The Complete Reference

J2EE

Chapter 2

J2EE Multi-Tier Architecture

The expectation for instant gratification was ratcheted up a notch with the growth of the Internet and the maturity of corporate infrastructure. Executives and customers alike demand instant access to information any time, any place—24 hours a day, 7 days a week. Whether it's accessing the corporation online catalog, placing an online order, retrieving account information, or sending and receiving email, they want an immediate response.

Information technology departments of corporations had to devise a scheme to revamp their networks and systems to accommodate thousands of people who wanted to simultaneously access corporate resources. To meet these expectations, technologists rethought the way in which information is stored, accessed by, and delivered to clients.

Focus was directed at the technology architecture model used to provide services to desktop and remote computers. Many IT departments used a two-tier, client/server architecture model where desktop software called *clients* request information over the corporate network infrastructure to *servers* running software that fulfilled a client's request.

However, this two-tier architecture depends heavily on keeping client software updated, which is both difficult to maintain and costly to deploy in a large corporation that has several intranets and a workforce that consists of field representatives and other remote users. Web-based, multi-tier systems don't require client software to be upgraded whenever presentation and functionality of an application are changed.

The infrastructure had to be revamped. The two-tier, client/server architecture had to be abandoned and a new, multi-tier architecture had to be built in its place. The Java development team at Sun Microsystems, Inc. with the collaboration of the Java Community Program (JCP) developed the Java programming language to be used to build software for this new multi-tier architecture.

In this chapter you'll learn about multi-tier architecture and the role each Java 2 Enterprise Edition component plays in the redevelopment of corporate America's infrastructure—and how Java 2 Enterprise Edition is becoming a key component in web services technology.

Distributive Systems

The concept of multi-tier architecture has evolved over decades, following a similar evolutionary course as programming languages. The key objective of multi-tier architecture is to share resources amongst clients, which is the fundamental design philosophy used to develop programs.

As you learned in the previous chapter, programmers originally used assembly language to create programs. These programs employed the concept of software services that were shared with the program running on the machine.

Software services consist of subroutines written in assembly language that communicate with each other using machine registers, which are memory spaces within the CPU of a machine. Whenever a programmer required functionality provided by a software service, the programmer called the appropriate assembly language subroutine from within the program. Although the technique of using software services made creating programs efficient by reusing code, there was a drawback. Assembly language subroutines were machine specific and couldn't be easily replicated on different machines. This meant that subroutines had to be rewritten for each machine.

The introduction of FORTRAN and COBOL brought the next evolution of programming languages, and with it the next evolution of software services. Programs written in FORTRAN could share functionality by using functions instead of assembly language subroutines. The same was true of programs written in COBOL. A function is conceptually similar to a Java method, which is a group of statements that perform a specific functionality. The group is named, and is callable from within a program.

Although both assembly language subroutines and functions are executed in a single memory space, functions had a critical advantage over assembly language subroutines. A function could run on different machines by recompiling the function.

No longer were software services exclusive to a particular machine. However, software services were restricted to a machine. This meant programs and functions that comprise software services had to reside on the same machine. A program couldn't call a software service that was contained on a different machine.

Programs and software services were saddled with the same limitations that affected data exchange at that time. Magnetic tapes were used to transfer data, programs, and software services to another machine. There wasn't a real-time transmission system.

Real-Time Transmission

Real-time transmission came about with the introduction of the UNIX operating system. The UNIX operating system contains support for Transmission Control Protocol/Internet Protocol (TCP/IP), which is a standard that specifies how to create, translate, and control transmissions between machines over a computer network.

It was also around the same time when technologists developed the Remote Procedure Call (RPC). RPC defined a way to share functions written in any procedural language such as FORTRAN, COBOL, and the C programming language. This meant that software services were no longer limited to a machine. Furthermore, a programmer could now call a function that was created by a different program using a different procedural language that resided on an entirely different machine as long as that machine was connected to the same network.

Another important development in the evolution of distributive systems came with the development of eXternal Data Representation (XDR). While RPC enabled programmers to call preprogrammed functions that were available on the network using TCP/IP, there remained a need to exchange complex data structures between programs and functions—and between functions.

The solution came with the introduction of XDR. XDR specified how complex data structures could be exchanged among programs and software services. This became the linchpin that changed the way programmers and system designers conceived applications. Instead of limiting an application to one machine, an application became a collaborative development effort that utilized software services that were available throughout the network.

Software Objects

The next evolutionary step in programming language gave birth to object-oriented languages such as C++ and Java. Procedural languages focused on functionality, where a program was organized into functions that contained statements and data that were necessary to execute a task. Functions were either internal software services within a program or external software services called by RPC.

Programs written in an object-oriented language were organized into software objects—not by functionality. A software object resembles a real-world object in that a software object encapsulates data and functionality in the same way data and functionality are associated with a real-world object. A software object is a software service that can be used by a program.

Although objects and programs could use RPC for communication, RPC was designed around software services being functionally centric and not software-object-centric. This meant it was unnatural for programs to call software objects using RPC. A new protocol was needed that could naturally call software objects.

Simultaneously two protocols were developed to access software objects. These were Common Object Request Broker Architecture (CORBA) and Distributed Common Object Model (DCOM). CORBA was developed by a consortium that included Sun Microsystems, Inc., IBM, and Oracle among others. Microsoft developed DCOM.

As you probably suspect, CORBA and DCOM were incompatible. This resulted in confusion in the marketplace, which some technologists believe caused the lack of widespread adoption of either protocol. Companies that embraced distributive object technology had to adopt either CORBA or DCOM. Otherwise, companies had to use a protocol converter as a gateway between environments that used CORBA and DCOM.

Web Services

The Internet indirectly shed new light on the conflict between these competing protocols. The Internet is based on a set of open protocol standards that centered on the Hypertext Transport Protocol (HTTP) that is used to share information between machines. HTTP isn't a replacement for TCP/IP. Instead, HTTP is a high-level protocol that uses TCP/IP for low-level transmission.

The Internet solidified the direction of distributive systems by proving to corporations that they can greatly improve efficiency through better utilization of computer networks. But there was another hurdle to overcome. Internet technology lacked the capability to share software services that businesses needed to fully integrate large-scale business applications.

The next evolution of software services was born and was called web services. There is a common misnomer regarding web services. Some believe the "web" component of web services comes from the relationship web services has with the Internet. This isn't true. Web services is a web of services where services are software building blocks that are available on a network from which programmers can efficiently create large-scale distributive systems.

Three new standards were developed with the introduction of web services. These are Web Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI), and Service Oriented Architecture Protocol (SOAP).

Programmers use WSDL to publish their web service, thereby making the web service available to other programmers over the network. A programmer uses UDDI to locate web services that have been published and uses SOAP to invoke a particular web service. You'll learn more about WSDL, UDDI, and SOAP in Part V of this book.

Many large-scale distributive systems and web services have something in common. They are written using J2EE because J2EE addresses the complex issues that a programmer faces when developing a large-scale distributive system. There are numerous web services used in a typical large-scale distributive system and each service is associated with a tier in the multi-tier architecture that is used to share resources over a corporate infrastructure.

The Tier

A tier is an abstract concept that defines a group of technologies that provide one or more services to its clients. A good way to understand a tier structure's organization is to draw a parallel to a typical large corporation (see Figure 2-1).

At the lowest level of a corporation are facilities services that consist of resources necessary to maintain the office building. Facilities services encompass a wide variety of resources that typically include electricity, ventilation, elevator services, computer network services, and telephone services.

The next tier in the organization contains support resources such as accounting, supplies, computer programming, and other resources that support the main activity of the company. Above the support tier is the production tier. The production tier has the resources necessary to produce products and services sold by the company. The highest tier is the marketing tier, which consists of resources used to determine the products and services to sell to customers.

Any resource is considered a client when a resource sends a request for service to a service provider (also referred to as a service). A service is any resource that receives and fulfills a request from a client, and that resource itself might have to make requests to other resources to fulfill a client's request.

Let's say that a product manager working at the marketing tier decides the company could make a profit by selling customers a widget. The product manager requests an accountant to conduct a formal cost analysis of manufacturing a widget. The accountant is on the support tier of the organization. The product manager is the client and the accountant is the service.

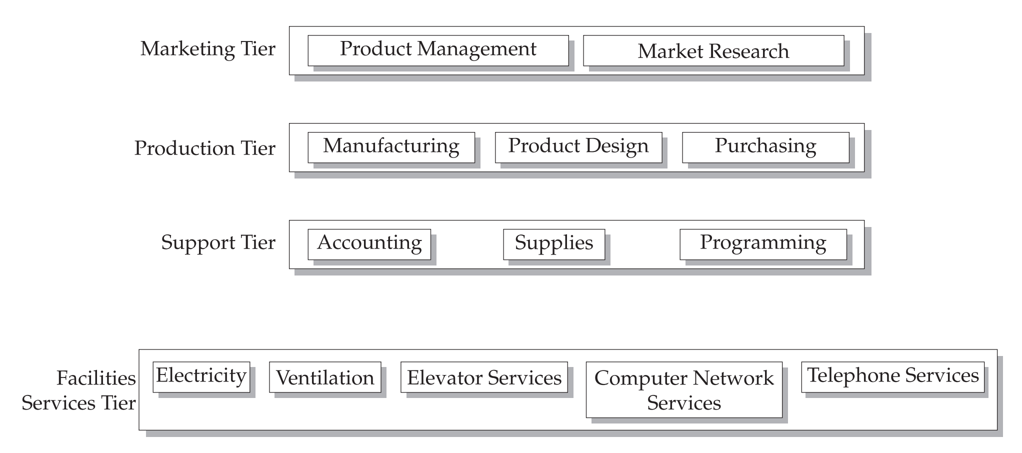

Figure 2-1. *Resources of a large organization are typically organized into a tier structure that operates similarly to the tier structure used in distributed systems.*

However, the accountant requires information from the manufacturing manager to fulfill the product manager's request. The manufacturing manager works on the production tier of the organization. The accountant is the client to the manufacturing manager who is the service to the accountant.

In multi-tier architecture, each tier contains services that include software objects, database management systems (DBMS), or connectivity to legacy systems. Information technology departments of corporations employ multi-tier architecture because it's a cost-efficient way to build an application that is flexible, scalable, and responsive to the expectations of clients. This is because the functionality of the application is divided into logical components that are associated with a tier. Each component is a service that is built and maintained independently of other services. Services are bound together by a communication protocol that enables a service to receive and send information from and to other services.

A client is concerned about sending a request for service and receiving results from a service. A client isn't concerned about how a service provides the results. This means that a programmer can quickly develop a system by creating a client program that formulates requests for services that already exist in the multi-tier architecture. These services already have the functionality built into them to fulfill the request made by the client program.

Services can be modified as changes occur in the functionality without affecting the client program. For example, a client might request the tax owed on a specific order. The request is sent to a service that has the functionality to determine the tax. The business logic for calculating the tax resides within the service. A programmer can modify the business logic in the service to reflect the latest changes in the tax code without having to modify the client program. These changes are hidden from the client program.

Clients, Resources, and Components

Multi-tier architecture is composed of clients, resources, components, and containers (see Figure 2-2). (In J2EE, the term "component" is used in place of the term "service," but both have the same philosophical meaning.) A *client* refers to a program that requests service from a component. A *resource* is anything a component needs to provide a service, and a *component* is part of a tier that consists of a collection of classes or a program that performs a function to provide the service. A *container* is software that manages a component and provides a component with system services.

The relationship between a container and a component is sometimes referred to as a contract, whose terms are governed by an application programming interface (API). An API defines rules a component must follow and the services a component will receive from the container.

A container handles persistence, resource management, security, threading, and other system-level services for components that are associated with the container. Components are responsible for implementation of business logic. This means programmers can focus on encoding business rules into components without becoming concerned about low-level system services.

This is an important concept in multi-tier architecture because modification can be made to low-level security, for example, without requiring any modification to a component. Only the container needs to be modified by the programmer.

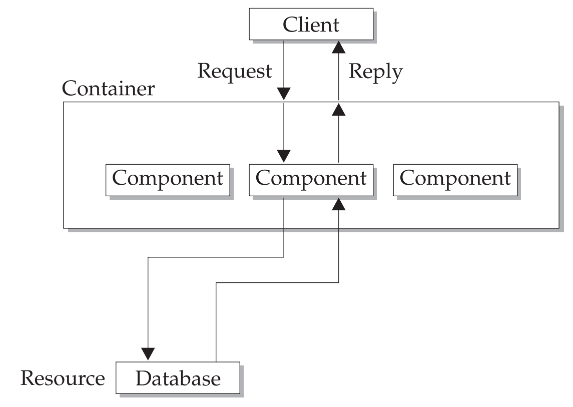

Figure 2-2. *A multi-tier architecture consists of clients, resources, components, and containers that are used by a programmer to create a distributive system.*

The relationship between a component and a container is very similar to the relationship between a program and an operating system. The operating system provides low-level system services such as I/O to a program. Programs don't need to be modified if a new disk drive is installed in the computer. Instead, the operating system is reconfigured to recognize the new disk drive.

Accessing Services

A client uses a client protocol to access a service that is associated with a particular tier. A protocol is a standard method of communication that both a client and the tier/component/resource understand. There are a number of protocols that are used within a multi-tier infrastructure because each tier/component/resource could use different protocols.

One of the most commonly implemented multi-tier architectures is used in web-centric applications where browsers are used to interact with corporate online resources. A browser is a client and requests a service from a web server using HTTP.

In a typical enterprise-wide application, a browser requests services from other components within infrastructures such as a servlet. A servlet uses a resource protocol to access resources that are necessary for the servlet to fulfill the request. For example, a servlet will use the JDBC protocol to retrieve data from DBMS. You'll be introduced to specific protocols throughout this book as you learn to build components and use resources.

J2EE Multi-Tier Architecture

J2EE is a four-tier architecture (see Figure 2-3). These consist of the Client Tier (sometimes referred to as the Presentation Tier or Application Tier), Web Tier, Enterprise JavaBeans Tier (sometimes referred to as the Business Tier), and the Enterprise Information Systems Tier. Each tier is focused on providing a specific type of functionality to an application.

It's important to delineate between physical location and functionality. Two or more tiers can physically reside on the same Java Virtual Machine (JVM) although each tier provides a different type of functionality to a J2EE application. And since the J2EE multi-tier architecture is functionally centric, a J2EE application accesses only tiers whose functionality is required by the J2EE application.

It's also important to disassociate a J2EE API with a particular tier. That is, some APIs (i.e., XML API) and J2EE components can be used on more than one tier, while other APIs (i.e., Enterprise JavaBeans API) are associated with a particular tier.

The Client Tier consists of programs that interact with the user. These programs prompt the user for input and then convert the user's response into requests that are forwarded to software on a component that processes the request and returns results to the client program. The component can operate on any tier, although most requests from clients are processed by components on the Web Tier. The client program also translates the server's response into text and screens that are presented to the user.

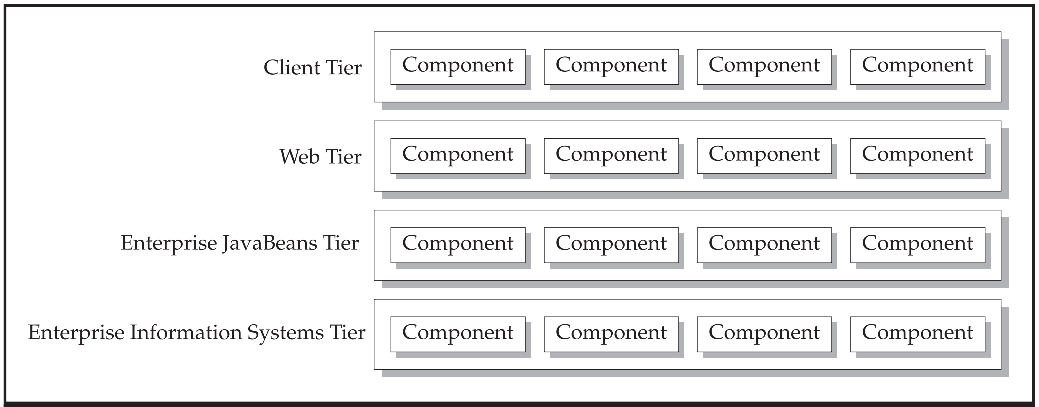

Figure 2-3. *J2EE consists of four tiers, each of which focuses on providing specific functionality to an application.*

The Web Tier provides Internet functionality to a J2EE application. Components that operate on the Web Tier use HTTP to receive requests from and send responses to clients that could reside on any tier. A client is any component that initiates a request, as explained previously in this chapter.

For example (see Figure 2-4), a client's request for data that is received by a component working on the Web Tier is passed by the component to the Enterprise JavaBeans Tier where an Enterprise Java Bean working on the Enterprise JavaBeans

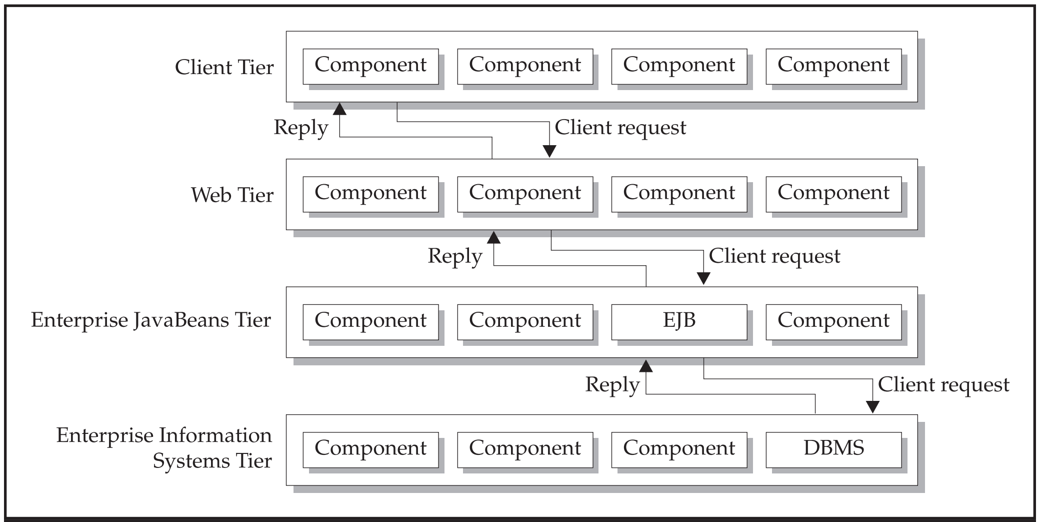

Figure 2-4. *A request is typically passed from one tier to another before the request is fulfilled.*

Tier interacts with DBMS to fulfill the request. Requests are made to the Enterprise JavaBeans by using the Java Remote Method Invocation (RMI) API. The requested data is then returned by the Enterprise JavaBeans where the data is then forwarded to the Web Tier and then relayed to the Client Tier where the data is presented to the user.

The Enterprise JavaBeans Tier contains the business logic for J2EE applications. It's here where one or more Enterprise JavaBeans reside, each encoded with business rules that are called upon indirectly by clients. The Enterprise JavaBeans Tier is the keystone to every J2EE application because Enterprise JavaBeans working on this tier enable multiple instances of an application to concurrently access business logic and data so as not to impede the performance.

Enterprise JavaBeans are contained on the Enterprise JavaBeans server, which is a distributed object server that works on the Enterprise JavaBeans Tier and manages transactions and security, and assures that multithreading and persistence are properly implemented whenever an Enterprise JavaBean is accessed.

Although an Enterprise JavaBean can access components on any tier, typically an Enterprise JavaBean accesses components and resources such as DBMS on the Enterprise Information System (EIS) Tier.

Access is made using an Access Control List (ACL) that controls communication between tiers. The ACL is a critical design element in the J2EE multi-tier architecture because ACL bridges tiers that are typically located on different virtual local area networks and because ACL adds a security level to web applications. Hackers typically focus their attack on the Web Tier to try to directly access DBMS. ACL prevents direct access to DBMS and similar resources.

The EIS links a J2EE application to resources and legacy systems that are available on the corporate backbone network. It's on the EIS where a J2EE application directly or indirectly interfaces with a variety of technologies, including DBMS and mainframes that are part of the mission-critical systems that keep the corporation operational. Components that work on the EIS communicate to resources using CORBA or Java connectors, referred to as J2EE Connector Extensions.

Client Tier Implementation

There are two components on the Client Tier that are described in the J2EE specification. These are applet clients and application clients. An applet client is a component used by a web client that operates within the applet container, which is a Java-enabled browser. An applet uses the browser as a user interface.

An application client is a Java application that operates within the application client container, which is the Java 2 Runtime Environment, Standard Edition (JRE). An application has its own user interface and is capable of accessing all the tiers in the multi-tier architecture depending how the ACLs are configured, although typically an application has access to only the web layer.

A rich client is a third type of client, but a rich client is not considered a component of the Client Tier because a rich client can be written in a language other than Java—and therefore J2EE doesn't define a rich client container.

A rich client is similar to an application client in that both are applications that contain their own user interface. And as with an application client, a rich client can access any tier in the environment, depending on the ACLs configuration, using HTTP, SOAP, ebXML, or an appropriate protocol.

Classification of Clients

Besides defining clients as an applet client, application client, or a rich client, clients are also classified by the technology used to access components and resources that are associated with each tier. There are five classifications: a web client, Enterprise JavaBeans client, Enterprise Information System (EIS) client, web service peers, and a multi-tier client.

A web client consists of software, usually a browser, that accesses resources located on the Web Tier. These resources typically consist of web pages written in HTML or XML. However, a web client can also access other kinds of information that is located on the Web Tier. Web clients communicate with Web Tier resources using either HTTP or the Hypertext Transmission Protocol Secured (HTTPS), which is used to transfer encrypted information.

Enterprise JavaBeans clients are similar to web clients in that an Enterprise JavaBeans client works on the Client Tier and interfaces the J2EE application with the user. However, an Enterprise JavaBeans client only accesses one or more Enterprise JavaBeans that are located on the Enterprise JavaBeans Tier rather than resources on the Web Tier.

This access is made possible by using the RMI API. RMI handles communication between the Enterprise JavaBeans client and the Enterprise JavaBeans Tier using either the Java Remote Method Protocol (JRMP) or the Internet Inter-ORB Protocol (IIOP).

EIS clients are the interface between users and resources located on the EIS Tier. These clients use Java connectors, appropriate APIs, or proprietary protocols to utilize resources such as DBMS and legacy data sources.

A web service peer is a unique type of client because it's also a service that works on the Web Tier. Technically, a web service peer forms a peer-to-peer relationship with other components on the Web Tier rather than a true client/server relationship. However, a web service peer is commonly referred to as a client because it requests service from other components on the Web Tier, although a web service peer can also access other tiers. Typically, a web service peer makes requests over HTTP using either electronic business XML or the Simple Object Access Protocol (SOAP).

Multi-tier clients are conceptually similar to a web service peer except a multi-tier client accesses components located on tiers other than the tier where the multi-tier client resides. Multi-tier clients typically use the Java Message Service (JMS) to communicate asynchronously with other tiers.

Web Tier Implementation

The Web Tier has several responsibilities in the J2EE multi-tier architecture, all of which is provided to the Client Tier using HTTP. These responsibilities are to act as an intermediary between components working on the Web Tier and other tiers and the Client Tier. Intermediary activities include

- Accepting requests from other software that was sent using POST, GET, and PUT operations, which are part of HTTP transmissions
- Transmit data such as images and dynamic content

There are two types of components that work on the Web Tier. These are servlets and JavaServer Pages (JSP), although many times they are proxied to the Application or EJB Tier. A servlet is a Java class that resides on the Web Tier and is called by a request from a browser client that operates on the Client Tier. A servlet is associated with a URL that is mapped by the servlet container.

A request for a servlet contains the servlet's URL and is transmitted from the Client Tier to the Web Tier using HTTP. The request generates an instance of the servlet or reuses an existing instance, which receives any input parameters from the Web Tier that are necessary for the servlet to perform the service. Input parameters are sent as part of the request from the client.

An instance of a servlet fulfills the request by accessing components/resources on the Web Tier or on other tiers as is necessary based on the business logic that is encoded into the servlet. The servlet typically generates an HTML output stream that is returned to the web server. The web server then transmits the data to the client. This output stream is a dynamic web page.

JSP is similar to a servlet in that a JSP is associated with a URL and is callable from a client. However, JSP is different than a servlet in several ways, depending on the container that is used. Some containers translate the JSP into a servlet the first time the client calls the JSP, which is then compiled and the compiled servlet loaded into memory. The servlet remains in memory. Subsequent calls by the client to the JSP cause the web server to recall the servlet without translating the JSP and compiling the resulting code. Other containers precompile a JSP into a .java file that looks like a servlet file, which is then compiled into a Java class.

Business logic used by JSP and servlets is contained in one or more Enterprise JavaBeans that are callable from within the JSP and servlet. The code is the same for both JSP and servlet, although the format of the code differs. JSP uses custom tags to access an Enterprise JavaBeans while servlets are able to directly access Enterprise JavaBeans. You'll learn how to create and use servlets in Chapter 10, JSPs in Chapter 11, and Enterprise JavaBeans in Chapter 12.

Enterprise JavaBeans Tier Implementation

J2EE uses distributive object technology to enable Java developers to build portable, scalable, and efficient applications that meet the 24-7 durability expected from an enterprise system. The Enterprise JavaBeans Tier contains the Enterprise JavaBeans server, which is the object server that stores and manages Enterprise JavaBeans.

The Enterprise JavaBeans Tier is a vital element in the J2EE multi-tier architecture because this tier provides concurrency, scalability, lifecycle management, and fault tolerance. The Enterprise JavaBeans Tier automatically handles concurrency issues that assure multiple clients have simultaneous access to the same object. The Enterprise JavaBeans Tier is the tier where some vendors include features that enable scalability of an application, because the tier is designed to work in a clustered environment. This assumes that vendor components that are used support clustering. If not, a Local Director is typically used for horizontal load balancing.

The Enterprise JavaBeans Tier manages instances of components. This means component containers working on the Enterprise JavaBeans Tier create and destroy instances of components and also move components in and out of memory.

Fault-tolerance is an important consideration in mission-critical applications. The Enterprise JavaBeans Tier is the tier where some vendors include features that provide fault-tolerant operation by making it possible to have multiple Enterprise JavaBeans servers available through the tier. This means backup Enterprise JavaBeans servers can be contacted immediately upon the failure of the primary Enterprise JavaBeans server.

The Enterprise JavaBeans server has an Enterprise JavaBeans container within which is a collection of Enterprise JavaBeans. As discussed in previous sections of this chapter, an Enterprise Java Bean is a class that contains business logic and is callable from a servlet or JSP.

Collectively the Enterprise JavaBeans server and Enterprise JavaBeans container are responsible for low-level system services that are required to implement business logic of an Enterprise Java Bean. These system services are

- Resource pooling
- Distributed object protocols
- Thread management
- State management
- Process management
- Object persistence
- Security
- Deploy-time configuration

A key benefit of using the Enterprise JavaBeans server and Enterprise JavaBeans container technology is that this technology makes proper use of a programmer's expertise. That is, a programmer who specializes in coding business logic isn't concerned about coding system services. Likewise, a programmer whose specialty is system services can focus on developing system services and not be concerned with coding business logic.

Any component, regardless of the tier where the component is located, can use Enterprise JavaBeans. This means that an Enterprise Java Bean client can reside outside the Client Tier. The protocol used to communicate between the Enterprise JavaBeans Tier and other tiers is dependent on the protocol used by the other tier.

Components on the Client Tier and the Web Tier communicate with the Enterprise JavaBeans Tier using the Java RMI API and either IIOP or JRMP. Sometimes software on other tiers, usually the middle tier, uses JMS to communicate with the Enterprise JavaBeans Tier. This communication isn't exclusively used to send and receive messages between machines. JMS is also used for other communication, such as decoupling tiers using the queue mechanism.

However, the Enterprise Java Bean that is used must be a message-driven bean (MDB). MDBs are commonly used to process messages on a queue that may or may not reside on the local machine. You'll learn more about MDB when Enterprise JavaBeans is discussed in detail in Chapter 12.

Enterprise Information Systems Tier Implementation

The Enterprise Information Systems (EIS) Tier is the J2EE architecture's connectivity to resources that are not part of J2EE. These include a variety of resources such as legacy systems, DBMS, and systems provided by third parties that are accessible to components in the J2EE infrastructure.

This tier provides flexibility to developers of J2EE applications because developers can leverage existing systems and resources currently available to the corporation and do not need to replicate them in J2EE.

Likewise, developers can utilize off-the-shelf software that is commercially available in the marketplace because the EIS Tier provides the connectivity between a J2EE application and non-J2EE software. This connectivity is made possible through the use of CORBA and Java Connectors or through proprietary protocols.

Java Connector technology enables software developers to create a Java Connector for legacy systems and for third-party software. The connector defines all the elements that are needed to communicate between the J2EE application and the non-J2EE software. This includes rules for connecting to each other and rules for conducting secured transactions. You'll learn more on how these connections are created in Part IV, Java Interconnectivity.

Challenges

Although J2EE enables programmers to design and build large-scale distributive systems that use web services connected together in a multi-tier architecture, there remain design considerations that could inhibit successful deployment of the system.

Let's begin with transactions. Typically, resources are locked until a transaction is completed, which is adequate for short-lived transactions. However, an issue occurs when a single transaction takes hours to complete and other components of the system require access to resources that are being used for the transaction. Programmers must build into an application a routine that correlates IDS and uses JMS to decouple tiers that are locked.

Reliability is another issue. The assumption is that a resource is always available and will provide optimal performance when called within a distributive system. However, the assumption is based on a false premise, as proven when a resource (i.e., database) within a corporate network environment is offline at the time a system requires database access. This situation is compounded when resources are outside the corporate network environment and become services that are provided by a third party.

This means that Java applications must provide proper error-handling routines. The compiler forces the developer to either catch or throw all exceptions except for RuntimeExceptions. It's up to the developer to provide a graceful response to an error condition. The developer must also be aware of possible RuntimeExceptions that may be thrown during the execution of a program. These can be a little subtler because the compiler does not require them to be caught or thrown.

Likewise, security becomes an issue. Even if access to resources is made using HTTP and Secured Sockets Layer (SSL), another security level is needed to assure that only authorized systems have access to the resource. SSL only protects data when it's en route between two systems. Similar security issues are successfully addressed in today's corporate infrastructure using a variety of techniques that include ACLs, IP filtering, and routing. However, security becomes complex in a web service multi-tier architecture application if resources used by the application are provided by organizations outside a corporation's infrastructure.

This leads to the issue of how to manage and test a large distributive system that uses web services. There are many resources used by such a system, and each resource must be acquired, integrated into the system, accessible, and accurately perform its function; otherwise, the system fails. A large distributive system can have numerous fail points that must be adequately tested.