gateway must be included in the JavaUserClasses entry in the Notes.ini.

```
JavaUserClasses=
   C:\Program Files\IBM\CICS Transaction Gateway\classes\ctgclient.jar;
   C:\Program Files\IBM\CICS Transaction Gateway\classes\ctgserver.jar
```

It is not sufficient to use the CLASSPATH setting in the Servlet section of the server document. Since these JAR files make reference to external libraries, and the security employed by the servlet manager disallows that, you must load the class files this way; otherwise, you will receive errors when trying to use the CICS Java Gateway.

16.4.2 Writing the Servlet

In the following section we explain how we developed the servlet. Only the important part of the code is shown here. For a complete listing of the servlet, refer to Appendix B, "Servlet Example" on page 223.

Import Statements

We first load all the required packages needed to create our servlet:

- Java package, to write Java code
- Java servlet extensions packages, to create an HTTP servlet
- Domino package, to use Domino backend classes
- CICS package, to connect to CICS

The following import statements perform the task:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import lotus.domino.*;
import com.ibm.ctg.client.*;
```

To access the local Domino server from the servlet we imported the *lotus.domino* package into the Java code. This enabled us to use the Domino backend classes from within the servlet. You can create servlets that access a Domino R5 server remotely through the CORBA-enabled Java classes. To support remote calls, the Domino server must be running the HTTP and DIIOP server tasks.

CICSServlet

Our class starts by extending the HttpServlet class, and defining a few globals for the class:

```
public class CICSServlet extends HttpServlet {
   JavaGateway jgaConnection;
   ServletContext context;

   private static int CICS_COMMAREALENGTH = 131;
   ......
```

init Method

The *init()* method is called once when the Web server loads the servlet. In this method, we open the connection to the Java Gateway.

The Java program creates and opens an instance of a *com.ibm.ctg.client.JavaGateway* object. The default JavaGateway constructor creates a blank JavaGateway object. We then set the correct properties in this object using the relevant *set* methods.

In our sample we used the local transaction gateway (*local:* protocol). The local JavaGateway object communicates directly to a locally installed CICS client. No network connection is used between the Java program and the CICS client, and no CICS Transaction Gateway is required to be running locally. Given the requirement to communicate to a locally installed CICS client, a local JavaGateway object is generally only applicable for use in a Java application or servlet.

We finally open the JavaGateway.

We also turn on the CICS tracing if we need it for debugging. The output is placed in the miscellaneous events section of the Notes log:

```
public void init (ServletConfig config) {
try {
   com.ibm.ctg.client.T.setDebugOn( true );
   com.ibm.ctg.client.T.setTimingOn( true );

   jgaConnection = new JavaGateway();
   jgaConnection.setURL( "local:");
   jgaConnection.open();
}
```

doGet Method

The *doGet* method is run on response of a GET request from HTML. The *doGet* method is run every time the Domino server receives a trigger URL for the servlet.

In this method we use the *getParameter* method of the *HttpServletRequest* class to fetch the parameter from the URL.

The *doGet* method then uses this parameter as a key to find a document in the Domino database.

If the document has been found, the *doGet* method call the CICS program using the *PerformCICSTransaction* method:

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response) throws IOException {
   try {
         ServletOutputStream out = response.getOutputStream();
         response.setContentType("text/html");
         // Kick off Notes
         NotesThread.sinitThread();
         // Get the value of the EmpNo variable from the URL
         String empKey = request.getParameter("EmpNo");
         if(empKey != null) // As long as we have a value to work with
         {
            // Open up the database and get the view we want
               Session s = NotesFactory.createSession();
               Database db = s.getDatabase("","test.nsf");
               View view = db.getView("CustomerLookupView");
            // Look for our key
               Document doc = view.getDocumentByKey(empKey, true);
            if (doc != null)
            {
               // Go get the data from CICS,
                performCICSTransaction(doc,out);
                // And return results to the browser
                createReturnHTML(doc,out);
             }
             ......
             ......
}
```

performCICSTransaction Method

Input to a CICS transaction is made through the CICS communication area (COMMAREA). The COMMAREA is the data area that can be passed to CICS programs when the programs are called by another program. The calling program—in our sample, the Java servlet— uses the ECI API.

In this method, we first fetch the *Surname* field from the NotesDocument object passed to us from the *doGet* method, and build up our COMMAREA with the *createCommArea* method described above.

We build an *ECIRequest* object. An ECIRequest object defines all the details of an ECI call to a CICS server. It defines which CICS server to make the request to, the CICS user ID and password to use, the CICS program to be run on the server, and the COMMAREA to pass to CICS.

We flow the ECIRequest object to the gateway or the client using the JavaGateway.flow method.

If we have results back from CICS to process we then place this information onto our NotesDocument:

```
public void performCICSTransaction(Document notesDoc,
                                 ServletOutputStream out) {
   ECIRequest eciRequest = null;
   try {
         .....
         // Get the surname to pass to the CICS transaction
         String lookName = notesDoc.getItemValueString("Surname");
         // Build up the CommArea that we will pass to CICS
         byte [] commArea = createCommArea(lookName);
      // Create our eciRequest
         eciRequest = new ECIRequest("CICSOS2",// CICS Server
                            "SYSAD",     // UserId, null for none
                            "SYSAD",     // Password, null for none
                            "VSAMSERV",   // Program name
                            commArea,  // Commarea
                            ECIRequest.ECI_NO_EXTEND,
                            ECIRequest.ECI_LUW_NEW);
         eciRequest.Cics_Rc = 0;
```

```
// Call CICS
jgaConnection.flow(eciRequest);
// If the CICS transaction happened OK
// then sort out the returned CommArea into our NotesDocument
if (eciRequest.Commarea != null) {
 ......}
```

Supporting Methods

We have also created some supporting methods that are used in the servlet. Here is a brief description of the methods:

- *createCommArea* method

  Format the COMMAREA exchanged with the CICS program.

- createReturnHTML method

  Put together the HTML that we return to the user.

- outputSimpleTableLine method

  Output a table row (with 2 columns) in HTML. This is called by createReturnHTML.

16.4.3 Triggering the Servlet

Once the servlet is completed you have to copy it to the Domino/Servlet folder (or wherever you have set the servlet home in the server document).

If your servlet is part of a named Java package then include the package name as new folder under your Servlet home folder. For example, a servlet named MyServlet.class which is part of a TestPackage Java package, should be placed in a folder as follows:

```
(My Servlet Folder)/TestPackage/
```

If you don’t do this you will receive *Wrong Name* errors on the Domino server console.

Finally if you wish to pass initialization parameters, provide aliases for your servlet, or dictate that the servlet should be loaded on startup of the servlet manager, you must set your *servlets.properties* file appropriately.

In our sample, we trigger the servlet using the following URL that passes an employee number:

```
www.myserver.com/Servlet/ExampleServlet2.ExampleServletV2?EmpNo=100
```

Chapter 17. A Comparison

In this chapter we compare the selection criteria for different types of Java programs. We also compare the use of Java and LotusScript for developing Domino applications.

17.1 Java Program Types

In this section, we summarize the criteria for the different Java programs that you can use:

- Applet
- Servlet
- Application
- Agent

In Table 6 we have highlighted these criteria and make some recommendations as to when to use each type.

Table 6. Comparison of Java Program Types

Function / Program	Applet	Servlet	Application	Agent
Supported by:				
Web browser	Yes	Yes (through URL)	No	Yes (through URL)
Notes client	Yes	No	No	Yes
Java application	No	No	Yes	No
Domino server	No	Yes	Yes	Yes
Other HTTP server	No	Yes	Yes	No
Access to DOM:				
Web browser	No (through URL)	N/A	N/A	No (through URL)
Notes client	No (through URL)	No	Yes	Yes
Java application	N/A	Yes	Yes	N/A
Domino server	N/A	Yes	Yes	Yes
Other HTTP server	N/A	Yes	Yes	N/A
Methods for Starting:				

Function / Program	Applet	Servlet	Application	Agent
Manually	No	Yes (using WebSphere)	Yes	Yes
Scheduled	No	No (exec could be used)	No (exec could be used)	Yes
URL	Yes?	Yes	No	Yes
HTML tag	Yes	Yes	No	No
Other event	No	No	No	Yes (including MQSeries Trigger Monitor for Lotus Notes agents)
Access enterprise	Yes (if signed)	Yes	Yes	Yes
Security	Signed applets offer increased access, while still providing security	Run with privilege of server		Run with privilege of given user or agent signer and ACL can be applied
Tracking	User defined. With CORBA and R5 has access to lotus.domino.log class	User defined. Has access to R4 lotus.notes.log class or R5 lotus.domino.log class	User defined. Has access to R4 lotus.notes.log class or R5 lotus.domino.lo g class	Notes log automatically tracks agents if appropriate setting is used ^{1}
Recommended use	Simple user oriented programs, that can be offloaded from the server	High volume Web or intranet situations. Servlets are cached by the servlet manager, so can be very responsive	When customized user interface required or program needs to be executable outside context of any other application or server	As with servlets, but where more robust security or better integration with Domino services is required. Agents are, however, only loaded when required
Notes: 1 - Set *LOG_AGENTMANAGER=1* in the *notes.ini* initialization file				

17.2 Use of Java versus LotusScript

This section provides some comparisons between the use of Java and the Lotus-unique language LotusScript.

Our experience in writing with Java, when coming from a background of LotusScript, shows that the two have great similarities. Java Notes classes parallel the LotusScript Notes backend classes for the most part. Although Java does provide multithreading support and additional access to external network Java classes (for example TCP and sockets).

You can use the Java classes from any Java program, within the Notes Designer environment or outside of it, although the requirement with Java is for Domino R4.6 or later. LotusScript can only be used in agents or other design elements of Domino as it is a language embedded in Domino.

Java programs are generally written in blocks, that is sequences of statements: a try block followed by one or more catch blocks (catching exceptions thrown by the try block) and possibly finishing with a finally block. This is not dissimilar from the structure in a LotusScript agent, if you consider the try block as the *initialize* event, the catch blocks as the *event handlers* and the finally block as the *terminate* event. However, in Java a try block and its associated catch blocks can follow another try block and catch block(s) or be nested within another try block.

The similarities between LotusScript and Java can cause you to expect everything to be approached in the same way and that is not the case. Some of the differences are purely syntactical, while others reflect the different origins of the two languages: Java evolving from C++, and LotusScript from BASIC.

Some simple examples of coding differences would include:

- A Java program is generally made up of a collection of (class) files. The file designated as the base class, is the starting point for the agent program.
- Even if several methods are included in the same file, Domino will look to begin code execution in a method called NotesMain.
- Java is very case sensitive! The name of the class, the variable name, the method name and URLs must all be written exactly.
- Java programs cannot be attached to front-end objects. That is to say, there is no equivalent to the LotusScript NotesUIDocument class. One noticeable consequence of this, is that it is not possible to refresh the

document while open on the Notes client. Instead the user must close the document and refresh the view.

- It is not possible for Java programs to be used in form events in the way LotusScript programs can be used. This and the fact that Java has no access to front-end objects, make it a less popular choice for a Notes client-based application.
- Another interesting comparison is that of access to the Domino object classes or Notes object interface (NOI) as it was known in R4.6. This is illustrated in Table 7 on page 192.

Table 7. Comparison of Domino Object Classes Access

Program	Java Client	LotusScript Client	Java Server	LotusScript Server
Applet	No	No	No	No
Servlet	No	No	Yes	No
Application	Yes	No	Yes	No
Agent	Yes	Yes	Yes	Yes

- Java only supports methods, while LotusScript provides classes with both methods and properties. In order to inquire on or set what would otherwise be properties of an object, Java implements methods of getXXX and setXXX:

  ```
  db.setTitle = "Set the title of the database";
  String dbTitle = db.getTitle();
  ```

  The empty parentheses are required on the getXXX as it is a method and all methods are followed with parentheses that may contain arguments.

- Java data types are not necessarily the same as the LotusScript data types; for example, Table 8 on page 192 shows us the three numeric data types that might commonly be used in a Domino agent:

Table 8. Data Types

Data Type	Java	LotusScript
Short	2 bytes	Not defined (implemented as 2 bytes in MQLSX, MQEI)
Integer	4 bytes	2 bytes
Long	8 bytes	4 bytes

- Variants used in LotusScript to contain any data type (including arrays and object instances) are not available in Java. Java instead supports method name overloading, where multiple methods share the same name. Overloaded methods are differentiated by the number and type of the arguments passed into the method. Arrays can be returned using the java.lang.Vector class, which uses the Object type.

- The Java "if" statement needs a boolean expression, for example:

  ```
  if (replyQ.isOpen)
  if (replyComments.equals(""))
  ```

- In LotusScript the NotesDocument class is an expanded class and form fields can be used there as properties of the document, leading to easy manipulation such as:

  ```
  doc.StatusField = "Loan approved"
  ```

  Java does not support expanded classes.

Part 4. Domino and WebSphere

For the time being, Enterprise JavaBeans (EJBs) and CORBA are considered by many analysts as the most promising and best technologies for implementing multitiered, enterprise-level, server-based, cross-platform applications.

In this part we look at the use of WebSphere with Domino and the advantages that can be gained.

WebSphere's support for Domino allows Java servlet applications to be written to run in either environment and allows the same base Java servlet code to execute in both products. Applications can also call back and forth between the WebSphere and Domino environments through the use of Java. In addition, the capabilities of Domino and WAS can be combined in a *server farm*, where multiple Web servers work together transparently to fulfill the needs of a single Web-based solution.

Together, WAS and Domino offer a comprehensive range of Web application server environments that support business applications from simple Web publishing through enterprise-scale transaction processing and collaborative business solutions.

WAS reflects the evolution from traditional Web servers that served HTML pages to a Web application server serving industrial-strength business applications. In order to set Domino and WAS up to work together, we should first understand the three tiers of WAS:

- HTTP engine

  Handles HTTP requests including requests for CGI programs, GIF files and HTML files. This tier can be scaled to run multiple engines in a cluster feed by an HTTP *sprayer* (or load balancer). The HTTP server itself can be selected from a list of supported servers:

  - Lotus Domino R5
  - Lotus Domino Go Webserver
  - Netscape Enterprise Server
  - Netscape FastTrack Server
  - Microsoft Internet Information Server
  - Apache Server

  In our configuration, WAS uses the Domino HTTP engine. Servlets requests are authenticated and authorized by the HTTP engine, but are then passed to the Java servlet engine.

- Servlet manager

  The Websphere Java Servlet engine plugs into the Domino HTTP engine using proprietary plug-in APIs. The engine manages servlet requests and passes the data back to the client. The engine also handles requests for JSP or server-side HTML scripting. Handling applications and dynamic content requests requires multiprocess support to ensure an industrial-strength solution. The Websphere Java Servlet engine supports a single process until it is required to provide support for multiple Java processes, at which time it adds servlet queues. As the WAS administrator, you determine how many queues can be established and define a policy for the use of each queue.

- Enterprise JavaBeans Server

  The Enterprise JavaBeans Server runs the business logic and ensures transactional integrity, with Java transactional services (JTS) and the Java naming and directory interface (JNDI).

In the following chapters we explain:

- How to use the WAS Servlet Engine with Domino
- How to access EJBs from a Domino application

Chapter 18. WAS Servlet Manager

We explained in 12.2.2, “Servlet Manager” on page 125 how to configure the WAS servlet manager to be an alternate servlet manager for Domino. In this configuration, we use the WAS servlet manager to manage our servlets. In Chapter 16, “Servlets” on page 181 we explained how to develop a CICS servlet called *ExampleServletV2*. In this chapter we use the WAS servlet manager to manage our servlet.

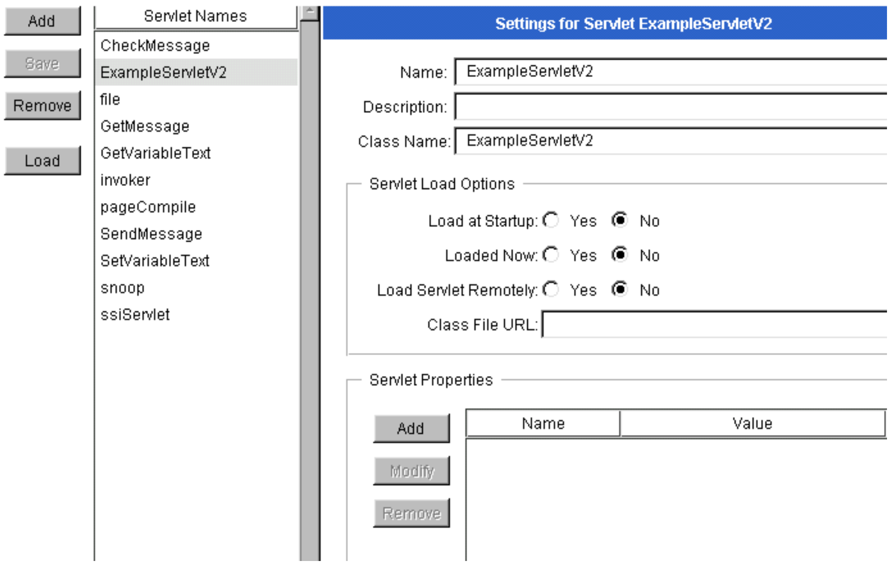
18.1 Installation

By default, WAS looks for servlet class files in the servlet root directory. Copy your compiled servlet class files to that directory. You can load servlets from an alternate servlet directory using a reloadable servlet directory. If the servlets are in a package, mirror the package structure as subdirectories under the servlet or reloadable servlet directory. If your servlets import additional classes that you have developed, it is recommended that you copy those classes to the servlet directory.

We copied the *ExampleServletV2* servlet into the servlet root directory.

18.2 Settings

If you want to set servlet initialization parameters, use the WebSphere Application Server Manager to configure the servlet (see Figure 45 on page 198).



Servlet Names
- CheckMessage
- ExampleServletV2
- file
- GetMessage
- GetVariableText
- invoker
- pageCompile
- SendMessage
- SetVariableText
- snoop
- ssiServlet

Settings for Servlet ExampleServletV2

Name: ExampleServletV2

Description:

Class Name: ExampleServletV2

Servlet Load Options

Load at Startup: Yes [✓ No]

Loaded Now: Yes [✓ No]

Load Servlet Remotely: Yes [✓ No]

Class File URL:

Servlet Properties

Name	Value

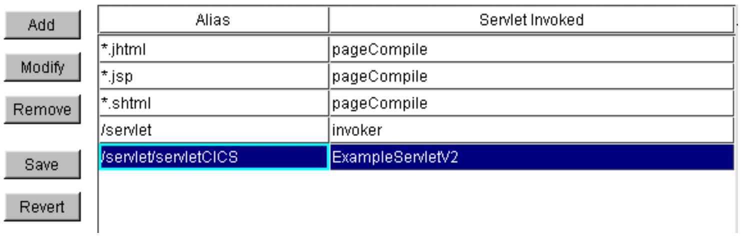
Figure 45. Servlet Configuration Settings

In this panel, you can define the settings of your servlet such as:

- The unique name of the servlet
- A text describing the servlet
- The associated class file for the servlet
- Whether the Web server loads the servlet when the server starts
- Whether the Web server should load or unload the servlet immediately
- The servlet properties

18.3 Alias

To associate servlets with aliases by which they will be known, use the Aliases page. After setting up a servlet alias, you can connect to the servlet by entering the alias into a server-side include or URL. Aliases are particularly useful for servlet chaining. Figure 46 on page 199 shows how we defined the *servletCICS* alias for the *ExampleServletV2* servlet. The alias must begin with */servlet/*.



Alias	Servlet Invoked
*.jhtml	pageCompile
*.jsp	pageCompile
*.shtml	pageCompile
/servlet	invoker
/servlet/servletCICS	ExampleServletV2

Figure 46. Servlet Alias

18.4 Security

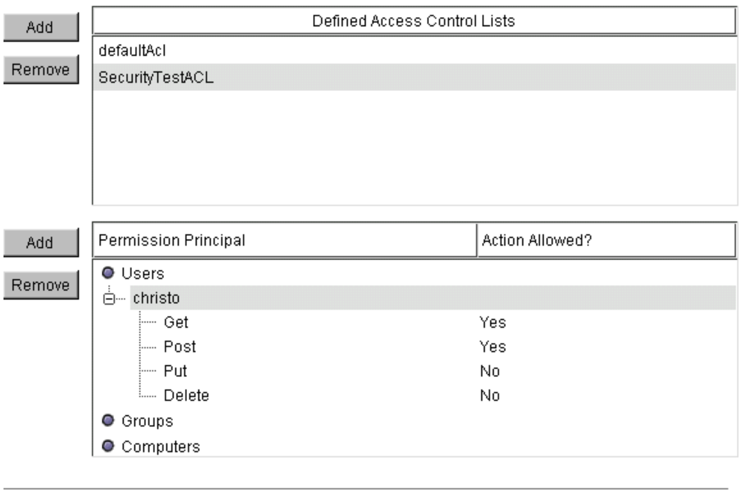
WAS provides a servlet security feature. You can create a user group, a set of users, and an access control list, and then add a servlet to the list of resources protected by that access list.

If the users are already defined in Domino, you don't need to create new IDs in WebSphere. WebSphere can reference the Domino users using LDAP.

Once a servlet has been configured in WAS, you can create Users, User Groups, Access Control, and declare the servlet as a Resource.

To test this security feature, you have to declare the users in WAS using a user ID and a password. You may also create user groups that contain multiple users.

You also can create access control lists (ACLs). Figure 47 on page 200 shows how we created the *SecurityTest* ACL.



Defined Access Control Lists
defaultAcl
SecurityTestACL

Permission Principal	Action Allowed?
● Users	
christo	
Get	Yes
Post	Yes
Put	No
Delete	No
● Groups	
● Computers	

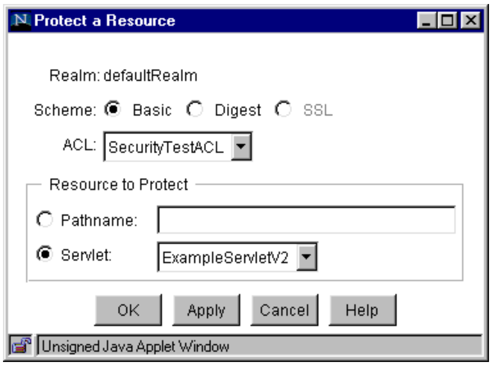
Figure 47. Access Control List

The *SecurityTestACL* ACL contains one user, *christo*. You can also add groups of users and computers. For every member of the ACL, you assign permissions. You can assign permissions for "files and folders" and "Servlets":

- The Servlets selection refers to permissions that an executing servlet has, such as opening sockets, reading files, etc.
- The files and folders selection sets permissions to access a servlet, such as GET and POST.

In our example, *christo* has permission to use the GET and POST methods on files, folders, and by extension, servlets that are protected by this ACL.

We used the Resource page to protect resources by assigning resources to an access control list that specifies who can use the resource. Figure 48 on page 201 shows how we set the protection of the *ExampleServletV2* servlet using the *SecurityTestACL* ACL. You can also protect a directory resource using its full path name.


Protect a Resource

Realm: defaultRealm

Scheme: [✓ Basic] Digest SSL

ACL: SecurityTestACL

Resource to Protect

Pathname:

[✓ Servlet:] ExampleServletV2

Figure 48. Protect a Resource Setting

When using a Web browser you enter the URL of the protected resource, then WAS prompts you for a user ID and a password. In our sample, only user *christo*, is allowed to access the servlet.

Chapter 19. WAS Enterprise JavaBeans Server

In this chapter we describe how to create a Domino agent and a Domino servlet that access an Enterprise JavaBean managed by WebSphere.

We first introduce the *Employee* EJB used during the test.

19.1 Employee EJB

In our test, we created a Domino application that could access an existing EJB shipped with WAS. The *Employee* EJB is part of the Phone book sample application.

Employee EJB is a container-managed persistence entity bean that is mapped onto the Employee table included as part of the Sample database distributed with the DB2 product. The Employee table contains over a dozen different columns, of which only six are used by the Employee bean. The Phone sample shows how the SQL statements in the persister class generated by the stand-alone deployment tool can be modified in a straightforward manner to accommodate the actual data contained in an existing DB2 table.

The Employee EJB is a container-managed bean; that is, it does not have any persistence code in the bean's method. Instead the container provider's tools have generated the necessary functions at deployment time and implemented them in the container. The *Employee* EJB is completely independent from the data store.

Our Domino application is just a client application that integrates existing EJBs. To access EJBs, clients need to be able to access the container that contains the Home and the Remote interfaces.

The client-side programming model for accessing EJBs is described in the EJB specification. These classes have the following characteristics:

- They hide the low-level complexity of accessing an EJB.
- They are JavaBeans (EJBs are not JavaBeans - they share the JavaBean name because they are component models for Java, not because they have anything in common technically).
- They can be used as session EJBs.

In the design of cooperative or composite EJB applications, there are programming dependencies between the EJBs that make up the applications.

In particular, it is necessary for the business logic in the implementation of some EJBs to have access to the client interfaces, stubs, and helper classes of other EJBs. Currently, you stage the development of the EJBs in order to make the stubs and helpers available to the other EJBs.

Home Interface

The Home interface of the *Employee* EJB provides information which is used by the container. The EJB provider has to supply:

- Extensions to the javax.ejb.EJBHome interface
- A create method which is used to create another session bean
- A finder method used by entity beans to identify a database
- Related exceptions

Our Domino application—the client application—uses the JNDI to look up the name of the *Employee* EJB class in the name space on the server using JNDI.

Here is the Home interface of the *Employee* EJB (*EmployeeHome.java*):

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.util.Enumeration;
import java.rmi.RemoteException;
public interface EmployeeHome extends EJBHome {
   public Employee create ( EmployeeKey employeeKey )
          throws RemoteException, CreateException;
   public Employee findByPrimaryKey ( Object employeeKey )
          throws RemoteException, FinderException;
   public Enumeration findByLastName ( String lastName )
          throws RemoteException, FinderException;
   public Enumeration findByDepartment ( String department )
          throws RemoteException, FinderException;
   public Enumeration findAll ( )
          throws RemoteException, FinderException;
}
```

In the example, the following methods have been extended:

- create
- findByPrimaryKey
- findByLastName
- findByDepartment

- findAll

Remote Interface

The Remote interface defines the business methods for use in the EJB. The EJB defines and extends the existing javax.ejb.EJBObject interface. The EJB provider must:

- Extend the javax.ejb.EJBObject interface.
- Provide a corresponding method in the EJB class for each method defined in the remote interface. For example, the matching method must have the same name, number, and types in its arguments and the same return type, as well as the relevant exceptions.

Here is the Remote interface of the *Employee* EJB (*Employee.java*):

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Employee extends EJBObject {
   public String getEmployeeNumber ( ) throws RemoteException;
   public String getLastName       ( ) throws RemoteException;
   public String getFirstName      ( ) throws RemoteException;
   public String getMiddleInitial  ( ) throws RemoteException;
   public String getDepartment     ( ) throws RemoteException;
   public String getPhoneNumber    ( ) throws RemoteException;
}
```

In this remote interface six business methods have been added:

- getEmployeeNumber
- getLastName
- getFirstName
- getMiddleInitial
- getDepartment
- getPhoneNumber

In the following sections, we show how to access this *Employee* bean from a Domino agent and from a Domino servlet.

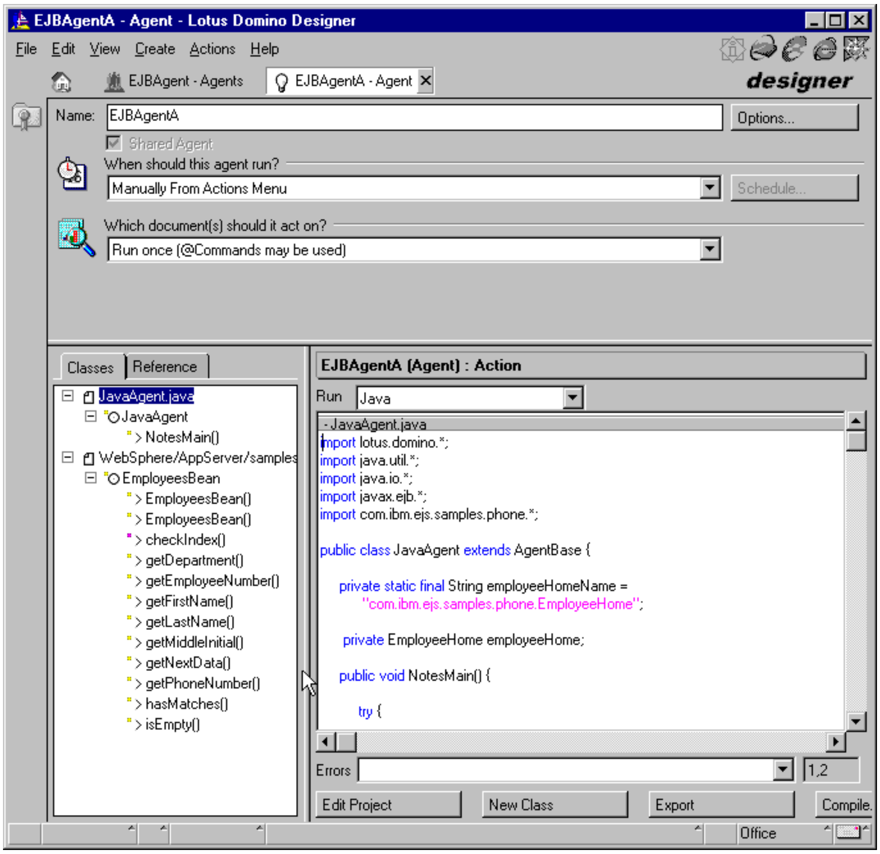
19.2 Domino Agent

Using Domino Designer R5, we created the *EJBAgentA* agent that accesses the *Employee* EJB.

Our *EJBAgentA* Domino agent uses the *JavaAgent* base class that extends the AgentBase class.

19.2.1 Settings

Figure 49 on page 206 shows how we created the *EJBAgentA* Domino agent.



Name: EJBAgentA

Shared Agent

When should this agent run? Manually From Actions Menu

Which document(s) should it act on? Run once (@Commands may be used)

```
JavaAgent.java
    JavaAgent
        NotesMain()
WebSphere/AppServer/samples
    EmployeesBean
        EmployeesBean()
        EmployeesBean()
        checkIndex()
        getDepartment()
        getEmployeeNumber()
        getFirstName()
        getLastName()
        getMiddleInitial()
        getNextData()
        getPhoneNumber()
        hasMatches()
        isEmpty()
```

Run: Java

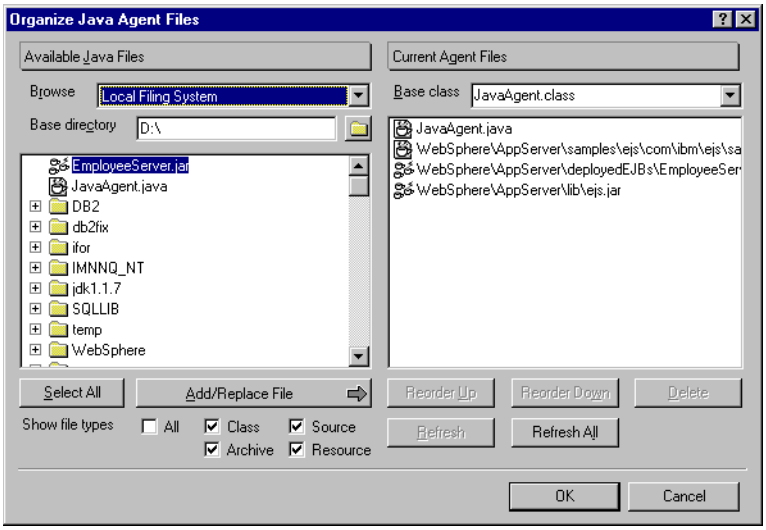
```
- JavaAgent.java
import lotus.domino.*;
import java.util.*;
import java.io.*;
import javax.ejb.*;
import com.ibm.ejs.samples.phone.*;

public class JavaAgent extends AgentBase {

    private static final String employeeHomeName =
        "com.ibm.ejs.samples.phone.EmployeeHome";

    private EmployeeHome employeeHome;

    public void NotesMain() {

        try {
```

Errors: 1,2

Figure 49. EJBAgentA Agent

Using Domino Designer, we created a new Java agent. This agent can run on the Domino server or the Notes client as long as the core EJB classes are accessible from the agent. In our example we import all the requested files in the document.

To specify the packages that are required by this agent, we click on the **Edit Project** button. Figure 50 on page 207 shows the windows where you can specify all the agent files.



Browse: Local Filing System

Base directory: D:\

EmployeeServer.jar
JavaAgent.java
DB2
db2fix
ifor
IMNNQ_NT
jdk1.1.7
SQLLIB
temp
WebSphere

Show file types: ☐ All ☑ Class ☑ Source ☑ Archive ☑ Resource

Base class: JavaAgent.class

JavaAgent.java
WebSphere\AppServer\samples\ejs\com\ibm\ejs\sa
WebSphere\AppServer\deployedEJBs\EmployeeSer
WebSphere\AppServer\lib\ejs.jar

Figure 50. EJB Classes Imported

We have to import the following files:

- *JavaAgent.java*, added automatically by Domino Designer, containing our agent source classes
- *EmployeeBean.java*, to ease the development by adding method definitions in the IDE
- *EmployeeServer.jar,* the deployed EJB
- *ejs.jar*, core EJB classes

19.2.2 Writing the JavaAgent Agent

In the following we explain how we developed the agent. Only the important part of the code is shown here. For a complete listing of the agent, refer to Appendix E.1, “EJBAgent” on page 241.

Import Statements

We first load all the required packages needed to create our agent:

- Java package, to write Java code
- Domino package, to use Domino backend classes
- EJB package, to use EJB classes
- *Employee* EJB package, to access the EJB

The following import statements perform the task:

```
import lotus.domino.*;
import java.util.*;
import java.io.*;
import javax.ejb.*;
import com.ibm.ejs.samples.phone.*;
```

JavaAgent Agent

Our class starts by extending the AgentBase class.

We define a private variable for the type of EJB and define a string containing the full name of the EJB:

```
public class JavaAgent extends AgentBase {
   private static final String employeeHomeName =
           "com.ibm.ejs.samples.phone.EmployeeHome";
   private EmployeeHome employeeHome;
```

NotesMain Method

The NotesMain() method is the entry point to the functional code of the agent:

```
   public void NotesMain() {
   try {
```

Notes Initialization

We first need to initialize a session with the Domino server. The Session class is the root of the Domino backend object containment hierarchy. We used the *getSession()* method of AgentBase as we are writing an agent.

AgentContext represents the agent environment as the current program is running as an agent.

As the agent is started from a Domino document, we get the document using the *getDocumentContext()* method of the AgentContext class, and read one of its fields using the *getItemValueString()* method of the Document class:

```
Session session = getSession();
AgentContext agentContext = session.getAgentContext();
Document doc = agentContext.getDocumentContext();
String surname = doc.getItemValueString("Surname");
```

EJB Home Interface Localization and Object Lookup

An EJB client uses the JNDI API to locate the EJB’s Home interface. When the interface has been found, the EJB client creates an instance of the EJB.

The first step for our client is to use the JNDI naming services to locate a name server. This involves creating an initial context. We also indicate the address of the name server:

```
java.util.Hashtable properties = new java.util.Hashtable(2);
javax.naming.InitialContext initContext = null;
try
{
   properties.put(javax.naming.Context.PROVIDER_URL,
               "iiop://krypton.almaden.ibm.com:9019");
   properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
               "com.ibm.jndi.CosNaming.CNInitialContextFactory")
   initContext = new javax.naming.InitialContext(properties);
} catch (javax.naming.NamingException e) {
   System.out.println("Error retrieving the initial context: "
                     + e.getMessage());
   return;
}
```

The next step is to look up the required object, the EJB’s Home interface. The Employee EJB was configured with a JNDI Home name of *EmployeeHome.* As we are using CORBA’s CosNaming, a CORBA object is returned by the *lookup()* method which was then subsequently narrowed to be of type *EmployeeHome:*

```
if ( employeeHome == null )
try {
   java.lang.Object o = initContext.lookup("EmployeeHome");
   if (o instanceof org.omg.CORBA.Object)
      employeeHome=EmployeeHomeHelper.narrow((org.omg.CORBA.Object) o);
}
catch (javax.naming.NamingException e) {
   System.out.println("Error retrieving the home interface: "
                     + e.getMessage());
   return;
}
```

Starting the EJB

After retrieving an *EmployeeHome* instance, we can now retrieve the collection of Employee instances using a last name. We pass the key for the request (surname) using the *findByLastName()* method of the EJB’s Home interface. We receive the result as an enumeration.

We used the *EmployeesBean* class that was developed in the Phone application to manipulate the *Employee* instances. This manipulation could have been done directly from the enumeration:

```
Enumeration employees = employeeHome.findByLastName(
                           surname.trim().toUpperCase() + "%" );
EmployeesBean emp = new EmployeesBean (employees);
```

Receiving the Data

We test the success of the request using the *isEmpty()* property of the Employee EJB. If the request was successful, that is the bean has matched names using the *hasMatches()* method of the Employee EJB, we can then receive the names with the *getLastName()* method of the Employee EJB and move them to the document:

```
if (!emp.isEmpty()) {
   if (emp.hasMatches()) {
      if (!employees.hasMoreElements()){
         doc.replaceItemValue("R_Surname_0", emp.getLastName(0));
         doc.replaceItemValue("R_FirstName_0", emp.getFirstName(0));
         ....
```

Ending the Agent

When we finish handling the results, we can save the updated document and close the connection to the EJB. We can run the garbage collector to remove the unused objects from memory. You can ignore this process, as Java runs the garbage collector automatically:

```
doc.save(true,true);
employees = null;
emp = null;
System.gc();
```

19.2.3 Form Running the Java Agent

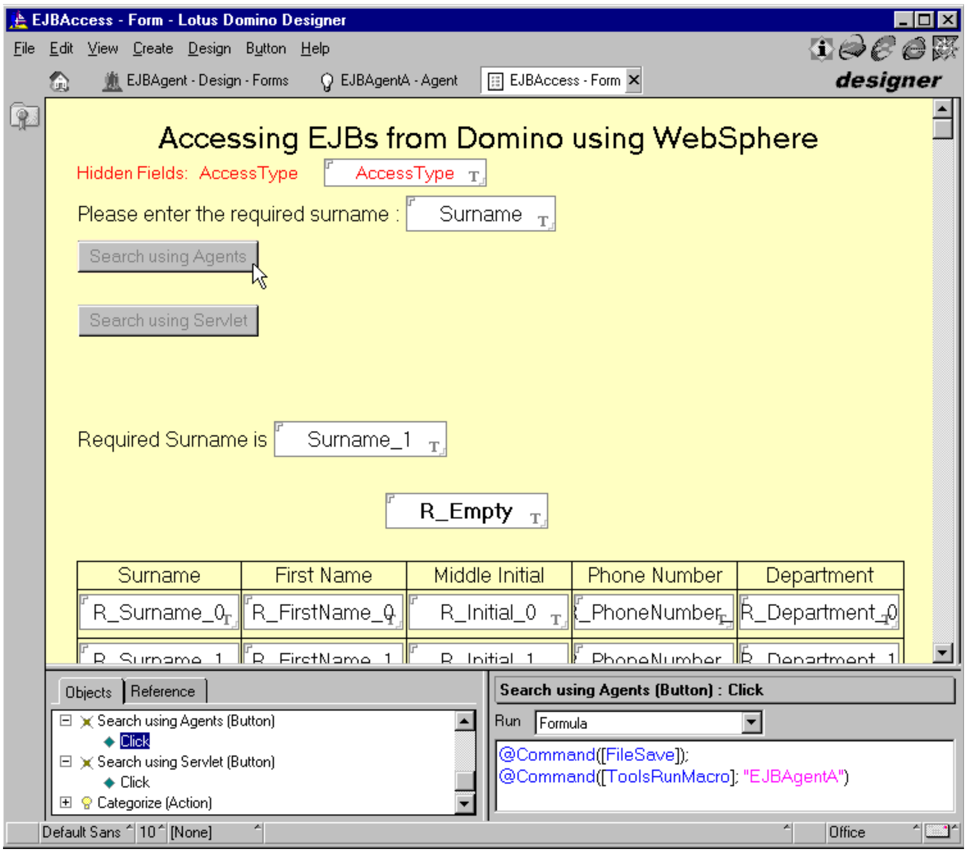
To run the EJBAgent, we created the *EJBAccess* form that contains the field used as input for the EJB. The input field is called *Surname*.

To run the Java agent we added the following formula to the form:

```
@Command([FileSave]);
@Command([ToolsRunMacro]; "EJBAgentA")
```

To receive the results from the bean, we added output fields (*R_Surname, R_FristName,...*) for each corresponding value received from the EJB, that is, each column of the DB2 table that we wanted to be displayed.

Figure 51 on page 211 shows the design of the *EJBAccess* form.


EJBAccess - Form - Lotus Domino Designer

Accessing EJBs from Domino using WebSphere

Hidden Fields: AccessType [AccessType]

Please enter the required surname : [Surname]

Search using Agents

Search using Servlet

Required Surname is [Surname_1]

[R_Empty]

Surname	First Name	Middle Initial	Phone Number	Department
R_Surname_0	R_FirstName_0	R_Initial_0	PhoneNumber	R_Department_0
R_Surname_1	R_FirstName_1	R_Initial_1	PhoneNumber	R_Department_1

Search using Agents (Button) : Click

Run: Formula

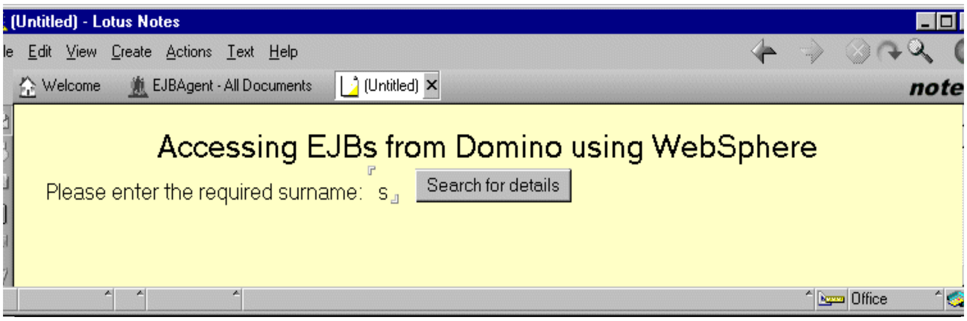
```
@Command([FileSave]);
@Command([ToolsRunMacro]; "EJBAgentA")
```

Figure 51. Example Domino Form for Running the Java Agent

19.2.4 Running the Example

To run the example, you need to:

1. Compose the *EJBAccess* Domino form.
2. Fill in the Surname field with any characters (see Figure 52 on page 212).

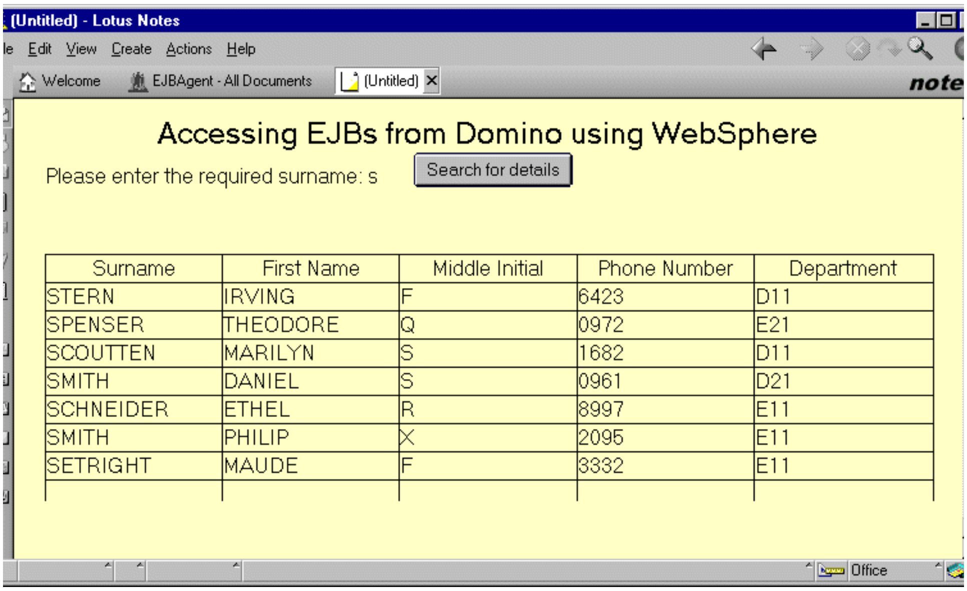


Accessing EJBs from Domino using WebSphere

Please enter the required surname: s

Figure 52. Fill the Surname Field with "s"

3. Click the **Search for details** button.
4. Using the EJB, the example displays the result in a table (see Figure 53 on page 212).



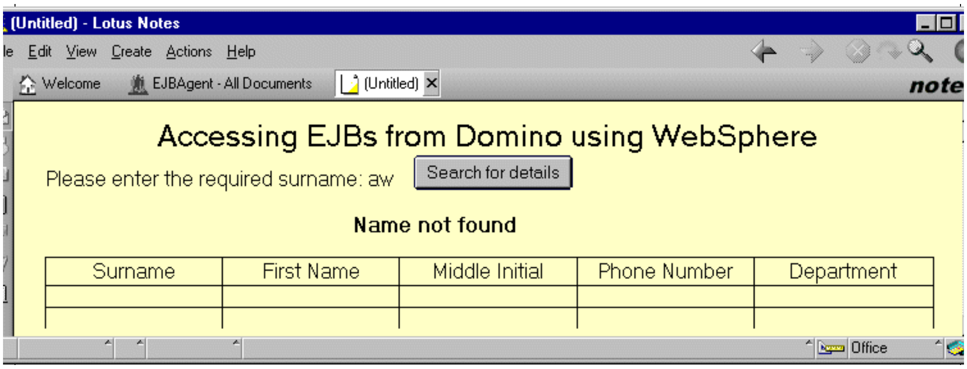
Accessing EJBs from Domino using WebSphere

Please enter the required surname: s

Surname	First Name	Middle Initial	Phone Number	Department
STERN	IRVING	F	6423	D11
SPENSER	THEODORE	Q	0972	E21
SCOUTTEN	MARILYN	S	1682	D11
SMITH	DANIEL	S	0961	D21
SCHNEIDER	ETHEL	R	8997	E11
SMITH	PHILIP	X	2095	E11
SETRIGHT	MAUDE	F	3332	E11

Figure 53. The Agent Result from the EJB

If the EJB cannot find the specified data, the sample application fills the *R_Empty* field with "Name not found" (see Figure 54 on page 213).



Accessing EJBs from Domino using WebSphere

Please enter the required surname: aw

Name not found

Surname	First Name	Middle Initial	Phone Number	Department

Figure 54. EJB Cannot Find the Data

19.3 Domino Servlet

We also tested the connection to an EJB using a Domino servlet. Here, two configurations are possible:

- The servlet is managed by the Domino Java servlet manager and can access the EJB managed by the WebSphere application server running on a separate machine.
- The servlet is managed by the WebSphere servlet manager running under the HTTP task of the Domino server.

19.3.1 Setup

To develop and run a servlet you need to add the EJB classes and the support classes into the ..**\domino\data\servlet\** directory.

We modified the *notes.ini* initialization file as follows (on one line):

```
JavaUserClasses=<other paths>;
                d:\websphere\appserver\samples\ejs\EmployeesBean.class;
                d:\websphere\appserver\deployedEJBs\EmployeeServer.jar;
```

Refer to 10.3.4, “Servlet Manager” on page 104 for additional information on how to set up the servlet manager in Domino R5.

19.3.2 Writing the Dom_Empl Servlet

In the following we explain how we developed the servlet. Only the important part of the code is shown here. For a complete listing of the servlet, refer to Appendix E.2, “Dom_Empl Servlet” on page 242.

Import Statements

We first load all the required packages needed to create our servlet:

- Java package, to write Java code
- Java servlet extensions packages, to create an HTTP servlet
- Domino package, to use Domino backend classes
- EJB package, to use EJB classes
- *Employee* EJB package, to access the EJB

The following import statements perform the task:

```
import lotus.notes.*;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.sun.server.http.HttpServiceRequest;
import com.sun.server.http.HttpServiceResponse;
import com.ibm.ejs.samples.phone.*;
```

Dom_Empl Servlet

Our class starts by extending the HttpServlet class, AgentBase class.

We define a private variable for the type of EJB and define a string containing the full name of the EJB:

```
public class Dom_Empl extends HttpServlet {
   private static final String employeeHomeName =
               "com.ibm.ejs.samples.phone.EmployeeHome";
   EmployeeHome employeeHome;
```

init Method

The init() method is called once when the Web server loads the servlet. In this method, we search the EJB and create a connection to its Home interface:

```
   public void init(ServletConfig config) {
```

An EJB client uses the JNDI API to locate an EJB’s Home interface. When the interface has been found, the EJB client creates an instance of the EJB.

The first step for our client is to use the JNDI name services to locate a name server. This involves creating an initial context. We also indicate the address of the name server:

```
java.util.Hashtable properties = new java.util.Hashtable(2);
javax.naming.InitialContext initContext = null;
try {
   properties.put(javax.naming.Context.PROVIDER_URL,
                  "iiop://krypton.almaden.ibm.com:9019");
   properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                  "com.ibm.jndi.CosNaming.CNInitialContextFactory");
   initContext = new javax.naming.InitialContext(properties);
```

The next step is to look up the required object, the EJB's Home interface. The Employee EJB was configured with a JNDI Home name of *EmployeeHome.* As we are using CORBA's CosNaming, a CORBA object is returned by the *lookup()* method which was then subsequently narrowed to be of type *EmployeeHome:*

```
if (employeeHome == null)
   try {
      java.lang.Object o = initContext.lookup("EmployeeHome");
      if (o instanceof org.omg.CORBA.Object) {
         employeeHome=EmployeeHomeHelper.narrow((org.omg.CORBA.Object) o);
      }
   } catch (javax.naming.NamingException e) {
      System.out.println("Error retrieving the home interface: "
                         + e.getMessage());
      return;
   }
}
```

doGet Method

The *doGet* method is run on response of a GET request from HTML. The *doGet* method is run every time the Domino server receives a trigger URL for the servlet. We set an output stream to be able to publish for Web users.

We call the static *sinitThread* method to explicitly start a Notes thread.

We use the *GetParameter* method of the *HttpServletRequest* class to fetch the parameter from a Domino document field called *Surname*.

We create a new session with the Domino server, access the database (*getDatabase()*), access the view (*getView()*), and get the last document from the view (*getLastDocument()*):

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
try {
   ServletOutputStream out = response.getOutputStream();
   response.setContentType("text/html");
   NotesThread.sinitThread();
   String lastName = request.getParameter("Surname");
```

```
.....
   Session session = Session.newInstance();
   Database db = session.getDatabase("", "EJBAgent.nsf");
   View view = db.getView("AllDocuments");
   Document doc = view.getLastDocument();
```

Starting the EJB

We have retrieved an *EmployeeHome* instance. We can now retrieve the collection of Employee instances using a last name. We pass the key for the request (surname) using the *findByLastName()* method of the EJB's Home interface. We receive the result as an enumeration.

We used the *EmployeesBean* class that was developed in the Phone application to manipulate the *Employee* instances. This manipulation could have been done directly from the enumeration:

```
Enumeration employees = employeeHome.findByLastName(
                           surname.trim().toUpperCase() + "%" );
EmployeesBean emp = new EmployeesBean (employees);
```

Receiving the Data

We test the success of the request using the *isEmpty()* property of the Employee EJB. If the request was successful, that is the bean has matched names using the *hasMatches()* method of the Employee EJB, then we can receive the names with the *getLastName()* method of the Employee EJB, and move them to the document or print them to the Web:

```
if (!emp.isEmpty()) {
   if (emp.hasMatches()) {
      if (!employees.hasMoreElements()){
         doc.replaceItemValue("R_Surname_0", emp.getLastName(0));
         doc.replaceItemValue("R_FirstName_0", emp.getFirstName(0));
         ....
         doc.save(true,true)
         col1 = doc.getItemValueString("R_Surname_0");
         col2 = doc.getItemValueString("R_FirstName_0");
         ...
         outputSimpleTableLine(out, col1,col2,...) ;
```

Ending the Servlet

When we finish handling the results, we can close the connection to the EJB. We can run the garbage collector to remove the unused objects from memory. You can ignore this process as Java runs the garbage collector automatically:

```
employees = null;
emp = null;
```

```
System.gc();
```

19.3.3 Form Running the Servlet

We used the same form defined for the agent process (see “Form Running the Java Agent” on page 210). To run the servlet, we added a button to the form with the following formula:

```
@Command([FileSave]);
@URLOpen( "http://krypton:999/servlet/Dom_Empl?Surname="+Surname)
```

19.3.4 Running the Servlet

You can run the servlet from a Web browser or a Notes client.

From a Web browser, you need to:

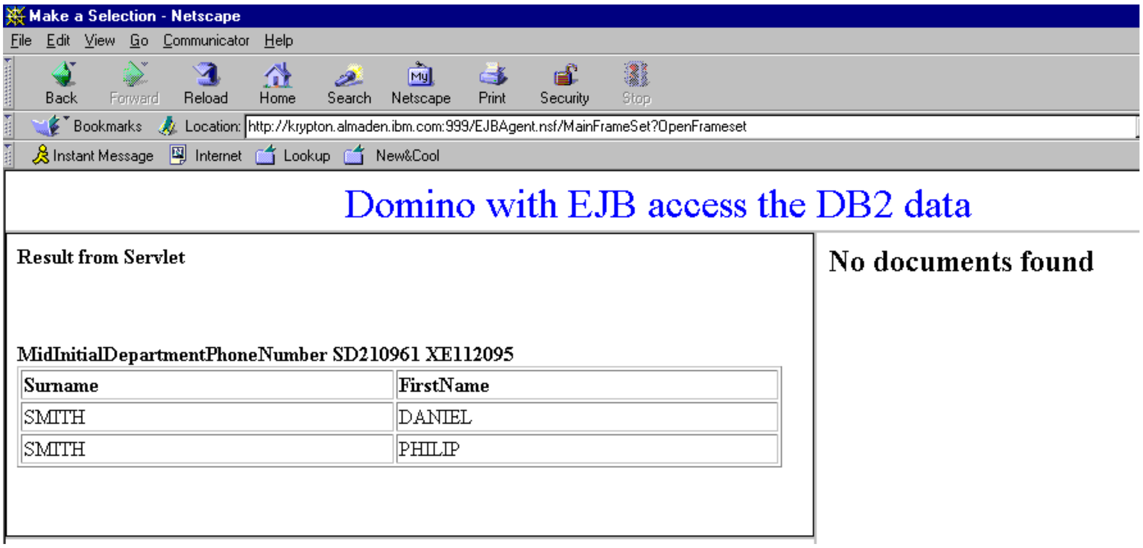
- Access the Domino form using the following URL:

  ```
  http://krypton/EJBAgent.nsf/EJBAccess?OpenForm
  ```

- Fill the Surname field with any characters.
- Click the **Search using a servlet** button.
- Using the EJB, the example displays the result in a table. The process may seem slow the first time it is run, as the server must initialize the servlet.

Note: If you change your servlet programs, you have to copy the new classes into the Domino servlet directory and reload the HTTP task using the commands:

```
tell http quit
load http.
```


Make a Selection - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://krypton.almaden.ibm.com:999/EJBAgent.nsf/MainFrameSet?OpenFrameset
Instant Message Internet Lookup New&Cool

Domino with EJB access the DB2 data

Result from Servlet

MidInitialDepartmentPhoneNumber SD210961 XE112095

Surname	FirstName
SMITH	DANIEL
SMITH	PHILIP

No documents found

Figure 55. Result Printed on the Web

19.3.5 Agent or Servlet When Using EJB

In this section we give some advice for selecting the type of Domino program when you need to use an EJB.

To develop a Domino agent that access an EJB, you need to include all the core EJB classes as well as all the supporting classes in the agent. As the Domino agent is discarded when it finishes, there is no way to keep the reference to the used EJB. Each time the agent is started, it creates a connection to the EJB Home interface and this process may take a while.

On the other hand, a Domino servlet has two important methods: *init()* and *doGet()*. The *init()* method is run at the initialization of the servlet; therefore, it runs per servlet lifecycle. In this method, we can create a connection to the EJB Home interface and keep it as long as the servlet is running. So, the first user may get a degraded response time as the servlet initializes the connection to the EJB. The next users running the servlet get a quick response time as the servlet processes only the *doGet()* method where we invoke the EJB to run the backend processes.

Appendix A. Applet Example

This appendix contains the code for creating the DB2 Domino applet. We used VisualAge for Java to create the applet. In the following list, we omitted all the methods generated directly by VisualAge for Java.

```
package itso.sg245425.applet;

import java.applet.*;
import java.awt.*;
import lotus.domino.*;
import COM.ibm.db2.*;
import java.sql.* ;
/**
 * This type was created in VisualAge.
 */
public class Db2DominoApplet extends AppletBase implements java.awt.event.ItemListener {
   //Intitialization for Domino
   Session s;
   Database db;
   DocumentCollection dcol;
   String dbname = "SG245425T";
   String server = "oxygen";
   String viewname = "Employee\\Last Name";
   String user = "Administrator";
   String pwd = "password";
   //Initialization for DB2
   Connection con;
   Statement stmt;
   ResultSet rs;
   ResultSetMetaData rsmd = null;
   String name;
   String sqlinit = "SELECT LASTNAME, FIRSTNME,SALARY, BONUS, COMM FROM christo.employee
where empno ='";
   String sqlend = "'";
   String userdb2 = "db2admin" ;
   String pwddb2 = "db2admin" ;

   Font font = new Font("Dialog", Font.BOLD, 24);
   String str = "Welcome to VisualAge";
   int xPos = 5;
   private TextField ivjBonusTextField = null;
   private TextField ivjCommTextField = null;
   private List ivjEmployeeNumberList = null;
   private TextField ivjFirstNameTextField = null;
   private Label ivjLabel1 = null;
   private Label ivjLabel2 = null;
   private Label ivjLabel21 = null;
   private Label ivjLabel211 = null;
   private Label ivjLabel2111 = null;
   private Label ivjLabel22 = null;
   private TextField ivjLastNameTextField = null;
   private TextField ivjSalaryTextField = null;
/**
 * Connect to DB2
 */
public void accessDB2Database(String empno) {
   try {
      String query ;
      stmt = con.createStatement() ;
      query = sqlinit + empno + sqlend ;
```

```
            rs = stmt.executeQuery(query) ;


        } catch (Exception e) {
            System.out.println("Error in AccessDB2Database") ;
            e.printStackTrace();
        }
    }
    /**
     * Connect to Domino
     */
    public void accessDominoDatabase() {
        try {

            // Access Database
            db = s.getDatabase(s.getServerName(), dbname);
            //List All Documents
            dcol = db.getAllDocuments();
            System.out.println("Database \"" + dbname + "\" has " + dcol.getCount() +
                                  " documents");
            //Get the Document through a view
            View view = db.getView(viewname);
            Document doc = view.getFirstDocument();
            // Document doc;
            String [] val = new String[dcol.getCount()];
            //String[] val;
            for (int i = 1; i <= dcol.getCount(); i++) {
                val[i] = doc.getItemValueString("EMPNO");
                getEmployeeNumberList().add(val[i]);
                doc = view.getNextDocument(doc);
            }
        } catch (NotesException e) {
            System.out.println("Error in Access Database");
            e.printStackTrace();
        }
    }
    /**
     * Connect to DB2
     */
    public void connectDB2() {
        try {
            String port = "999";
            Class.forName("COM.ibm.db2.jdbc.net.DB2Driver"); // .newInstance();
            // construct the URL ( sample is the database name )
            String url = "jdbc:db2://" + server + ":" + port + "/sample";
            // connect to database with userid and password
            con = DriverManager.getConnection(url, userdb2, pwddb2);
            System.out.println("Connection DB2 OK");
        } catch (Exception e) {
            System.out.println("Error in Connecting DB2 Server") ;
            e.printStackTrace();
        }
    }
    /**
     * Connect to Domino
     */
    public void connectDomino() {
        try {
            s = this.openSession("Administrator","password");
            System.out.println("Connection OK on server :" + s.getServerName());
            System.out.println("Connection OK for User  :" + s.getCommonUserName());
        } catch (NotesException e) {
            System.out.println("Error in Connecting Domino Server");
            e.printStackTrace();
        }
    }
    /**
     * Comment
     */
    public void employeeNumberList_ItemStateChanged(java.awt.event.ItemEvent event,
                                                String empno) {
        try {
            if (event.getStateChange() != java.awt.event.ItemEvent.SELECTED) {
                return;
            }
            accessDB2Database(empno);
            if (!rs.next()) {
                System.out.println("No Corresponding row");
            } else {
                getLastNameTextField().setText(rs.getString(1));
                getFirstNameTextField().setText(rs.getString(2));
                getSalaryTextField().setText(rs.getString(3));
                getBonusTextField().setText(rs.getString(4));
                getCommTextField().setText(rs.getString(5));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * main entrypoint - starts the part when it is run as an application
     * @param args java.lang.String[]
     */
    public static void main(java.lang.String[] args) {
        try {
            Frame frame;
            try {
                Class aFrameClass = Class.forName("com.ibm.uvm.abt.edit.TestFrame");
                frame = (Frame)aFrameClass.newInstance();
            } catch (java.lang.Throwable ivjExc) {
                frame = new Frame();
            }
            Db2DominoApplet aDb2DominoApplet;
            Class iiCls = Class.forName("itso.sg245425.applet.Db2DominoApplet");
            ClassLoader iiClsLoader = iiCls.getClassLoader();
            aDb2DominoApplet =
    (Db2DominoApplet)java.beans.Beans.instantiate(iiClsLoader,"itso.sg245425.applet.Db2Domin
    oApplet");
            frame.add("Center", aDb2DominoApplet);
            frame.setSize(aDb2DominoApplet.getSize());
            frame.setVisible(true);
        } catch (Throwable exception) {
            System.err.println("Exception occurred in main() of java.applet.Applet");
            exception.printStackTrace(System.out);
        }
    }
    /**
     * Initializes the applet.
    */
    public void notesAppletInit() {
        super.notesAppletInit();
        try {
            setName("Db2DominoApplet");
            setLayout(null);
            setSize(364, 416);
            add(getLabel1(), getLabel1().getName());
            add(getEmployeeNumberList(), getEmployeeNumberList().getName());
            add(getLabel2(), getLabel2().getName());
            add(getLastNameTextField(), getLastNameTextField().getName());
            add(getLabel21(), getLabel21().getName());
            add(getFirstNameTextField(), getFirstNameTextField().getName());
            add(getLabel211(), getLabel211().getName());
            add(getBonusTextField(), getBonusTextField().getName());
            add(getLabel22(), getLabel22().getName());
            add(getSalaryTextField(), getSalaryTextField().getName());
            add(getLabel2111(), getLabel2111().getName());
            add(getCommTextField(), getCommTextField().getName());
            initConnections();
            connEtoC2();
            connectDomino() ;
            connectDB2() ;
            accessDominoDatabase() ;
        } catch (java.lang.Throwable ivjExc) {
            handleException(ivjExc);
        }
    }
    /**
     * Called to start the applet. You never need to call this method
     * directly, it is called when the applet's document is visited.
    */
    public void notesAppletStart() {
        super.notesAppletStart();
    }
    /**
     * Called to stop the applet. It is called when the applet's document is
     * no longer on the screen. It is guaranteed to be called before destroy()
     * is called. You never need to call this method directly.
    */
    public void notesAppletStop() {
        super.notesAppletStop();
    }
    }
```

Appendix B. Servlet Example

This appendix contains the source code of the Domino servlet that accesses the enterprise using CICS.

```
package ExampleServletV2;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import lotus.notes.*;
import com.ibm.ctg.client.*;

/**
 * This type was created in VisualAge.
 */
public class ExampleServletV2 extends HttpServlet {
    JavaGateway jgaConnection;
    ServletContext context;
    private static int CICS_COMMAREALENGTH = 131;
    private static int CICS_FIRSTNAME_POS  = 24;
    private static int CICS_FIRSTNAME_LEN  = 10;
    private static int CICS_ADDRESS_POS  = 34;
    private static int CICS_ADDRESS_LEN  = 15;
    private static int CICS_CITYSTATE_POS  = 49;
    private static int CICS_CITYSTATE_LEN  = 15;
    private static int CICS_POSTCODE_POS  = 64;
    private static int CICS_POSTCODE_LEN  = 15;
    private static int CICS_CREDIT_RATING_POS  = 79;
    private static int CICS_CREDIT_RATING_LEN  = 2;
    private static int CICS_COMMENTS_POS  = 81;
    private static int CICS_COMMENTS_LEN  = 50;
/**
 * ExampleServletV2 constructor comment.
 */
public ExampleServletV2() {
    super();
}
/**
 * This method was created in VisualAge.
 * @param lookName java.lang.String
 */
public byte [] createCommArea(String lookName) {
    // Set up Commarea buffer
    ByteArrayOutputStream byteBuf = new ByteArrayOutputStream(131);
    try
    {
        // Start writing the COMMAREA
        String byteStr = null;
        byteStr = "2";
        byteBuf.write(byteStr.getBytes());//Server state
        byteStr = "0";
        byteBuf.write(byteStr.getBytes());//Return Value
        byteBuf.write((byte)5);//Server_key_size (2 Byte Short)
        byteBuf.write((byte)0);
        int c;                           // Record Number 5 bytes
        for (c=0;c<5;byteBuf.write((byte)0),c++);
        lookName = lookName.toUpperCase();// Force the surname to uppercase
        byteBuf.write(lookName.getBytes(),0,lookName.length());  // Surname to look for
        int size = byteBuf.size();
        // Pad out rest of Commarea to required length
        for (c=0;c<(CICS_COMMAREALENGTH-size);c++)
            byteBuf.write((byte)0);
    }
    catch (IOException e) {};
    return byteBuf.toByteArray();
}
/**
 * This method was created in VisualAge.
 * @param document doc
 * @param outstuff out
 */
public void createReturnHTML(Document doc, ServletOutputStream out)
{
    try
    {
        String col1;
        String col2;
        out.println("<B>This frame has been generated by the Servlet.</B><BR><BR>");
        out.println("<TABLE WIDTH=\"100%\" BORDER=1>");
        outputSimpleTableLine(out,"<B>DATA FROM DOMINO (VIA DOM)</B>",
                            "<B>DATA FROM CICS (VIA JAVA GATEWAY)</B>");
        col1 = "Name : " + doc.getItemValueString("Firstname") + " "
                      + doc.getItemValueString("Surname");
        col2 = "Address : " + doc.getItemValueString("CICS_ADDRESS");
        outputSimpleTableLine(out,col1,col2);
        col1 = "Job : " + doc.getItemValueString("Job");
        col2 = "Postcode: " + doc.getItemValueString("CICS_POSTAL_CODE");
        outputSimpleTableLine(out,col1,col2);
        col1 = "";
        col2 = "Credit Rating : " + doc.getItemValueDouble("CICS_CREDIT_RATING");
        outputSimpleTableLine(out,col1,col2);
        out.println("</TABLE>");
    } catch (Exception e) {e.printStackTrace();}
}
/**
 * This method was created in VisualAge.
 * @param request javax.servlet.http.HttpServletRequest
 * @param response javax.servlet.http.HttpServletResponse
 * @exception java.io.IOException The exception description.
 */
public void doGet (HttpServletRequest request, HttpServletResponse response) throws
IOException {
    try {
            ServletOutputStream out = response.getOutputStream();
            response.setContentType("text/html");
            NotesThread.sinitThread();     // Kick off Notes
            // Get the value of the EmpNo variable from the URL
            //eg. www.mysite.com?Empno=1000
            String empKey = request.getParameter("EmpNo");
            if(empKey != null) // As long as we have a value to work with
            {
                // Open up the database and get the view we want
                Session s = NotesFactory.createSession();
                Database db = s.getDatabase("","test.nsf");
                View view = db.getView("CustomerLookupView");
                Document doc = view.getDocumentByKey(empKey, true);// Look for our key
                if (doc != null)      // ..and as long as we find something
                {
                    performCICSTransaction(doc,out); // Go get the data from CICS,
                    createReturnHTML(doc,out);// And return results to the browser
                }
                else
                    out.println("<B>Sorry, data not found. Please try again</B><BR>");
            }
            else
                out.println("<B>Sorry the EmpNo parameter can not be found</B><BR>");
    }
    catch (Exception e) { e.printStackTrace(); }// Basic error handler
    finally {NotesThread.stermThread();}// Clean up
}
/**
 * This method was created in VisualAge.
 * @param config javax.servlet.ServletConfig
 */
public void init(ServletConfig config) {
    try {
            com.ibm.ctg.client.T.setDebugOn( true );
            com.ibm.ctg.client.T.setTimingOn( true );
            jgaConnection = new JavaGateway();
            jgaConnection.setURL( "local:");
            jgaConnection.open();
        }
        catch (IOException e) {};
}
/**
 * This method was created in VisualAge.
 * @param out javax.servlet.ServletOutputStream
 * @exception java.lang.Exception The exception description.
 */
public void outputSimpleTableLine(ServletOutputStream out,String col1, String col2)
{
    try {
            out.println("<TR VALIGN=top><TD WIDTH=\"50%\">"
                + col1 + "</TD><TD WIDTH=\"50%\">"
                + col2 + "</TD></TR>");
    }catch (Exception e) {e.printStackTrace();}
}
/**
 * This method was created in VisualAge.
 * @param Document doc
 */
public void performCICSTransaction(Document notesDoc,ServletOutputStream out) {
    ECIRequest eciRequest = null;
    try
    {
        if (!jgaConnection.isOpen())// check we do actually have a connection to CICS
        {
            out.println("<BR><B>JGate Connection is not open,</B>");
            return;
        }
        // Get the surname to pass to the CICS trasaction
         String lookName = notesDoc.getItemValueString("Surname");
        // And build up the CommArea that we will pass to CICS
         byte [] commArea = createCommArea(lookName);
        // Create our eciRequest
        eciRequest = new ECIRequest("CICSOS2", // CICS Server
                                "SYSAD", // UserId, null for none
                                "SYSAD", // Password, null for none
                                "VSAMSERV", // Program name
                                commArea,// Commarea
                                ECIRequest.ECI_NO_EXTEND,
                                ECIRequest.ECI_LUW_NEW);
                    eciRequest.Cics_Rc = 0;
        // Call CICS
        jgaConnection.flow(eciRequest);
```

```
             // If the CICS transaction happened OK
             // then sort out the returned CommArea into our NotesDocument
            if (eciRequest.Commarea != null) {
                       String tmpString;
                        tmpString = new String(eciRequest.Commarea,
                                  CICS_FIRSTNAME_POS,CICS_FIRSTNAME_LEN);
                       notesDoc.replaceItemValue("CICS_FIRST_NAME",tmpString);
                       tmpString = new String(eciRequest.Commarea,
                                  CICS_ADDRESS_POS,CICS_ADDRESS_LEN);
                       notesDoc.replaceItemValue("CICS_ADDRESS",tmpString);
                       tmpString = new String(eciRequest.Commarea,
                                  CICS_CITYSTATE_POS,CICS_CITYSTATE_LEN);
                       notesDoc.replaceItemValue("CICS_CITY_STATE",tmpString);
                       tmpString = new String(eciRequest.Commarea,
                                  CICS_POSTCODE_POS,CICS_POSTCODE_LEN);
                       notesDoc.replaceItemValue("CICS_POSTAL_CODE",tmpString);
                       // Credit rating is a 2 byte short,  Low byte, High Byte)
                       // and with 255 as an integer to flip byte valus > 127
                       int creditRatingValueByte1 =
                           eciRequest.Commarea[CICS_CREDIT_RATING_POS] & 255;
                       int creditRatingValueByte2 =
                           eciRequest.Commarea[CICS_CREDIT_RATING_POS+1] & 255;
                       Integer creditRatingValue  =
                           new Integer(creditRatingValueByte1 + (255*creditRatingValueByte2));
                       notesDoc.replaceItemValue("CICS_CREDIT_RATING",creditRatingValue);
                       tmpString = new String(eciRequest.Commarea,
                                  CICS_COMMENTS_POS,CICS_COMMENTS_LEN);
                       out.println("<BR>" + tmpString);
                      }
                }
               catch (Exception e){ e.printStackTrace();};
     }
     }
```

Appendix C. Application Example

This appendix contains the source code of the three Java applications.

C.1 Domino JDBC Driver

```
package itso.sg245425.application;

import java.math.*;
import java.sql.*;
import java.util.*;
import lotus.jdbc.domino.*;
/**
 * This type was created in VisualAge .
 */
public class DominoJDBC {
/**
 * DominoTest constructor comment.
 */
public DominoJDBC() {
   super();
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void fill(String s, int times)
   {
      if (times <= 0)
         return;
       for (int i=0; i<times; i++)
       {
           System.out.print(s);
       }
   }
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
   // Insert code to start the application here.
   Connection con;
   Statement stmt;
   ResultSet rs;
   ResultSetMetaData rsmd = null;
   String name;
   String sql = "SELECT * FROM \"Employee\\Last Name\"";
   String connStr = "jdbc:domino/sg245425\\SG245425T.nsf/oxygen";
   java.text.DecimalFormat moneyForm = new java.text.DecimalFormat("##,###.##");
   try {
      try {
         Class.forName("lotus.jdbc.domino.DominoDriver");
      } catch (ClassNotFoundException e) {
         System.out.println("ClassNotFoundExecption: " + e.getMessage());
      }
      // GET CONNECTION
      System.out.println();
      System.out.println("Connecting to URL " + connStr);
      con = DriverManager.getConnection(connStr, "", "");
```

```
            System.out.println();
            // Create Statement
            stmt = con.createStatement();
            // Execute statement
            rs = stmt.executeQuery(sql);
            System.out.println("Executing... " + sql);
            System.out.println();
            // Get Result set metadata
            rsmd = rs.getMetaData();
            // Find number of columns in the result set
            int colCount = rsmd.getColumnCount();
            // Array to hold max display size per column
            int[] len = new int[colCount + 1];
            //print the column labels
            printColLabel(colCount, len, rsmd);
            // New line
            System.out.println();
            //print the rows
            printColText(colCount, len, rs);
            // Close the statement
            stmt.close();
            // Close the connection
            con.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    /**
     * Starts the application.
     * @param args an array of command-line arguments
     */
    public static void printCol(int len, String s)
        {
            System.out.print(s);
            fill(" ",len-s.length());
            System.out.print("  ");
        }
    /**
     * Starts the application.
     * @param args an array of command-line arguments
     */
    public static void printColLabel(int colCount, int[] len, ResultSetMetaData rsmd) {
        try {
            // Print column Labels as header
            for (int i = 1; i <= colCount; i++) {
                // Get column label.
                String label = rsmd.getColumnLabel(i);
                // Store the maximum of display size or label length
                if (label.length() > 10)
                    len[i] = label.length();
                else
                    len[i] = 10;
                // Print label
                System.out.print(label);
                // Pad with blanks
                fill(" ", len[i] - label.length());
                // Column seperator
                System.out.print("  ");
            }
            // New line
            System.out.println();
            for (int i = 1; i <= colCount; i++) {
                fill("-", len[i]);
                System.out.print("  ");
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    /**
     * Starts the application.
     * @param args an array of command-line arguments
     */
    public static void printColText(int colCount, int[] len, ResultSet rs) {
        try {
            // Fetch all rows in the result set
            while (rs.next()) {
                for (int i = 1; i <= colCount; i++) {
                    Object obj = rs.getObject(i);
                    boolean nl = rs.wasNull();
                    if (nl)
                        printCol(len[i], "null");
                    else
                        printCol(len[i], obj.toString());
                }
                // New line
            System.out.println();
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    }
```

C.2 Domino Objects Classes

```
package itso.sg245425.application;

import java.math.*;
import java.sql.*;
import java.util.*;
import lotus.domino.*;
/**
 * This type was created in VisualAge.
 */
public class DominoApplicationNotesThread extends NotesThread {
/**
 * DominoTest constructor comment.
 */
public DominoApplicationNotesThread() {
    super();
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
    // Insert code to start the application here.
    try {
        DominoApplicationNotesThread thread1 = new DominoApplicationNotesThread();
        thread1.start();
        thread1.join();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void printCol(View view) {
    try {
        Document doc = view.getFirstDocument();
        while (doc != null) {
            System.out.println("Last Name  : " + doc.getItemValueString("LASTNAME") );
            System.out.println("First Name : " + doc.getItemValueString("FIRSTNME"));
            System.out.println("Employee # : " + doc.getItemValueString("EMPNO"));
            System.out.println("Salary     : " + doc.getItemValueInteger("SALARY"));
            System.out.println("Bonus      : " + doc.getItemValueInteger("BONUS"));
            System.out.println("Commission : " + doc.getItemValueInteger("COMM"));
            System.out.println();
            doc = view.getNextDocument(doc);
        }
    } catch (NotesException e) {
        e.printStackTrace();
    }
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void printColLabel(View view) {
    try {
        Vector columns = view.getColumns();
        if (columns.size() != 0) {
            for (int i = 0; i < columns.size(); i++) {
                ViewColumn column = (ViewColumn) columns.elementAt(i);
                String vtitle = column.getTitle();
                if (vtitle == null)
                    vtitle = "No Title";
                System.out.print(" " + vtitle);
            }
        }
        System.out.println();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public void runNotes() {
    String dbname = "SG245425\\SG245425T";
    String servername = "oxygen";
    String viewname = "Employee\\Last Name";
    try {

        // GET Session
        System.out.println("Session");
         Session s = NotesFactory.createSession();

        // Access Database
        System.out.println("Database");
        Database db = s.getDatabase(servername, dbname);
        // Open Database
        System.out.print("Database " + dbname + "has been last modified on");
        System.out.println(db.getLastModified());

        // get View
        System.out.println("View  " + viewname + "contains the following columns:");
        View view = db.getView(viewname);
        printColLabel(view);
        // Get Documents from the view
        System.out.println("View  " + viewname + "contains the following documents");
        printCol(view);
    } catch (NotesException e) {
        e.printStackTrace();
    }
}
}
```

C.3 Lotus Connector

```
package itso.sg245425.application;


import lotus.lcjava.*;
/**
 * This type was created in VisualAge.
 */
public class LotusConnectorNotesData  {
/**
 * LotusConnectorNotesData constructor comment.
 */
public LotusConnectorNotesData() {
    super();
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
    // Insert code to start the application here.
    LCSession session = null;
    LCConnection connection = null;
    try {
        // Create LC session
        session = new LCSession(0);
        System.out.println("Session ready");
        // Create LC Connection to Notes
        connection = new LCConnection("notes", 0);
        System.out.println("Connection to Notes OK");
        connection.setPropertyJavaString(LCTOKEN.SERVER, "oxygen/Almaden");
        connection.setPropertyJavaString(LCTOKEN.DATABASE, "sg245425\\sg245425T.nsf");
        connection.setPropertyJavaString(LCTOKEN.METADATA, "Employee");
        connection.connection();
        System.out.println("Connection to Notes database ready");
        //Set the data field lists
        LCFieldlist keyList = new LCFieldlist(1, 0);
        LCFieldlist resultList = new LCFieldlist(1, 0);
        LCFieldlist fetchList = new LCFieldlist(1, 0);
        //Set up the key
        LCField keyField = new LCField();
        // the key column
        keyList.append("WORKDEPT", LCTYPE.TEXT, keyField);
        keyField.setFlags(LCFIELDF.KEY);
        // the key value
```

```
            LCStream Workdept = new LCStream("D11");
            keyField.setStream(1, Workdept);
            System.out.println("Key Initialized");
            // Set up the result set
            LCField lastName = new LCField();
            LCField firstName = new LCField();
            LCField salary = new LCField();
            LCField bonus = new LCField();
            resultList.append("LASTNAME", LCTYPE.TEXT, lastName);
            resultList.append("FIRSTNME", LCTYPE.TEXT, firstName);
            resultList.append("SALARY", LCTYPE.FLOAT, salary);
            resultList.append("BONUS", LCTYPE.FLOAT, bonus);
            System.out.println("result set Set Up");
            // Get the result set (LASTNAME, FIRSTNME, SALARY, BONUS)
            connection.setPropertyJavaString(LCTOKEN.FIELD_LIST,
                                 "LASTNAME,FIRSTNME,SALARY,BONUS");
            Integer returnedRows = new Integer(0);
            connection.select(keyList, 1, resultList, returnedRows);
            System.out.println("Result Set OK, #" + returnedRows);
            // Set up and fetch the result
            LCField lastNameF = new LCField();
            LCField firstNameF = new LCField();
            LCField salaryF = new LCField();
            LCField bonusF = new LCField();
            fetchList.append("LASTNAME", LCTYPE.TEXT, lastNameF);
            fetchList.append("FIRSTNME", LCTYPE.TEXT, firstNameF);
            fetchList.append("SALARY", LCTYPE.FLOAT, salaryF);
            fetchList.append ("BONUS", LCTYPE.FLOAT, bonusF) ;
            int rc = 0;
            rc = connection.fetch(fetchList, 1, 1);
            while (rc != LCFAIL.END_OF_DATA) {
                System.out.println("Name: " + lastNameF.toJavaString());
                System.out.println("First Name: " + firstNameF.toJavaString());
                System.out.println("Salary: " + salaryF.toJavaString());
                System.out.println( "Bonus: " + bonusF.toJavaString()) ;
                rc = connection.fetch(fetchList, 1, 1);
            }
        } catch (LCException e) {
            int err = e.getLCErrorCode();
            System.out.println(err) ;
            String errmsg = session.getStatusText(err);
            System.out.println(errmsg);
        }
    }
    }
```

Appendix D. Agent Example

This appendix contains the source code of the Domino Java agents that access the enterprise using MQSeries.

D.1 Domino Agent to the Enterprise Using MQSeries

```
// ===========================================================================
//  mqdom2ent
// ===========================================================================
//
// Example of Using MQSeries Client for Java within Domino Agent
//  Notes Client or Web Browser initiation (Web publish with if statement)
//  Pointed to queue manager DEF_QMNGR on OB machine
//
//  Input to enterprise:
//      part number (3 characters)
//      quantity (short)
//
//  Output from enterprise:
//      part number (3 characters)
//      quantity (short)
//      inventory level (integer)
//      comments (101 characters)
//
// ===========================================================================
import java.lang.*;
import java.io.PrintWriter;
import lotus.notes.*;           // Include Notes package
import com.ibm.mq.*;            // Include MQ package
public class mqdom2ent extends AgentBase
{
  //*----------------------------------------------------------------------
  // Variables
  //*----------------------------------------------------------------------

  String hostname = "ob.almaden.ibm.com";  // hostname to connect to
  String channel  = "SYSTEM.DEF.SVRCONN";  // name of channel for client to use
  int    iPort = 1414;                     // port, assumed to be 1414 unless specified

  String qManager = "DEF_QMNGR";           // queue manager to connect to
  String requestQueue = "MQAPPL.Q";        // queue to put request to
  String replyQueue = "MQAPPL.RQ";         // queue to get reply from

  MQQueueManager qMgr;                     // queue manager object use in try and finally
  MQQueue requestQ;                        // queue object use in try and finally
  MQQueue replyQ;                          // queue object use in try and finally

  // Method: Main Calling Method
  // ---------------------------

  public void NotesMain()

  {
    System.out.println("Now executing mqdom2ent class, last modified 22/02/99 09:50");
    System.out.println("NotesMain method starting");

    try
    {
```

```
// Notes initialization...

Session session = getSession();
AgentContext ac = session.getAgentContext();
Database db = ac.getCurrentDatabase();
Document doc = ac.getDocumentContext();
System.out.println("Document in use, UID: " + doc.getUniversalID());
PrintWriter pw = getAgentOutput();

// Extract required information from document...

String partNumber = doc.getItemValueString("I_PARTNO");
String quantity = doc.getItemValueString("I_QUANTITY");

// MQ initialization...

MQEnvironment.hostname = hostname;
MQEnvironment.port = iPort;
MQEnvironment.channel  = channel;

System.out.println("About to try connecting to qmgr: " + qManager);
System.out.println("Using channel: " + channel);
System.out.println("and hostname: " + hostname);

// Create a connection to the queue manager...

qMgr = new MQQueueManager(qManager);

if (qMgr.isOpen)
{
System.out.println("Connection to qmgr open");
}

// Set up the options on the queues we wish to open...
// Note: All MQ Options are prefixed with MQC (i.e. as constants) in Java
// Note: MQOO_INQUIRE & MQOO_SET are always included by default

int openOptionsOut = MQC.MQOO_OUTPUT;
int openOptionsIn = MQC.MQOO_INPUT_SHARED;

// Now open the queues...

System.out.println("About to try opening the queues: "
               + requestQueue + ", " + replyQueue);
requestQ = qMgr.accessQueue(requestQueue,
                            openOptionsOut,
                            null,          // default q manager
                            null,          // no dynamic q name
                            null);         // no alternate user id
replyQ = qMgr.accessQueue(replyQueue,
                          openOptionsIn,
                          null,          // default q manager
                          null,          // no dynamic q name
                          null);         // no alternate user id

// Create the MQ request message object..

MQMessage requestMsg = new MQMessage();

System.out.println("CharacterSet set to: " + requestMsg.characterSet);
System.out.println("Encoding set to: " + requestMsg.encoding);
requestMsg.characterSet = 437;
requestMsg.encoding = 546;
```

```
        System.out.println("Explicitly set the characterSet and encoding");
        System.out.println("CharacterSet set to: " + requestMsg.characterSet);
        System.out.println("Encoding set to: " + requestMsg.encoding);

        System.out.println("Write part number " + partNumber + " into message");
        requestMsg.writeString(partNumber);
        System.out.println("Write quantity " + quantity + " into message");
        // if the quantity field is blank, the writeShort will fair, so make it zero
        if (quantity==null)
           requestMsg.writeShort(0);
        else
           requestMsg.writeShort(Short.parseShort(quantity));

        requestMsg.messageType = MQC.MQMT_REQUEST;
        requestMsg.format = MQC.MQFMT_STRING;
        requestMsg.replyToQueueName = replyQueue;
        requestMsg.replyToQueueManagerName = qManager;

        // Specify the put message options...
        // accept the defaults, MQPMO_DEFAULT
        MQPutMessageOptions pmo = new MQPutMessageOptions();

        // put the message on the queue

        requestQ.put(requestMsg,pmo);
        System.out.println("The request message was successfully put");

        // Define an MQ message buffer to receive the reply message into...

        MQMessage replyMsg = new MQMessage();
        replyMsg.correlationId = requestMsg.messageId;

        // Set the get message options..

        MQGetMessageOptions gmo = new MQGetMessageOptions();
        gmo.waitInterval = 1000;
        gmo.options = MQC.MQGMO_WAIT;

        // Now get the reply message off the queue...

        replyQ.get(replyMsg, gmo, 300);

        System.out.println("The reply message was successfully received");
        System.out.println("reply characterSet = " + replyMsg.characterSet);
        System.out.println("reply encoding = " + replyMsg.encoding);

//      replyMsg.encoding = MQC.MQENC_INTEGER_REVERSED;

        if (replyMsg.getMessageLength() != 0)
        {
           String replyPartNumber = replyMsg.readString(3);
           String replyQuantity = Short.toString(replyMsg.readShort());
           String replyInventory = Integer.toString(replyMsg.readInt());

           System.out.println("The part number received: " + replyPartNumber);
           System.out.println("the quantity: " + replyQuantity);
           System.out.println("and inventory: " + replyInventory);
           System.out.println("The datalength is now: " + replyMsg.getDataLength());
           if (replyMsg.getDataLength() != 0)
           {
             String replyComments =replyMsg.readString(replyMsg.getDataLength());
             doc.replaceItemValue ("I_COMMENTS", replyComments.trim());
           }
           doc.replaceItemValue("I_PARTNO", replyPartNumber);
           doc.replaceItemValue("I_QUANTITY", replyQuantity);
           doc.replaceItemValue ("I_INVENTORY", replyInventory);
       }
       else
       {
           // If there is no message data...
           doc.replaceItemValue("I_COMMENTS", "No data available");
       }

       // Save the updates to the document...

       doc.save(true, true);

       // If agent was invoked by Web user, publish the updated document back to the Web...

       String accessType = doc.getItemValueString("AccessType");
       System.out.println ("Access type is: " + accessType);
       if (accessType.equals("WebClient"))
       {
          pw.println ("[/"+db.getFileName()
                 +"/All+Documents/"+doc.getNoteID()+"?OpenDocument]");
       }

      }

      // If an error has occured in the above, try to identify what went wrong.
      // Was it an MQ error?
      catch (MQException mqex)
      {
        System.out.println("An MQ error occurred: Completion code " +
                           mqex.completionCode +
                           " Reason code " + mqex.reasonCode +
                           " from: " + mqex.exceptionSource);

        if (mqex.reasonCode==2085)
        {
           System.out.println("The explanation is: Unknown object name.");
        }
        else
        {
           if (mqex.reasonCode==2033)
           {
             System.out.println("The explanation is: No message available.");
           }
           else
           {
             System.out.println("No specific explanation available for this error,
                    please refer to the MQ Application Programming Reference");
           }
         }
      }

      // Was it a Notes error?
      catch (NotesException nex)
      {
        System.out.println("An Notes error occurred: " + nex);
      }

      // Was it a Java buffer space error?
      catch (java.io.IOException jex)
      {
        System.out.println("An error occurred whilst writing to the message buffer: "
                           + jex);
      }
      // Any other error...
      catch (Exception e)
      {
        e.printStackTrace();
      }

      finally
      {
        // Tidy up before exiting (error or not)...
        System.out.println("and finally...");

        try
        {
          if (requestQ.isOpen)
              requestQ.close();

          if (replyQ.isOpen)
              replyQ.close();

          if (qMgr.isOpen)
          {
             System.out.println ("Disconnecting from queue manager");
             qMgr.disconnect();
          }
        }
        catch (Exception e)
        {
          e.printStackTrace();
        }
      }

    } // end of NotesMain

  } // end of Agent
```

D.2 The Enterprise to a Domino Agent Using MQSeries

```
// ============================================================================
//  mqent2dom
// ============================================================================
//
// Example of Using MQSeries Client for Java within Domino Agent
//  Enterprise initiation
//  Pointed to queue manager DEF_QMNGR on OB machine
//
//  Output from enterprise:
//      part number (3 characters) aligned to 4 bytes
//      inventory level (integer)
//      comments (101 characters)  aligned to 104 bytes
//
// ============================================================================
//


import com.ibm.mq.*;            // Include MQ package
import lotus.notes.*;           // Include Notes package
import java.lang.*;

public class mqent2dom extends AgentBase
```

```
{
  //*-----------------------------------------------------------------------
  // Variables
  //*-----------------------------------------------------------------------

  String hostname = "ob.almaden.ibm.com";  // hostname to connect to
  String channel  = "SYSTEM.DEF.SVRCONN";  // name of channel for client to use
  int    iPort = 1414;                     // port, assumed to be 1414 unless specified

  String qManager = "DEF_QMNGR";           // queue manager to connect to
  String inputQueue = "MQAPPL.TQ";         // triggered queue to get input from

  MQQueueManager qMgr;                    // queue manager object use in try and finally
  MQQueue inputQ;                          // queue object use in try and finally

  // Method: Main Calling Method
  // ---------------------------

  public void NotesMain()

  {
   System.out.println("Now executing mqent2dom class, last modified 22/02/99 at 14:30");
    System.out.println("NotesMain method starting");

    try
    {
      // Notes initialization

      Session session = getSession();
      AgentContext ac = session.getAgentContext();
      Database db = ac.getCurrentDatabase();
      View view = db.getView("Inventory Summary");
      Document doc = view.getLastDocument();
      System.out.println("Document in use, UID: " + doc.getUniversalID());

      // MQ initialization

      MQEnvironment.hostname = hostname;
      MQEnvironment.port = iPort;
      MQEnvironment.channel  = channel;

      System.out.println("About to try connecting to qmgr: " + qManager);
      System.out.println("Using channel: " + channel);
      System.out.println("and hostname: " + hostname);

      // Create a connection to the queue manager...

      qMgr = new MQQueueManager(qManager);

      if (qMgr.isOpen)
      {
      System.out.println("Connection to qmgr open");
      }

      // Set up the options on the queue we wish to open...
      // Note: All MQ Options are prefixed with MQC (i.e. as constants) in Java
      // Note: MQOO_INQUIRE & MQOO_SET are always included by default

      int openOptionsIn = MQC.MQOO_INPUT_SHARED;

      // Now open the queue...

      System.out.println("About to try opening the queue");
      inputQ = qMgr.accessQueue(inputQueue,
                                openOptionsIn,
                                null,           // default queue manager
                                null,           // no dynamic queue name
                                null);          // no alternate user id

      // Define an MQ message buffer to receive the message into..

      MQMessage inputMsg = new MQMessage();

      // Set the get message options..
     // accept the defaults (MQGMO_DEFAULT)
      MQGetMessageOptions gmo = new MQGetMessageOptions();

      // Now get the message off the queue...

      inputQ.get(inputMsg, gmo, 300);

      System.out.println("The message was successfully received");
      System.out.println("The encoding is: " + inputMsg.encoding);
      System.out.println("The character set is: " + inputMsg.characterSet);

     // inputMsg.encoding = MQC.MQENC_INTEGER_REVERSED;

      if (inputMsg.getMessageLength() != 0)
      {
         String inputPartNumber = inputMsg.readString(3);
         String dummy = String.valueOf(inputMsg.readByte());
         String inputInventory = String.valueOf(inputMsg.readInt());

         String fieldToUpdate = "PN"+inputPartNumber+"_INVENTORY";
         System.out.println("The field to update with inventory is: " + fieldToUpdate);
         doc.replaceItemValue (fieldToUpdate, inputInventory);
         String inputComments = inputMsg.readString(inputMsg.getDataLength());
         String commentsField = "PN"+inputPartNumber+"_COMMENTS";
         doc.replaceItemValue (commentsField, inputComments.trim());

     doc.save(true, true);
     view.refresh();
     }

    }

    // If an error has occured in the above, try to identify what went wrong.
    // Was it an MQ error?
    catch (MQException mqex)
    {
      System.out.println("An MQ error occurred: Completion code " +
                         mqex.completionCode +
                         " Reason code " + mqex.reasonCode);
    }
    // Was it a Notes error?
    catch (NotesException nex)
    {
      System.out.println("An Notes error occurred: " + nex);
    }
    // Was it a Java buffer space error?
    catch (java.io.IOException jex)
    {
      System.out.println("An error occurred whilst writing to the message buffer: " +
                         jex);
    }
    catch (Exception e)
```

```
    {
      e.printStackTrace();
    }

    finally
    {

      System.out.println("and finally...");

      try
      {
        if (inputQ.isOpen)
            inputQ.close();

        if (qMgr.isOpen)
        {
           System.out.println ("Disconnecting from queue manager");
           qMgr.disconnect();
        }
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }

  } // end of NotesMain

} // end of Agent
```

Appendix E. WebSphere Example

This appendix contains the source code of the Domino agent and the Domino servlet that access the *Employee* EJB.

E.1 EJBAgent

```
import lotus.domino.*;
import java.util.*;
import java.io.*;
import javax.ejb.*;
import com.ibm.ejs.samples.phone.*;

public class JavaAgent extends AgentBase {

     private static final String employeeHomeName =
           "com.ibm.ejs.samples.phone.EmployeeHome";

    private EmployeeHome employeeHome;

   public void NotesMain() {

      try {

         // Get Domino Session
         Session session = getSession();
         AgentContext agentContext = session.getAgentContext();
         Document doc = agentContext.getDocumentContext();

         // get Surname from Domino Document
         String surname = doc.getItemValueString("Surname");
         System.out.println("Surname : "+surname);
         System.out.println(employeeHomeName);
         System.out.println("employees home : "+employeeHome);


         // Find EJB

         java.util.Hashtable properties = new java.util.Hashtable(2);
         javax.naming.InitialContext initContext = null;
         try       // Attempt to get Context for our EJB
         {
            properties.put(javax.naming.Context.PROVIDER_URL,
                       "iiop://krypton.almaden.ibm.com:9019");
            properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                       "com.ibm.jndi.CosNaming.CNInitialContextFactory");
            initContext = new javax.naming.InitialContext(properties);
         }
         catch (javax.naming.NamingException e) {
            System.out.println("Error retrieving the initial context: "
                              + e.getMessage());
            return;
         }

           if ( employeeHome == null )
            try    // Now try and get a handle to our EJB
            {
                   System.out.println("Retrieving the home interface...");
                   // this is the JNDI name
```

```
                java.lang.Object o = initContext.lookup("EmployeeHome");
            if (o instanceof org.omg.CORBA.Object)
                employeeHome = EmployeeHomeHelper.narrow((org.omg.CORBA.Object) o);
        }
        catch (javax.naming.NamingException e) {
            System.out.println("Error retrieving the home interface: "
                                  + e.getMessage());
            return;
        }

    Enumeration employees = employeeHome.findByLastName(
                          surname.trim().toUpperCase() + "%" );
    EmployeesBean emp = new EmployeesBean (employees);

    if (!emp.isEmpty()) {
        if (emp.hasMatches()) {
            // if number of the results = 1
            if (!employees.hasMoreElements()){
                doc.replaceItemValue("R_Surname_0", emp.getLastName(0));
                doc.replaceItemValue("R_FirstName_0", emp.getFirstName(0));
                doc.replaceItemValue("R_PhoneNumber_0", emp.getPhoneNumber(0));
                doc.replaceItemValue("R_Initial_0", emp.getMiddleInitial(0));
                doc.replaceItemValue("R_Department_0", emp.getDepartment(0));
            } else{
                // if number of the results > 1
                int i = 0;
                while (employees.hasMoreElements()) {
                    doc.replaceItemValue("R_Surname_"+i, emp.getLastName(i));
                    doc.replaceItemValue("R_FirstName_"+i, emp.getFirstName(i));
                    doc.replaceItemValue("R_PhoneNumber_"+i,
                                          emp.getPhoneNumber(i));
                    doc.replaceItemValue("R_Initial_"+i, emp.getMiddleInitial(i));
                    doc.replaceItemValue("R_Department_"+i, emp.getDepartment(i));
                    emp.getNextData(i);
                    i++;
                }
            }
    } else {
    doc.replaceItemValue("R_Empty", "Name not found" );
     }

    doc.save(true,true);
    employees = null;
    emp = null;
     System.gc();
}
} catch(Exception e) {
    e.printStackTrace();
}
}
}
```

E.2 Dom_Empl Servlet

```
//Import
import lotus.notes.*;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.sun.server.http.HttpServiceRequest;
import com.sun.server.http.HttpServiceResponse;
import com.ibm.ejs.samples.phone.*;

//PhoneBook' servlet class:

public class Dom_Empl extends HttpServlet {
   private final static String kIBMCopyright = "(c) Copyright IBM Corporation 1999";

   // Constants
   private static final String employeeHomeName =
                         "com.ibm.ejs.samples.phone.EmployeeHome";
   // Fields
   EmployeeHome employeeHome;
   // Static fields
   //  Service request handler
   public void doGet(HttpServletRequest request,
                HttpServletResponse response)
                throws ServletException, IOException {
      try {
         ServletOutputStream out = response.getOutputStream();
         response.setContentType("text/html");
         NotesThread.sinitThread();
         String lastName = request.getParameter("Surname");
         if ((lastName == null) || lastName.equals("")) {
            return;
         }
         // CREATE A CONNECTION TO DOMINO
         Session session = Session.newInstance();
         Database db = session.getDatabase("", "EJBAgent.nsf");
         View view = db.getView("AllDocuments");
         Document doc = view.getLastDocument();
         System.out.println("finding the data........");
         // FIND THE DB2 DATA USING EJB
         Enumeration employees =
                employeeHome.findByLastName(lastName.trim().toUpperCase() + "%");
         EmployeesBean emp = new EmployeesBean(employees);
         if (!emp.isEmpty()) {
            if (emp.hasMatches()) {
               String col1;
               String col2;
               String col3;
               String col4;
               String col5;
               out.println("<B>Result from Servlet<B><BR><BR>");
               out.println("<BR><BR>");
               out.println("<TABLE WIDTH=\"100%\" BORDER=1>");
               outputSimpleTableLine(out, "<B>Surname</B>",
                          "<B>FirstName</B>", "<B>MidInitial</B>",
                          "<B>Department</B>", "<B>PhoneNumber</B>");
               // if number of the results = 1
               if (!employees.hasMoreElements()) {
                  doc.replaceItemValue("R_Surname_0", emp.getLastName(0));
                  doc.replaceItemValue("R_FirstName_0", emp.getFirstName(0));
                  doc.replaceItemValue("R_PhoneNumber_0", emp.getPhoneNumber(0));
                  doc.replaceItemValue("R_Initial_0", emp.getMiddleInitial(0));
                  doc.replaceItemValue("R_Department_0", emp.getDepartment(0));
                  doc.save(true, true);
                  col1 = doc.getItemValueString("R_Surname_0");
                  col2 = doc.getItemValueString("R_FirstName_0");
                  col3 = doc.getItemValueString("R_Initial_0");
                  col4 = doc.getItemValueString("R_Department_0");
                  col5 = doc.getItemValueString("R_PhoneNumber_0");
```

```
                    outputSimpleTableLine(out, col1, col2, col3, col4, col5);
                } else {
                    // if number of the results > 1
                    int i = 0;
                    while (employees.hasMoreElements()) {
                        doc.replaceItemValue("R_Surname_" + i, emp.getLastName(i));
                        doc.replaceItemValue("R_FirstName_" + i, emp.getFirstName(i));
                        doc.replaceItemValue("R_PhoneNumber_" + i,
                                      emp.getPhoneNumber(i));
                        doc.replaceItemValue("R_Initial_" + i,
                                      emp.getMiddleInitial(i));
                        doc.replaceItemValue("R_Department_" + i,
                                      emp.getDepartment(i));
                        emp.getNextData(i);
                        i++;
                    }
                    doc.save(true, true);
                    int j = 0;
                    while (j < i) {
                        col1 = doc.getItemValueString("R_Surname_" + j);
                        col2 = doc.getItemValueString("R_FirstName_" + j);
                        col3 = doc.getItemValueString("R_Initial_" + j);
                        col4 = doc.getItemValueString("R_Department_" + j);
                        col5 = doc.getItemValueString("R_PhoneNumber_" + j);
                        outputSimpleTableLine(out, col1, col2, col3, col4, col5);
                        j++;
                    }
                }
                out.println("</TABLE>");
                System.out.println("Done");
            }
        } else {
            doc.replaceItemValue("R_Empty", "Name not found");
            out.println("<B>Name not found<B>");
        }
        employees = null;
        emp = null;
        System.gc();
    } catch (Exception exception) {
    }
    return;
}
public void init(ServletConfig config) {
    // FIND THE EJB and CREATE A CONNECTION WITH THE EJB HOME INTERFACE
    System.out.println("Running the Initialization ..... ");
    java.util.Hashtable properties = new java.util.Hashtable(2);
    javax.naming.InitialContext initContext = null;
    try // Attempt to get Context for our EJB
    {
        properties.put(javax.naming.Context.PROVIDER_URL,
                    "iiop://krypton.almaden.ibm.com:9019");
        properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                    "com.ibm.jndi.CosNaming.CNInitialContextFactory");
        initContext = new javax.naming.InitialContext(properties);
    } catch (javax.naming.NamingException e) {
        System.out.println("Error retrieving the initial context: " + e.getMessage());
        return;
    }
    if (employeeHome == null)
        try // Now try and get a handle to our EJB
        {
        // this is the JNDI name
        java.lang.Object o = initContext.lookup("EmployeeHome");
```

```
            if (o instanceof org.omg.CORBA.Object) {
                employeeHome = EmployeeHomeHelper.narrow((org.omg.CORBA.Object) o);
            }
        } catch (javax.naming.NamingException e) {
            System.out.println("Error retrieving the home interface: " + e.getMessage());
            return;
        }
    }


    // Private methods
    public void outputSimpleTableLine(ServletOutputStream out,
                    String col1, String col2, String col3, String col4, String col5) {
        try {
            out.println("<TR VALIGN=top><TD WIDTH=\"50%\">" + col1
                    + "</TD><TD WIDTH=\"50%\">" + col2 + "</TD></TR>"
                    + col3 + "</TD></TR>" + col4 + "</TD></TR>" + col5 + "</TD></TR>");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Appendix F. Special Notices

This publication is intended to help information systems architects, Java and Domino developers to integrate Domino applications with enterprise ressources using Java. The information in this publication is not intended as the specification of any programming interfaces that are provided by Domino, DB2, CICS, and MQSeries. See the PUBLICATIONS section of the Lotus Programming Announcement for Domino Release 5.0 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, 500 Columbus Avenue, Thornwood, NY 10594 USA.

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 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

Appendix G. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

G.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 253.

The books of the Lotus Solution for the Enterprise Collection are:

- Volume 1 - *Lotus Notes: An Enterprise Application Platform*, SG24-4837
- Volume 2 - *Using DB2 in a Domino Environment*, SG24-4918
- Volume 3 - *Using the IBM CICS Gateway for Lotus Notes*, SG24-4512
- Volume 4 - *Lotus Notes and the MQSeries Enterprise Integrator*, SG24-2217
- Volume 5 - *NotesPump: The Enterprise Data Mover*, SG24-5255

These publications are also relevant as further information sources:

- *Using VIsualAge for Java to Develop Domino Applications,* SG24-5424
- *Lotus Domino Release 5: A Developer’s Handbook*, SG24-5331
- *Designing Web Applications Using Lotus Notes Designer for Domino 4.6*, SG24-2183
- *The Domino Defense: Security in Lotus Notes and the Internet*, SG24-4848
- *Java Network Security, SG24*-2109
- *Enterprise Integration with Domino for S/390*, SG24-5150
- *Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications*, SG24-5276
- *Revealed! CICS Transaction Gateway with More CICS Clients Unmasked*, SG24-5277

G.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

G.3 Other Publications

This publication is also relevant as further information source:

- *Programming Domino 4.6 with Java, Groupware for the Internet*, by Bob Balaban ISBN: 1558515836

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com/`.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER** – to order hardcopies in the United States
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). 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.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** – send orders to:

	IBMMAIL	**Internet**
In United States	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** – send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** – send orders to:

United States (toll free)	1-800-445-9269
Canada	1-800-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1) 408 256 5422 (Outside USA)** – ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). 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.

IBM Redbook 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.

Glossary

application programming interface (API). A set of calling conventions defining how a service is invoked through a software package.

applet. A Java program designed to run within a Web browser. Contrast with application.

application. In Java programming, a self-contained, stand-alone Java program that includes main() method. Contrast with applet.

bean. A definition or instance of a JavaBeans component.

browser. An Internet-based tool that lets users browse Web sites.

call level interface (CLI). A callable application program interface (API) for database access, which is an alternative to an embedded SQL application program interface. In contrast to embedded SQL, CLI does not require precompiling or binding by the user, but instead provides a standard set of functions to process SQL statements and related services at run time.

Customer Information Control System (CICS). A distributed online transaction processing system designed to support a network of many terminals. The CICS family of products is available for a variety of platforms ranging from a single workstation to the largest mainframe.

CICS Access Builder. A VisualAge for Java Enterprise tool that generates beans to access CICS transactions through the CICS Gateway for Java and CICS Client.

CICS Client. A server program that processes CICS ECI calls, forwarding transaction requests to a CICS program running on a host.

CICS Gateway for Java. A server program that processes Java ECI calls and forwards CICS ECI calls to the CICS Client.

class. An aggregate that defines properties, operations, and behavior for all instances of that aggregate.

client. As in client/server computing, the application that makes requests to the server and, often, handles the necessary interaction with the user.

client/server. A form of distributed processing, in which the task required to be processed is accomplished by a client portion that requests services and a server portion that fulfills those requests. The client and server remain transparent to each other in terms of location and platform. See *client* and *server*.

commit. The operation that ends a unit of work to make permanent the changes it has made to resources (transaction or data).

Common Gateway Interface (CGI). A standard protocol through which a Web server can execute programs running on the server machine. CGI programs are executed in response to requests from Web client browsers.

Common Object Request Broker Architecture (CORBA). A middleware specification which defines a software bus—the Object Request Broker (ORB)—that provides the infrastructure

communications area (COMMAREA). In a CICS transaction program, a group of records that describes both the format and volume of data used.

conversational. A communication model where two distributed applications exchange information by way of a conversation; typically one application starts (or allocates) the conversation, sends some data, and allows the other application to send some data. Both applications continue in turn until one decides to finish (or deallocate). The conversational model is a synchronous form of communication.

Data Access Builder. A VisualAge for Java Enterprise tool that generates beans to access and manipulate the content of JDBC/ODBC-compliant relational databases.

database. (1) A collection of related data stored together with controlled redundancy according to a scheme to serve one or more applications. (2) All data files stored in the system. (3) A set of data stored together and managed by a database management system.

database management system (DBMS). A computer program that manages data by providing the services of centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency control, privacy, and security.

DB2 Call Level Interface (CLI). The DB2 call level interface is an alternative SQL interface for the DB2 family of products and takes full advantage of DB2 capability.This implementation closely follows industry standards, such as X/OPEN, to enhance application portability. Currently, the DB2 Call Level Interface functions are compatible with ODBC 2.0, and contain DB2-specific APIs to help exploit DB2 capability.

DB2 for MVS/ESA. An IBM relational database management system for the MVS operating system.

DCE. Distributed Computing Environment. Adopted by the computer industry as a de facto standard for distributed computing. DCE allows computers from a variety of vendors to communicate transparently and share resources such as computing power, files, printers, and other objects in the network.

distributed processing. Distributed processing is an application or systems model in which function and data can be distributed across multiple computing resources connected on a LAN or WAN. See *client/server computing*.

distributed program link (DPL) enables an application program executing in one CICS system to link (pass control) to a program in a different CICS system. The linked-to program executes and returns a result to the linking program. This process is equivalent to remote procedure calls (RPCs). You can write applications that issue RPCs that can be received by members of the CICS family.

dynamic link library (DLL). A file containing executable code and data bound to a program at run time rather than at link time. The C++ Access Builder generates beans and C++ wrappers that let your Java programs access C++ DLLs.

e-business Either (a) the transaction of business over an electronic medium such as the Internet or (b) a business that uses Internet technologies and network computing in their internal business processes (via intranets), their business relationships (via extranets), and the buying and selling of goods, services, and information (via electronic commerce.)

external call interface (ECI). An API that enables a non-CICS client application to call a CICS program as a subroutine. The client application communicates with the server CICS program using a data area called a *COMMAREA*.

external presentation interface (EPI). An API that allows a non-CICS application program to appear to the CICS system as one or more standard 3270 terminals. The non-CICS application can start CICS transactions and send and receive standard 3270 data streams to those transactions.

Enterprise Access Builders (EAB). In VisualAge for Java Enterprise, a set of code-generation tools.

See also CICS Access Builder and Data Access Builder.

file transfer protocol (FTP). The basic Internet function that enables files to be transferred between computers. You can use it to download files from a remote, host computer, as well as to upload files from your computer to a remote, host computer. See Anonymous FTP.

gateway. A host computer that connects networks that communicate in different languages. For example, a gateway connects a company's LAN to the Internet.

graphical user interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

HotJava A Java-enabled Web and intranet browser developed by Sun Microsystems, Inc. HotJava is written in Java.

hypertext markup language (HTML). The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted (called *rendering*) by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphic images, special effects, hypertext jumps to other Internet locations, and information forms.

hypertext transfer protocol (HTTP). The protocol for moving hypertext files across the Internet. Requires an HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web (WWW). See also Client, Server, WWW.

HTTP request. A transaction initiated by a Web browser and adhering to HTTP. The server usually responds with HTML data, but can send other kinds of objects as well.

hypertext. Text in a document that contains a hidden link to other text. You can click a mouse on a hypertext word and it will take you to the text designated in the link. Hypertext is used in Windows help programs and CD encyclopedias to jump to related references elsewhere within the same document. The wonderful thing about hypertext, however, is its ability to link—using HTTP over the Web—to any Web document in the world, yet still require only a single mouse click to jump clear around the world.

Internet Inter-ORB Protocol (IIOP). An industry standard protocol that defines how General Inter-ORB Protocol (GIOP) messages are exchanged over a TCP/IP network. The IIOP makes it possible to use the Internet itself as a backbone ORB through which other ORBs can bridge.

integrated development environment (IDE). A software program comprising an editor, a compiler, and a debugger. IBM's VisualAge for Java is an example of an IDE.

interface. A set of methods that can be accessed by any class in the class hierarchy. The Interface page in the Workbench lists all interfaces in the workspace.

Internet. The vast collection of interconnected networks that all use the TCP/IP protocols and that evolved from the ARPANET of the late 1960’s and early 1970’s.

intranet. A private network inside a company or organization that uses the same kinds of software that you would find on the public Internet, but that is only for internal use. As the Internet has become more popular, many of the tools used on the Internet are being used in private networks. For example, many companies have Web servers that are available only to employees.

Internet protocol (IP). The rules that provide basic Internet functions.

See TCP/IP.

Java. Java is a new programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files. Using small Java programs (called *applets*, Web pages can include functions such as animations, calculators, and other fancy tricks. We can expect to see a huge variety of features added to the Web using Java, since you can write a Java program to do almost anything a regular computer program can do, and then include that Java program in a Web page.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security. Because the JAR format is written in Java, JAR files are fully extensible.

JavaBeans. In JDK 1.1, the specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called *beans*) may have methods, properties, and events.

Java Database Connectivity (JDBC). In JDK 1.1, the specification that defines an API that enables programs to access databases that comply with this standard.

Java Development Kit (JDK) The Java Development Kit 1.1 is the latest set of Java technologies made available to licensed developers by Sun Microsystems. Each release of the JDK contains the following: the Java Compiler, Java Virtual Machine, Java Class Libraries, Java Applet Viewer, Java Debugger, and other tools.

Java Foundation Classes (JFC) Developed by Netscape, Sun, and IBM, JFCs are building blocks that are helpful in developing interfaces to Java applications. They allow Java applications to interact more completely with the existing operating systems.

LAN. Local area network. A computer network located at a user's establishment within a limited geographical area. A LAN typically consists of one or more server machines providing services to a number of client workstations.

LU type 6.2 (LU 6.2). A type of logical unit used for CICS intersystem communication (ISC). LU 6.2 architecture supports CICS host-to-system-level products and CICS host-to-device-level products. APPC is the protocol boundary of the LU 6.2 architecture.

logical unit of work (LUW). An update that durably transforms a resource from one consistent state to another consistent state. A sequence of processing actions (for example, database changes) that must be completed before any of the individual actions can be regarded as committed. When changes are committed (by successful completion of the LUW and recording of the synch point on the system log), they do not need to be backed out after a subsequent error within the task or region. The end of an LUW is marked in a transaction by a synch point that is issued by either the user program or the CICS server, at the end of task. If there are no user synch points, the entire task is an LUW.

messaging. A communication model whereby the distributed applications communicate by sending messages to each other. A message is typically a short packet of information that does not necessarily require a reply. Messaging implements asynchronous communications

method. A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

Multipurpose Internet Mail Extension (MIME). The Internet standard for mail that supports text, images, audio, and video.

online transaction processing (OLTP). A style of computing that supports interactive applications in which requests submitted by terminal users are processed as soon as they are received. Results are returned to the requester in a relatively short period of time. An online transaction-processing system supervises the sharing of resources to allow efficient processing of multiple transactions at the same time.

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, though it has unique values assigned to its attributes.

ODBC Driver. An ODBC driver is a dynamically linked library (DLL) that implements ODBC function calls and interacts with a data source.

ODBC Driver Manager. The ODBC driver manager, provided by Microsoft, is a DLL with an import library. The primary purpose of the Driver Manager is to load ODBC drivers. The Driver Manager also provides entry points to ODBC functions for each driver and parameter validation and sequence validation for ODBC calls.

Open Database Connectivity (ODBC). A Microsoft-developed C database application programming interface (API) that allows access to database management systems calling callable SQL, which does not require the use of a SQL preprocessor. In addition, ODBC provides an architecture that allows users to add modules called *database drivers* that link the application to their choice of database management systems at run time. This means applications no longer need to be directly linked to the modules of all the database management systems that are supported.

Object Request Broker (ORB). A CORBA term designating the means by which objects transparently make requests and receive responses from objects, whether they are local or remote.

protocol. (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate. It's similar to the way that mail, in countries around the world, is addressed in the same basic format so that postal workers know where to find the recipient's address, the sender's return address and the postage stamp. Regardless of the underlying language, the basic protocols remain the same.

proxy. An application gateway from one network to another for a specific network application such as Telnet of FTP, for example, where a firewall's proxy Telnet server performs authentication of the user and then lets the traffic flow through the proxy as if it were not there. Function is performed in the firewall and not in the client workstation, causing more load in the firewall. Compare with socks.

Remote Method Invocation (RMI). In JDK 1.1, the API that allows you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

Remote Procedure Call (RPC). A communication model where requests are made by function calls to distributed procedure elsewhere. The location of the procedures is transparent to the calling application.

sandbox. A restricted environment, provided by the Web browser, in which Java applets run. The sandbox offers them services and prevents them from doing anything naughty, such as doing file I/O or talking to strangers (servers other than the one from which the applet was loaded). The analogy of applets to children led to calling the environment in which they run the *sandbox.*

schema. In the Data Access Builder, the representation of the database that will be mapped. In the Data Access Builder, the mapping contains a set of definitions for all attributes matching all the columns for your database table, view, or SQL statement, information required to generate Java classes.

server. A computer that provides services to multiple users or workstations in a network; for example, a file server, a print server, or a mail server.

Socket Secure (SOCKS). The gateway that allows compliant client code (client code made socket secure) to establish a session with a remote host.

Swing. A Java GUI component kit that simplifies and streamlines the development of windowing components, such as menus, tool bars, dialogs and the like that are used in graphically based applets and applications. Swing is integrated into the JDK.

Transmission Control Protocol/Internet Protocol (TCP/IP). The basic programming foundation that carries computer messages around the globe via the Internet. The suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

thin client Thin client usually refers to a system that runs on a resource-constrained machine or that runs a small operating system. Thin clients don't require local system administration, and they execute Java applications delivered over the network.

transaction. A unit of processing (consisting of one or more application programs) initiated by a single request. A transaction can require the initiation of one or more tasks for its execution.

transaction processing. A style of computing that supports interactive applications in which requests submitted by users are processed as soon as they are received. Results are returned to the requester in a relatively short period of time. A transaction processing system supervises the sharing of resources for processing multiple transactions at the same time.

Uniform Resource Locator (URL). Standard to identify resources on the World Wide Web

virtual machine (VM) A software program that executes other computer programs. It allows a physical machine, a computer, to behave as if it were another physical machine.

Web server The server component of the World Wide Web. It is responsible for servicing requests for information from Web browsers. The information can be a file retrieved from the server's local disk or generated by a program called by the server to perform a specific application function.

workstation. A configuration of input/output equipment at which an operator works. A terminal or microcomputer, usually one that is connected to a mainframe or a network, at which a user can perform applications.

World Wide Web (WWW or Web). A graphic hypertextual multimedia Internet service.

List of Abbreviations

API	application programming interface
AR	application requester
AS	application server
ASP	active server page
ACL	access control list
APPC	Advanced Program-to-Program Communication
ASCII	American National Standard Code for Information Interchange
AWT	abstract window toolkit
BLOB	binary large object
BOA	basic object adapter
CAE	client application enabler
CGI	Common Gateway Interface
CICS	Customer Information Control System
CLI	call level interface
CLOB	character large object
CORBA	Common Object Request Broker Architecture
DBCLOB	double-byte character large object
DBMS	database management system
DB2LSX	DB2 LotusScript Extension
DCE	distributed computing environment
DECS	Domino Enterprise Connection Services
DLL	dynamic link library
DPL	dynamic program link
DNS	domain name server
DRDA	Distributed Relational Database Architecture
EBCDIC	extended binary coded decimal interchange code
ECI	external call interface
EPI	external presentation interface
EJB	Enterprise JavaBean
ERP	enterprise resource planning
ESA	Enterprise Systems Architecture
FTP	file transfer protocol
GIOP	General Inter-ORB Protocol
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IDE	integrated development environment
IDL	interface definition language
IIOP	Internet Inter-ORB Protocol
IMAP	Internet Message Access Protocol
ITSO	International Technical Support Organization

JDBC	Java database connectivity
JDK	Java Development Kit
JNDI	Java naming and direcory interface
JSDK	Java Servlet Development Kit
JSP	Java server page
JTS	Java Transaction Service
JVM	Java virtual machine
LAN	local area network
LC	Lotus Domino Connector
LDAP	Lightweight Directory Access Protocol
LEI	Lotus Enterprise Integrator for Domino
LS:DO	LotusScript Data Option
LSX	LotusScript Extension
LUW	logical unit of work
MAPI	messaging application program interface
MIME	Multipurpose Internet Mail Extension
MQEI	MQSeries Enterprise Integrator
MQI	message queue interface
MQLSX	MQSeries link LotusScript Extension
MVS	Multiple Virtual Storage
MTA	message transfer agent
NC	network computer
NCF	network computing framework
NDS	NetWare Directory Service
NetBIOS	Network Basic Input/Output System
NNTP	NetNews Transfer Protocol
NOI	Notes object interface
NOS	Notes object store
NT	Microsoft Windows NT (new technology)
ODBC	open database connectivity
OLAP	online analytical processing
OLE	object linking and embedding
OLTP	online transaction processing
OMG	Object Management Group
ORB	object request broker
OS/2	Operating System/2
OSF	Open Software Foundation
PC	personal computer
PIM	personal information manager
POP	Post Office Protocol
RACF	Resource Access Control Facility
RAD	rapid application development
RMI	remote method invocation
RPC	remote procedure call
RSA	Rivest Shamir Adleman
RUW	remote unit of work
SDK	software developer's kit

SET	secure electronic transaction
SMP	symmetric multiprocessors
SMTP	Simple Mail Transfer Protocol
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
SMTP	Simple Mail Transfer Protocol
SQL	structured query language
SSL	secure sockets layer
TCP/IP	Transmission Control Protocol/Internet Protocol
UDB	Universal Database
UDF	user-defined function
UDT	user-defined data type
UPM	User Profile Manager
URL	uniform resource locator
WAN	wide area network
WAS	WebSphere Application Server
XML	extended markup language

Index

ITSO Redbook Evaluation

Connecting Domino to the Enterprise Using Java
SG24-5425-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation 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

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

SG24-5425-00

Printed in the U.S.A.

Connecting Domino to the Enterprise Using Java

SG24-5425-00