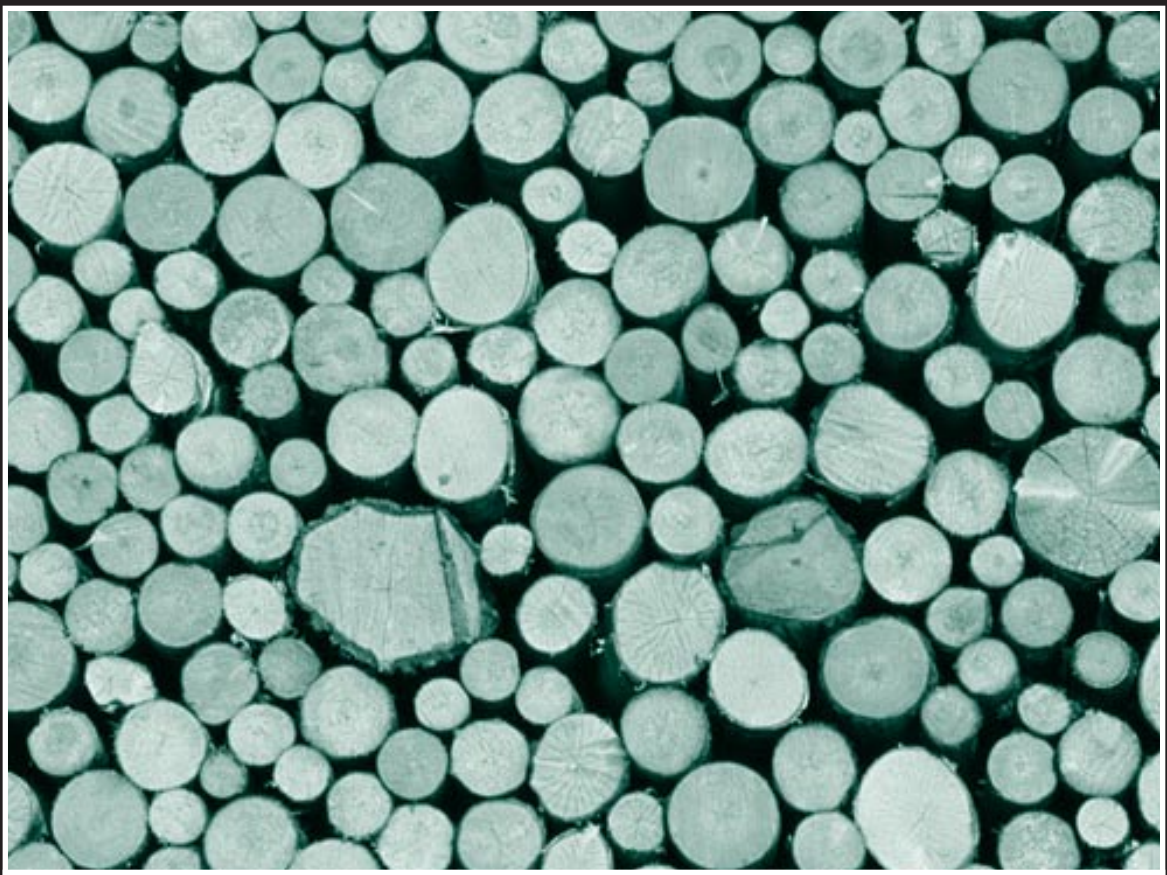
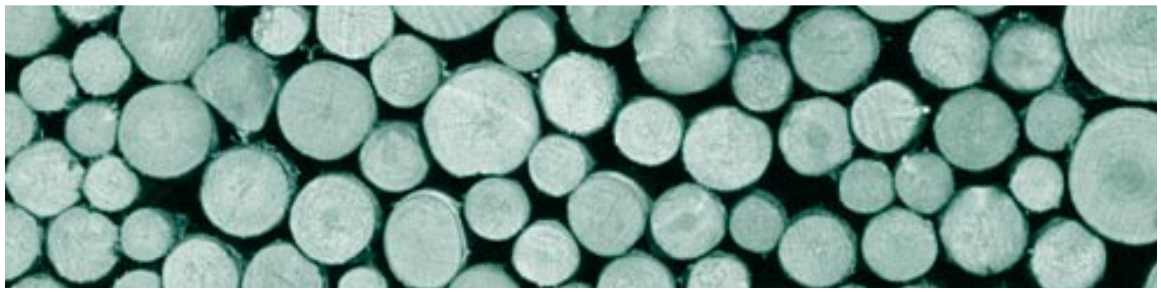


Java Server Programming: Principles and Technologies

Subrahmanyam Allamaraju, Ph.D.



mightywords™

Java Server Programming:
Principles and Technologies

By Subrahmanyam Allamaraju, Ph.D.

Published by MightyWords, Inc.
2550 Walsh Avenue
Santa Clara, CA 95051
www.mightywords.com

ISBN 0-7173-0642-9

Table of Contents

Summary

Building and managing server-side enterprise applications has always been a challenge. Over the last two decades, the role and importance of server-side applications has increased. The twenty-first century economy dictates that e-commerce and other enterprise applications are transparent, networked, adaptable, and service-oriented. These demands have significantly altered the nature of applications. As enterprises migrate from closed and isolated applications to more transparent, networked, and service-oriented applications to enable electronic business transactions, server-side technologies occupy a more prominent place. Despite the disparity of applications that exist in any enterprise, server-side applications are what power twenty-first century enterprises!

The purpose of this article is two-fold. Firstly, this article attempts to highlight the technical needs for server-side applications, and thereby establish a set of programming models. Secondly, based on these programming models, this article introduces the reader to the Java 2, Enterprise Edition (J2EE).

If you are a beginner to server-side programming, or the J2EE, this article will help you gain an overall perspective of what this technology is about. If you are familiar with one or more J2EE technologies, this article will provide the basic principles behind server-side programming, and help you relate these principles to specific J2EE technologies.

Introduction

Building and managing server-side enterprise applications[1] has always been a challenge. Over the last two decades, the role and importance of server-side applications has increased. The twenty-first century economy dictates that e-commerce and other enterprise applications are transparent, networked, adaptable, and service oriented. These demands have significantly altered the nature of applications. As enterprises migrate from closed and isolated applications to more transparent, networked, and service-oriented applications to enable electronic business transactions, server-side technologies become more prominent. Despite the disparity of applications that exist in any enterprise, server-side applications are the ones that power twenty-first century enterprises.

The history of server-side applications dates back to the mainframe era, an era during which only mainframes roared across enterprises. They were centralized in nature, with all the computing—from user interfaces, to complex business process execution, to transactions—performed centrally. The subsequent introduction of minis, desktops, and relational databases fueled a variety of client-server style applications. This contributed to a shift in the way applications could be built, leading to various architectural styles, such as two-, three-, and multi-tier architectures. Of all the styles, database-centric client-server architecture was one of the most widely adapted. Other forms of client-server applications include applications developed using remote-procedure calls (RPC), distributed component technologies, such as Common Object Request Broker Architecture (CORBA), and Distributed Component Object Model (DCOM), etc. Just as these technologies and architectures were diverse, so too were the nature of clients for such applications. The most commonly used client types for these applications were desktop based (i.e., those developed using Visual Basic or other similar languages). Other types of clients included other applications (viz., as in enterprise application integration) and even Web servers.

Subsequently, what we see today are Internet-enabled server-side applications. The underlying technologies for building Internet-enabled server-side applications have evolved significantly and are more mature and less complex to deal with, as you will see later in this article. Apart from providing more flexible programming models, today's server-side technologies are infrastructure intensive. They eliminate some of

[1] As per Merriam Webster's dictionary, the word *enterprise* means a business organization. Enterprise applications are those software applications that facilitate various activities in an enterprise.

the complex infrastructure-related tasks, such as network-level programming, multiple threading and concurrency, distributed transactions, security, and so on.

The way in which server-side applications are built and managed has changed significantly. Unlike applications developed in the '80s and '90s, today's applications are neither centralized (as in mainframes) nor distributed between clients and servers (as in two-tier client-server applications). What we are attempting to build now are loosely coupled distributed applications with the following requirements:

1. **Transparent (not locked to departments within enterprises):** This represents a shift from both centralized and decentralized client-server applications. The need for transparency arises from a need to access information irrespective of where the data is stored and which application maintains the data.
2. **Networked (not isolated):** Applications can be linked to execute several complex business processes.
3. **Adaptable (rather than those built for each specific need):** In an economy where speed matters, it is no longer feasible to build end-to-end applications from scratch for every business requirement. Instead, what is needed is the ability to quickly adapt and enhance existing applications with changing requirements. This implies that enterprise applications cannot be viewed as monolithic units of software, each built for a separate task. Instead, such applications should be viewed (and built) as loosely coupled and flexible components.[2] Depending on the need, you should be able to compose such components to implement various business processes.
4. **Service Oriented (not user-interface oriented):** This requires a marked shift from earlier views, in which enterprise applications were seen as a set of databases and user interfaces. Instead, an application must be seen as a set of services accessed from a variety of channels and user interfaces.

Note: None of these requirements are entirely new. They are simply manifestations of abstraction, encapsulation,[3] and loose-coupling at the enterprise level.

[2] Here I am using the word *component* in a loose sense, implying a piece of loosely coupled unit of software, not necessarily a distributable piece of software.

[3] Booch, G. (1994). Object-oriented analysis and design with applications. Addison-Wesley Publishing Company.

What triggered the possibility of building such applications? Until a few years ago, the technologies that enabled Internet and conventional server-side applications were disparate, complex and difficult to manage; they include the following:

- Common Gateway Interface (CGI) and related Web-development technologies.[4]
- Database-based client-server technologies (for instance, from Oracle and Microsoft).
- Distributed component development technologies, such as CORBA[5] and COM/DCOM.

These technologies attempted to address two different categories of problems and left the problem of unifying them to developers. For instance, the CGI and related technologies helped in building dynamic Web pages, while distributed component technologies considered the network and related distribution issues alone. The client-server technologies did not consider either, and instead left us with a large number of applications that aren't easily adaptable and cannot easily be networked.

This situation has changed significantly over the last couple of years. The Windows Distributed Internet Applications Architecture (DNA), now called *Web Solution Platform*[6], and a part of the recently announced .NET platform[7] suite of technologies from Microsoft, and the J2EE[8] from Sun Microsystems (in collaboration with other leading technology vendors) attempt to integrate the distributed component technologies and Web technologies together, thus avoiding the burden of integrating them at the application level. In addition, these technologies provide more simplified programming models that hide most of the complexity associated with the underlying technologies. You will find more details later in this article.

In this article, my objective is to discuss the fundamental ideas behind server-side programming and discuss server-side programming using the J2EE technology. Sun

[4] Castro, E. (1998). Perl and CGI for the world wide web: Visual QuickStart guide. Addison-Wesley Publishing Company.

[5] Slama, D. et al. (1999). Enterprise CORBA. Prentice Hall Inc.

[6] Web Solution Platform at: http://microsoft.com/business/products/webplatform.

[7] http://microsoft.com/net

[8] http://java.sun.com/j2ee

Microsystems introduced J2EE in 1999. J2EE is essentially a set of specifications produced by Sun Microsystems with participation from several other leading vendors. Application servers[9] complying with J2EE began to appear in 1999 and 2000. Today, J2EE offers a cross-platform (and partly vendor independent) means for developing large-scale server-side applications using the Java programming language.

Following are some of the questions I will address in this article:

1. What is server-side programming? What are the fundamental requirements of server-side technologies?
2. What are the common programming models for server-side programming?
3. What is J2EE? How does J2EE simplify developing distributed component applications for the Internet? How does it relate to the conventional technologies previously mentioned?
4. What are some of the common patterns adapted in J2EE that add flexibility to the infrastructure?

If you are beginning to learn server-side programming and/or J2EE, this article should help you get a quick overview of the underlying principles of server-side programming using J2EE[10] and its applicability for electronic commerce.[11]

This article adopts a *discovery* approach to discussing the Java server-side technologies and their applicability. In this approach, rather than starting with specific technologies and trying to see their details with examples, we will walk through the business/technical problems first, and then lead to specific Java server-side technologies. You will find pointers to reference books, papers, and Web sites for further reading in footnotes. I encourage you to go through these resources for more details.

---

[9] See: http://www.techmetrix.com for a list of application servers offering J2EE capabilities and their relative features. Since these products are evolving rapidly, you should also consult the respective vendor Web sites to get more up-to-date product details. Another source of information is http://www.TheServerSide.com where experienced developers post and debate product reviews. You may also try http://www.jGuru.com for tips and answers to questions on several Java technologies including J2EE.

[10] Allamaraju, S. et al. (2000). Professional java server programming, J2EE edition. Wrox Press Inc.

[11] Allamaraju, S. et al. (2001). Professional java e-commerce. Wrox Press Inc.

Java in Retrospect

Today, Java is the one of the most commonly used programming languages for developing enterprise and e-commerce applications. Since its introduction by Sun Microsystems in 1995, the Java programming language has replaced other programming languages, such as C and C++ (and to a lesser extent even Visual Basic). While developers grappled with these programming languages to implement enterprise and e-commerce applications, the Java programming language offered a platform-neutral alternative such that developers could program object-oriented applications in a platform-independent manner. Although programming languages like C and C++ were flexible enough to develop a variety of complex applications (including client-server and other enterprise applications), they lacked a core architecture capable of integrating the programming requirements of the ’90s that were geared towards the Internet.

It is not an overstatement to say that software development needs changed rapidly during late 1990s. This was fueled partly by various business and technical developments related to e-commerce. The most important of these developments includes the need for rapid development, adaptability, and security. The Java programming language attempts to meet these needs at the infrastructure level. Unlike other standard languages, such as C and C++, and proprietary languages, such as Visual Basic, Java provides a more consistent and infrastructure-intensive runtime. The Java platform’s core runtime (the Java Run-time Environment) has built-in security, multiple threading, and memory management (garbage collection). The object-oriented features of this language, coupled with the standard *application programming interfaces* (APIs) that are included in the platform, help developers achieve rapid development and adaptability. Due to the built-in threading and memory management in Java, it is not uncommon to find Java programs significantly shorter than those developed using other languages, which results in greater software reliability and programmer productivity. Those of you who have attempted to port an application developed in C language to various flavors of UNIX and Windows platforms would be quick to see these advantages.

Apart from Java’s basic object-oriented features, its built-in security and networking facilities and runtime made it a popular language; in fact, Sun Microsystems continues to emphasize these features to this day. Java offers a viable means for developing and deploying a variety of applications—ranging from consumer products and mobile devices, to browser-based applets and multi-tiered

enterprise applications. The fact that the original Java platform could be extended to cater to such diverse application-development needs indicates the strength of its core architecture.

Over the last few years, the Java language has been significantly improved. While the core architecture of Java remained the same, the language has been enhanced in terms of security, usability of APIs, etc. Today, there are three major editions of Java technologies, as follows:

1. **Java 2 Standard Edition (J2SE)[12]:** This is the most commonly used form of Java technology. Also referred to as *Java Development Kit* (JDK), J2SE is a software development kit that includes a set of APIs, tools, and a runtime for developing general-purpose applications.
2. **Java 2 Enterprise Edition (J2EE)[13]:** J2EE is the basis for Java-based enterprise applications. J2EE specifies a standard for developing multi-tier distributed component-based applications. J2EE provides an integrated technical framework for developing Internet-enabled enterprise, and e-commerce applications.
3. **Java 2 Micro Edition (J2ME)[14]:** J2ME provides a lightweight and optimized Java environment for applications and consumer and mobile devices.

In addition, there are several APIs and related technologies under development via a process known as the *Java Community Process*.[15] Through this process, individuals as well as vendors can participate in enhancing existing APIs and can develop new APIs.

Of these editions, the J2SE is the standard platform for developing Java applications for most of the common platforms, which range from Windows-based personal computers, to Macs, to several variants of UNIX, to even some mainframes.[16]

---

[12] See: http://java.sun.com/j2se for more details including API documentation, and downloads.

[13] See: http://java.sun.com/j2ee for the J2EE specification, links to various J2EE technologies, and to download the reference implementation.

[14] See: http://java.sun.com/j2me for more details.

[15] See: http://java.sun.com/aboutJava/communityprocess/index.html. There are over 100 Java Specification Requests (JSRs) under progress now. These JSRs indicate some of the future directions of Java technologies.

[16] See: http://java.sun.com/cgi-bin/java-ports.cgi for a list of ports available from various vendors.

The J2EE consolidates most of the server-side technical infrastructure and development lifecycle requirements into a set of specifications. The purpose of J2EE is to simplify development and management of distributed component-based Web-centric applications.

While J2EE is targeted at large-scale enterprise applications, the J2ME is targeted at consumer electronics and embedded devices. Similar to J2SE and J2EE, the J2ME includes a Java virtual machine and a set of APIs tailored to the needs of applications for consumer electronics and embedded devices.

In this article, my focus is limited to server-side programming with J2EE. As you will see, the J2EE is driving a major section of the Java programming community because the technology addresses most of the infrastructure issues that server-side systems face.

What is server-side programming?

The terms *server* and *server-side programming* are very commonly used. But what is a server? Generally speaking, a server is an application hosted on a machine that provides some services to the other applications (clients) requesting the services. As is obvious from the definition, a server typically caters to several clients, often concurrently. The services offered by a server application can be very diverse—ranging from data, and other information, management to the execution of specialized business logic. Although several types of server applications are possible, you can quickly recognize the following two kinds of servers:

1. **Commercial Database Servers (or Database Management Systems):** Servers that manage data, which come from vendors, such as Oracle, Sybase, and Microsoft. Client applications use standard APIs, such as Open Data Base Connectivity (ODBC) or Java Data Base Connectivity (JDBC), in combination with SQL (Structured Query Language) to access/manipulate the data. In addition to these standard APIs, you may use vendor-specific client access APIs to communicate with the database server. These servers are responsible for consistently managing data such that multiple clients can manipulate the data without loss of consistency.
2. **Web Servers:** The other easily recognizable types of servers are Web servers. A Web server hosts content, such as HTML files. Clients (Web browsers) access this content by submitting requests using Hyper Text Transfer Protocol (HTTP). In its basic mode, a Web server maps a client request to a file in the file system, and then sends the content of the file to the client making the request. Similar to database servers, Web servers can deal with concurrent requests—often several thousands.

Because we are accustomed to working with these types of servers, we often tend to overlook the fact that they are server-side applications, albeit standardized. However, when I use the term *server-side programming*, I do not mean developing such databases or Web servers. Such applications were commodified years ago. These general-purpose server-side applications provide standard, predefined functionality. In the case of server-side programming, we are more interested in developing special-purpose applications to implement specific business-use cases.

Why is it important or relevant to consider server-side applications? Why is it necessary to build applications that can provide services to several client applications? Following are some common scenarios that illustrate the need for server-side applications:

1. **Execute some logic available with a different application.** This is one of the reasons that prompted development of early distributed technologies, such as the RPC.[17] Consider that you are developing an application implementing certain new business-use cases. Suppose that there is some business logic already implemented in a different application. In cases where such functionality is available as separate/isolated class or function libraries, you might consider reusing the same code in the new application. But, what if such logic (and code) depends heavily on some data being managed by the existing application? In such a case, the class/function library cannot be reused as these class/function libraries may fail to execute without the availability of such data. Alternatively, what if such code is dependent on several other parts of the existing application, making it difficult to reuse the code in the new application? One possible solution to deal with this is by executing this logic in the existing application; that is, instead of reusing the code from the existing application, the new application can send a request (with any required data) to the existing application to execute this logic and obtain any results.

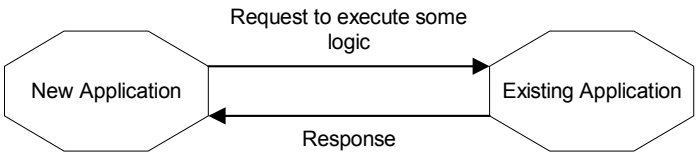


Figure 1: Execute logic available with a different application.

Figure 1 shows a sample scenario. In this example, we are dealing with two different applications executing as two separate processes, but we can extend this case further and execute these applications on two different machines. In the above example, the new application acts as a client for the existing application (server) that provides the required service, that is, executing some business logic.

[17] For a brief introduction to RPC, see: http://www.sei.cmu.edu/str/descriptions/rpc_body.html.

2. **Access information/data maintained by another application.** This is yet another scenario that mandates server-side applications and clients. Database servers and other applications maintain information in a persistent storage fall under this category; however, database servers, such as relational database management systems, typically manage *raw data* (rows of data in various tables). Such data may or may not be directly usable (in the business sense) by other applications that rely on the data. For instance, consider a database holding various accounts of all customers in a bank. A common requirement in such systems is to consolidate all the accounts of a given customer and present a summarized view of all accounts. Depending on that customer's account types, the process of generating the summarized view may involve performing some complex calculations including evaluating some business rules.

   Now consider several applications within the bank performing other banking tasks some of which require a summarized view of all accounts of a customer. For instance, an application responsible for calculating monthly account charges may require the value of the consolidated account balance. When there are several such applications that require this data, it is appropriate to let one of the applications handle this calculation for all other applications.

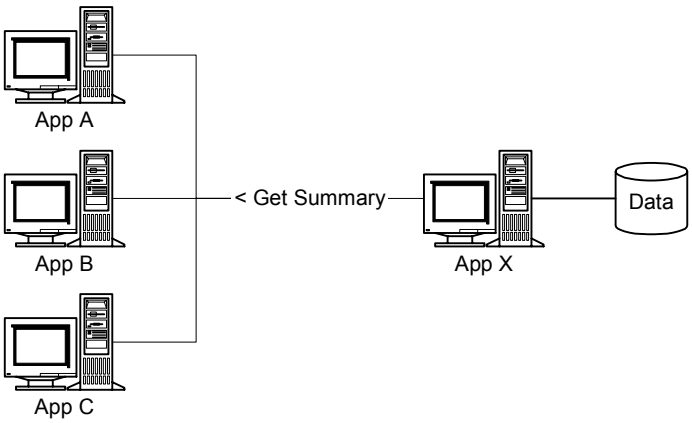


Figure 2: Access information from another application.

Figure 2 illustrates such a case. In this figure, App X is capable of making the necessary calculations and arrives at a summarized view of all customer accounts, while App A, App B, and App C request this information from App X. Instead of each individual application attempting to calculate the summarized view, these applications can delegate the responsibility to App X. Similar to the previous

scenario, this illustration also requires of clients and servers. In this case, App X acts as a server (for the account summary information) for App A, App B, and App C.

Standard Web servers also fall under this category. As discussed previously, Web server technologies let you access and search for data/content maintained by Web servers.

3. **Integrate applications.** In any given enterprise, it is not uncommon to find several applications developed over years that are used for executing different business tasks. You may even find that such stand-alone applications are maintained by different departments of enterprises and developed using different programming languages and technologies. To some extent, such applications were a result of the drive to decentralize businesses from mainframe-based applications. However, as we discussed earlier, one of the requirements encountered by almost all enterprises today is the ability to internetwork such applications so that business workflows can be established across the enterprise.

   To elaborate further, let us take the case of a traditional retail business collecting orders for products via a call center, surface mail, etc. Let us assume that an order-entry application is used to record all orders (in a database) captured via these channels. Let us also assume that the company has developed an in-house order-fulfillment application, which takes the orders recorded in the database and conducts whatever tasks are need to fulfill each order. These two applications are stovepiped because each of these applications addresses specific business tasks in a decoupled manner without any knowledge of the overall business flow. In this case, the overall business flow is to receive (and enter) orders, and then to fulfill each of the orders. Such stovepiped applications cannot, individually, implement the end-to-end business flow. In order to implement this, you need to integrate the applications such that the order-entry application can automatically trigger the fulfillment application, directly or indirectly. This is the enterprise application integration[18] problem, but there are several ways to achieve such integration. One way is to make the order-entry application invoke (or notify) the fulfillment application directly after an order has been captured. The fulfillment application can then start the fulfillment activities thereby streamlining a business flow across these applications.

[18] Linthicum, D. S. (2000). Enterprise application integration. Addison Wesley Longman.

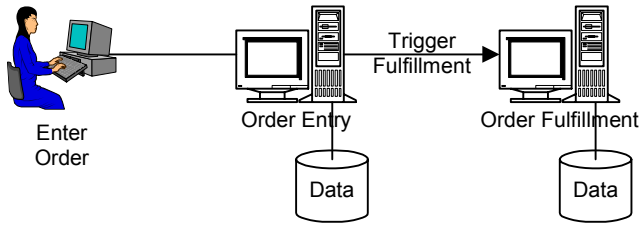


Figure 3: Integrate applications.

Note: This is a simplified application. In practice, you may find that the fulfillment application was developed using some commercial, off-the-shelf enterprise resource planning (ERP) application (such as from SAP). In such cases, the problem of integration gets more challenging; however, most commercial products offer some technical plumbing to allow for such integration. We'll discuss more about such pluming in the next section.

These three cases illustrate some of the typical scenarios that require solutions based on server-side applications. Although simplified, and not exhaustive, the purpose of these illustrations is to identify the need for server-side applications.

In the previous examples, we did not discuss specific details of how clients communicate with the server. Neither did we discuss the programming requirements and the underlying technologies. We'll shortly address these issues.

The server-side technologies (including the Java server-side technologies) can be categorized based on the connectivity model for client-server communication. In the next section, you will find that the connectivity model is what dictates the programming model for different styles of server-side applications. Once we identify the requirements for various programming models, we'll proceed to map these needs to the various technologies under J2EE.

Server-Side Applications—Programming Models

In this section, we will discuss various programming models for developing server-side applications. But, what do we mean by a programming model? A programming model describes the structure of programs and the associated abstractions. In the case of server-side applications, the programming model describes how to develop client and server applications, how to distribute various infrastructure-related responsibilities, and how the connectivity between clients and servers is established.

In the case of server-side applications, the style of connectivity largely dictates the programming model. The style of connectivity has to do with the following questions:

1. **What is the underlying network protocol?** Outside of special situations, most networks and operating platforms use Transport Control Protocol/Internet Protocol (TCP/IP) as the network-level protocol. Unless, the client-server applications use other protocols, such as System Network Architecture Protocol (SNA), TCP/IP is the only protocol that applications have to deal with at the wire level.
2. **What is the underlying application protocol?** Although the most commonly used network level protocol is TCP/IP, client-server applications may use either a connectionful or connectionless approach for application-level communication.
3. **How do clients locate servers?** In a client-server world, there may be several services available across the network. At the network level, the network address specifies a location; however, at the application level, clients will be interested in the services available—not how or where they are located.
4. **Do clients and server communicate synchronously or asynchronously?** In a synchronous connection, the client connects to the server and waits while the server processes the request. With asynchronous communication, the client does not wait, but instead continues to execute.
5. **What is the application-level contract between clients and servers?** Do clients see servers as interfaces with methods, or do clients exchange messages with servers? The answer depends on the application-level protocol and the synchronous/asynchronous communication model. You will see the possibilities shortly.

Depending on the application-level protocol, and whether the communication model is synchronous or asynchronous, server-side application programming models can be classified into the following:

1. **Synchronous Request-Response Model:** This model is meant for method-level communication between clients and servers. In this model, clients invoke methods on server-side objects, and the programming model is similar to regular method invocation on local objects.
2. **Synchronous Stateless Request-Response Model:** This is the programming model for the Web. In this model, clients send requests using HTTP, and the server responds with responses over HTTP. In the last few years, this programming model has gained tremendous popularity, as it is the model that is driving the externally visible portions of electronic commerce.
3. **Asynchronous Communication Model:** In this model, clients communicate with servers by sending messages asynchronously. Clients rely on intermediate message brokers (or message buses) for posting messages. This programming model allows highly decoupled communication from clients to servers, and is widely used for integrating enterprise applications.

Now we will discuss each of these approaches and identify the technical infrastructure needs and the application models.

Synchronous Request-Response Model

The fundamental idea behind this model is that it allows client applications to invoke methods/functions executing on remote applications. Technologies, such as CORBA, Java Remote Method Invocation (RMI), and Enterprise JavaBeans (EJB) follow this model.

The following features characterize the synchronous request-response model approach:

1. **The contract between clients and servers is interface based.** The interface between client and server is the remote interface. It specifies the operations/methods that clients can invoke on servers.
2. **The communication between the client and the server is synchronous.** This is no different from local method calls. Upon invoking a method, the client thread waits until the server responds. This also implies that the server should be up and running for the client to invoke methods on the server.
3. **The communication is connectionful.** The application contract can maintain various requests and responses between the client and the server.

These two features narrow the line between invoking a method on a local object and invoking a method on a remote object.

Basic Requirements

What are the technical requirements for implementing such an approach? Before we can identify the technical requirements, let us study this approach in more detail. The following **Figure 4** shows as application (client) invoking a method in a different process. In object-oriented languages, such as Java, methods exist on interfaces or classes. Therefore, in a Java environment, this approach involves a client invoking a method on an object that exists in a different process (possibly on a different machine).

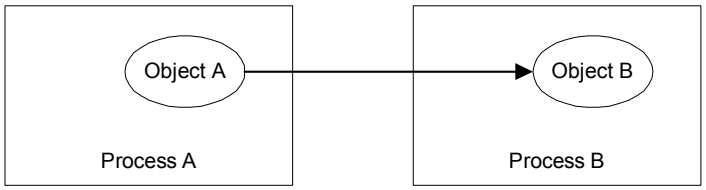


Figure 4: Remote method invocation.

Once you start considering the details of implementing the above, you will come across the following infrastructure-level challenges:

1. **Object References:** In an object-oriented runtime environment, an object can invoke methods on another object as long as the calling object holds a reference to the object being invoked. In Java, an object reference is essentially a handle to an object located in the heap of a process. Because the actual object exists in the heap, an object reference is always local. That is, an object cannot maintain a reference to an object existing in the heap of a different process (unless we are dealing with shared-memory systems, which we are not!). However, the inability of obtaining a reference to a remote object precludes any method invocation in the **Figure 4**.

   In **Figure 4**, because object B is remote to object A, object A needs to obtain, and maintain, a *remote* reference to object B. Such a *remote* reference should be able to maintain a valid reference to an object B existing in process B, and the reference should be valid across the process and network boundaries. But, how to create such a reference? The reference should typically include the following information:

   - The network location of the process holding the object.
   - Any identity associated with the object in the process.

   Figure 5 shows a remote reference to B. The remote reference is an object encapsulating the previously listed information. Note that, the client object maintains a reference to the remote reference, while the remote reference object maintains information necessary for connecting to object B. Thus, the role of this remote reference is to be a proxy for object B.

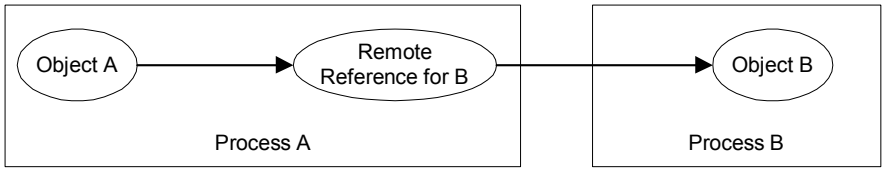


Figure 5: Remote reference.

2. **Method Invocation:** Once the client obtains a remote reference to a remote object, the next step is to use the remote reference to invoke methods on the remote object. To do this, the remote reference should support all of those methods on the remote object meant for remote invocation. For instance, if object B has five methods that are meant for execution by remote clients, the remote reference should support these methods such that client objects can invoke these methods without having to deal with the remote object directly. One of the ways to exposing these methods is to specify an interface including all the methods. Such interfaces are called *remote interfaces*. The remote object (object B) implements this interface as it provides the behavior required of the methods specified in the interface. At the same time, the remote reference should also implement this interface, as the clients deal with the remote interface. In this manner, the remote reference acts like a *proxy* to the remote object. In general, remote references are referred to as *proxy objects* or *stubs*.

   Although both the proxy object and the remote object support (implement) the remote interface, there is one difference—the proxy object does not implement the methods, rather it delegates the calls on this object to equivalent calls on the remote object. **Figure 6** shows the proxy, and shows the remote interface as a circle attached to an object implementing it.

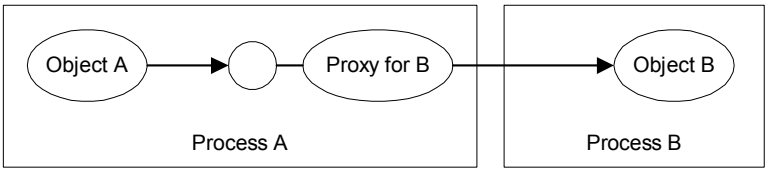


Figure 6: Client-side proxy.

To summarize, a proxy object is responsible for the following:

- To act as local representative for the remote object in the client process.
- To delegate method implementation (over the network) to the corresponding remote object.

Method invocation, however, does not just end by invoking methods on the proxy object. How does this result in a method invocation on the remote object? Should the remote object be network aware and listen for requests over the network? No, it complicates development of the server-side object (i.e., object B).

On the client-side, the proxy object (the remote reference) prevents the calling object (client) from having to deal with the location of the remote object and network semantics. Similarly, on the server-side, we can designate another object to isolate the remote object from the network details. Such an object can receive network-level method requests and translate them into method invocations on the remote object. **Figure 7** completes **Figure 6** by introducing a skeleton on the server side.

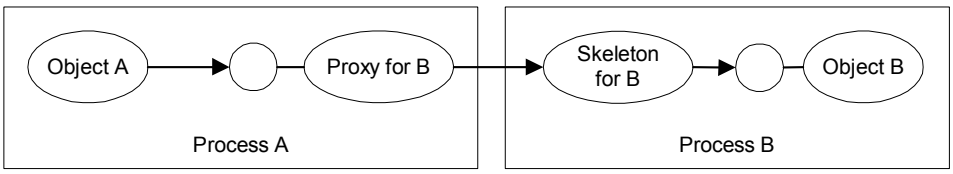


Figure 7: Proxy and skeleton isolating the client from the remoteness of the remote object.

The role of this skeleton is similar to that of the proxy on the client side. Both are intermediate objects that prevent the client and the remote object from having to deal with any network-level details. As far as the client is concerned, it is invoking a method on a local object (the proxy). Similarly, as far as the remote object is concerned, it is being invoked by a local object (the skeleton). The proxy and the skeleton handle the true client-server implementation at the network level.

3. **Transportation of Data:** Let us suppose that we obtain a remote reference to object B. In terms of local objects, in which an object invokes a method on another object within the same process, the underlying runtime (the Java virtual machine [JVM]) passes the arguments, return types, and exceptions between the calling

object and the object being called. Depending on the argument types, the runtime either copies (call by value), or sends references (call by reference) of arguments to the method being invoked. With primitives, the Java runtime copies the primitives such that these exist in the stack of the method being invoked. In the case of objects as parameters, the runtime copies the references such that the references still point to the same set of objects on the heap. But, what happens in the case of remote objects?

- Because references cannot be maintained across processes, all arguments should be copied by value. This applies to both primitives and objects—with one exception, when the argument itself is a remote reference, in which case the remote reference itself can be copied by value.
- Because arguments (primitives as well as objects) can not copied and sent directly across network to the remote process, the state of each argument should be transformed into a raw format that can be transmitted across the wire. This process is called *marshalling*, and can be implemented using Java object serialization.
- Once the argument data is transferred across the wire to the target process, the data must be converted back into the original format. For instance, if an argument is a `java.util.Collection` type, the marshalling process converts the collection into a raw format. On the target process, this raw data must be transformed into an object of the `java.util.Collection` type, equal by value to the object that was originally marshalled. This process is called *unmarshalling*. Object deserialization techniques can be used for *unmarshalling*. **Figure 8** shows these steps.

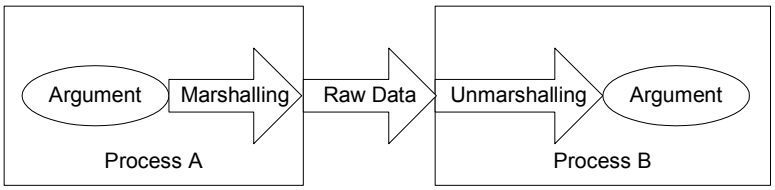


Figure 8: Marshalling and unmarshalling.

The previously listed steps apply to return types, as well. Similar to the arguments, return values should be marshalled into raw data on the server side, and the raw data be unmarshalled into the corresponding Java type on the client side.

Although marshaling and unmarshalling applies to exceptions, too, exceptions are more complicated than argument and return values. Apart from copying the exception value to the client side, the same exception should be raised on the client side so that the client application sees the exception as though it was thrown in the same thread.

Let us now summarize the solutions for the previous three technical requirements:

1. **Solution requires interface specification for each remote object.** Distributed technologies, such as CORBA and COM/DCOM, rely on interfaces specified using interface definition language (IDL). In the case of CORBA, an IDL interface is a programming language-neutral interface. CORBA vendors provide tools using which such interfaces can be converted into programming language equivalent interfaces/classes. In the case of Java RMI, you can specify remote interfaces by extending the `java.rmi.Remote` interface.
2. **Proxies and skeletons are classes that are capable of performing all network-level tasks between clients and server objects.** These tasks include marshalling method requests to the server object, unmarshalling return types, arguments, and exceptions.

Implementing proxy and skeleton classes requires network-level programming. All available distributed technologies today provide tools to automatically generate these classes based on the remote interface specifications. For example, with Java RMI, the RMI compiler generates these classes based on the remote interface specification. **Figure 9** summarizes the technical infrastructure for client-server communication.

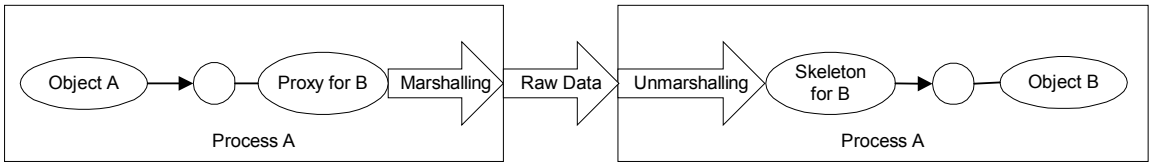


Figure 9: Proxy and skeleton mediating network level method invocation.

Java RMI

Let us now illustrate how to develop client and server objects, and generate proxy and skeleton classes with Java RMI.

Java RMI is the CORBA alternative in a pure Java environment. Java RMI defines a framework for specifying remote interfaces, which is specified in the `java.rmi` package. The J2SE also includes the runtime support for remote objects and client-server communication.

You can specify remote interfaces by making your interface extend the `java.rmi.Remote` interface (this indicates the Java runtime that it is a remote interface) as follows:

```
public interface Process extends java.rmi.Remote {
   public void process(String arg) throws java.rmi.RemoteExcetion;
}
```

This code snippet defines a remote interface with one method. The exception clause is required to denote that this method can throw communication-related exceptions.

The next step is to implement a class that implements the remote interface. This is the server object.

```
   public class ProcessImpl extends UnicastRemoteServer implements
Process {
          public ProcessImpl() {
                super();
          }
          public void process(String arg) throws java.rmi.RemoteExcetion
{
                // Implement the method here
          }
   }
```

Once you have the remote interface, the next step is to use the `rmic` compiler on the implementation class to generate the proxy and skeleton classes. You will find this compiler under the J2SE distribution. This compiler has options to create source files for the proxy and skeleton classes.

Other technologies, such as CORBA or EJB, use similar approaches to specify remote interfaces and generate proxies and skeleton.

Although proxies and skeletons eliminate the need for network-level programming, the following two questions should be considered:

1. **How does the skeleton maintain instances of object B?** Should the server application create an instance initially, for each request, or should it share one instance for all requests? What are the threading/concurrency implications of any strategy?
2. **Location transparency: How do you indicate the location of the server to the proxy object?** Can you predetermine this and force the proxy to invoke a specific implementation of the remote interface available at a given network location? If so, the client application would be dependent on the location of the server. But, is it possible to achieve location transparency?

Instance Management

Once the remote interface, and its implementation are available, the next question is how to make instances of the implementation class available in a runtime (process). Creating a new instance of an implementation object is not adequate to make this implementation available via a remote interface. In order for the server implementation object to be available and be able to process requests coming from client applications over the network, the following additional steps are required:

1. **The process hosting the instance should be reachable via network.** Typically, this boils down to a server socket listening for requests over the network (i.e., the server process should have the necessary infrastructure to do so).
2. **The server process should allocate a thread for processing each incoming request.** Servers are generally required to be multi-threaded.
3. **The server process may host implementations of several remote interfaces.** Depending on some of the information contained in the incoming network request, the server process should be capable of identifying the correct implementation object. For instance, the name of the interface could specify the implementation object for the server. In this case, the server should have knowledge of which class implements which remote interface.

4. **Once the server identifies the correct interface, it should invoke the associated skeleton class.** This class can unmarshall the request parameters and invoke the appropriate method for the implementation class.

In the above list, the task of creating instances depends on whether the client-server communication is required to be stateful or stateless. Address this question when you expect more than one client to invoke methods on a remote object.

The main difference between stateful and stateless communication is whether the remote instance can maintain and rely on instance variable between multiple method calls from a client. For instance, consider that client A is invoking methods on a remote object B. Let's also assume that the remote interface has several methods. The client may invoke these methods one after the other. While implementing a method, the remote object may store data in its instance variables. For instance, it may store some calculation results in an instance variable. If subsequent methods on this object depend on the value stored in the instance variable, the communication is stateful; that is, the server should make sure that the same instance is being used for all requests from a given client. This also means that, while an instance is serving a client, the server should not use the same instance for other client requests. One important ramification of statefulness is that the server will maintain a separate instance for each client. In systems that have a very large number of clients, statefulness may affect performance.

On the other hand, stateless communication does not require the same instance for a given client (i.e., invoking methods). In this case, the server may be able to use ten objects for these ten clients. As a result, stateless communication is more efficient as it converses memory on the server side.

A general practice is to implement remote objects using the stateless approach. Technologies, such as RMI, CORBA, and COM/DCOM provide for stateless remote objects alone. On the other hand, the EJB technology allows for stateful server-side objects, too. We shall discuss more about this later. **Figure 10** summarizes the instance management.

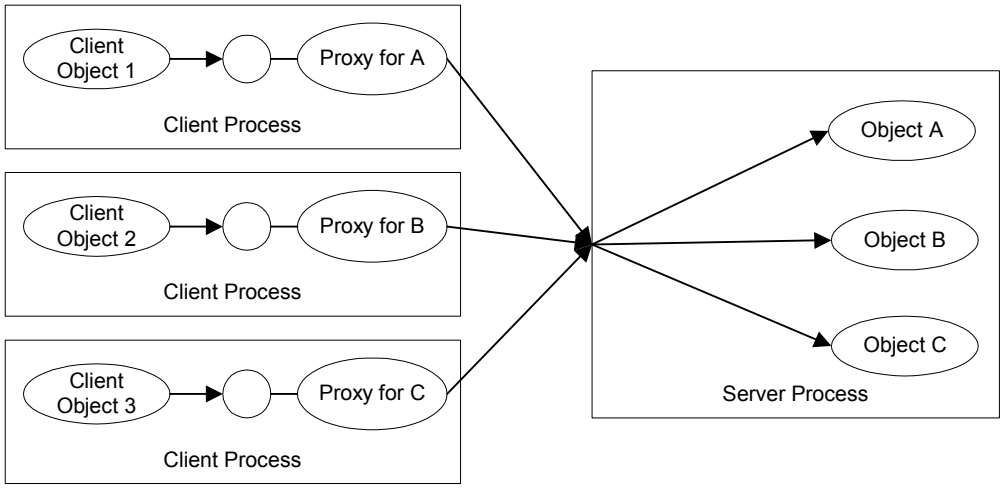


Figure 10: Instance management.

Figure 10 shows three client applications communicating with three different remote objects, all being maintained by a single process. Depending on the interface being used, the server process locates an instance and delegates the incoming request to the instance.

Although this figure shows a single server process, in a typical distributed client-server environment, there may be several server processes, each maintaining a set of remote objects. When there is one single known server process, it is easy to indicate the network location of the server process to the proxy object; however, when the possibility exists for several such processes, it is difficult to manage the network locations for each proxy. We'll address this question in the following section.

Location and Naming

As discussed in the previous section, it is important to achieve location transparency in distributed applications with several clients invoking methods on a number of remote objects that are being managed by several server processes across the network. There is one additional question that we did not consider in the previous question. How does the client application create an instance of the proxy object? In the previous discussion we assumed implicitly that the client would create an instance and use it; however, the client should also be aware of the network location of the server process.

Although these two tasks seem independent, both are typically solved using the same approach, with naming and naming services. The basic idea behind naming is

that remote references can be named. Unlike a remote reference (which essentially points to a memory address within the process), a name is portable, and can uniquely determine a remote interface. Therefore, clients can use names to refer to remote objects; however, a name should still correspond to a remote object available at a given network location. How can this association between names and proxies be maintained?

A naming service is one that can manage names and the corresponding remote references (proxies). Let's consider the following steps in order to understand the role of names and naming services:

1. When the server is started, the server creates a proxy object for each remote object that it is responsible for maintaining, and stores it against a name in another process. Now, this new process has a list of names and the associated proxy objects. The server process can store the network location within the proxy object, or provide sufficient information for the proxy to locate the server.
2. The client supplies the name of a remote object to this new process, and obtains the proxy object corresponding to that name.
3. Because the server originally created this proxy object with the network location embedded, the proxy is network aware. There is no need for the client application to supply this address to the proxy object.

In simple terms, the previous list describes the role of names and naming services. While proxy objects ought to be network aware, the names need not be. During compile time, the client application developer uses a name. At runtime, the client application looks up the naming service to retrieve a proxy object. **Figure 11** shows the role of names and naming service.

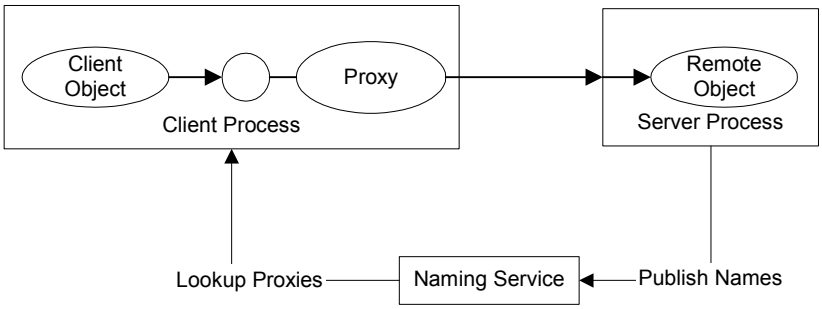


Figure 11: Role of naming service.

In the case of RMI, there is an RMI registry that is equivalent to a naming service. During the server process implementation, the server *binds* (publishes) the name of the remote object, as shown in the following:

```
public class MyServer {
        public static void main(String args[]) {
                try {
                ProcessImpl processImpl = new ProcessImpl();
                        java.rmi.Naming.bind("myProcess", processImpl);
                } catch(Exception e) {
                        // Exception handling
        }
        }
}
```

In this code snippet, the `MyServer` class is the one that hosts an instance of `ProcessImpl`, which implements the `Process` remote interface. When you invoke this class, the main method creates an instance of `ProcessImpl`, and publishes it in the RMI registry. Although it appears that the actual `ProcessImpl` is what is being published in the registry, it is actually a proxy object that gets published. This task is handled under the wraps by RMI.

Note: In order for the previously listed code to work, the RMI registry should be up and running.[19,20]

[19] See: Java Tutorial from Sun at: http://java.sun.com/tutorial for instructions on how to setup RMI.

Once the name is published, the client can look up the RMI registry for the proxy object. The following code snippet in a client application illustrates this task:

```
public class MyClient {
       public static void main(String args[]) {
              try {
               Process                    process                    =
(Process)java.rmi.Naming.lookup("myProcess");
           process.process("xyz");
               } catch(Exception e) {
                     // Exception handling
       }
       }
}
```

In this snippet, the client uses the name `"myProcess"` to locate the proxy from the RMI registry, and invokes a method process with appropriate arguments. Under the covers, this method call will be marshalled via the proxy object to the server process, unmarshalled by the server-side skeleton, and then handled to the `processImpl` object awaiting requests.

Programming Model

What is the programming model for client-server applications based on synchronous requests and responses? The following list explains:

1. Remote interfaces that specify the contract between server-side objects and clients.
2. Proxies and skeletons for handling the network level programming. As discussed in this section, these are tool generated, and you need not implement these classes explicitly.

[20] Also see: Chapter 3 of *Professional Java Server Programming, J2EE Edition* by Subrahmanyam Allamaraju et al. (2000, Wrox Press Inc.) for a more advanced discussion on Java RMI.

3. A server process to maintain server-side remote objects. As you will see later, the J2EE specifies a full-fledged process for maintaining remote objects. Instead of custom developing, you rely on the runtime support provided by commercial J2EE platforms.
4. Naming services for publishing proxy objects. This is an essential piece that allows location transparency.
5. Clients to lookup for proxy objects using names.
6. Clients invoke methods on remote objects.

The purpose of this programming model is to isolate the network-level tasks involved in client-server communication. The previously listed code snippets contain very minimal additional code to indicate that you are dealing with remote objects and not local objects. Once a proxy object is obtained from the naming service, the client uses it as if it were a local object.

The same is true with the server-side object. The object does not know if the caller is local or remote; the only indicator to point to such communication is the task of publishing a name in a naming service.

The goal of this programming model is, therefore, to make the programming task as close as possible to programming local objects. While technologies, such as RMI and CORBA, meet these criteria to a fair extent, the J2EE environment offers a much more simplified approach to developing clients and server objects.

Synchronous Stateless Request-Response Model

In the previous section, we focused on an approach that was based on remote interfaces. In this section, we'll discuss another approach for client-server communication that is based on HTTP.

HTTP[21] is the most widely used protocol across the Internet, connecting browser-based clients and Web servers. The primary purpose of HTTP is to define requests and responses between browsers and Web servers. In this model, a browser sends an HTTP request to a Web server (also called an *HTTP server*), which generates some response and sends back to the client as a HTTP response.

HTTP is a generic protocol that only specifies certain types of requests, as well as the structure of requests and responses. Unlike the remote interface-based model in which you specify a remote interface, with HTTP, the contract between the client (browser) and the server (Web server) is fixed by the HTTP. Clients and servers can however express parameters and results within HTTP requests and responses.

In the following discussion, the term *client* means Web browser, and the word *server* means a Web server. However, note that it is possible to build custom HTTP clients that are not Web browsers and custom HTTP servers that are not full-fledged Web servers.[22] In fact, technologies, such as XML-RPC[23] and Simple Object Access Protocol (SOAP) are based on non-browser clients interacting with server-side objects via HTTP.

---

[21] The HTTP/1.0 protocol is documented in RFC 1945. A later version of this protocol, HTTP/1.1 is documented in RFC 2068. These protocols are released as "requests for comments" by the Internet Engineering Task Force (IETF). Although these documents are not called specifications, these documents do not change once released. Any changes to these documents are released as new RFCs. See http://www.ietf.org for a large collection of RFCs that deal with different layers and protocols of Internet technologies. Refer to http://www.ietf.org/rfc/rfc1945.txt for RFC 1945, and to http://www.ietf.org/rfc/rfc2068.txt for RFC 2068.

[22] It is important to remember that a browser is a general purpose HTTP client. Based on user interaction, the browser makes HTTP requests to specified addresses (in links or in the address bar) to Web servers, receives HTTP responses (downloads), and renders the response (typically HTML with embedded images, documents etc.). However, in the context of client-server applications, it is possible to build more specialized HTTP clients and servers that are meant for implementing specific application logic that does not involve content at all.

[23] See: http://www.xml-rpc.org. XML-RPC is based on the idea of using HTTP as the wire-level protocol for invoking remote objects. SOAP is also based on the same idea. For more historic information on SOAP and XML-RPC refer to "SOAP Part 1" by Simeon Simeonov, *XML Journal*, Volume 1, Issue 4.

The HTTP

The following demonstrates how the basic HTTP protocol works between Web browsers and Web servers:

- The client composes a HTTP request and sends it to a HTTP server (located at a HTTP address as http://). You can identify a Web server with a HTTP URL.[24] An HTTP request includes a request header and a request body. The request header includes a request URI[25] and optionally certain parameters (known as query parameters). These parameters are specified as name-value pairs at the end of the request URL. Depending on the type of the request (which will be discussed shortly), the request body might be empty, or it might contain MIME[26] messages.
- The HTTP server receives the message and attempts to identify the resource specified in the request. For instance, if your HTTP request points to http://www.mightywords.com/bestsellers/index.html, the Web server identified by www.mightywords.com tries to map the resource /bestsellers/index.html to a physical HTML file available under the Web server's file system. Alternatively, the Web server could map a request, such as http://www.mightywords.com/buynow, to an application that can generate custom HTML.
- If the server is able to identify a resource, it sends the resource as an HTTP response indicating what type of resource is being sent. The server also sends a status message.[27]

---

[24] *URL* stands for Universal Resource Locators, and is used to locate resources on the Internet. A HTTP URL uses the HTTP protocol to locate resources, and starts with *http://*. Other types of URLs include *ftp://*, *file://*, *mailto*, etc. URLs are documented in RFC 1738, and can be accessed at: http://www.ietf.org/rfc/rfc1738.txt.

[25] *URI* stands for Universal Resource Identifier, and is the way you can identify resources over the Internet. Although URIs are usually used for Internet resources, this scheme can be applied to any resource in an application, or in a network. Note that a URL is different from a URI. One of the basic differences is that, a URI does not indicate a protocol. Secondly, a resource identified by a URI may be physical (say, a file) or may be abstract (a name). URLs are documented in RFC 1738, and can be accessed at: http://www.ietf.org/rfc/rfc1808.txt

[26] *MIME* stands for Multipurpose Internet Mail Extensions, and is documented in RFC 1521 and RFC 1522. MIME allows complex message bodies (not necessarily human-readable) for Internet messages.

[27] Refer to http://www.ietf.org/rfc/rfc1945.txt for a list of possible HTTP status messages.

- If the server cannot resolve the request (for instance, if the file was not found), the server sends a status message indicating the possible error. This process is shown in **Figure 12**.

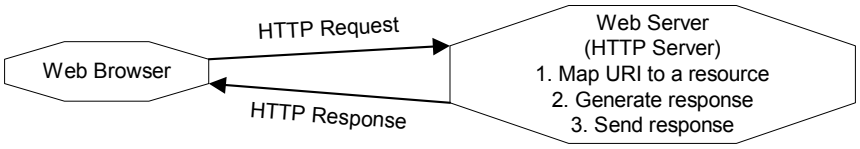


Figure 12: HTTP request-response process.

As you can see from the description, this is a very simple protocol and is primarily used to send *independent atomic* requests and receive responses. The Web server treats each request as a new request and executes the logic listed in the **Figure 12**.

The HTTP/1.0 describes two types of requests—GET and POST, as follows:

1. **GET Request:** The GET request queries for a resource on the Web server. It may optionally include query parameters to indicate to the server the criteria used to querying a resource. The idea behind GET is that it is a read-only request.[28] That is, the server should not modify any data on the server side while it is invoking a request.
2. **POST Request:** This is the most commonly used request type for submitting information (for instance, using HTML forms) to Web servers. Unlike GET requests, POST requests help clients send information to Web servers such that the server can implement business logic and generate a response based on the results.

The HTTP/1.1 adds six more types of HTTP requests: HEAD, OPTIONS, PUT, TRACE, DELETE, and CONNECT. In general, the most commonly supported and used request types are GET and POST. Other types of requests are rarely implemented by Web servers, and are rarely used. If you are interested in more details, refer to RFC 2068.

[28] Such requests are also called as *idempotent requests*.

HTTP Programming Needs

HTTP does not specify how to implement the logic needed to respond to HTTP request. It is up to the HTTP server to honor the request in any manner deemed satisfactory to generate response. That is, it is up to the implementation of an HTTP server to implement the logic or provide a programming model and framework such that you can plug-in programs to process HTTP requests. Consider the following possible scenarios:

1. **Standard HTTP servers:** By default, HTTP servers provide documents, such as HTML pages, to incoming requests. This HTTP application is document-centric and is the crucial element for the World Wide Web.
2. **Standard HTTP Servers with Additional Programming Capabilities:** A Web server may provide programming capabilities such that you can write arbitrary programs to receive HTTP requests and generate HTTP responses. We will focus on this model later in this section.
3. **Custom HTTP Servers:** In certain cases, you may consider building custom HTTP servers for specific application requirements. For instance, you may consider this route if your application needs to process special-purpose HTTP requests (not necessarily generated by Web browsers).

All three approaches can be used in document-centric (or content-centric) applications; however, the third approach may also be used for data-/service-centric applications in which the response need not be human-readable or browser-renderable content.

Of these scenarios, standard Web servers, such as Apache Web server[29], iPlanet Web Server[30], or Microsoft's Internet Information Services (IIS)[31], provide the first and second capabilities, albeit in various forms. The following list summarizes these models:

[29] See: http://httpd.apache.org for more information. This is most widely used Web server developed as open-source.

[30] See: http://www.iPlanet.com/products for more information. The iPlanet Web server used to be known as *Netscape Enterprise Server*.

[31] See: http://www.microsoft.com/iis for more information. Note that this product is usually bundled with various flavors of Windows software.

1. **Common Gateway Interface (CGI):** This is a very popular model for generating dynamic content. The basic idea behind CGI is to invoke programs registered with a Web server. For instance, you may specify that /bar/buy should invoke a buy.pl (a Perl program). When the server receives a request with this URI, it immediately invokes this Perl program using a Perl interpreter. This program will have access to whatever information is contained in the incoming HTTP request. The information includes query parameters and the POST data. The Perl program can implement any logic (database query, updates, etc.) and output HTML content. The Web server then sends this content as HTTP response.

   You can use programming languages, such as Perl, C, Tcl, and even shell scripts to develop CGI applications. Almost all Web servers support CGI. Refer to vendor specific documentation for more details.

   CGI was one of the earliest models to address the need for generating dynamic content. Although, this technology gave way to later technologies, such as Java servlets, JSP pages, and active server pages (ASP), there are still several very popular Web sites built using this technology.

2. **Web Server Extensions:** There are two popular Web server extensions available from iPlanet's Netscape Server Application Programming Interface (NSAPI)[32], and Microsoft's Internet Server Application Programming Interface (ISAPI).[33] Both of these APIs let you develop applications for processing HTTP requests. Although both technologies can be used for developing custom Web applications, in the context of Java, you will find that intermediate layers developed using NSAPI and ISAPI help glue Web servers with commercial J2EE platform implementations. (Note that both ISAPI and NSAPI predate server-side Java technologies.)

   Common to both models is the ability to specify some means of application logic while processing an HTTP request, as shown in the **Figure 13**.

[32] See: http://developer.netscape.com/docs/manuals/enterprise/nsapi/index.htm.

[33] See: http://msdn.microsoft.com and browse through "Platform SDK Documentation | Web Services | Developing ISAPI Extensions and Filters."

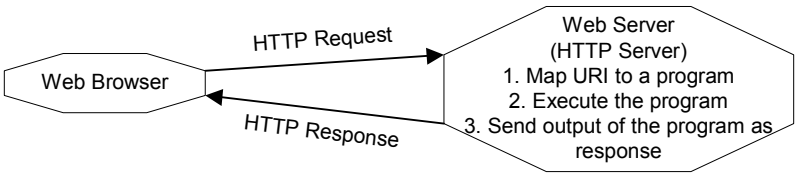


Figure 13: Custom logic execution during HTTP request-response process.

Upon mapping a URI to a program, the Web server executes the associated program and sends the program's output—the response—to the client.

Some of the CGI technologies execute the custom programs outside the server process with an external process launched for each HTTP request. However, such an approach may not scale well as it consumes too many server resources. Other CGI technologies, such as mod_perl[34], execute the custom programs within the server process.

Given the previous overview of these technologies, let's identify the requirements for a programming model for HTTP-based applications, as follows:

1. **Custom Programming Logic:** The first requirement is the ability to plug custom application logic during the HTTP request-response process. Depending on the type and contents of a request, the Web server should be able to identify the corresponding program and invoke it. For example, consider a customer self-registration form in a Web site. When the user submits the form (with a POST request), the Web server can invoke a program that you registered for this purpose. In this program you can open a database connection and store the data in a database.
2. **Dynamic Content:** The second requirement is the ability to allow creation of dynamic content. Without such a facility, you will be forced to custom generate HTML content (programmatically) using the program that was used to process the data. The easiest method of content generation, in terms of usability, is to rely on templates. A template is a sort of a placeholder for static and dynamic content. While the static content is fixed for all requests, the dynamic content may depend on the nature of the request. For instance, a welcome page in an Internet e-mail site

[34] See: http://perl.apache.org.

shows the name of person logged in. This name is dynamic (as are the e-mail folders, etc.) while the rest of the page, including the structure, is static.

The first set of requirements stem from the need to implement custom logic and to generate content dynamically for each request. Ultimately, what the client (the browser) is interested in is a response that can be rendered by a browser. For the developer, what is required is a technology that lets him or her prepare content without having to program (the second point above addressed this need).

Note: Most of the time, HTTP is used for document-centric (or content-centric) applications. This is how most of the e-commerce sites are developed today.

There is, however, another reason to use HTTP as the communication protocol between clients and servers that is not based on content. Consider a client—not a browser—application sending some data to be processed on the server side (not necessarily in a Web server) and receiving the response from the server. In the previous section, we discussed interface-driven technologies, such as RMI, however, there are certain situations in which you may prefer HTTP to RMI. For example, you may not have access the naming service from the client side due to security restrictions. This could happen if, for instance, your servers (including the naming service) were within the secure network and external access was restricted to Web access, or if your client was developed using a different programming language and couldn't perform naming and remote method invocation.

Under such conditions, HTTP provides a better alternative. It is programming language independent—all you need is the capability to make HTTP requests programmatically. Rather than obtaining a proxy and invoking methods on the proxy, you can send HTTP POST requests (with parameters sent as POST parameters) to a Web server. The Web server can then execute the necessary logic (refer to point 1 above), and send an HTTP response. **Figure 14** illustrates this approach.

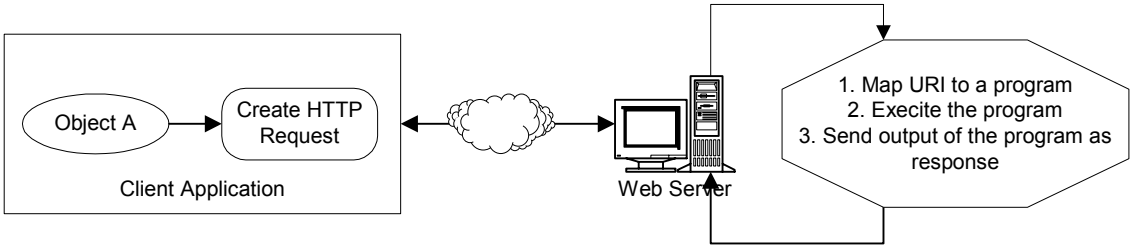


Figure 14: HTTP for client-server communication.

While you might find this approach forces you to deal with custom HTTP code on the client side, note that it offers greater flexibility. Secondly, XML-based protocols, such as SOAP, function this way and future Web service-based e-commerce applications are expected to depend on HTTP more than they do today.

Having discussed the HTTP-based alternative for client-server communication, let us go back to the requirements for the programming model. So far, we've identified the need for a programming model that includes the custom logic to process HTTP requests, and a technology that generates dynamic content without requiring programming; but there are other needs to consider when dealing with HTTP.

HTTP is a stateless protocol. In order to understand what statelessness is, consider the remote interfaces discussed in the previous section. Once you obtain a remote reference, you can continue to invoke methods on the remote reference. All such method invocations will use the same network connection. The server can also maintain state, if required, for each client. In this approach, the client and server maintain a connection throughout. The same happens when you are dealing with File Transfer Protocol (FTP) or even a database connection. Because multiple requests use the same connection, the server object can keep track of the state of conversation with the client.

In the case of HTTP, however, connections are closed[35] at the end of each request. In this protocol, either the browser or the server may close the connection during, or at the end of, a request. For example, if you make two HTTP requests to a Web server, as

[35] It is worth mentioning here that HTTP/1.1 has a special request header called *keep-alive*. When this header is specified, the server does not close connection at the end of each request. In this manner, the browser may use the connection for multiple requests. However, this does not make HTTP a stateful protocol, since HTTP does not rely on the fact that the connection is being reused.

far as the client is concerned, it makes two different connections to the Web server. As far as the Web server is concerned, these two requests are independent.

This is a serious limitation with Web-based transactional applications (such as e-commerce applications) in which users submit a series of requests interactively (much like in a conversation). In such a case, the Web server must identify the client in some manner such that conversation can occur. In order to solve this, several solutions evolved, including cookies[36], and URL encoding, both of which work over HTTP.

The purpose of both of these solutions is to continuously exchange a token with each request between the browser and the server. When the browser makes the first request, the server sends a special token along with the response. The browser sends this token with every subsequent request. The server can, therefore, use this token to identify the state of the conversation. **Figure 15** explains this technique.

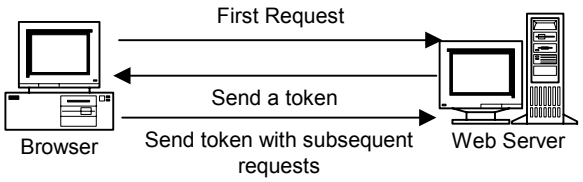


Figure 15: Session via a token.

Both cookies and URL encoding rely on the approach illustrated in **Figure 15**. The only difference is that cookies may be exchanged automatically between browsers and servers, whereas for URL encoding, the programmer has to encode the token programmatically with each hyperlink in the content being generated. The result of these solutions is that, with the help of this special token, the server will be able to establish a *session* for each client.

Yet another requirement for this programming model is the ability to manage state for each session (that is, the server-side programs should be able to store some information during a request and retrieve it later during the same session). Although you may use databases for such purposes, it is sometimes more convenient to store this state in memory, particularly when such a state is temporary. You may, for example, store the information that was read from the database during a request in memory. Alternatively, you may store some information posted by the user in a request. This is

[36] See: http://hotwired.lycos.com/webmonkey/webmonkey/geektalk/96/45/index3a.html.

what is called *state management*. With such a facility, server-side programs can store the conversational state in memory.

Programming Model

To summarize, following are the requirements for the programming model:

1. A programming framework for including custom programming logic during the HTTP request-response process.
2. A facility to generate dynamic content with minimal programming requirements.
3. The ability to manage sessions.
4. The ability to manage state in memory for each session.

We will examine such a model in terms of Java servlets and JSP pages later in this article.

Asynchronous Communication Model

In the previous sections, we discussed two models—synchronous request-response model and synchronous stateless request-response model. While these models follow different approaches for client-server communication, they are both synchronous in nature (that is, they require that the client application is connected to the server application in order for the client to request services of the server). Without such connectivity, the client would not be able to communicate with the server. This is a natural model for client-server communication; it happens between Web browsers and Web servers, and between database clients and database servers. In a given enterprise, it is easy to spot various clients and servers that rely on connections for communications. There are, however, certain criteria for which such a model is not appropriate, as follows:

- Scalability: Synchronous communication between clients and servers affects scalability. *Scalability* is an application's ability to process increased number of requests without significantly losing the response time. With today's Web-based applications, scalability is of paramount importance as the openness and availability of the Internet allows an unlimited number of users to access such applications. With several synchronous communication requests across various parts of an application, scalability will be affected as a result of increased network traffic (and associated tasks, such as marshalling, unmarshalling, etc.).
- Responsiveness: One of the factors affecting responsiveness is the deterioration of scalability. As the amount of time spent in synchronous network communication increases, the servers tend to respond more slowly. In certain instances when this is not the case, the server may not be required to respond to the client at all, or it may be allowed to process the request later. In both the circumstances, synchronous communication is not the preferred communication model.
- Availability: In addition to the above, synchronous communication is not suitable when the availability of the server is not guaranteed. For instance, the server application may not be up and running to receive client requests, process the requests, and respond immediately. The server could be down (completely unavailable), or it could be busy performing other tasks and, therefore, incapable of accepting client requests.

Under these circumstances, asynchronous communication is more appropriate. But, how do you achieve asynchronous communication? How can you even send a request when the recipient is not guaranteed to be available to receive a request? In order to answer this question, consider the analogy of a person sending a surface mail to another person.

In this mode, the recipient need not be available to receive the mail at the time of the sender sending it. The sender delivers the mail to a third party (the postal service). The postal service then routes the mail as per the destination address. Once the mail reaches its destination, the postal service delivers the mail to the recipient's mailbox. The recipient opens the mailbox at his or her leisure.

In this mode, the recipient need not be available during any stage of the communication—not when the sender posts the mail or even when the postal service delivers it to his or her mailbox. The presence of the intermediaries (the postal service and the mailbox) helps this process. Without both of these, it is impossible for the sender to send a message when the recipient is not available to receive it.

Compare this analogy to a telephone conversation. The conversation cannot happen if the call recipient is not available to answer the telephone. An answering machine can, however, receive a message from the caller, and then relay it to the recipient on demand.

Therefore, for asynchronous communication to take place, we need an intermediary to collect and retain the message until the recipient (the server) is ready to receive the message. This is the basic idea behind asynchronous client-server communication, as depicted in **Figure 16**.

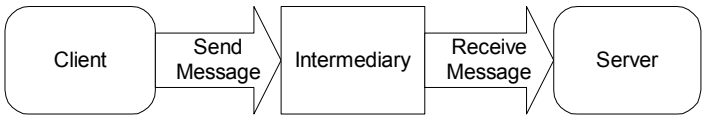


Figure 16: Role of an intermediary for asynchronous client-server communication.

In this figure, the client posts the message intended for the server to the intermediary. The intermediary holds the message until the server receives the message. Thus, the responsibility of availability now shifts from the server to the intermediary. For this communication to take place, the intermediary should be available to take and dispatch messages. Compare this with the previously mentioned analogy of the postal service. If the postal service was not available to accept or

deliver mail, you couldn't send or receive mail. Of course, the postal service preserves *availability* by having mailboxes where you can drop mail and mailboxes from which you can pick it up.

This type of communication is also referred to as *message queuing* and is considered crucial for enterprise and business-to-business (B2B) application integration.[37,38] The technology that facilitates such communication is traditionally called, *message-oriented middleware*.[39]

In the following discussion, we use the terms *sender* and *recipient* in place of *client* and *server,* respectively. The terms *sender* and *recipient* are more commonly used when dealing with applications that exchange messages asynchronously.

Following are two fundamental approaches you can use to establish asynchronous communication between communicating parties:

1. Message queuing
2. Publish and subscribe

Although both of these approaches can be implemented using similar infrastructure, they differ in the way senders send messages and recipients receive messages.

Message Queuing

Message queuing is the simplest form of asynchronous communication between senders and recipients. In this form of communication, the intermediary maintains several message queues, one queue per sender-recipient pair. The sender puts messages into the queue, and the recipient gets messages from the queue; thus, the queue acts as a buffer for this communication. **Figure 17** shows how message queuing can take place between senders and recipients.

[37] Linthicum, D. S. (2000). Enterprise application integration. Addison-Wesley.

[38] Linthicum, D. S. (2001). B2B application integration. Addison-Wesley.

[39] Message oriented middleware also includes message broker services in addition to facilitating basic message handling. A message broker typically provides for message transformation, business level message routing (basic on certain configuration, or based on the contents of a message) etc. See the above references for more details on message brokers.

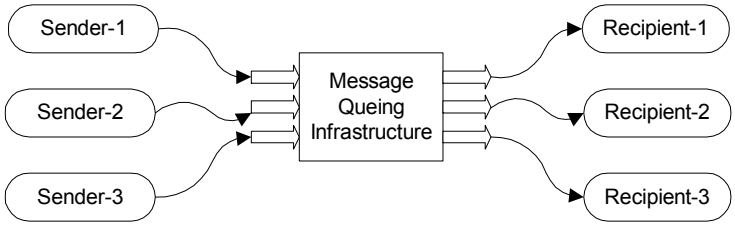


Figure 17: Message queuing.

As the figure shows, the message queuing infrastructure (the message-oriented middleware) maintains the queues for each sender-recipient pair. With this approach, the sender puts messages into a queue meant for a recipient; the recipient can then retrieve the message from the queue. **Figure 17** shows three queues in which senders can put messages and from which recipients fetch them.

But, how does the recipient know there is a message in the queue to retrieve? There are two possible approaches, as follows:

1. **Polling:** With this approach, the recipient keeps polling the queue to check if there is a message in the designated queue.
2. **Notification:** With this approach, the recipient receives a callback handler with the message queuing infrastructure, and it invokes this handler when there is a message to be fetched. The queuing infrastructure can repeat its attempts in invoking this handler until the queue receives it.

Of these approaches, the second approach relieves the recipient of having to check the queue continuously. Vendors of message queuing products provide one or both of these approaches for the recipient.

Message senders may use the following procedure to put messages into a queue:

1. Connect to the message queuing infrastructure for a specific queue.
2. Create a message.
3. Send a message.

Message recipients may use the following steps to receive messages from the queue:

1. Connect to the message queuing infrastructure for a specific queue.
2. Poll for messages or register a handler.
3. Receive message.

Message queuing is also called, *point-to-point messaging,* as a queue is specific to a sender-recipient pair. (Note that a sender may act as a recipient for another queue.)

Note: We have not discussed the contents of a message. Because the infrastructure is concerned only with messages and queues, the format is up to the sender and recipient. Senders and recipients may use any format (text or binary) for describing the contents of messages.

Publish and Subscribe

Unlike the message queuing approach, the *publish and subscribe* approach is based on certain applications that publish messages and certain other applications that subscribe to certain kinds of messages based on a topic. Message queuing allows for asynchronous communication between two applications, but what is the relevance of the publish and subscribe approach?

As an analogy, consider a simple stock-tracking application. Depending on the values of different stocks, the application may fire events, such as "XYZ stock fell below USD50.00." If you're interested in such an event, you may register with the stock-tracking application and request that it notify you whenever the "XYZ stock fell below USD50.00" event occurs. You will then be notified whenever the specific stock falls below the specified value, and you will not receive any notification as long as the stock is either below or above USD50.00. As a *consumer*, you specify the kinds of events (or, more generally, messages) that you are interested.

This is the basic idea of the publish and subscribe approach. This approach is appropriate when there are several applications generating different kinds of messages and certain applications interested in specific types of messages. For instance, in an enterprise, you may find several applications performing different business processes. Each of the processes operates on business data, and change the state of business data. Depending on the specific state of a business data (i.e., the value of a balance), you may have to initiate one or more business processes. In this case, the application

processing the data may post a message stating that a certain state has occurred or that a certain event happened. There may other applications posting similar events. Now, consider a messaging infrastructure that allows applications to be registered for notification on specific message topics. Whenever matching messages are posted, the infrastructure notifies the applications that are registered against specific topics. **Figure 18** shows such a scenario.

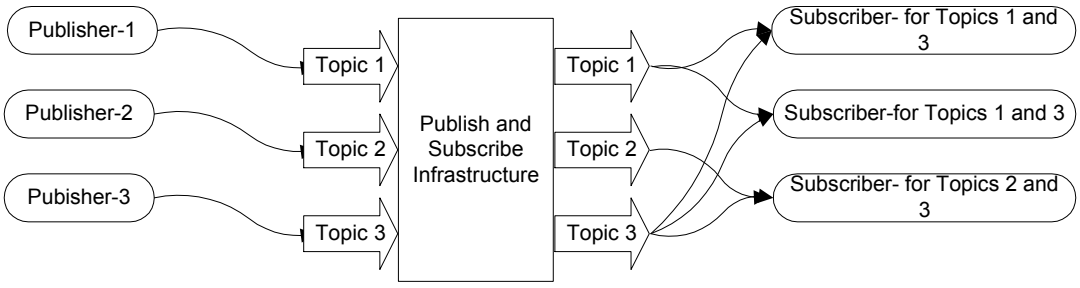


Figure 18: A system of publishers and subscribers of messages.

This figure shows three publishers, three subscribers, and three topics. Each of these publisher applications publishes messages against a topic. The arrows on the left show this process. On the right, there are three message subscribers interested in different kinds of topics. The first subscriber is interested in topics 1 and 3, while the third subscriber is interested in topics 2 and 3. These subscribers would receive only messages published against the respective topics.

Note: A publisher may publish messages against several topics and, similarly, a subscriber may subscribe for messages against several topics.

The publish and subscribe approach differs from message queuing approach in the following ways:

1. While message queuing is based on queues between two applications (sender and recipient), publish and subscribe is based on topics for messages.
2. The sender (publisher) publishes messages against topics and not into queues.
3. The recipient (subscriber) receives messages based on topics, irrespective of the publisher's identity.
4. The publisher and subscriber are coupled via messages and not queues.

In this approach, message publishers may use the following procedure to put messages into a queue:

1. Connect to the messaging infrastructure for a specific topic.
2. Create a message.
3. Publish a message.

Message subscribers may use the following steps to receive messages from the queue:

1. Connect to the message queuing infrastructure for a specific topic.
2. Poll for messages or register a handler.
3. Receive message for the specific topic.

Programming Model

Having seen the two possible messaging approaches, let us now consolidate the programming requirements, as follows:

1. An infrastructure for setting up queues and topics.
2. An API for connecting to the messaging infrastructure and putting or publishing messages.
3. Infrastructure capable of registering recipients/subscribers against queues/topics.
4. An API for getting messages based on queues or topics.

In addition to these basic requirements, the following features are also be required:

1. **Message Durability:** The infrastructure should allow for message durability (that is, once posted, a message should not be lost until it is delivered).
2. **Guaranteed Delivery:** The infrastructure should guarantee delivery.

3. **Once-Only Delivery:** The infrastructure should deliver the message once and only once. This is crucial as duplicate messages may adversely affect the business processes implemented by recipients.
4. **Message Confirmation:** The infrastructure should also allow confirmation notifications to be sent to the senders.

We will examine some of these features in more detail when we discuss Java Messaging Service (JMS).

Other Server-Side Requirements

In the previous section, we focused on programming aspects that depend how clients and servers communicate with each other.

In this section, we will discuss the two most important infrastructure-related issues applicable to server-side applications in an enterprise—*distributed transactions* and *security*.

Distributed Transaction Processing

Transaction processing is fundamental to any enterprise application that deals with data shared across various business processes.

If you are familiar with basic database programming, you might also be familiar with the basic idea of transaction processing. With JDBC[40], the `setAutoCommit(boolean)`, `commit()`, and `rollback()` methods on the `java.sql.Connection` interface let you commit or undo a group of database operations as a unit of work. Or, if you have developed client-server applications using, say, the rapid application-development programming environments from Oracle or Microsoft, you should have used the commit and rollback operations.

What purpose do these operations serve? As you may recall, these operations allow you to permanently record (with commit) or undo (with rollback) all database changes made with a connection. For example, consider that you are implementing a few database operations. Using the commit/rollback operations, you can either complete or abort all changes, as shown in **Figure 19**.

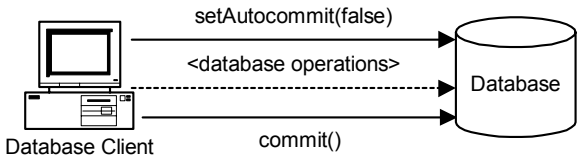


Figure 19: A JDBC transaction.

[40] White, S. et al. (1999). JDBC API tutorial and reference second edition. Addison-Wesley Publishing Company.

The figure shows how you can implement a transaction using the JDBC API. With the enclosing `setAutocommit(false) and commit()`, all of the database operations performed in between are treated as a single unit of work. Rather than invoke `commit()`, you may also invoke `rollback()` such that all operations performed prior to `rollback()` will be undone in the database. The `setAutocommit(false)` and `commit()` calls let you mark the boundaries of a transaction. Databases use the notion of a connection to implement such transactions.

One of the basic requirements for transaction processing is the preservation of ACID[41] properties of the data held in databases. The acronym *ACID* stands for *atomicity*, *consistency*, *isolation*, and *durability*. These properties imply the following:

- You should be able to group multiple database operations into atomic units of work.
- Database operations should be consistent and complete with respect to the business logic modifying the data.
- Database operations performed by multiple applications on the same data should not affect each other.
- Effects of database operations should be permanent.

While these properties of data can be maintained by today's relational database systems, databases alone cannot preserve the properties in the case of server-side applications. What is special about server-side distributed applications? The next section will introduce you to the issues associated with handling such transactions in distributed server-side environments.

In order to understand the need for a distributed transaction-processing infrastructure for distributed applications, consider the following figure:

[41] For more discussion on transaction process, see: *Nuts and Bolts of Transaction Processing* by Subrahmanyam Allamaraju at:

http://www.subrahmanyam.com/articles/transactions/NutsAndBoltsOfTP.html.

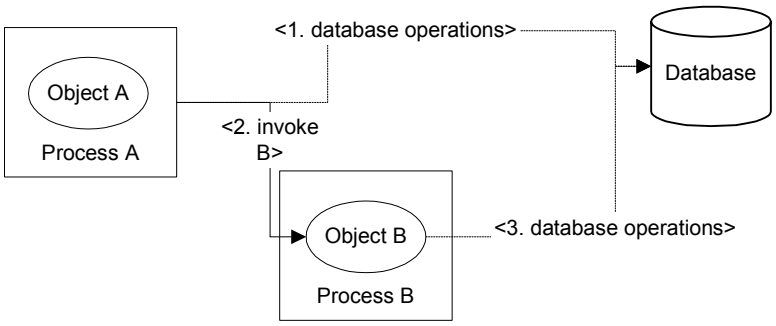


Figure 20: Distributed objects participating in database operations.

In **Figure 20**, there are two remote objects. For the sake of illustration, consider the following sequence of invocations:

1. Client (not shown) invokes a method on object A.
2. Object A performs a few database operations.
3. Object A invokes object B.
4. Object B invokes performs a few more database operations.
5. Object B returns.
6. Object A returns.

As far as the database server is concerned, there are two different connections and, hence, two database transactions. But, can we treat these two independent transactions as part of a single unit of work? This question might seem more involved when each of the objects is operating on a different database. But, what is required to maintain a single unit of work across these operations? How can we provide the semantics for commit and rollback in this case? The basic operations defined on a connection object are not sufficient for this because we are dealing with two database connections. In order to implement a unit of work (i.e., transaction) across the two connections, both sets of database operations must be either committed or rolled back together. However, the two sets of operations happen in the context of two different database connections, each of which is local to the object that obtained the connection. Coordinating the two sets of operations such that both commit or rollback together requires a third party and uses the following approach:

1. Client (not shown) invokes a method on object A.
2. Object A informs the third party that it will perform some database operations.
3. Object A performs a few database operations.
4. Object A invokes object B.
5. Object B performs a few more database operations.
6. Object B returns.
7. Object A informs the third party once it has completed its database operations.
8. The third party verifies that all of the database operations can be either committed or rolled back.
9. The third party instructs the database to either commit or rollback the respective database operations.
10. Object A returns.

The list of steps now seems longer, but before we look at what is happening, we will discuss the role of the third party.

- Because A and B are concerned only with their respective database operations and could be unaware of all database operations being performed by the other, we need a third party to monitor all operations across both objects.
- Because A and B can commit/rollback the respective sets of database operations, we need the third party to coordinate a single commit/rollback across both.

This third party is called, *transaction manager* or *transaction monitor*; and the process of coordinating commit/rollback across the above-listed objects is called, *two-phase commit*.[42] In this process, the transaction manager polls the database(s) to find out if the database operations can be committed. If the answer is yes, the transaction manager requests to commit the operations.

As you can see from the previous list of steps, object A must inform the transaction manager that it is about to conduct database transactions before starting the operations. Similarly, at the end of all operations, it must inform the transaction

[42] Bernstein, P. A., & Newcomer, E. (1997). Principles of transaction processing. Morgan Kaufmann Publishers.

manager that all operations are complete. This process is called, *transaction demarcation*. It is the process through which boundaries are created for transactional operations. All operations conducted within these boundaries are treated as a single unit of work.

Note: In addition to object A, from the previous example, the client can also demarcate the transaction.

The Open Group[43] (formerly the X/Open group) specified the first widely used transaction-processing model, which is known as the *distributed transaction processing model* (DTP).[44] This model defines a set of APIs, which are the basic building blocks for distributed transactional applications. Following are two of the APIs:

- **XA Interface:** This defines operations between databases and transaction managers such that databases can register themselves with the transaction manager and the transaction manager can interact with databases to coordinate transactions.
- **TX Interface:** This defines operations that the transaction manager should implement such that applications (objects A and B above) can demarcate transactions and control transactions when required.[45]

Most databases support the TX interface and the database part of the XA interface. Commercial transaction processing systems[46], such as Tuxedo (from BEA Systems, Inc) and Encina (now part of IBM's WebSphere suite of products), implement the TX interface for transaction managers.

Another transaction-processing model is Object Management Group's (OMG's) [47] Object Transaction Service (OTS).[48] This model is intended for distributed object

[43] See: http://www.OpenGroup.com.

[44] See: http://www.OpenGroup.org/publications/catalog/g504.htm for information on how to get documentation for the DTP reference model.

[45] See: *Nuts and Bolts of Transaction Processing* by Subrahmanyam Allamaraju at http://www.subrahmanyam.com/articles/transactions/NutsAndBoltsOfTP.html for an overview of various operations supported by these APIs.

[46] See: http://www.opengroup.org/business_transactions/relationship.htm for more information on available transaction processing products and vendors.

[47] See: http://www.omg.org.

applications that use CORBA, and it specifies certain interfaces that support distributed transactions across CORBA objects.

In the Java realm, the following are two APIs that deal with distributed transactions:

1. **Java Transaction Service (JTS)[49]:** This is the Java mapping of the OTS; it is suitable for implementing Java transaction managers.
2. **Java Transaction API (JTA)[50]:** This is a high-level transaction API for Java server-side applications; it is the model used for transaction processing in J2EE applications.

These technologies provide transaction management programmatically (that is, the clients/objects are responsible for transaction demarcation). The J2EE, as well as the Microsoft Transaction Server (MTS), provide an alternative mechanism called, *declarative transaction processing*. Using this process, you can enable transaction processing via certain configuration rather than implementing some transaction logic.

Security

In both business and technical circles, security is one of the most commonly misused and misunderstood concepts. Despite security concerns expressed by various parties who deal with enterprise application development and usage, security is the one of the least cared about requirements. One reason for this is because security lapses can occur at any of the following levels:

1. **Network Security[51]:** The lapses at this level include breach of the networks and servers. Solutions for such breaches involve firewalls, demilitarized zones, etc.
2. **Operating System Security[52]:** This is yet another level of security that deals with protecting various operating system-level resources (files, threads, processes, etc.).

---

[48] See: http://www.omg.org/technology/documents/formal/transaction_service.htm.

[49] See: http://java.sun.com/products/jts.

[50] See: http://java.sun.com/products/jta.

[51] Slawter, B. (1999). Network security: Developing and implementing effective enterprise strategies. Computer Technology Research Corporation.

Once an intruder breaks the network, the operating system is the next target for accessing operating system-level resources.

3. **Server-Side Application Security:** This level deals with preventing unknown or unauthorized users from accessing server-side applications.
4. **Database Security:** In addition to the previous levels, database vendors also provide their own security mechanisms to protect data from unauthorized access.

Because we are dealing with server-side applications in this piece, coverage of security will be limited to security mechanisms at the application level.

In enterprises, security concerns arise whenever applications provide resource access to several internal, as well as external, users and client applications.

As far as security is concerned, a resource could be business data (i.e., a purchase order or customer profile) maintained by various enterprise applications, or it could be the execution of, or participation in, a business process implemented by one or more enterprise applications. The following list shows some typical security concerns with enterprise applications:

- **Identity of Users—Authentication:** When your applications deal with privileged business processes and business sensitive data, it is important to establish the identity of a user/client before providing the user/client access to data or business processes. The process of *authentication* helps establish user/client identity. Authentication commonly involves the user/client exchanging a token (a password) with the application to which the user/client is trying to gain access. For this mechanism, the token should be known to both the user/client and the application. (More elaborate mechanisms include digital certificates, biometrics, etc.)
- **Allow/Bar Access—Authorization:** Authentication alone is not always enough, particularly when there are different types of users/clients with different privileges in a given environment. In a typical enterprise, you may encounter different users participating in different business processes under different identities and roles. Only those users/clients allowed to participate in certain business processes or access to some data should be allowed to access/do the task. All other users/clients without the required identity or role should not be provided access. The process of

[52] Pipkin, D. (1996). Halting the hacker: A practical guide to computer security. Prentice Hall.

authorization (sometimes called, *access control*) provides or denies access based on the identity/role of users. (Note the difference between authentication and authorization. While authentication deals with establishing identity, authorization deals with providing or denying access based on identity.)

- **Wire-Level Security—Data Confidentiality and Integrity:** With today's Internet-networked applications, data travels via several public networks, raising concerns over whether or not someone should be allowed to record/monitor or even modify the data while in transit. Depending on the nature of the data, such breaches may lead to consequences that are very harmful to businesses. In order to prevent this, it is necessary to guarantee data confidentiality and integrity. *Data confidentiality* prevents a person from beginning to decipher the data while in transit. *Data integrity* involves of the introduction checks that enable communicating parties (clients and servers) to detect any type of tampering with the data. Digital signatures, various encryption technologies[53] and protocols, such as Secure Sockets Layer (SSL), help in maintaining data confidentiality and integrity.

Following are key requirements for the server-side applications and programming models discussed in this article:

1. **Means to authenticate users and clients:** some means for authenticating users and clients and for maintaining the identity across all parts of an application. (This requirements holds for Web-based and other forms of clients).
2. **URL access control:** This is required when server-side applications can be reached via HTTP.
3. **Control of method invocation:** Control over the invocation of methods on remote objects.
4. **Control of message queue access:** Control over the access to message queues and topics.
5. **Ability to plug technologies at the wire level:** The ability to plug technologies at the wire level guarantees data confidentiality and integrity.

   We will see how these are achievable in J2EE applications in the next section.

[53] Schneier, B. (1995). Applied cryptography. Second Edition. John Wiley & Sons.

J2EE for Server-Side Programming

Up to this point in this article, we have discussed various principles behind server-side computing. During the discussion, though I have made references to some of the server-side Java technologies, I have intentionally left out details of how these principles map to various server-side technologies with Java. From this section forward, we will discuss specific Java server-side technologies within J2EE.

The J2EE is an integrated platform combining various server-side programming models previously discussed in this article. Sun Microsystems introduced J2EE in late 1999[54], though some J2EE APIs, such as servlets, Java Naming and Directory Interface (JNDI), etc., were introduced before; Sun Microsystems integrated all such technologies with a coherent and integrated architecture for server-side programming with J2EE. Today, J2EE is the de facto standard for building server-side and Internet applications using Java and is endorsed by several vendors offering J2EE-compatible application servers.

J2EE is essentially a specification that stipulates how the various J2EE technologies work together. Each of these technologies is discussed in various specifications released by Sun Microsystems. Sun maintains a reference implementation of J2EE. Nonetheless, J2EE is still a specification with several commercial and open-source implementations. Although such implementations are typically referred to as *J2EE application servers*, we will instead use the term *J2EE platform* to indicate an implementation of the J2EE technologies. J2EE includes the following technologies:

- **Enterprise JavaBeans (EJBs):** Used for developing distributed components (http://java.sun.com/products/ejb).
- **Java servlets and Java Server Pages (JSP Pages):** Used to build Web-based applications

  (http://java.sun.com/products/servlet, http://java.sun.com/products/jsp).

[54] As of writing this article, the public version of J2EE is 1.2, and the final draft of the next version 1.3 is under review. Apart from various enhancements to the existing APIs, the next release will support the following new technologies: (1) Java Connector, to provide a uniform framework for integrating to enterprise information systems such as SAP, or mainframes, and (2) Java Authentication and Authorization Service (JAAS) to enhance the basic Java security model for server-side applications. See http://java.sun.com/j2ee for the latest information.

- **Java Naming and Directory Interface (JNDI):** Used for naming and directory services (http://java.sun.com/products/jndi).
- **Java Message Service (JMS):** Used for asynchronous messaging (http://java.sun.com/products/jms).
- **Java Transaction API (JTA):** Used for distributed transactions (http://java.sun.com/products/jta).
- **JDBC:** Used for enterprise-level database access (http://java.sun.com/products/jdbc).
- **Java API for XML Parsing (JAXP):** Used for parsing XML documents (http://java.sun.com/xml).
- **Java Mail:** Used for sending and receiving electronic mail (http://java.sun.com/products/javamail).

In the previous list, I included some Web site references where you will find the specifications, APIs, associated products, white papers, etc.

Note: Some of these technologies may be used individually, not necessarily as a part of a J2EE environment (i.e., you may use implementations of JAXP, JMS, Java Mail, and JNDI outside a J2EE environment). However, most of the implementations of these technologies are bundled with other J2EE technologies and using them generally requires a J2EE platform.

There are several ways to learn various J2EE technologies. You can start with individual technologies, later gaining an overall view of J2EE that will enable you to architect server-side applications using J2EE; however, this article attempts to map various programming needs to respective J2EE technologies such that, depending on specific requirements, you may study each of these technologies in more detail.

Using this approach, we will not discuss all of the technologies in detail in this article. Instead, we will focus mainly on J2EE programming models (with EJBs, Java servlets, and JSP pages), JMS, transactions, and security.

We have discussed the following programming requirements thus far:

- Programming models and the infrastructure that supports for the following:

- Synchronous request-response type client-server communication.
- Synchronous stateless request-response type client-server communication.
- Asynchronous communication.

- The need for an infrastructure that supports distributed transactions and security, as shown in **Figure 21**.

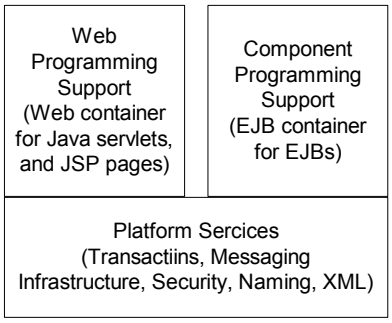


Figure 21: High-level view of J2EE.

Let us now discuss how J2EE provides for each of these needs.

Before we go into the details on how these programming models are supported, we will look at some of the most commonly used J2EE terminology, as follows:

1. **Container:** We use the term *container* to denote a runtime that supports application programs. There are two types of containers for developing server-side applications using J2EE.

    - **EJB Containers:** EJB containers are capable of hosting server-side objects that are developed for the synchronous request-response model. In the upcoming enhancement via J2EE1.3 and EJB2.0, EJB containers are also capable of hosting server-side objects that can be activated asynchronously via messages.
    - **Web Containers:** Web containers are capable of hosting objects developed for the stateless request-response model with communication over HTTP. The Web container maintains instances of Java servlets and JSP pages.

2. **Application Components:** The J2EE specification uses the term *application components* to denote container-managed objects, such as EJBs, Java servlets, and JSP pages. These are managed, as the container (i.e., the runtime) maintains the lifecycle of these instances. (Developers do not explicitly create these instances.) The containers also manage such things as transactions and security as we will see later in this article.
3. **Modules:** Apart from the notion of application components, J2EE employs the notion of modules. A *module* consists of one or more of a given type application components. For instance, an EJB module consists of one or more EJB components and, similarly, a Web module consists of one or more servlets and JSP pages (along with static content, such as HTML files, images, etc.). The primary purpose of a module is to allow flexible packaging structure. In J2EE, modules are packaged into Web archive (WAR) or Java archive (JAR) files.
4. **J2EE Applications:** A J2EE application is a combination of or more modules. J2EE applications can be packaged into enterprise archive (EAR) files. Apart from serving as a means for high-level application module packaging, a J2EE application can also define virtual boundaries between various applications deployed on a given container.
5. **Deployment:** This is another word you will encounter frequently in the J2EE world. While a strict definition of deployment is outside the scope of this article, in simple terms, *deployment* is the process of adding your applications to a J2EE platform. This could mean simply copying the classes (suitably packaged as applications) into a directory accessible by a J2EE platform implementation, or it could mean using vendor-specific deployment tools to add applications to a J2EE platform. Once deployed, J2EE application components are available for invocation by J2EE containers based on clients' requests (i.e., requests from Web browsers or other clients) or even other J2EE application components.
6. **Deployment Descriptor:** In simple terms, a *deployment descriptor* is a configuration file expressed using XML. Along with application components, each module consists of a deployment descriptor. Depending on the type of application component, the deployment descriptor consists of specific configuration information. Deployment descriptors help containers to not only decipher the contents of modules, but also to let containers manage the lifecycle of various application components, transactions, and even security.

Distributed Components for Synchronous Programming—EJBs

In the *Synchronous Request-Response Model* section, we discussed the intricacies of invoking methods on remote objects synchronously. Specifically, we identified the following programming needs:

- Remote interfaces, which specify the contract between server-side objects and clients.
- Proxies and skeletons to handle the network-level programming.
- A server process to create and maintain server-side remote objects.
- Naming services to publish proxy objects (this is an essential piece that allows location transparency).
- Clients to look up proxy objects using names.
- Clients to invoke methods on remote objects.

The EJB technology coupled with the JNDI meets these programming needs.

An *EJB* is essentially an instance of a remote object maintained by what is known as an *EJB container*. In simple terms, a container is nothing more than a runtime (or process) that creates and manages all lifecycle aspects of instances of EJB instances. Earlier in this article, we presented a process for creating an instance of an RMI remote object. The container is similar to a process for creating (on-demand) instances of remote objects and maintaining them based on the type of bean and certain configuration details.

In any given remote object, the logic implemented includes some business logic (validation, computation, executing some rules, etc.) and persistence logic[55] (database queries, database updates, etc.). Unlike other distributed component technologies, such as CORBA or RMI, the EJB technology attempts to isolate both sets of programming tasks by defining two basic types of EJBs—*session beans* and *entity beans*.

[55] In general, any logic dealing with data stored in non-volatile storage, such as file system, databases, etc. is persistence logic.

1. **Session Beans:** General purpose EJBs, which are similar to remote objects in CORBA or RMI. Any kind of logic can be implemented in session beans.
2. **Entity Beans:** Special purpose beans that encapsulate persistence logic, such as creating/updating/deleting/finding data from databases.

Because these two types of beans have different purposes, there are separate interfaces defined for them. Although there is nothing to prevent you from implementing non-persistence logic in entity beans, or from implementing persistence logic in session beans, such a separation of concerns allows for better layering of applications.

The upcoming EJB 2.0 specification introduces another kind of bean, called *message driven bean*. We will discuss these types of beans in some detail later in this article.

Developing an EJB

Before we discuss the various types of beans, let us consider the steps involved in creating an EJB. They are as follows:

1. **Remote Interface:** Similar to the remote objects discussed previously, EJBs are primarily described by their remote interfaces. A remote interface includes all of the methods that a remote object provides for its clients.
2. **Home Interface:** Each EJB has a home interface that lets EJB clients participate in certain instance lifecycle operations, and each has an instance of the home interface that provides instances of remote interfaces to clients.
3. **Bean Class:** This class provides implementations for methods specified in both the remote and home interfaces. At runtime, instances of this class are the *remote objects*.

Following are examples of each type of bean:

1. **Remote Interface:** You can specify a remote interface by extending the `javax.ejb.EJBObject` interface (which, in turn, extends the java.rmi.Remote interface). This is similar to the remote interface of Java RMI.

While the `java.rmi.Remote` interface is a marker interface with no methods specified, the `javax.ejb.EJBObject` interface specifies certain methods applicable for different types of beans.

Following is a sample remote interface of an EJB:

```
public interface OrderManager extends javax.ejb.EJBObject {
    public void createOrder(...)
                    throws java.rmi.RemoteExcetion;
            public void amendOrder(...)
                                                throws
java.rmi.RemoteException;
  }
```

The above remote interface specifies certain methods for clients to invoke.

2. **Home Interface:** You can specify the home interface by extending the `javax.ejb.EJBHome` (which also extends the `java.rmi.Remote` interface). Because the home interface extends the `java.rmi.Remote` interface, clients can invoke methods on this interface remotely. The `javax.ejb.EJBHome` interface specifies certain common methods.

   The following is a sample home interface for the above EJB:

```
public interface OrderManagerHome extends javax.ejb.EJBHome {
     public Process create() throws java.rmi.RemoteException;
}
```

3. **Bean Class:** After specifying the remote and home interfaces, the next step is to provide an implementation for methods specified in those interfaces. Depending on the type of the bean, the bean class implements either the `javax.ejb.SessionBean` or `javax.ejb.EntityBean` interface.

```
public class OrderManagerBean extends javax.ejb.SessionBean
{
   public Process ejbCreate() throws java.rmi.RemoteExcetion
{
          // Implementation
   }
   public void ejbRemove() {
          // Any cleanup
   }
   public void createOrder(...)
                  throws java.rmi.RemoteExcetion {
              // Implementation
          }
          public void amendOrder(...)
                                          throws
java.rmi.RemoteException {
              // Implementation
          }
    public void ejbActivate() {
          // After activation
   }
   public void ejbPassivate() {
          // Before activation
    }
    public void setSessionContext(
             javax.ejb.SessionContext ctx) {
              // Implementation
          }
}
```

4. **Sample Client:** The previous three steps complete the programmatic aspects of creating an EJB. Now we will create a sample client that invokes methods on the EJB.

```
public class OrderClient {
   public static void main(String[] arg) {
      // 1. Lookup for a proxy object for the home
      Context context = new InitialContext();
      Object obj = context.lookup("orderManager");
      // 2. Cast the home object to OrderManagerHome
      OrderManagerHome omHome =
                javax.rmi.PortableRemoteObject.narrow(obj);
       // 3. Create a remote proxy
       OrderManager om = omHome.create();
       // 4. Invoke methods on the remote proxy
       om.createOrder(...);
       //
   }
}
```

5. **Deployment Descriptor:** As discussed previously, the purpose of a deployment descriptor is to provide deployment-time configuration information to the container. Following is a sample deployment descriptor for the previous bean:

```
<ejb-jar>
   <enterprise-beans>
          <session>
              <ejb-name>Order_Manager</ejb-name>
              <home>OrderManagerHome</home>
              <remote>OrderManager</remote>
              <ejb-class>OrderManagerBean</ejb-class>
              <session-type>Stateless</session-type>
              <transaction-type>Container</transaction-type>
          </session>
    </enterprise-beans>
    <assembly-descriptor>
          <container-transaction>
```

```
                    <method>
                        <ejb-name>Order_Manager</ejb-name>
                        <method-intf>Remote</method-intf>
                        <method-name>*</method-name>
                    </method>
                    <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

For EJBs, deployment descriptors are created as ejb-jar.xml files. These are XML files with a document type definition (DTD) specified in the EJB specification. For this discussion, consider the element `<session> … </session>` in this descriptor. This element describes the session bean with a name (any name), as well as the names of the home interface, remote interface, and bean class. This element also specifies that is a stateless bean, and that the container should manage the transactions for the bean. We will discuss transactions shortly.

We specified one method in the home interface and two methods in the remote interface; the bean class has several other methods. Further, the client is dealing with two proxy objects instead of one. Following is a discussion of details:

- **ejbCreate and ejbRemove:** These methods resemble constructors and destructors in object oriented programming languages. You can compare these methods to the constructor and finalize methods in Java. While we explicitly specified a `create()` method in the home interface, the `javax.ejb.EJBObject` interface has a `remove()` method. The `ejbCreate()` and `ejbRemove()` methods correspond to these `create()` and `remove()` methods. But what role do these methods play?

  As discussed previously, the EJB container creates and maintains EJB instances (that is, instances of `ProcessBean` in this case). Because the client is remote, it cannot create/delete objects explicitly in a remote process. Such requests have to be remote.

- In order to create an EJB instance, the client calls the `create()` method specified in the home interface. When the container receives this call, it creates an instance

of the bean (that is, `ProcessBean` class) and calls its `ejbCreate()` method. Thus, this method allows the bean instance to do any initialization. Note that because the `create()` method is specified by the bean developer, the bean developer may specify arguments for this method in the home interface. Accordingly, the `ejbCreate()` method will need the same set of arguments, and the container passes these arguments directly to the bean instance while calling `ejbCreate()` method.

- The same logic applies to the `remove()` method except that there are no arguments for this method. Remote invocation of the `remove()` method on the remote interface translates to the `ejbRemove()` method on the bean instance.

What are the roles and purposes of these two methods? While the container can automatically create instances based on requests from remote clients, home's purpose is to allow the client to participate in the lifecycle of remote objects. For the bean, this allows it to perform client-specific initializations. The home is comparable to a factory[56] that can produce remote objects.

- **Naming Convention:** Note that the names of the methods implemented above are `ejbCreate()` and `ejbRemove()` instead of `create()` and `remove()`. This is a convention adopted in the EJB technology, and it implies that the methods starting with *ejb* are not business methods, and are meant for invocation by the EJB container—not clients. The same convention applies to the other methods starting with *ejb* in the previous example. These methods are called *container callbacks,* as EJB containers invoke these methods during the lifecycle of beans.
- **Proxies and Stubs:** As shown in the client class, for each EJB, the client must deal with two proxies. The first proxy is for the home interface, and the second is for the remote interface. The client first obtains an object (a proxy) that confirms to the home interface, and uses one of its create methods to obtain an object (another proxy) that confirms to the remote interface. After these two steps, the client uses the EJB as though it were local.

With EJBs, the container creates these two proxy objects during the deployment process. The same is the case with the stubs on the server side. The

[56] Gamma, E. et al. (1995). Design patterns. Addison-Wesley Publishing Company.

exact structure of these proxies and stubs is vendor specific. Ultimately, these container-generated classes map the client calls to appropriate methods on the bean instance.

- **Exceptions:** Note that the methods in the remote interface throw `java.rmi.RemoteException`. Because these methods execute on the server side (in the EJB container), all exceptions are remote. In addition to this exception, the methods on the remote interface may also throw application-specific exceptions, such as `OrderCreationException`. Such exceptions are called *application exceptions*, and the container faithfully reproduces these exceptions on the client side.
- **Activation and Passivation:** Once the remote proxy is obtained, the client can continue to invoke the methods on the remote interface. But, what if the client is holding a reference, but not using it frequently? In such cases, the container may reuse the instance for another proxy. In this manner, the container may maintain a pool of instances and allocate them to different client requests. When an instance is allocated, the remote proxy will be associated with an instance from the pool. For beans that manage state (such as stateful beans, to be discussed shortly), the container needs to store the state of the instance before deallocating an instance, and retrieve it during allocation. The process of deallocation and the storage of the state (in some storage area) is called *passivation*. The reverse process is called *activation*. Before activation or passivation, the container gives the instance a chance to perform any special tasks, such as allocating or deallocating any resources, by calling `ejbActivate()` and `ejbPassivate()` methods.
- **Context:** For both session and beans, the container maintains an instance of a context (`javax.ejb.SessionContext` or `javax.ejb.EntityContext`) that encapsulates the context in which the instance is being used. For instance, with session beans, the context lets the bean get an instance of the remote proxy for the session bean. The bean may pass this instance wherever needed to pass this reference.
- **Threading:** With EJBs, the bean developer is concerned about managing any threading; in fact, the bean developer is not allowed to use threads within the bean implementation. For a given client request, the container allocates a thread and invokes the corresponding method on the bean instance. The purpose of this

constraint is to simplify the bean-development task without concern about multiple threading and synchronization issues.

Session Beans

As discussed previously, the purpose of a session bean is to encapsulate any server-side logic (the example just discussed illustrates this). As we saw in the example, clients use the home proxy to obtain a remote proxy, and then invoke methods specified in the remote interface. Note that a session bean is client specific—not shared across multiple clients. This prevents the programmer from having to deal with any multiple-threading issues.

Depending on whether or not a bean instance manages internal state, a session bean can be one of the following two types:

1. **Stateful Session Beans:** Beans that can manage state across multiple client invocations. The `OrderManagerBean` class can maintain private variables holding some data. When such a bean is specified to be stateful (in the deployment descriptor), the container guarantees that the client is always associated (via the proxy) with the same bean instance. That is, a stateful session bean can maintain a *conversational state* between different method invocations. When a bean is stateful, and when the container decides to recycle an instance, the container performs bean passivation and activation.
2. **Stateless Session Beans:** The most commonly used beans. With stateless session beans, the container does not guarantee that a client will be associated with the same bean instance and, therefore, it guarantees that any state maintained by a bean instance will be valid from the client's view.

As seen in the deployment descriptor of the previous example, you can indicate if a bean is stateless in the deployment descriptor.

In general, stateless beans are suitable when you expect a large number of clients, such as in the Internet scenario. When a bean is stateful, for each active client, the container will be required to maintain a separate instance, thus leading to increased memory usage on the container process. In addition, the frequent passivation and activation may affect overall performance. Stateful session beans may be used when the number of expected clients is minimal.

Entity Beans

The notion of entity beans originates from the idea of using domain objects to encapsulate data in persistent storage, such as a relational database. Consider the following cases:

- **Runtime Representation of Persistent Data:** If you have a relational database holding customer profile records, a typical object-oriented application represents each record as an object of, say, `CustomerProfile` class.
- **Persistence of Domain Objects:** Alternatively, based on a domain analysis, you may identity certain domain objects that need to be persisted. Both cases involve runtime representation of persistent data as objects.

Apart from representing persistent data, these objects simplify runtime manipulation of data. A given record in the persistence store corresponds to an instance of an object in memory.

Such objects should typically provide the following semantics:

- **Accessors and mutators[57] to manipulate the state of objects:** For instance, the `CustomerProfile` class may provide methods such as `setName()` and `getName()` to manipulate the name of the customer.
- **Methods to deal with persistence, which include the following:**
    - Given some identity of data existing in a database (the primary key), create an object that represents the data associated with the identity).
    - Given some identity, create a record in the database such that the record corresponds to the state of the object in memory.
    - Update the state of object in memory and transfer the changes to the corresponding database records.
    - Delete the records in the database with the help of the object.

[57] An *accessor* is a method to access information, and a *mutator* is a method to modify information. For instance, the method setName("foo") changes the name of the customer to "foo," and therefore is a mutator. On the other hand, you can use the getName() to get the current name, and therefore this method is an accessor.

- Given some criteria, search for corresponding database records and create objects to hold the search results.

- Identity to maintain the notion of the identity (primary key) for each object.

The goal of such an abstraction is to isolate database-level implementation details, away from the rest of the application relying on data. Rather than dealing with database-level operations directly, applications can deal with higher-level objects that encapsulate the data.

Entity EJBs employ the above-listed approach. Entity beans provide a remote object representation of persistent data. Unlike session beans, entity beans deal with persistence logic. The following example illustrates entity beans.

Consider a `CustomerProfile` entity bean that encapsulates the name and telephone number of a customer. Assume that, within the database, these attributes are stored in each row of a table, and the name is the primary key.

1. **Remote Interface:** Following is a sample remote interface of the `CustomerProfile` bean:

```
public interface CustomerProfile extends javax.ejb.EJBObject {
   public void setName(String name)
                    throws java.rmi.RemoteExcetion;
   public String getName()
                    throws java.rmi.RemoteException;
   public void setTelNumber(String number)
                    throws java.rmi.RemoteExcetion;
   public String getTelNumber()
                    throws java.rmi.RemoteException;
}
```

These methods let the client access/manipulate the name and telephone number of a given customer.

2. **Home Interface:** Following is a sample home interface for the above-listed EJB:

```
public interface CustomerProfileHome extends javax.ejb.EJBHome
{
    public CustomerProfile create(String name,
              String telNumber) throws java.rmi.RemoteExcetion,
                                       javax.ejb.CreateException;
    public CustomerProfile findByPrimaryKey(String name)
                              throws javax.ejb.FinderException,
                                        java.rmi.RemoteException;
}
```

These methods let the client create new `CustomerProfile` objects.

3. **Bean Class:** To implement an entity bean, the bean class should implement the `javax.ejb.EntityBean` interface.

```
public class CustomerProfileBean extends javax.ejb.EntityBean {
   String name;
   String telNumber;
   public void setName(String name) {
        this.name = name;
   }
   public String getName() {
        return name;
   }
   public void setTelNumber(String number) {
        this.telNumber = telNumber;
   }
   public String getTelNumber() {
        return telNumber;
   }
   public String ejbCreate(String name, String telNumber)
throws java.rmi.RemoteExcetion {
        // Insert a row in the database, and return the name
```

```
    }
    public void ejbPostCreate(String name, String telNumber) {
          //
    }
    public void ejbRemove() {
          // Delete the row from the database
    }
    public void ejbLoad() {
         // Load the row from the database
    }
    public void ejbStore() {
         // Store/update the row in the database
    }
    public String ejbFindByPrimaryKey(String name)
             throws ObjectNotFoundException
    }
             // Select the row from the database and
             // return the primary key
    }
    public void ejbActivate() {
             // After activation
    }
    public void ejbPassivate() {
             // Before activation
    }
    public void setEntityContext(javax.ejb.EntityContext ctx)
    {
             // Implementation
    }
    public void unsetEntityContext() {
             // Implementation
    }
    public void ejbActivate() {
```

```
            // After activation
    }
    public void ejbPassivate() {
            // Before passivation
    }
}
```

4. **Sample Client:** The following client illustrates the client view of the above entity bean.

```
public class CustomerProfileClient {
public static void main(String[] arg) {
    // 1. Lookup for a proxy object for the home
    Context context = new InitialContext();
    Object obj = context.lookup("customerProfile");
    // 2. Cast the home object to CustomerProfileHome
    CustomerProfileHome cpHome =
                javax.rmi.PortableRemoteObject.narrow(obj) ;
    // 3. Create a remote proxy
    CustomerProfile cp = cpHome.create("Joe Smith");
    // 4. Invoke methods on the remote proxy
    cp.setTelNumber("555-5555");
    ...
    ...
    // 5. Search CustomerProfile bean for "Joe Smith".
    cp = cpHome.findByPrimaryKey("Joe Smith");
    ...
  }
}
```

Note: An entity bean implementation looks significantly different than that of a session bean.

- On the remote interface are methods for accessing/changing the name and telNumber attributes of the bean. For entity beans, these are the proper business methods.
- The home interface has a `create()` method, which can create a new customer profile given the name and telephone number. (Note that this method returns a `CustomerProfile`, the remote interface, instance.) The other method is `findByPrimaryKey`, which returns an instance of `CustomerProfile` given the name. In our example, the name of the customer is the primary key. (Note that these two methods are equivalent to SQL CREATE and SELECT statements for relational databases.)
- On the bean class, the `create()` and `findByPrimaryKey()` methods correspond to the `ejbCreate()` and `ejbFindByPrimaryKey()` methods, respectively. Apart from the addition of *ejb* to these methods (and case conversion), these methods return strings. In our example, the type of the primary key is a string. When the client calls `create()` or the `findByPrimaryKey()` method on the home interface, the container invokes either the `ejbCreate()` or `ejbFindByPrimaryKey()` method, and then creates a proxy to correspond with the returned primary key (that is, the container creates a client view of the primary key and returns it).
- The `findByPrimaryKey` method is called a *finder*. The purpose of a finder is to provide search semantics on the home interface. In relational database terms, finders abstract SELECT statements with the finder arguments corresponding to WHERE clauses. You can add several other finders depending on possible retrieval criteria. You may, for instance, add a `findAll()` method to retrieve all customer profiles in the home interface:

```
public Collection findAll()
                            throws javax.ejb.FinderException,
                                   java.rmi.RemoteException;
```

Since this method may return more than one `CustomerProfile` instance, the return type is a `java.util.Collection`. You may also use `java.util.Iterator` instead of a collection.

On the bean class, the findAll() corresponds to ejbFindAll().

```
public Collection ejbFindAll()
     throws ObjectNotFoundException
{
     // Select all rows from the database and
     // return a collection of primary keys
}
```

This method returns a collection of primary keys. The container converts this collection into a collection of proxies.

- Also, note the exceptions thrown by the `ejbFindByPrimaryKey` method. If no data is found in the database for the given primary key, this method throws an `javax.ejb.ObjectNotFoundException` indicating that no rows exist in the database for the given primary key.

Note the following differences between entity beans and session beans:

- The above example illustrates that an entity bean corresponds to some data persistent in the database. The lifecycle of an entity bean thus corresponds to the data (a row in the above example) present in the database. This is one of the main differences between session beans and entity beans. The lifecycle of an entity bean extends that of a session bean. The lifetime of a session bean cannot extend beyond a container shutdown. On the other hand, an entity bean lives as long as the data in the database.
- The second difference has to do with identity. Session bean identity is specific to the client that is using it. Moreover, this identity does not extend beyond a container shutdown. On the other hand, an entity bean has an identity spanning across its lifetime.

In the previous example, the bean developer was responsible for coding the persistence (using JDBC and SQL) logic. This approach is, therefore, called *bean managed persistence (BMP).* The EJB technology also has a more sophisticated approach whereby the container manages the persistence logic. This approach is called *container managed persistence (CMP).* With CMP, the bean developer does not implement such persistence logic; instead, the bean developer maps the attributes of

the bean to the tables/fields in the underlying relational database. How this mapping is done depends on the container implementation, and is vendor-specific. Some vendors provide tools, which you can use to do this mapping. Other tools rely on vendor-specific deployment descriptors to describe such mappings. In either case, the CMP lets you manage persistence at deployment time as opposed to development time.

The CMP eliminates the need for custom JDBC/SQL implementation, and is better suited for simple mappings between entity beans and relational databases. In addition, CMP is appropriate when the bean developer cannot predetermine the structure of the database during the development time.

Naming and Object Location—JNDI

While discussing Java RMI, we also discussed the role of naming services in distributed computing. The JNDI[58] provides this service for J2EE applications.

As discussed previously, a naming service provides for the storage of, and access to, named objects. In general, naming services exist with directory services. A directory service is a naming service, which includes metadata that describes the object referenced by a name. Using this metadata, you can search the directory service to find an object without knowing its name.

The architecture of the JNDI is such that it allows pluggable implementations for naming and directory services. While it is possible to provide a completely non-persistent implementation of JNDI, most implementations of the JNDI are persistent (that is, the names exist in a persistent storage and can be accessed via the JNDI API). A JNDI implementation may use a variety of service providers[59] for the actual implementation of naming directory services—ranging from relational databases to Light-Weight Directory Access Protocol (LDAP)[60] servers, to files in a file system.

In the context of EJB, the JNDI provides the naming service, irrespective of its underlying implementation of JNDI; however, unlike RMI, in the case of EJBs, the client starts by looking up for a proxy for the home interface. The container is responsible for binding a proxy for each home interface deployed. The client looks up the home proxy and calls a create method to get a proxy for the home interface. From the client's perspective, the following is the only step needed to interact with the JNDI:

```
try {
    // Get an InitialContext
    Properties h = new Properties();
    h.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
          "weblogic.jndi.WLInitialContextFactory");
```

[58] Lee, R., & Seligman, S. (2000). JNDI API tutorial and reference: Building directory-enabled java applications. Addison-Wesley Publishing Company.

[59] See: http://java.sun.com/products/jndi/serviceproviders.html for a list of JNDI service providers.

[60] Howes, T. et al. (1999). Understanding and deploying LDAP directory services. MacMillan Technical Publishing.

```
        h.put(javax.naming.Context.PROVIDER_URL,
              "t3://localhost:8001");
        javax.naming.Context context =
                                  new javax.naming.InitialContext(h);
        // Now lookup
        Object obj  = context.lookup("OrderManager");
        ...
    } catch (NamingException ne) {
        // Handle the exception.
    }
```

In a naming service, names exist in a context. A *context* provides a hierarchical structure to the set of names. In order to look up a name in JNDI, the first step is to get a hold of the context. The first few lines in the previous code sample obtain this context; however, in order to obtain a context, you need to specify an implementation for creating a context and the location of the JNDI service provider. The previous example is specific to the WebLogic[61] server as indicated in the context factory name (`javax.naming.Context.INITIAL_CONTEXT_FACTORY`) and the location (`javax.naming.Context.INITIAL_CONTEXT_FACTORY`). Other providers will have similar implementations and the procedure for obtaining the context remains the same.

Following are other usages for JNDI in J2EE applications:

1. **Transactions:** The `javax.user.UserTransaction` object for managing transactions (to be discussed shortly).
2. **Messaging:** Queues and topics setup for asynchronous messaging based on JMS.
3. **JDBC:** Datasource[62] objects for obtaining database connections.
4. **Environment Variables:** You can also publish environment variables in the JNDI for various EJBs (i.e., you can specify certain configuration parameters as environment variables in the deployment descriptor). The container publishes

[61] See: http://www.bea.com/products/weblogic/server for documentation. Also see: *Professional J2EE Programming with BEA WebLogic Server* by Paco Gomez and Peter Zadrozny (2000, Wrox Press).

[62] See: Chapter 3 of *Professional Java Server Programming, J2EE Edition* by Subrahmanyam Allamaraju (2000, Wrox Press).

these variables in the JNDI (under the context "java:/comp/env"), and the bean classes can look up such information.

5. **Other System-Level Objects:** Apart from the home proxy objects, you can bind any other object to JNDI and look up other EJBs or clients.

For the first four cases, the container will be responsible for creating and binding the objects.

Java Servlets and JSP Pages for Synchronous Connectionless Programming

In the *Synchronous Stateless Request-Response Model* section, we discussed a programming model for client-server communication via HTTP. In particular, we identified the following technical requirements for such communication between clients, such as Web browsers, and servers hosting Web applications:

1. A programming framework for including custom programming logic during the HTTP request-response process; Java servlets provide for this requirement.
2. A facility to generate dynamic content with minimal programming requirements. JSP pages allow you to construct templates and generate content dynamically based on HTTP requests.
3. The ability to manage sessions and state. The Java servlet API has provisions for managing state using session tracking based on cookies and URL encoding.

In the following section, we will discuss Java servlets and JSP pages in some detail.

Java Servlets

Java servlets were introduced by Sun Microsystems in early 1997. The primary goal of the initial servlet model was to provide a Java language alternative to the CGI. Accordingly, the initial servlet model was designed to serve dynamic content for incoming HTTP requests. Later versions of the Java servlet API (currently 2.2, with the 2.3 version under final review) introduced session management facilities. Today, servlets are an integral of the J2EE architecture. The Java servlet API is specified in `javax.servlet`, and `javax.servlet.http` packages.

A Java servlet is a class implementing the `javax.servlet.Servlet` interface or the `javax.servlet.http.HttpServlet` class. The `javax.servlet.http.HttpServlet` is an abstract class that provides common abstractions necessary for all servlets communicating with clients via HTTP. The other core classes of Java servlets are `javax.servlet.http.HttpServletRequest`, `javax.servlet.http.HttpServletResponse`,

`javax.servlet.http.HttpSession`, `javax.servlet.ServletContext` and `javax.servlet.ServletConfig`. Of these, the `HttpServlet` is the class that your application servlets extend from, while the rest are interfaces. J2EE platform vendors provide implementations for all these classes/interfaces.

Let us illustrate servlet programming with the following example:

```
public class HelloServlet extends
javax.servlet.http.HttpServlet {
    public void init() {
    // Any initialization of the servlet instance
}
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
    // Read parameters from the request
    String name = request.getParameter("name");
    // Send HTML response.
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    out.println("<html><body>");
    out.println("<p>Hello " + name + ".</p>");
    out.println("</body></html>");
    out.close();
}
public void destroy() {
    // Any cleanup code
 }
}
```

The purpose of this simple servlet is to read an HTTP request parameter (called a *name*) and send an HTML response to the requesting client. For example, the HTTP request could be http://www.xyz.com/sample/hello?name=foo. In this case, the servlet sends a response "Hello foo." formatted as HTML. Now we will discuss the details of this servlet.

Similar to EJBs, servlets are also container managed. The Web container creates and manages instances of servlets. In the previous example, the Web container creates a single instance of the servlet and directs all requests to this instance in separate threads. Upon creating an instance, the Web container lets the servlet developer perform any initialization by calling the `init()` method on the servlet. Similarly, before withdrawing a servlet out of service, the container invokes the `destroy()` method to provide for any cleanup.

Now we will discuss the anatomy of the previous example. The `javax.servlet.http.HttpServlet` class implements the `javax.servlet.Servlet` interface that specifies a `service()` method with `javax.servlet.Request` and `javax.servlet.Response` as arguments. These two arguments encapsulate the request and response streams associated with a client request. The `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` interfaces extend the `javax.servlet.Request` and `javax.servlet.Response` interfaces, respectively. These derived interfaces abstract HTTP requests and responses. They also provide access to the underlying input and output streams associated with the HTTP connection from the client to the server. When the Web container receives an HTTP request (apart from creating/locating a servlet instance), the container also creates objects that implement the `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` interfaces.

The `javax.servlet.http.HttpServlet` class has methods responsible for handling GET, POST, DELETE, OPTIONS, PUT, and TRACE requests specified in HTTP/1.1. These methods have default implementations, and you can override them to suit your requirements. In the previous example, we chose to override the implementation for the GET method.

After writing a servlet, the developer is also responsible for providing a deployment descriptor with it. The deployment model for servlets, as well as JSP pages, is a Web application. A *Web application* is a collection of servlets, JSP pages, and static resources (such as HTML files, images, etc.), as well as a deployment descriptor (specified in a web.xml file).

Following is a sample deployment descriptor for the previous servlet:

```
<web-app>
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>HelloServlet</servlet-name>
        </servlet>
</web-app>
```

This deployment descriptor indicates that the Web application includes a servlet with name *hello* implemented by `HelloServlet` class.

When you deploy a Web application into a Web container, you also specify a URI for the application. This URI helps the Web container identify the Web application for a given HTTP request. If we specify the URI as */sample* for the previous Web application, an HTTP request to http://www.xyz.com/sample/hello?name=foo will be mapped to this Web application. The container then uses the servlet name *hello* to identify a servlet to handle the request. In this manner, the Web container can map HTTP requests to servlet instances.

This simple model, therefore, allows for programmatic response to HTTP requests. Another important feature of the servlet API is its session management capabilities. The `javax.servlet.http.HttpSession` interface is an abstraction of an HTTP session. As discussed previously, cookies and URL coding are the two most commonly used means for tracking sessions. Irrespective of which technology is used for tracking sessions, the servlet API (in association with the Web container) tracks sessions transparently. In addition, the `javax.servlet.http.HttpSession` interface provides the following capabilities:

- Creation of a new session.
- Invalidation of the existing session.
- Storage of name/value pairs of data (for instance, you can store an object with a name during execution of one servlet, and retrieve it during execution of another servlets).

Note: In the servlet API, HTTP sessions are associated with HTTP requests; therefore, you can use the `javax.servlet.http.HttpServletRequest` object to gain access to the HTTP session.

The servlet API provides the following approaches for managing state:

1. **Session:** As discussed previously, you can associate name/value pairs of data with HTTP sessions. The state is specific to each session. Because a session is specific to a given client, you can use HTTP session to store client-specific state (for example, you can store the name of the user in the HTTP session).
2. **Servlet Context:** The servlet API also has another means of storing state across all servlets and sessions in a given application. For each Web application, the Web container maintains a `javax.servlet.ServletContext` object in which you can store name/value pairs of data. The state can be shared across all servlets and HTTP sessions. This context is typically used to store state common across the application.
3. **Request:** In addition to the previous two approaches, you can also associate name/value pairs of data with HTTP requests. The scope of such state is equal to a single HTTP request. This method may be used in combination with request forwarding.[63]

JSP Pages

While servlets provide you with a means of programmatically responding to HTTP requests, servlets are not suitable for responses that involve visual rendering. In the majority of cases, HTTP clients are browsers and require responses that contain formatted content. In such cases, Java servlets make it hard to generate content and even harder to maintain the content. Any changes to the content require reprogramming all the servlets.

JSP pages are suitable for sending formatted content in response to HTTP requests. Unlike Java servlets, JSP pages are based on templates.

The following is an example of a JSP page that produces the same response as is produced by the servlet in the previous example:

[63] See: Chapter 9 of *Professional Java Server Programming, J2EE Edition* by Subrahmanyam Allamaraju et al. (2000, Wrox Press).

```
<html>
   <body>
         <p>Hello <%=request.getParameter("name")%>.</p>
   </body>
<html>
```

When compared to the servlet, the JSP looks more like an HTML page, except for the script enclosed within <% and %>. Developing and maintaining such pages is much simpler than trying to use servlets to generate the same content.

A JSP page contains the following two parts:

1. **HTML or XML:** Used for the static content.
2. **JSP tags and scriptlets written in the Java programming language:** Used to encapsulate the logic that generates the dynamic content.

In the previous example, the content enclosed within <% and %> is the dynamic part that prints the parameter *name* contained in the HTTP request. This part of the JSP is called a *scriptlet*.

Using a scriptlet, you can embed Java statements within JSP pages. From within these scriptlets, you can access certain standard servlet variables, such as *request* (corresponding to the `javax.servlet.http.HttpServletRequest`), *response* (corresponding to the `javax.servlet.http.HttpServletResponse`), *session* (corresponding to the `javax.servlet.http.HttpSession`), *context* (corresponding to the `javax.servlet.ServletContext`), etc. Using these variables, you can embed any servlet-like code within Java servlets.

While JSP pages allow dynamic content based on templates, in reality, JSP technology is an extension of the Java servlet technology, and both servlets and JSP pages confirm to the same lifecycle model within a Web container. Before serving a JSP page, the Web container converts (compiles) the JSP page into an equivalent servlet class and invokes the servlet to generate the response. Therefore, you may consider that a JSP provides a much quicker route for developing servlets. Ultimately, what serves the content is the dynamically generated class—not the JSP page. In order to optimize this process, the container generates the servlet class either at startup or when the JSP page is invoked for the first time. For subsequent requests, the container

uses the pre-generated servlet class. Nonetheless, when compared to servlets, which are pure Java programs, JSP pages are text-based documents.

In addition to scriptlets, JSP technology also provides for custom tags that look like standard HTML/XML tags. JSP tags bypass the need for embedding Java statements within JSP pages, as shown in the following sample JSP, which uses a custom tag to display the name:

```
<html>
   <body>
         <p>Hello <name/>.</p>
   </body>
<html>
```

This JSP is much closer to standard HTML markup, except the markup uses a new tag called *name*. However, it is the programmer's responsibility to provide an implementation for this tag using the JSP tag extension mechanism.[64,65]

Java Servlets or JSP Pages?

Since the introduction of JSP pages in 1999, there have been discussions about whether to use servlets or JSP pages. Some JSP proponents consider servlets legacy and cumbersome.

As seen in the previous subsection, JSP pages simplify the task of generating dynamic content because they follow a page approach as opposed to a program approach. This is a well-suited approach when the target client is a browser or a similar content rendering application. Servlets are completely inappropriate for this purpose; however, there is a case in which servlets are more suitable for responding to HTTP requests.

In the *Synchronous Stateless Request-Response Model* section, we discussed two uses for the HTTP request-response model. The first one was to generate dynamic content; the second was to allow programmatic communication between clients and

[64] See: Chapters 12 and 13 of *Professional Java Server Programming, J2EE Edition* by Subrahmanyam Allamaraju (2000, Wrox Press).

[65] Avedal, K. et al. (2000). Professional JSP. Wrox Press.

servers using HTTP as the protocol. For example, rather than invoking an EJB from a client, you can send an HTTP request to a servlet, and the servlet can in turn invoke the EJB. This approach is predominant in emerging technologies, such as SOAP[66] and Web services based on universal description, discovery, and integration (UDDI)[67].

[66] SOAP is joint initiative from Microsoft and IBM, and enables remote object invocation over HTTP. See http://www.w3.org/TR/SOAP/ for the SOAP specification. See http://xml.apache.org/soap for a Java reference implementation of SOAP.

[67] http://www.uddi.org/

Asynchronous Programming—JMS

The JMS is a J2EE technology that provides for asynchronous messaging between application components. Before the introduction of JMS, there was no single model for messaging. Vendors had proprietary APIs and programming models for asynchronous messaging. With the introduction of JMS as one of the J2EE technologies, this situation has changed. JMS provides a common API for asynchronous messaging, which abstracts most of the commonly used programming models and techniques. As a result, today, you can program to JMS without depending on a specific implementation (JMS provider) or any proprietary API.

In the *Asynchronous Communication Model* section, we discussed two approaches for asynchronous messaging: point-to-point and publish and subscribe messaging. The JMS abstracts both of these models and provides similar APIs for them.

Basic Programming Model

The JMS programming model is very simple and intuitive; the steps are as follows:

1. **Open a Connection:** Any application component that deals with messaging must first open a connection to the underlying messaging system with respect to a queue (for point-to-point messaging) or a topic (for publish and subscribe messaging). The application may have to supply certain credentials such that the underlying messaging system can authenticate the user. The application uses a connection factory implementation provided by the JMS provider.
2. **Create a Session:** The application then creates a session, during which the application may send/receive messages.
3. **Send a Message:** Following are steps to send a message:

    a. The application creates a message using the session as a factory for messages.

    b. The application then creates a sender/publisher using the session as a factory for senders/publishers.

    c. Using the sender/publisher, the application sends the message.

4. **Receive a Message:** Following are steps to receive a message:

   a. The application creates receiver/subscriber using the session as a factory for receivers/subscribers.

   b. Using the receiver/subscriber, the application receives the message.

As you from these steps, JMS programming is simple, and the APIs follow the same pattern for most of the common tasks.

Point-to-Point Messaging

Let us consider an application to send a message to a queue. The following sample illustrates the steps involved:

```
// Import JMS package
import javax.jms.*;

// Get a JNDI context
javax.naming.Context context = ...;
// Get a QueueConnectionFactory implementation
QueueConnectionFactory factory =  (QueueConnectionFactory)
                        context.lookup("connectionFactory");
// Create a connection
QueueConnection connection =
                        factory.createQueueConnection();
// Create a session
QueueSession session = connection.createQueueSession(false,
                        Session.AUTO_ACKNOWLEDGE);
connection.start();
// Lookup for a queue in JNDI
Queue queue = (Queue) context.lookup("myQueue");
// Create a QueueSender
QueueSender sender = session.createSender(queue);
TextMessage message = session.createMessage();
```

```
// Set the message
message.setText("Hello");
sender.send(message);
// Close
sender.close();
session.close();
connection.close();
```

In this sample, both the queue connection factory and message queue are set up and bound to the JNDI by the underlying J2EE platform administrator. The exact procedures for setting these up is vendor dependent, and you should consult vendor supplied documentation to find the exact procedure for setting up the connection factory and the queue.

The steps for sending a message to a queue follow the model discussed in the previous subsection; however, please note the following points:

- While creating a session, the sender application can indicate if the session is bound by a transaction. The first argument in the call to `createQueueSession` indicates that the session should be non-transactional. When the session is transactional, all of the messages being sent in the session are treated as a single unit of work (for example, after sending a few messages, the application can rollback the session such that none of the messages are actually sent). In order for the JMS provider to finally send the messages, the application should call `commit()` on the session.
- While creating a session, the sender application can also indicate an acknowledgement mode for each message. In the previous example, for the recipient, we used an auto-acknowledgement mode in which the JMS infrastructure was notified once the receiver got the message.

The procedure for receiving messages from a queue depends on whether the receiver receives messages synchronously or asynchronously. In the synchronous mode, the receiver waits until a message is in the queue. In the asynchronous mode, the receiver registers a listener interface that will be invoked when a message is in the queue.

The following sample illustrates the message receipt in synchronous mode:

```
QueueReceiver receiver = session.createReceiver(queue);
TextMessage message = receiver.receive();
```

The second statement is a blocking call and waits until there is a message. Alternatively, the receiver may specify a timeout interval to abort if no message arrives during the specified interval of time.

The receiver can instead register a message listener for receiving messages, as follows:

```
QueueReceiver receiver = session.createReceiver(queue);
receiver.setMessageListener(myListener);
```

In this code, the receiver registers a listener that implements the `javax.jms.MessageListener` interface. This interface has an `onMessage()` method that gets invoked when a message is in the queue. The JMS provider also supplies the message while invoking this method.

Publish and Subscribe Messaging

The programming model for publish and subscribe messaging is similar to that of point-to-point messaging. The main difference is in the class/method names. Rather than using queue senders, the publishing application uses topic publishers. The same applies to message receipt. Rather than using queue receivers, the subscribing application uses topic subscribers. The following sample publishes a message:

```
// Get a TopicConnectionFactory implementation
TopicConnectionFactory factory = (TopicConnectionFactory)
                        context.lookup("connectionFactory");
// Create a connection
TopicConnection connection =
                         factory.createTopicConnection();
// Create a session
TopicSession session = connection.createTopicSession(false,
                                  Session.AUTO_ACKNOWLEDGE);
```

```
connection.start();
// Lookup for a topic in JNDI
Topic topic = (Topic) context.lookup("myTopic");
// Create a TopicPublisher
TopicPublisher publisher = session.createPublisher(queue);
TextMessage message = session.createMessage();
// Set the message
message.setText("Hello");
publisher.send(message);
// Close
sender.close();
session.close();
connection.close();
```

The code for the subscriber is also similar. The following sample illustrates the message subscription in synchronous mode:

```
TopicSubscriber subscriber = session.createSubscriber(topic);
TextMessage message = subscriber.receive();
```

The subscriber can instead register a message listener for receiving messages, as follows:

```
TopicSubscriber subscriber = session.createSubscriber(topic);
subscriber.setMessageListener(myListener);
```

Point-to-Point or Publish and Subscribe?

Having seen both models, how do you choose between these two models for asynchronous communication?

Consider the following questions in making your choice:

- Is your message exchange specific to senders and recipients? Or, is it based on type of message, irrespective of who is sending and who is receiving?

- Are there too many applications exchanging messages? Or, are there too many message types when compared to the number of senders/recipients?

In some cases, the message exchange can be specific to senders and recipients (i.e., an *order-capture* application and a *fulfillment* application may be exchanging several messages of various types). This situation qualifies for point-to-point messaging. On the other hand, if there are several applications generating certain business events, such *as purchase order arrived, requisition matched,* etc.), you should consider the publish and subscribe model.

Similarly, if too many applications are exchanging messages, point-to-point may involve too many couplings between applications. In such a case, publish and subscribe messaging may limit the coupling to the number of message types involved. On the other hand, when you are dealing with a large number of message types across a small number of applications, point-to-point messaging is simpler to implement.

Other JMS Features

Following are some of the features provided by JMS[68]:

- **Persistent/Non-Persistent Delivery:** You can set up a queue/topic in persistent or non-persistent mode. When the delivery mode is set to persistent, the JMS infrastructure persists each message until it is finally received (and acknowledged) by the receiver/subscriber. In the non-persistent mode, the JMS infrastructure does not persist messages. This implies that if the sender/subscriber is not available to receive the message, the message may be lost (i.e., such messages may not survive a server restart). Depending on the type of the data contained in messages, you should choose one of these modes. Because persistent mode involves durable storage, it is less efficient than the non-persistent mode.

  In addition to these modes, the sender/publisher may also indicate a time for the expiration of each message. The JMS provider discards a message when it cannot deliver it within this interval.

[68] Monson-Haefel, R., et al. (2000). Java message service. O'Reilly & Associates.

- **Message Priority:** When a sender/publisher creates a message, it can also indicate a priority for delivery. By default, all messages are delivered in the order they are received; however, this behavior can be changed by setting different priority levels on each message.
- **Durability:** With publish and subscribe messaging, subscribers can receive JMS messages in durable mode. Durability allows the subscriber to receive all pending messages posted prior to connecting to the JMS provider. (By default, topic subscribers are not durable.)
- **Types of Messages:** JMS provides for five different types of messages. They are as follows: byte messages, map (name-value pairs) messages, text messages, stream messages (stream of Java primitives), and object messages (containing serializable objects).

The upcoming EJB2.0 specification includes a new type of EJB that can be a message receiver/listener.

Note: Session/entity beans cannot receive messages synchronously/asynchronously because they are meant for synchronous invocation from clients; receiving messages requires waiting for messages or registering a listener for messages. Session/entity beans are not capable of this.

In order to address this problem, the EJB2.0 specification introduces a new type of beans called, *message-driven EJBs*. Similar to other types of EJBs, these beans are also container managed; however, clients cannot directly invoke these beans. Instead, the container invokes these beans when there are messages to be processed.

J2EE Transactions

The J2EE environment is a distributed component environment. The J2EE application components (servlets, JSP pages, and EJBs) may be distributed across multiple virtual machines, which points to the need for distributed transactions across the database logic performed by such components.

Within a J2EE environment, you may encounter the following situations in implementing transactions:

- A JSP page or a servlet provides some database access using one or more connections. In this case, the JSP page or servlet should demarcate the transactional database access.
- A JSP page or a servlet provides some database access, and then invokes methods on one or more session or entity EJBs. In this case, the JSP page or servlet should also demarcate the transactional database access such that all database access (including database logic performed by the EJBs) is considered a single unit of work.
- A JSP page or a servlet invokes one or more session or entity beans, each of which is possibly invoking other session or entity EJBs. In this case, all database logic performed by all of the beans should be considered a single unit of work.
- A Java client invokes one or more session or entity EJBs. In this case, all database logic performed by all the beans should be considered a single unit of work.

In a J2EE environment, the Java transaction API provides a `javax.transaction.UserTransaction` interface provides methods for you to explicitly begin and end transactions. In cases 1 or 2 above, the servlet or JSP could begin and end a transaction as follows:

```
// Import javax.transaction.UserTransaction
import javax.transaction.UserTransaction;
...
...
// Get a JNDI context
javax.naming.Context context = ...;
// Get the UserTransaction object
UserTransaction utx = (UserTransaction)
        context.lookup("javax.transaction.UserTransaction");
ctx.begin(); // Begin the transaction
// Perform database access
...
// Invoke EJBs that also perform database access
...
ctx.commit(); // End the transaction
```

Using this user transaction, the developer can explicitly control the transaction demarcation. This approach is called, *programmatic demarcation*. In a J2EE environment, the underlying transaction infrastructure implements two-phase commit (if required) to commit the transaction.

Despite the flexibility of programmatic transactions, in certain cases—particularly with EJBs, this approach requires explicit programmer intervention in beginning and ending transactions. When you are dealing with EJBs, the J2EE also offers a different approach called *declarative* or *container managed* transactions.

With declarative transactions, the bean developer specifies certain transactional attributes in the bean deployment descriptor. For example, the `OrderManager` bean discussed previously has the following segment in the deployment descriptor:

```
<ejb-jar>
    <enterprise-beans>
           <session>
                  ...
                  <transaction-type>Container</transaction-type>
           </session>
     </enterprise-beans>
    <assembly-descriptor>
           <container-transaction>
                  <method>
                       <ejb-name>Order_Manager</ejb-name>
                       <method-intf>Remote</method-intf>
                       <method-name>*</method-name>
                  </method>
                  <trans-attribute>Required</trans-attribute>
           </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

This descriptor indicates the container is responsible for managing transactions for the `OrderManager` bean. Neither the bean developer nor the EJB client developer needs to explicitly begin a transaction before using this bean.

The second of this descriptor specifies that all methods on the remote interface of this bean require a transaction. This is indicated by the `<trans-attribute>` set to "Required." With this description, the container automatically starts a transaction before invoking any method on the remote interface of this bean. Following are the other possible values for this attribute (note that these values apply to one or more methods combined in a `<container-transaction>` element):

- **NotSupported:** This value indicates to the container that the specified method should not be invoked within a transaction. If a transaction is already open, the container commits the transaction before invoking this method.
- **Required:** The container always invokes the bean within a transaction. If a transaction is already open, the container invokes this method within the same

transaction; otherwise, the container starts a new transaction before invoking this method.

- **Supports:** The bean is transaction-ready. If the client invokes the bean within a transaction, the bean is executed within the same transaction; if not, the bean is executed without a transaction.
- **RequiresNew:** This value indicates that the method requires a new transaction irrespective of whether or not there is already a transaction.
- **Mandatory:** This value indicates that the container must invoke the bean within a transaction (that is, the caller of this method is responsible for beginning a transaction). For example, if the caller is a JSP page or a servlet, it should programmatically start the transaction. If the caller is a bean, the bean should have started the transaction programmatically or declaratively.

Container-managed transactions, therefore, declare transactions in deployment descriptors rather than programmatically beginning and ending transactions at compile time. Declarative transactions are generally preferable, as they are more flexible to manage; however, if transactional database access is implemented outside EJBs in Java servlets or JSP pages, you will need to program transactions explicitly.

Also note that `javax.transaction.UserTransaction` cannot demarcate transactions outside the J2EE environment. If, for instance, you have a Java client invoking methods on EJBs in a J2EE platform, the client cannot start/end transactions using the `javax.transaction.UserTransaction`. Instead, you will be required to use the JTS (Java transaction services)[69] API to do so.

[69] Vogel, A., & Rangarao, M. (1999). Programmig with enterprise JavaBeans, JTS, and OTS: Building distributed transactions with java and C++. John Wiley & Sons.

J2EE Security

The minimal security requirements for J2EE applications include the following:

- **Authentication of Users:** Authenticate and maintain identity of users within the J2EE platform such that all application components can rely on the authenticity and identity of users.
- **Authorization:** Allow/deny access to invocation of J2EE application components based on users roles.

Next we will discuss how J2EE handles security.

Security for Web Applications

Unlike Web sites that serve static pages, Web applications provide access to secure information, as well as for executing business processes. In a typical on-line bank, an Internet user can not only view all of the details of his or her accounts, but he or she can also execute business processes, such as setting up on-line bill payment, transferring funds or even opening new accounts.

One of the traditional ways to secure such a site is to let the user authenticate himself or herself. In most cases, this process requires that you provide a login page wherein the user enters certain credentials, such as a login name and a password. The Web application validates this data against stored credentials, and when the credentials match, allows the user to start using the functionality. This approach is called *programmatic security*. The application developer a priori determines what Web pages require secure access and, based on this data, implements programming logic that verifies whether or not the user has been authenticated.

Note: that in addition to authenticating a user, the application developer must also authorize users. As discussed previously, authorization is the process of verifying whether or not the current user is authorized to access a certain resource (i.e., in the case of an on-line bank, a question of authorization might be whether or not the user is allowed to set up on-line bill payment).

It is not hard to implement both authentication and authorization programmatically on a simple application that does not have many different user types or business processes. With this approach, the developer must deal with the following:

1. Perform authentication and record credentials in the HTTP session.
2. Check to see if the user has the right credentials to access the page (for each page).

Most of today's Web sites follow this approach; however, it is prone to certain limitations that can lead to security violations. Consider the following questions:

1. What happens if the developer forgets to make checks for authentication? (This could result in users executing functionality that they are not supposed to execute).
2. What happens if there are a number of user kinds/roles? (In this case, the developer must programmatically check for all roles in each page. This is a tedious task, particularly when Web sites do not remain static but grow with more functionality and pages).

In order to simplify these tasks, J2EE offers an approach called, *declarative security* for J2EE Web applications. The following shows how this approach functions:

1. The developer classifies users into various roles. A role is the role played by a user in the context of the application (i.e., some such roles might consist of customers, premium customers, or guests). Note that a given person may possess many roles (i.e., an account manager and customer relationship manager).
2. The developer develops all of the pages in the Web application without any consideration as to authentication or authorization.
3. At deployment time, the developer examines the pages and groups the pages (based on URI patterns) for access based on user roles, and then defines resource collections. All pages starting with /customer/regular/, for example, could be meant for access by customers, while /customer/premium/ could be meant for access by premium customers. In this case, there are two resource collections. Depending on the structure and the number of roles, the developer may define several such resource collections.

4. The developer then specifies security constraints of the deployment descriptor for the application. The constraints would indicate the container of roles required to access a specific collection of resources.
5. The developer implements certain vendor-specific mechanisms to create user accounts and to set up various roles.
6. The developer then sets up a login mechanism for the Web application. J2EE allows three forms of login mechanisms—HTTP basic authentication, SSL mutual authentication, and form-based authentication. Of these, the form-based authentication[70] is most commonly used. In this model, you can specify a login page to collect user credentials.

This procedure is completely declarative and keeps the application security unaware. Most of this setup can be done at the deployment time. Moreover, when the security considerations change, only the deployment descriptor needs to be changed.

This approach shifts authentication/authorization from the Web application to the Web container. The following show how the container performs authentication/authorization declaratively:

- Whenever the user requests a page, the Web container examines whether or not the page belongs to any resource collection specified in the deployment descriptor. If it doesn't, the container serves the page as usual.
- If the requested page (URL) belongs to one of the resource collections, the Web container checks to see if the current HTTP request is associated with a role required by the resource collection. In J2EE, a Web user has no default role; so, authorization fails the first time a user requests a page requiring a specific role.
- The Web container then invokes a login mechanism specified in the deployment descriptor. With form-based authentication, the container sends the login page to the client (browser). The user fills in a login and password and submits the page. The container then verifies the credentials, performs authentication, and identifies the role of the user.

[70] See: Chapter 10 of *Professional Java Server Programming, J2EE Edition* by Subrahmanyam Allamaraju (2000, Wrox Press) for a detailed example.

- If any user role matches the required role, the Web container sends the originally requested page. If not, the container sends an error page (to be set up along with the login page).

As you can see from this description, the container performs authentication on demand (that is, there is no need to authenticate the user at the entry point to a Web application). Whenever the required roles are insufficient, the container performs authentication.

Apart from simplifying the authentication/authorization tasks, this approach relives the developer of having to examine each page in the application for possible security violations. This, of course, does not relieve the developer of having to examine the constraints set up in the deployment descriptor.

In addition to authentication/authorization, Web containers can also verify whether or not a wire-level security constraint is satisfied. In the deployment descriptor of a Web application, you can indicate whether the wire-level communication (from client to Web server) should be confidential or integral (to indicate whether the communication should guarantee confidentiality or integrity). If one of these constraints is set up for a resource collection in the deployment descriptor, the container verifies whether or not incoming requests satisfy it (otherwise, containers refuse to serve requested page). However, because browsers always initiate the communication, the container cannot enforce a specific type of communication, such as SSL.

Note: Currently, J2EE does not address how users or roles can be managed in a J2EE environment. Different vendors provide different approaches/APIs for managing users and roles.

Security for EJBs

The J2EE security model for EJBs is limited to authorization only. This model protects methods on both home and remote interfaces from being invoked without a proper role associated with the caller. Similar to configuring transactions declaratively on methods of beans, you can also specify security requirements declaratively in the bean deployment descriptor. The following is an example:

```
<method-permission>
    <role-name>customer</role-name>
    <method>
        <ejb-name>OrderManager</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
```

This deployment descriptor[71] indicates that all methods on the OrderManager bean require a role customer for the caller. If the caller does not possess such a role, the EJB container throws security exceptions.

But, how does the container know the identity? This depends on the caller. With Web applications containing JSP pages and servlets, the Web application can authenticate the user and establish the identity of the caller using the procedure described in the previous subsection. Once the Web container identifies a user, his or her identity is preserved until the end of the HTTP session. The forthcoming J2EE specification includes support for Java Authentication and Authorization Services (JAAS) to allow authentication for non-Web clients. Using this service, non-Web Java applications, such as Swing applications, can establish identity, which in turn are propagated to the EJB container when the client invokes methods on EJBs.

[71] See Chapter 21 of *Professional Java Server Programming, J2EE Edition* by Subrahmanyam Allamaraju (2000, Wrox Press) for a detailed example.

Conclusion

In this article, my main objective has been to establish the principles behind server-side programming and to map these principles and programming needs to various J2EE technologies. Wherever possible, I have tried to achieve this from first principles.

This article, however, is by no means complete. There are many details and topics you ought to consider before developing or managing server-side applications using J2EE. There are several places in this article in which you will find references to books, papers, and on-line resources. Please refer to these resources for more details.

On choosing J2EE to develop enterprise applications, developers face some of the following questions:

1. Which technology should I use for this problem? Such questions should not be treated simply and require a thorough understanding of the current, as well as future, application requirements. While JSP pages, servlets, and EJBs are the most commonly used J2EE technologies, in some cases, technologies, such as JMS, let you build a better, loosely coupled architecture.
2. Should I use EJBs for this problem? The answer depends on what you are considering EJBs for. As discussed in this article, EJBs are remote objects and are meant for remote clients. Such calls typically involve[72] marshalling, unmarshalling, and transmission across the network. Given this consideration, it is important to treat EJBs as coarse-grained, and to limit the number of calls from clients to EJBs. If your application logic is suitable for coarse-granularity, you should consider EJBs. Also, EJBs have the advantage of declarative transactions and security.
3. Which vendor should I choose? Today, there are several implementations—both free/open-source and commercial. Some of the popular free/open-source implementations include Orion Application Server[73] and JBoss Application

[72] Note that when the caller and the EJB are in the same virtual machine (this is called *collocation*), the EJB container implementation may treat the call as a local call. For instance, if you invoke an EJB from another EJB running in the same EJB container, the container need not treat this call as a remote call. However, the bean developer cannot make assumptions about this, as how beans are deployed and where they are located is a deployment decision.

[73] http://www.OrionServer.com

Server.[74] Both products are popular among developers. Of these, Orion Application Server has free non-commercial purposes, while JBoss Application Server is being developed as an open-source implementation. Currently, there are several commercial implementations, of which you can obtain evaluation copies from vendor Web sites. Some of the popular products include WebLogic Server[75] from BEA Systems Inc, WebSphere[76] from IBM, iPlanet Application Server[77] from iPlanet, Borland App Server from Inprise, etc. Each of these products has their merits and demerits. For your specific requirements, you should study various features of the products, try to work with their evaluation copies, and choose the one that comes closest to your expectations. A comparison[78] of these products is outside the scope of this article.

As mentioned previously in this article, the next version is J2EE (version 1.3) is due to be released this year. This version has enhancements in the following areas:

- **Servlets 2.3:** The new version introduces certain advanced features, such as request filtering. Request filters allow you to register filter programs to be invoked before processing HTTP requests, thus giving you a chance to do any preprocessing.
- **EJB 2.0:** The upcoming EBJ specification enhances the container-managed persistence model.
- **Java Connector API:** This major enhancement provides a common infrastructure for integrating J2EE applications with enterprise information systems, such as ERP systems, mainframes, etc. In addition, this API attempts to integrate all data access (including JDBC) into a common framework.
- **Security:** J2EE 1.3 also includes support for the JAAS.

---

[74] http://www.JBoss.org

[75] http://www.bea.com/products/weblogic/server/index.shtml

[76] http://www.ibm.com/websphere

[77] http://www.iplanet.com/products/iplanet_application/home_2_1_1n.html

[78] See: http://www.techmetrix.com for a high-level comparison. Another source of such information is http://www.TheServerSide.com, where you will find reviews of various products from developers. However, be vary of the fact that these products have frequent releases, and the features sometimes improve rapidly.

- **XML:** While J2EE 1.2's JAXP considers only the pluggability of document parsing, J2EE 1.3 includes support for pluggability of XML document transformation.

Note: We have not addressed any of these technologies in this article. As mentioned previously, these technologies are too advanced for this introductory article.

After discussing a number of topics leading to J2EE, is J2EE the killer technology? Is it the silver bullet for all of the perils of building and managing server-side applications? These same questions can, in fact, be asked of other competing technologies, such as Microsoft's .NET suite.

Irrespective of what proponents of these technologies might say to these questions, note that there are no silver bullets for the complexity of server-side applications. These technologies simplify most of the infrastructure-related issues but do not offer solutions—and rightly so. Due to the consolidation of infrastructure needs, these technologies reduce the amount of time you spend in building enterprise applications; however, as a developer, you are still responsible for architecting and building flexible and manageable applications. These technologies provide the tools; you are responsible for learning to use the tools and benefiting from them.

About the Author

Subrahmanyam Allamaraju is a senior engineer with BEA Systems Inc. and has been working with Java and distributed technologies for over four years. He is the author of technical papers published in the *Java Developers' Journal*, *XML Journal*, and TheServerSide.com. He has coauthored two books: *Professional Java Server Programming, J2EE Edition* (2000) and *Professional Java E-Commerce* (2001), both published by Wrox Press. His interests include J2EE, metadata, and XML-driven architectures for enterprise applications, business objects, and new paradigms for software architecture and programming. You can get updates on his current interests at his Web site: http://www.Subrahmanyam.com, where he also maintains a collection of technical papers.

Acknowledgment

The author would like to thank John Mueller for his patient review and elaborate comments.

Copyright, Trademarks and Liability