J2EE - Basics

What Is J2EE?

Before defining what J2EE is, let us understand that J2EE is not simply a language, package, utility, or service.

Java 2 Platform, Enterprise Edition (J2EE)

In simpler terms, J2EE is essentially a standard middleware architecture, proposed by Sun Microsystems for developing and deploying multitier, distributed, e-enabled, enterprise scale business applications. Applications written for J2EE standards enjoy certain inherent benefits such as portability, security, scalability, load-balancing, and reusability.

Middleware are essentially server-side software solutions that provide the much-required foundation for linking several disparate systems and resources that are scattered across the network. Prior to the introduction of J2EE, middleware solutions were highly proprietary and restrictive to specific vendors and products—with limited features and compatibility, and no interoperability or portability across different solutions. There was no common/acceptable industry standard in place to adhere to, and many of the features were left to the choice of vendors.

J2EE represents the maturity and seasoning that middleware technology has undergone over the years by learning from the mistakes of the past and addressing all the essential requirements of the industry. It also provides enough room for future developments. While developing this standard, Sun collaborated with other major vendors of middleware, operating systems, and database management systems—including IBM and Oracle.

At its core, J2EE is a set of standards and guidelines that defines how distributed n-tier applications can be built using the Java language. Developers build their applications on the top of these standards while middleware infrastructure vendors ensure compatibility to these guidelines set forth by J2EE. Thus, J2EE applications can be ported and deployed across several application servers, with minimal or no code-level changes. This concept is represented in Figure 1.

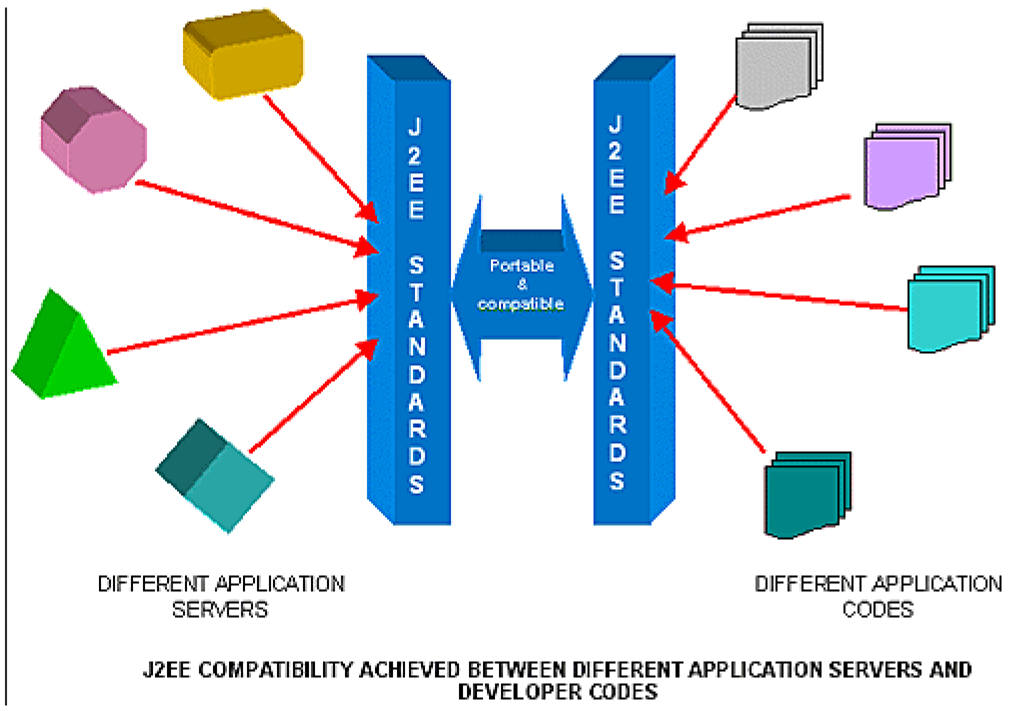
J2EE STANDARDS
Portable & compatible
J2EE STANDARDS
DIFFERENT APPLICATION SERVERS
DIFFERENT APPLICATION CODES
J2EE COMPATIBILITY ACHIEVED BETWEEN DIFFERENT APPLICATION SERVERS AND DEVELOPER CODES

Figure 1 J2EE compatibility between application servers and developer codes.

Perspectives on J2EE

J2EE offers several perspectives, as discussed in the following sections.

J2EE: A Syntax for Multitier, Distributed Middleware

J2EE clearly demarcates various tiers that are involved in application development, and defines components that can be hosted in those tiers. These tiers include the clients tier, presentation logic tier, business logic tier, and enterprise information systems tier. All J2EE applications are built on the top of this basic framework, and they naturally evolve as multitier systems, even without conscious effort. Each tier may be physically distributed across several servers.

With J2EE, distributed application development is no longer a complex task. J2EE components make no assumptions about the server environment in which they exist—and all resources are accessed through distributed directories. This means that there is no deliberate effort required on the part of application developers to distribute their components and resources.

J2EE: A Platform for Enterprise Scale Applications

J2EE, implemented as specific Web application servers such as BEA Web logic or IBM Web sphere, is a platform for building enterprise scale distributed applications. Applications can be built on top of the J2EE application-programming model, and can be deployed in one or more J2EE-compatible Web application servers.

A specific application server platform that is best suited for a given enterprise IT infrastructure can be chosen from a wide variety of J2EE compatible products—each enjoying its own distinctive advantage over the others. Irrespective of their unique features, all J2EE application servers provide a common groundwork for developing and deploying enterprise scale applications.

J2EE: A Model for e-Enabled Application Development

J2EE applications can be very easily exposed to Web, Palm, and handheld devices; mobile phones; and a variety of other devices. In other words, application components can be "e-enabled" without much effort. The J2EE application-programming model ensures that the business logic and back-end systems remain untouched as their facilities are exposed to different types of client access.

Yet another great feature of J2EE platform is automatic load-balancing, scaling, fault-tolerance, and fail-over. Components deployed in J2EE environment automatically inherit these facilities, and there is no deliberate coding effort required.

These features are significantly important for constructing Web portals that are available to clients 24/7/365.

J2EE: The Widely Adapted Standard in Web Application Servers

J2EE is perhaps the first industry standard to relish widespread industry recognition and adoption in the middleware world. Almost all top-notch Web application servers (BEA's Weblogic, IBM's Web sphere, HP's Application servers, Sun-Netscape's iPlanet, and Macromedia's Jrun, to name a few) are all J2EE-certified application servers. No other standard advocated before has been supported by such a long list of middleware infrastructure providers.

Moreover, with J2EE, companies are no longer nailed down to a specific vendor or application server provider. As long as the application components stick to J2EE specifications, they can be deployed across different application servers along the enterprise network. To ensure compatibility and coherence across different J2EE application servers, Sun has released a compatibility test suite.

Vision of J2EE

The primary vision that propels J2EE can be summarized as follows: "Developers should be writing codes to express their business and presentation logic, whereas the underlying middleware infrastructure takes care of system-level issues such as memory management, multithreading, resource allocation, availability, and garbage collection—automatically."

How Does J2EE Meet the Challenge?

Now, let us see how J2EE addresses the core issues and problems faced by enterprises—and how it provides scope for future developmental needs.

Hardware- and OS-Independent

The J2EE runtime resides over the Java virtual machine (JVM)—as shown in Figure 2. JVM is essentially hardware- and OS-independent. The Java runtime environment (JRE), which is the installable version of JVM and other essential components, is available for almost all popular hardware/OS compositions. Thus, by adopting Java, J2EE relieves enterprises from reinvesting in expensive hardware and operating systems.

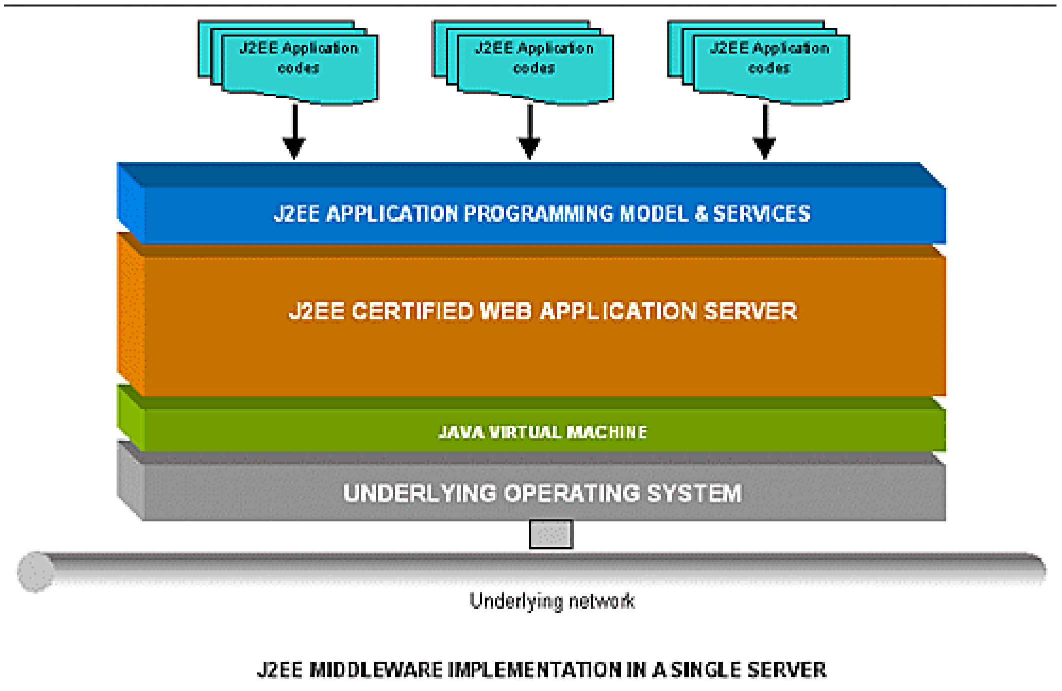
J2EE Application codes
J2EE Application codes
J2EE Application codes
J2EE APPLICATION PROGRAMMING MODEL & SERVICES
J2EE CERTIFIED WEB APPLICATION SERVER
JAVA VIRTUAL MACHINE
UNDERLYING OPERATING SYSTEM
Underlying network
J2EE MIDDLEWARE IMPLEMENTATION IN A SINGLE SERVER

Figure 2 J2EE middleware implementation.

Another significant advantage is that it is possible to accommodate the J2EE enterprise integration tier much closer to the actual back-end systems. In some cases, it can be hosted well within the EIS server environment itself, saving network traffic and increasing performance.

Adherence to Object-Oriented Design and Component Methodology

Being strictly an object-oriented language, Java lends itself to well-disciplined and structured coding. Almost all features of object-oriented programming are fully supported in Java. The J2EE application-programming model is built on top of the object-oriented methodologies and components-based design.

J2EE has a variety of component models for different tiers. Thus, by adopting suitable combinations for a given project at hand, developers achieve a high degree of extendibility and agility in developing and assembling components—often resulting in rapid application development. The implication of all this for enterprises is that the resulting applications are well-structured, modular, flexible, and reusable.

Flexibility, Portability, and Interoperability

Java codes are portable across different operating systems, and J2EE components are portable across different application-server environments. This implies that enterprises can instantaneously embark on a unified development platform whose codes are transferable across all machines.

Applications can be hosted in more than one J2EE application server. J2EE components that sit across several servers in different environments can still work in orchestration with each other. This feature gives unprecedented interoperability in assembling application components.

For example, it is possible to host EJBs that interact directly with mainframe/CICS environments in IBM Websphere environments, and access them from EJBs in BEA Weblogic servers that are hosted elsewhere.

Effortless Enterprise Information Systems Integration

J2EE has rapidly brought in watertight industry standards—such as JDBC, JMS, and JCA—that simplify the seamless integration of enterprise information systems such as legacy systems, ERP implementations, and relational databases.

Almost all major relational database vendors support JDBC. Thus, J2EE applications enjoy the capability to communicate with all popular databases, with the help of necessary JDBC drivers. Similarly, a number of EI adapters are appearing in the industry—based on Java connector architecture standards—that provide seamless integration with various legacy systems and ERP/CRM implementations.

Adopting Services-Oriented Architectures in J2EE

With the advent of Web services and open standards such as SOAP, interoperability across several disparate systems in an enterprise becomes a possibility. J2EE, being an extensible platform, naturally lends itself to exposing its core components and resources to Web services. Sun has released a comprehensive set of APIs, called the JAX package, which support all kinds of Web service requirements such as XML parsing, XML binding, SOAP message consumption and delivery, Registry lookup, XML RPCs, and XML messaging.

Although the J2EE platform was advocated long before the advent of Web services technologies, it provides room for fitting in the latest developments. It can be assumed with reasonable confidence that J2EE will be capable of accommodating more technological advancements in the future without disturbing its core framework and application-programming model.

Conclusions

J2EE is a middleware standard that has been widely adopted in the industry. It is the ideal platform for developing multitier, distributed, platform-independent enterprise applications that can scale and load-balance. The primary focus of J2EE is to provide a standard middleware infrastructure to take care of all low-level system issues, so that developers can concentrate on coding their business logic.

J2EE addresses the needs of the industry very effectively. It provides an operating system-independent environment; and resulting applications are flexible, portable, and reusable across different vendor implementations. Above all, J2EE infrastructure is capable of accommodating newer technologies and developments in the industry.

J2EE: Core Concepts

J2EE can be understood as an enterprise-scale, middleware architecture or platform that links several resources and applications scattered across the network. It provides a set of application components and a runtime environment to construct and host scalable business applications.

Physically, the J2EE environment can exist in more than one server, and a single business application can be deployed as a suite of distributed components in one or more servers across the network.

Understanding J2EE involves learning about the following core concepts that make up J2EE:

- J2EE n-tier application architecture: the basic infrastructure provided by a J2EE platform
- J2EE application components: the software elements with which J2EE applications can be contrived
- J2EE enterprise services: the common facilities available for application components
- J2EE containers: the J2EE runtime environment

J2EE n-Tier Application Architecture

J2EE middleware defines a set of four independent tiers, over which applications can be built:

- Client tier
- Presentation logic tier
- Business logic tier
- Enterprise information systems tier

A pictorial representation of these tiers is shown in Figure 3.

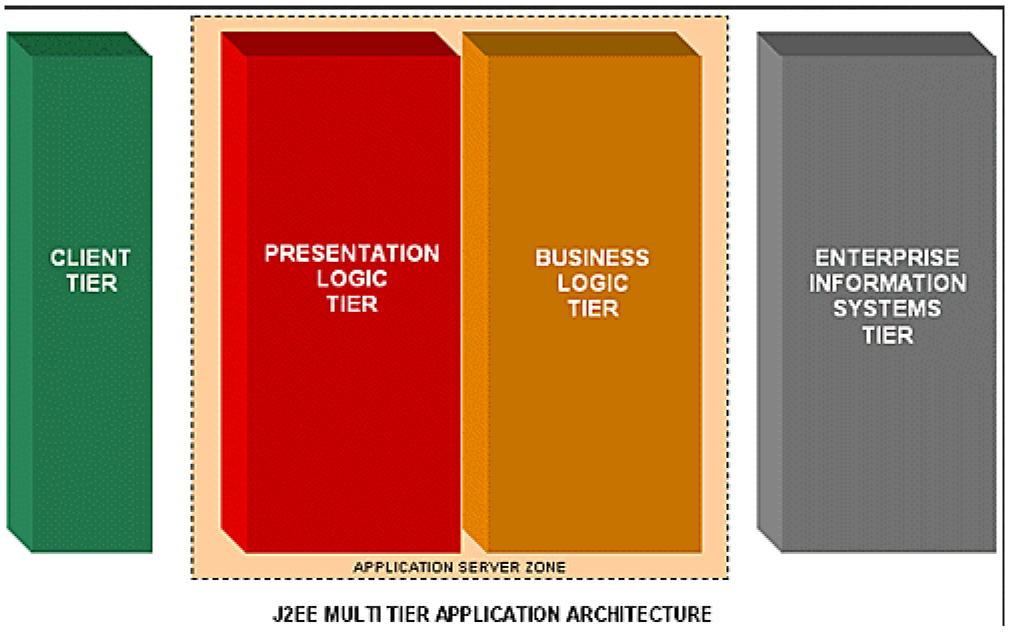

Figure 3 J2EE: multitier application architecture.

Presentation logic and business logic tiers fall under the application server zone—which is nothing but the J2EE platform implementation.

Each of these tiers may physically be sitting over several machines. Moreover, within the application server zone the presentation logic can be hosted in one application server, and business logic can be hosted in another.

For example, it is possible to use iPlanet (a J2EE application server product from Sun) as an HTTP Web server and presentation logic container, and use Weblogic (another J2EE application server from BEA) to deploy business logic components.

It is common to associate the presentation logic tier with Servlet and JSP containers (discussed later in this article) and business logic tier to the Enterprise Javabeans container. It will be easier to understand tiers as conceptual entities—meant for ease of design—and containers as physical software instances—meant for providing the runtime environment for application components.

J2EE's multitier architecture is inspired by Model View Controller (MVC) architecture: A software methodology used for demarcating the boundaries and scope of individual application components in a complex system.

MVC advocates that those codes, which represent the business logic or core application functionality, should not be intermingled with those that present the proceed results to clients or other applications. They should possibly be independent of each other, and a controller should mediate the interactions between them.

For example, let us consider an Internet-based, bank account-transfer process. According to MVC, this process can be broken down to four independent tasks, as shown in Figure 4.

- Task 1: The task of invoking the transfer from the browser (view)
- Task 2: The task of invoking the mechanism that does the account transfer (controller)
- Task 3: The task of doing the actual account transfer (business process or model)
- Task 4: The task of showing the status of the transfer (success or failure) to the browser (view)

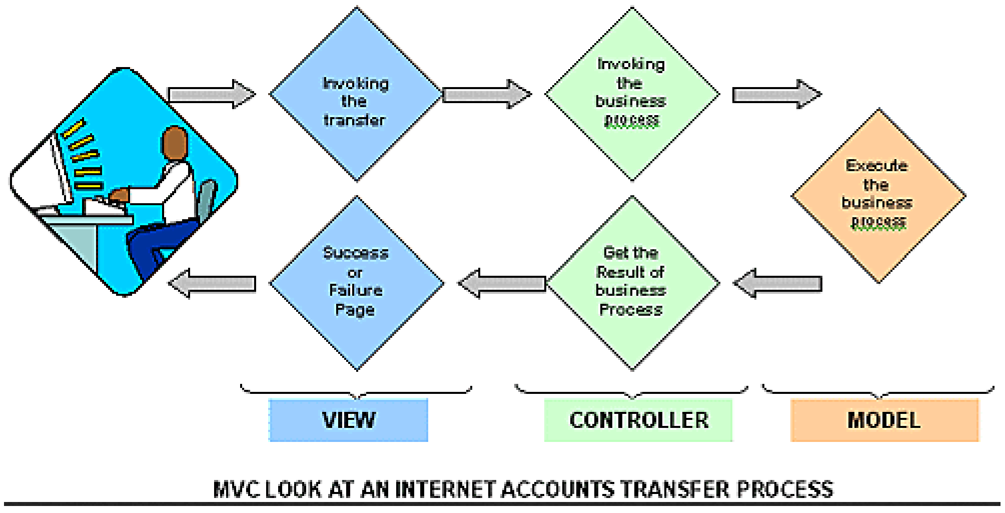
Invoking the transfer
Invoking the business process
Execute the business process
Success or Failure Page
Get the Result of business Process
VIEW
CONTROLLER
MODEL
MVC LOOK AT AN INTERNET ACCOUNTS TRANSFER PROCESS

Figure 4 MVC look at an Internet account transfer process.

MVC states that the core business process (Task 3) should not assume anything about the clients. Instead of a browser, another application or a back office system may well invoke it. The process of orchestrating between the business logic and view elements is to be assigned to a dedicated entity called the controller.

Having been based on MVC methodology, J2EE's architecture naturally demarcates business logic from presentation logic tier. Controllers can be placed in either of these tiers or both. By doing so, J2EE provides room for reusability of business logic components.

J2EE Application Components

The J2EE standard defines a rich set of application component framework, with which almost all kinds of business applications can be built. These applications range from simple Web portals to complex, distributed, enterprise-scale transactional applications. Application components, which are the brick and mortar of enterprise systems, are built on the top of this framework.

J2EE component framework is only a rudimentary infrastructure—in the form of libraries, base classes, and interfaces. Application components that are built over them usually embody business logic and the presentation/controller logic that makes up the application.

For example, let us consider servlets, which are standard application components provided by J2EE. Servlets for a given business application are to be developed on top of a basic servlet interface provided by J2EE. While doing so, developers can make use of servlet libraries and services that come along with the servlet package. Many of the system level services, such as reading data from an HTTP input stream or writing data to an HTTP output stream, are available within these libraries, so that applications can readily make use of them.

Hence, we term the component infrastructure provided by J2EE as an *application framework*, and those codes written on top of it as J2EE *application components*. But I'll be using these two terms interchangeably, throughout this series for the sake of simplicity.

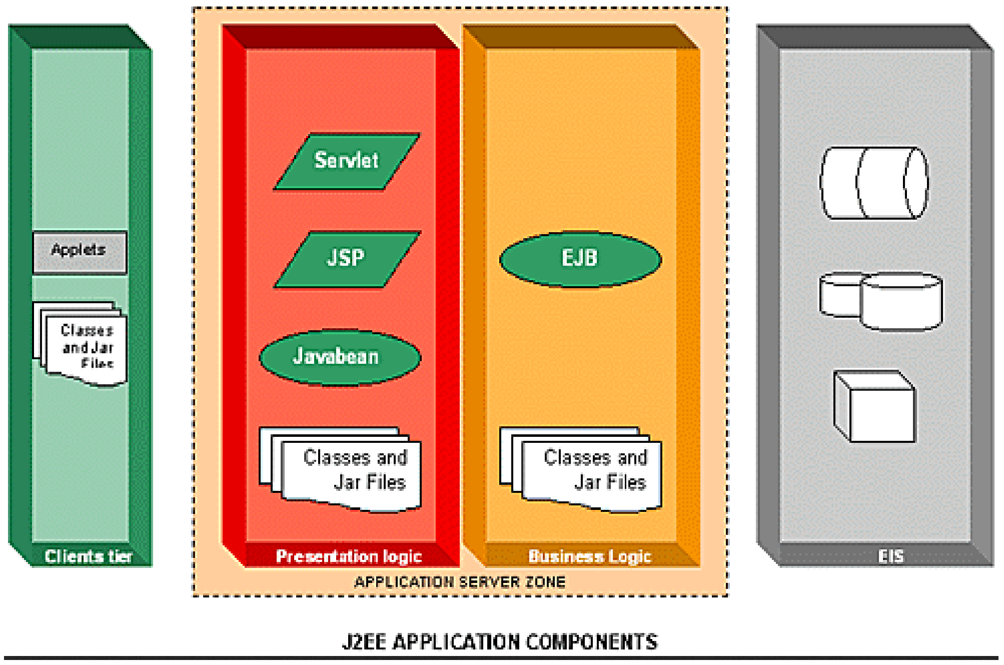

Figure 5 J2EE application components.

J2EE application components are available for the client tier, presentation tier, and business logic tiers, as shown in Figure 5.

These components include the following:

- Client tier: applets, Javabeans

- Presentation logic tier: servlets, Java server pages, Javabeans
- Business logic tier: Enterprise Javabeans

It is very important to note that normal Java classes and Jar archives can be hosted in all of these tiers very well, apart from the J2EE-specific components listed above. In many cases, we will find application codes spilling outside the J2EE components into normal Java classes.

Components are not available for the Enterprise Information System (EIS) tier because EIS encompasses all enterprise backend resources —such as databases, BackOffice/legacy systems, and ERP implementations—in the J2EE realm. Obviously, all we need is *access* to these resources from our application components, which are taken care of by J2EE enterprise services. Thus, there is no scope for application components in the EIS tier.

Applications are to be coded by the programmer, using one or more of these components, and deployed in respective containers.

Arriving at a proper mix of application components for a given business scenario is the crafty job of a J2EE architect. Because there is a wide variety of components available, it requires considerable knowledge and experience to choose the right mixture of components that will result in optimized code and performance-rich applications.

By assigning specific and well-defined roles to each application component, J2EE aims to provide room for logical application architecture, design, and development. The component interfaces ensure that the components adhere to certain standards and exhibit common behavior, which provides the base for interoperable J2EE server environments.

These application components are loosely coupled across the tiers to ensure flexibility and reusability while interacting with each other. A whole range of enterprise services—such as mailing, database connectivity, messaging, and transaction processing—are available at the disposal of application components.

On the flip side, unlike .Net framework, which is Microsoft's middleware architecture, all J2EE application components are to be coded only in the Java language!

J2EE Enterprise Services

The nutrients for the blossoming of enterprise applications—such as mailing, database connectivity, messaging, and transaction processing—are readily catered to by the J2EE environment as enterprise services.

These essential services are made available in the form of interfaces, class libraries, drivers, and adapters. As in the case of operating systems, J2EE provides a common Java interface over these services, so that the application codes are shielded from the subtleties of specific service implementation provided by the application vendors.

For example, JDBC is one service that provides connectivity to a wide variety of databases available in the market. Connectivity to different databases (such as Oracle, MySQL, Informix, and SQL Server) is achieved through specific JDBC drivers that are provided by the database manufacturers themselves or by third-party vendors. Application developers need not worry about the intricacies of specific databases; all they need to do is to make use of standard JDBC calls to access these resources, and the underlying JDBC driver takes care of implementing them.

J2EE enterprise services include the following:

Connectivity Services

- Java Database Connectivity (JDBC): provides database connectivity services
- Java Connector Architecture (JCA): provides connectivity to legacy systems

Communication Services

- Java Messaging Service (JMS): provides messaging services across the tiers and components
- Java Activation Framework/Java Mailing services (JAF/Javamail): provides e-mail services
- Java Interface Definition Language/RMI—IIOP: enable CORBA communication with J2EE
- Java XML APIs (JAX): provides XML parsing/binding services

Identification Services

- Java Naming and Directory Interface (JNDI): provides distributed naming and directory services

Other Services

- Java Transaction Service (JTS/JTA): provides transaction-monitoring services
- Java Authentication and Authorization service (JAAS): provides access control services

Specific drivers and adapters for selected services usually come bundled with the application server itself. For example, Web logic comes with a JDBC driver for connecting J2EE applications to Informix. This becomes an important selection criterion when evaluating different application servers for a given enterprise infrastructure.

J2EE Runtime Environment (Containers)

J2EE application components in different tiers come to life in their runtime environments, which are called *containers* in J2EE terminology. These containers are vendor-specific products that adhere to certain common interfaces, and provide the much-required low-level infrastructure facilities for J2EE middleware components.

After being coded to J2EE standards, application components need to be deployed in the respective containers by using vendor-specific deployment tools.

J2EE standard defines four different containers:

- Applet container: hosts applets
- Application client container: hosts standard Java application clients (including swing windows applications)
- Web container: hosts servlets and JSPs in the presentation logic tier
- EJB container: hosts Enterprise Javabeans in the business logic tier

It is typical to include an HTTP Web server along with the Web containers to host static HTML Web pages, and almost all popular J2EE application servers have built-in HTTP Web servers.

It is worthwhile to note that one of the principal visions of J2EE architecture was that developers should be writing codes for their business logic, and should not worry about system-level capabilities. This vision is achieved only through the concept of containers.

The infrastructure facilities provided by the containers typically include memory management, synchronization/threading, garbage collection, availability, scaling, load balancing, and fault tolerance.

The basic interfaces and facilities that should be implemented by the containers are defined in J2EE specifications, but the means by which they are to be accomplished, is left to the container vendor. Thus, J2EE provides the scope for vendor-specific codes while ensuring compatibility across different application servers. In fact, it is mainly with the help of containers, apart from other facilities, that vendors are able to distinguish themselves in the middleware industry.

Conclusions

The J2EE platform consists of multitier application architecture, based on Model View Controller methodology. Varieties of application component models are available for each of these tiers, by which all kinds of enterprise applications can be built.

These application components are serviced by a rich set of drivers and utilities. They take care of all essential services that might be required by the applications (for example, database connectivity, e-mailing, authentication, and distributed directory access.

Application components can be hosted in J2EE runtime environments called containers.

J2EE Application Programming Model

We have already discussed various components, enterprise services and runtime environments that constitute J2EE in earlier articles. Now it is time to assemble all of them in one place, and take a comprehensive look at the overall J2EE platform.

Have a look at Figure 6 below.

This famous model is often known as the J2EE Application Programming Model. All enterprise J2EE application architectures are built on top of this basic design.

Because our main focus in this series is on the application server area, we will first consider Web and EJB containers in our discussions.

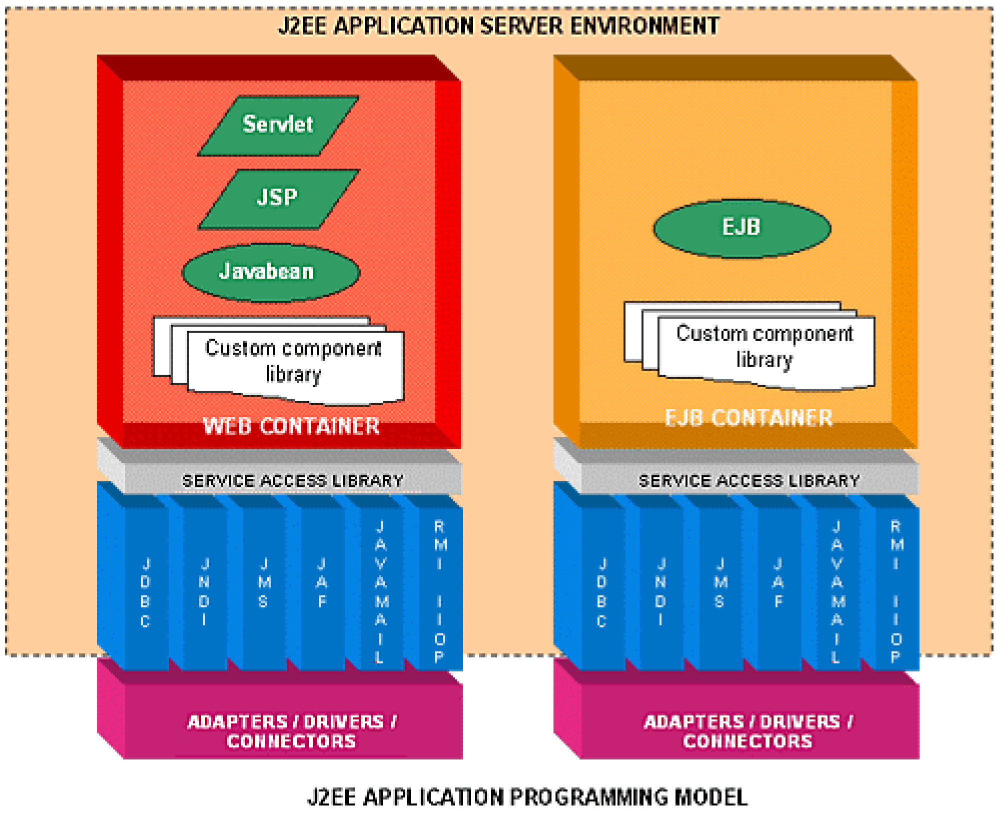

Figure 6 J2EE Application Programming Model.

Points to be noted in Figure 6 are as follows:

- There is a clear-cut distinction between Web containers and application containers—implying that they are relatively independent of each other, and may represent products from two different vendors.
- Enterprise services are available to both containers. In other words, access to facilities such as database connectivity, e-mailing, directory services, and messaging are available to Web container components as well as EJBs.
- Components can access EIS resources only through one or more enterprise service implementations in the form of drivers and adapters.
- The Java Virtual Machine permeates the application server environment, implying that the Java Runtime Environment (JRE) is available with all application servers.
- In any project, it is generally a good practice to consolidate all custom-made services and libraries that are frequently used by application components into a separate layer. In the J2EE model provided, I've termed these services *custom component libraries*.
- These libraries are developed in-house, and are not a part of standard J2EE application server. These libraries typically tend to grow over a period of time as the projects mature.
- An example of this is a configuration file utility, which reads a configuration filename, finds out its exact physical location in the file system, and returns the necessary parameters as integers or strings.
- It is also typical to find developers ending up on the top of core enterprise services provided by J2EE when writing their own customized access routines. It is worthwhile to consolidate them into a separate layer called *service access libraries*.

For example, let us say we are using IBM's MQ series as the messaging system in a project. We often end up writing some simple APIs on the top of the JMS codes used to access the MQ series services—such as adding standard, enterprise-defined headers and footers for all outgoing messages. It is better to consolidate these services as common APIs, and make them available across the enterprise platform.

In short, service access libraries provide a level of abstraction above the J2EE APIs, which is very specific to a given enterprise platform.

Kindly, note that custom component libraries and service access libraries are not shown in the official J2EE Application Programming Model released by Sun.

J2EE Application Components

Having seen how different J2EE components and services fit in, let us take a closer look at each of them to understand the roles they play in the complex game of enterprise computing.

Web Container Components

Java Servlets

Servlets are server-side, presentation logic components that reside in the Web container. As applets extend the basic functionality of a Web browser, servlets extend the functionality of a Web server—providing programming capabilities and the capability to generate dynamic contents, apart from serving static HTML.

Servlets are capable of receiving the HTTP requests emanating from a client's Web browser, process the input parameters received, and deliver the output as an HTTP response that can then be displayed on the client's browser.

For example, servlets can be used to develop a simple Web-based authentication system, receive the username and password from a client browser, process the request, and send back an authorization success or failure message.

Apart from accepting requests from client browsers, servlets can also respond to calls from other servlets.

Although servlets can generate HTTP response stream to clients by themselves, a better methodology to display the processed results is to use Java server pages.

Java Server Pages

Java server pages (JSPs) are also presentation logic components that reside, along with servlets, in the Web container. But they are slightly different from the latter in their scope and capacity.

JSPs are a cross between HTML and Java; that is, JSPs can contain HTML codes as well as Java codes. Although the HTML codes are directly sent to the client's browser, Java codes are stripped off and interpreted in the server itself. Thus, JSPs are most useful for incorporating intelligence inside HTML tags.

A structured and reusable way of incorporating Java intelligence in JSPs is to use JSP tag libraries. These libraries are easy to develop, and result in cleaner segregation of Java codes from HTML codes.

JSPs complement servlets in the Web container, and are often used to display the results processed by a servlet. It is also typical to find application designs, which use JSPs—in powerful association with Javabeans—without any servlets involved.

JSPs, in combination with Javabeans and/or servlets, can result in compact and yet extensible application designs.

Javabeans

Javabeans are basic data model components that were introduced to the Java community long before the advent of J2EE. In J2EE architecture, Javabeans usually reside in a client tier or presentation logic tier, and usually complement applets or Java server pages.

Javabeans define a certain basic grammar for application codes to be qualified as bean components. For example, a simple Java class with a few variables, and get_attribute() and set_attribute() methods for those variables, can be qualified as a standard Javabean component.

Setting aside the naming similarity, Javabeans are no way related to Enterprise Javabeans (EJBs), which are completely different breeds of enterprise Java application components.

EJB Container Components

Enterprise Javabeans

Enterprise Javabeans (or EJBs) are distributed, scalable business logic components of J2EE middleware. They encapsulate core business logic and data model elements in all enterprise Java projects.

EJBs reside in specialized environments called EJB containers, which are provided by the application server vendor. The J2EE specifications define the relationship (*contracts*) between the application EJBs coded by developers and the container environments.

Developers adhere to certain interfaces as they develop their EJBs. Although certain methods defined in the interface are to be implemented by the developers themselves, the rest are left to the EJB container provider (or the application server vendor). Under the hood, we can understand this grammar as a methodology to segregate business logic implementations (developer-coded methods) from infrastructure provisions (container-implemented methods).

An EJB can talk to any other EJB—either within the same container or in a different container that is hosted in a remote server. They can make use of all enterprise services, as well as *custom component libraries* and *service access libraries*.

EJB specifications define several breeds of EJBs that serve the needs of different application scenarios:

- Stateless session beans
- Stateful session beans
- Entity beans
- Message driven beans

In combination with servlets and Java server pages, Enterprise Javabeans result in highly flexible and reusable MVC-based application architecture.

J2EE Enterprise Services

J2EE enterprise services are used by Web container and EJB container application components to access the resources and services available within the enterprise. The requirements may vary—from a simple database access to a complex mainframe access through a messaging/CICS environment.

The concept behind all J2EE enterprise service APIs is shown in Figure 7.

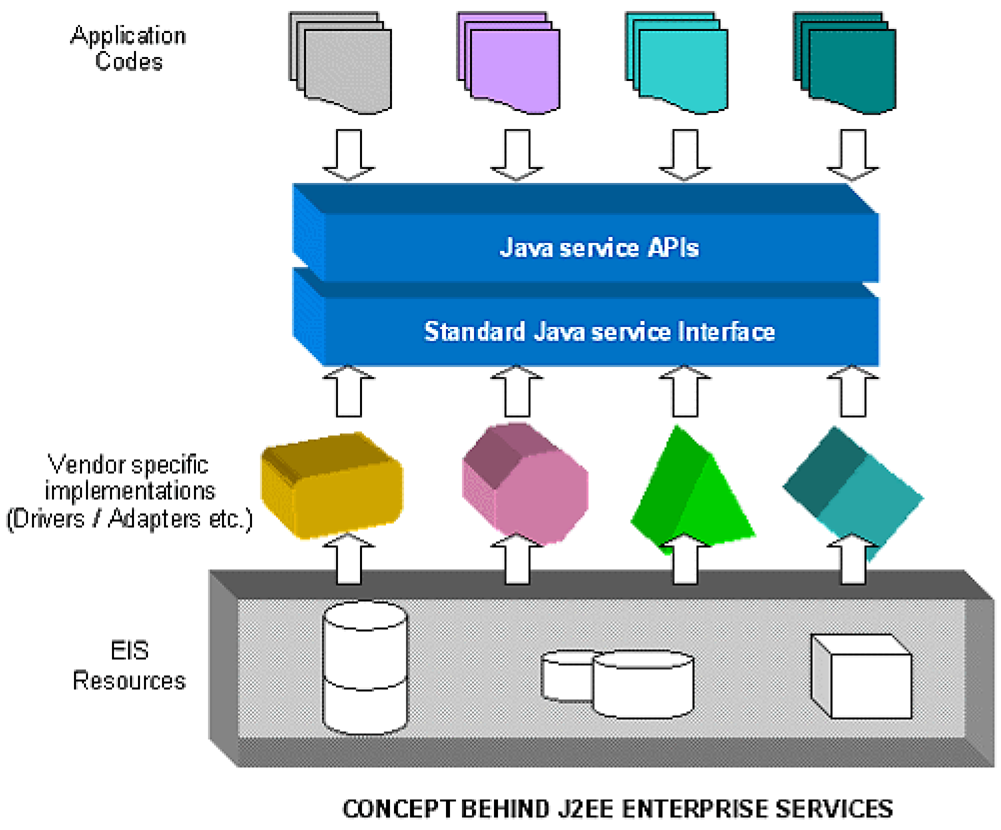
Application Codes
Java service APIs
Standard Java service Interface
Vendor specific implementations (Drivers / Adapters etc.)
EIS Resources
CONCEPT BEHIND J2EE ENTERPRISE SERVICES

Figure 7 Concept behind J2EE enterprise services.

As shown, J2EE service APIs provide a standard Java interface above specific services such as database connectivity and directory access—hiding the implementation-specific details from the eyes of the application. Thus, application programmers are exposed to a standard set of APIs, instead of those very specific to a given resource. This helps them to adopt easily across different service implementations.

For example, from a J2EE application component perspective, the way to access Sybase and Oracle databases is pretty much the same. All we need to change is the underlying JDBC driver that provides connectivity to the specific database; and, of course, to the database URL itself. Thus, with no code-level changes, programmers can access different databases effortlessly.

JDBC

Java database connectivity API helps J2EE components to connect to a wide variety of relational databases.

The JDBC API provides a set of standard interfaces for all common database functions that are used by J2EE components to query and update the database. The specific implementations of these interfaces are taken care of by JDBC drivers.

Thus, the application codes stand independent of database-specific calls, and it is possible to switch over from one database to another with minimal effort.

JNDI

In a distributed environment, naming and directory services enable the location and identification of different resources that are scattered across the network. Resources bind themselves with a unique name in a distributed hierarchical tree structure. Clients obtain a handle to the required resource by accessing the directory service.

Java Naming and Directory Interfaces (JNDI) standardize calls to enterprise naming and directory services. JNDI introduces a service layer above the specific vendor directory services such as LDAP, Novell Directory Service, or Active Directory Service, so that they can be accessed from J2EE components in a consistent manner.

JMS

Messaging services play a vital role in exposing heterogeneous systems, such as mainframes and other legacy systems, to J2EE middleware components. The essential communication link between these disparate systems is established by means of installing messaging service processes on either end. Applications talk to the messaging processes, which in turn transfer the message on a point-to-point or asynchronous methodology across other messaging processes along the network.

Java Messaging Services (JMS) APIs standardize calls to industry messaging services such as IBM's MQ series, Microsoft's MSMQ, or TIBCO's Rendezvous. JMS defines a layer above proprietary messaging API calls, so that J2EE applications stand independent of the actual messaging-service implementations.

Javamail/JAF

As the name indicates, Javamail API is meant for providing e-mail services to J2EE components. It sits over specific mail server implementations such as SMTP and POP3, providing uniform access to all common services that might be required (such as message formatting and sending and receiving e-mails), with or without attachments.

Javamail API works in synchronization with the Javabean Activation Framework (JAF).

RMI—IIOP/Java IDL

These two technologies allow intercommunication between J2EE components and CORBA (Common Object Request Broker Architecture) components.

CORBA is a famous distributed application framework from the Object Management Group that has been in existence for several years.

Although Java IDL is the recommended methodology for accessing all existing CORBA objects from J2EE environment, RMI—IIOP can be used for exposing J2EE components to CORBA clients.

JAX

The Java XML APIs make the usage of XML documents within the J2EE realm easy. The services provided by JAX APIs include XML parsing (SAX and DOM), XML binding to Java objects, XML messaging, and XSLT transformations.

The JAX APIs lie at the foundation of exposing J2EE business logic components as Web services, as well as consuming Web services from a variety of environments.

JCA

Java connector allows the seamless integration of existing enterprise information systems such as ERP, mainframes, transaction-processing environments, databases, and other legacy systems with J2EE middleware. As in the case of other service APIs, JCA provides room for vendors to develop their own resource adopters, which then establish system-level contracts with any J2EE-compatible application server and application-level contracts with the application components that access the EIS resource.

Currently, J2EE-compatible resource adapters for many popular back-end environments have started appearing in the market.

Conclusion

The J2EE standard defines a rich set of components with which complex distributed applications can be built, such as servlets, Java server pages, and Enterprise Javabeans. Each component has a specific and well-defined set of roles, which they can play in a complex distributed system.

J2EE enterprise service APIs interface between specific vendor drivers/adapters and J2EE components. They expose standard API calls over vendor-specific service implementations, so that the application components are independent of vendor-dependent codes.

Services available in J2EE include database connectivity, e-mailing, distributed directory access, XML parsing/binding, and CORBA communications.

J2EE Clients

J2EE clients are those applications, components, systems, and services that access the J2EE environment to fulfill a request or access a service. Depending upon the scope and horizons of a given enterprise J2EE environment, they can range from simple HTTP requests to complex interactions between several J2EE servers across the networks.

By *enterprise J2EE environment*, we mean the enterprise distributed application backbone over which all of the system components are deployed. Because the platform is distributed across several servers and network(s), there could be wide-ranging adaptations of J2EE services by various back office and front office applications within a given enterprise environment.

Types of J2EE Client Access

We can summarize the types of client access in J2EE environment as follows.

- ***Accessing business logic services in EJB:*** This is perhaps the most popular form of client access in any given J2EE environment. If the core business logic rules can be encapsulated into loosely coupled EJBs or Java components, then they can be reused over and over across several applications, within and even outside the enterprise. Sometimes, enterprises set forth far-reaching ambitions in building their J2EE components and services (for example, evolving global business logic repositories or common EIS access gateways). In such cases, business logic modules are carefully designed and crafted under the supervision of experienced architects, so that reusability and encapsulation are best evolved.

  Example: A bank's funds transfer service, which transfers money from one account to another. This service, when implemented as an EJB (or group of EJBs behind a façade), can be reused by different types of clients—such as a bank's back office application, an Internet banking Web site, and mobile banking services; and even by its sister companies, associates, and other subsidiary banks.

- ***Accessing presentation logic services:*** Presentation logic services are very specific to certain types of client front ends; hence, they are hardly reused across different applications. There are certain architectures and designs available, however, by which we can segregate *what is being presented* from *how it is presented*. A typical methodology is to use XML and XSLT pages, which ensure the clear-cut demarcation of presentation logic (XSLT) from data (XML).

  Example: Consider the same funds-transfer application that was described earlier. Let us say that we need to send an acknowledgment and balance account status after the successful execution of the service. The data that is presented across different clients is the same (status of transfer + balance account). In a Web page, it may be presented with an image, but not in the case of a mobile client. In such scenarios, if the presentation elements (images, fonts, etc.) are defined in separate XSLT pages (web.xsl, mobile.xsl, etc.), and the two are combined either at the server end or at the client end, then we achieve significant levels of segregation between presentation *data* and presentation *logic*. Browsers such as Internet Explorer 5 help further by comprehending and combining XSL and XML only during rendering time.

- ***Accessing enterprise services:*** There are situations in which components outside the J2EE realm might wish to access one or more enterprise services available in the platform. This is more pronounced when the enterprise architects build the J2EE environmental services so they are available to a wide variety of applications and components.

  For example, we can have a common gateway over J2EE enterprise e-mail services. Thus, they are available not only to other peers in the J2EE platform, but also to a whole range of enterprise applications that need to send and receive email. Similarly, all database access calls can be centralized inside the J2EE environment. Such arrangements enable exercising greater control and collaboration on the usage of infrastructure services across the enterprise.

- ***Accessing EIS services through J2EE:*** Apart from exposing its own application components to different types of client access, J2EE can also serve as a centralized facade for accessing one or more enterprise information system resources in the back end. This is particularly useful if the back end systems are bulky, complex, or are not directly accessible by many other applications.

  Example: Let us say we have a Siebel CRM system in the back end, and we access the resources through messaging systems/Siebel adapters. It is possible to construct access gateways inside J2EE—by making use of JMS, EJBs, and other Java libraries—so that Siebel resources can be accessed by a wide variety of Java clients and other applications in a consistent manner.

Types of J2EE Clients

Depending upon the scope of infrastructure and design within the enterprise, J2EE services can be availed by a wide variety of client applications, some of which are listed below:

- Web (HTML) clients: Clients that access J2EE services through the Internet HTTP protocols.
- Device clients: Clients that access J2EE services through hand-held devices and mobile phones using J2ME, WAP, and/or other technologies.
- Applet clients: Java applet clients within the browser or applet container that can access directly to J2EE business logic environment.
- Stand-alone Java clients: Stand-alone Java application clients that also enjoy direct access to business logic components such as applets.
- EJB clients from other servers: Business logic components (normal classes as well as EJBs) from other J2EE-compatible servers and containers that can access the EJBs directly in a given J2EE environment.
- CORBA clients: CORBA (Common Object Request Broker Architecture) application clients that reside in CORBA ORB environments and access EJBs through remote IIOP calls.
- Legacy clients: Legacy systems such as ERP/CRM and mainframes that access resources and services within the J2EE environment.
- JMS clients: Clients that access J2EE services through enterprise messaging systems and JMS API.
- Windows clients: Windows application clients such as VB/VC ++.

- Web service clients: Clients that access J2EE resources through vendor-neutral Web service technologies such as SOAP/WSDL and UDDI.
- Clients from other environments: All other types of J2EE client access.

Having seen different types of clients that access J2EE, let us see how J2EE components expose themselves to these clients.

Servicing Web-based HTTP Clients

Web-based clients are most popular with J2EE, and many J2EE projects begin as Web portals (during the initial stages) in any enterprise. We had earlier seen that HTTP Web servers are often bundled with J2EE Web application servers.

Components that expose J2EE to Web/Internet via HTTP protocols include servlets and Java server pages. Although servlets are most suitable for accepting the client requests and invoking the required business logic, JSPs are most suitable for rendering the processed result(d)

Servicing Java Applet Clients

Java applets are much more powerful than normal HTML pages—they can go as far as invoking the business logic beans (EJBs directly). But it is generally not a good practice to make applets very heavy by embedding view and controller logic in it, unless there are specific reasons to do so. This is because as more and more codes are embedded into applets, we might as well be deviating from the basic Model-View-Controller architecture. It also takes more time on the client side to download and execute applet codes in the browser's virtual machine.

It is better to do only client side validations inside the applets and push all other processing codes to the server side components.

Servicing Stand-alone Java Client Applications

Stand-alone Java clients can access all business logic components by making use of Java RMI calls. EJBs internally use RMI over IIOP to communicate between each other, so it is possible to directly execute RMI-based remote procedural calls from stand-alone Java clients.

When EJB access is required, Java clients obtain a handle to the given EJB by making a lookup call through JNDI in the distributed directory service. After the handle is available, Java clients can directly start invoking the business logic codes. Because EJBs are distributed components, it is possible to invoke different EJBs spread across different J2EE server environments in a consistent manner.

Servicing EJB Clients from Other J2EE/EJB Servers

It is possible that EJBs in a given enterprise environment are accessed by other EJBs from another J2EE environment! For example, if a company and its associates decide to embark on a unified middleware platform (J2EE), they may agree to allow accessing of their business logic components across their environments.

Because EJBs have very good authorization and authentication features—including method level security—it is quite possible to exchange EJB services across different environments while maintaining control over the access.

Care should be taken to ensure that only authorized EJBs are accessing a given business logic or a service.

Servicing CORBA clients from CORBA Object Request Brokers

J2EE promotes total interoperability with CORBA components that reside in CORBA Object Request Brokers (ORBs). The Java Interface Definition Language (Java IDL) exposes Enterprise Java beans as yet another CORBA object—to the eyes of CORBA clients. Similarly, when CORBA components need to be accessed from EJBs, RMI–IIOP calls are viable solutions.

Servicing Legacy System Clients (ERP/CRM/Mainframes)

When legacy systems become clients to J2EE environment, the popular methodology adopted is messaging. Hence, we can say that legacy system clients are in turn JMS clients to the J2EE environment.

The Java connector architecture might be useful in case relevant legacy connectors are available from the resource vendor.

Servicing JMS Clients

JMS clients are other applications and components that access J2EE services through one or more JMS-supported messaging systems such as IBM MQ series. Because the need for closer interactions with messaging systems was high within the J2EE realm, a new type of EJBs called message driven beans was introduced.

Message driven beans enable synchronous as well as asynchronous interactions with the JMS-based messaging systems, and can be effectively used to access business logic components inside a given EJB container from other messaging applications.

Servicing Windows/.Net Clients

Prior to the introduction of the .Net framework, there were limited attempts to expose Enterprise Java beans to native Windows application calls directly by means of Java client access services (CAS). But after the advent of .Net, the focus has drifted slightly away from CAS. Now, architects are looking at better ways to interact between the two environments by means of vendor-neutral standards and Web service technologies.

Servicing Web service client

Of late, Web service-related technologies such as SOAP, UDDI, and WSDL are gaining wide popularity and acceptance in the enterprise computing community. Any component or application can express its services as Web services by understanding SOAP calls and publishing its services to a directory such as UDDI.

The Java XML APIs (JAX Package) such as JAXP (Java API for XML Processing), JAXR (Java API for XML Registries such as UDDI), JAXB (Java APIs for XML Binding), JAXM (Java APIs for XML Messaging) and JAX-RPC (Java API for XML-based Remote Procedural Calls) enable total interoperability between the Web services world and J2EE.

Servicing Clients from Other Environments

Apart from all of the above-mentioned types of client access, there are other proprietary forms of J2EE component access. With the advent of neutral technologies such as Web services, however, these vendor-specific forms of client access methodologies are expected to be discouraged in the enterprise computing world, and may well disappear in due course.

Conclusions

Different types of clients access various kinds of services that are available in the J2EE environment. All Java clients are powerful enough to access J2EE business logic components directly, whereas non-Java clients have to go through enterprise messaging systems or other methodologies to access the business logic.

With the advent of .Net and growth of Web services, Windows clients (the most popular client platforms) can access J2EE business logic services without much complexity. This might pave the way for a new age computing world, in which J2EE and .Net coexist peacefully.

Web Application Servers

Web application servers (or simply *application servers*) are distributed middleware software implementations that interface between various types of clients on one end, and back-end system resources on the other end.

When a medium or large-scale enterprise migrates to the world of e-commerce and Web-based transactions, it needs to reorganize its systems in three-tier or multitier application architecture in which Web application servers are the essential middleware components.

Definition

A thematic definition of an application server has been provided by Forrester Research:

"An application server is a software server product that supports thin clients with an integrated suite of distributed computing capabilities. Application servers manage client sessions, host business logic, and connect to back-end computing resources, including data, transactions, and content."

Benefits

Application servers offer several benefits to the enterprise. They provide

- A single robust platform over which all kinds of applications can be built and deployed.
- Support to a wide variety of component models, resulting in compact system designs and code reusability.
- Platform/OS independence (applicable mainly to Java-based application servers).
- Linkage to different breeds of established enterprise information systems—such as mainframes, ERP, databases, and file systems—to distributed application middleware.
- Clustering over a host of hardware, and treating it as a single pool of resources. When one box goes down, the other takes up the additional load, thus ensuring continuous availability of services. This reliable back-end runtime environment is important for all Web-based applications.
- Monitoring services over sensitive database transactions, thus increasing security and control over the existing database applications.
- System resilience and application performance by caching, pooling, allocating resources, and load balancing.
- Efficient remote administration and management capabilities to the system.
- Flexibility and openness in the overall enterprise system architecture because each and every resource can be "plugged in" or "plugged out" of a single backbone.
- A single robust gateway to access all types of EIS resources.
- Application and component scaling to meet the increasing traffic.

Evolution

Historically, Web application servers emerged from different breeds of middleware products and technologies.

During the initial days of the Web, HTTP servers were mostly used to serve static HTML pages to the clients. As the Internet matured, several proprietary technologies—such as CGI/Perl scripts and Coldfusion programs—offered business logic and database-access capabilities to Web servers. The limitations of these technologies, the demanding needs of Web-based transactions, and the overall growth of e-commerce spearheaded several traditional middleware vendors to offer solutions for the Web.

They combined their traditional strengths in areas such as transaction monitors, CORBA Object request brokers and low-level database handling to evolve a new breed of server-side infrastructure solutions called Web application servers. Around this time, Sun released J2EE standards, with a vision of evolving a standard middleware infrastructure and environment from several proprietary technologies.

The new breed of J2EE Web application servers was born.

J2EE Web Application Servers

J2EE application servers are Web application server products that adhere to standards and technologies advocated by Sun's J2EE specifications. They all offer a standardized development and runtime environment for application components.

Codes built on top of J2EE application servers are portable across one another. For example, a servlet developed over BEA's Weblogic server can be deployed in IBM's Websphere server without any code-level changes.

To ensure compatibility of application components across different breeds of servers, Sun has released a J2EE licensing scheme and a J2EE compatibility test suite (CTS). Vendors who have obtained the J2EE license and passed CTS can call their products "J2EE-certified."

Choice of Application Servers

Because it is a lucrative market with a growth potential of several billion dollars, almost all major IT companies have jumped onto the application server bandwagon. These companies include ATG, BEA Systems, Borland Corp, Broad Vision, Brokat, Compaq, Computer Associates, Fujitsu, Hewlett-Packard, Hitachi, IBM, Macromedia, NEC, Nokia, Oracle, SAP, Silver Stream, Sybase, TIBCO Software, WebGain... All of them are J2EE licensees, and many of them have rolled out their very own J2EE-certified application servers to the market.

Combine this list of J2EE licensee companies with other application server companies that are yet to join the licensing scheme, and you end up with a highly crowded market place, with each product trying to carve out its own niche segments.

According to various market survey reports, BEA's Weblogic server and IBM's Websphere dominate the J2EE application server market, followed by other players such as Oracle. Ironically, Sun's own J2EE application server product (called iPlanet, and now renamed as the Sun ONE application server) is yet to make a major impact in the application server arena.

One application server that is not yet J2EE-certified, but is well worth taking a look at is JBoss. This open source application server is making significant inroads into the enterprise environments.

How to Choose Your Application Server

Choosing a particular application server for a given enterprise environment is no easy task. With a rich set of development and runtime features, each application server vendor lures companies to adopt its own product.

In the following sections, we try to identify all major features of application servers that are important for an integrated, Web-based enterprise IT architecture. The relevance and importance of specific features in a given enterprise environment should ultimately dictate the choice of decision-makers.

Essential Features of Application Servers

Scaling

Scaling is the capability of the application server to meet dynamic site traffic and serve the increasing number of client requests.

Ambiguous Web traffic volume is a potential problem for the server administrators. When not handled properly, it may well take the server to its limits, resulting in system crashes.

Load Balancing

Load balancing is the capability to distribute the client requests to different servers under the same cluster. Simple hardware solutions are already available for less-intelligent request routing. In the case of application servers, however, we are talking about a much higher level of load balancing, in which distributed application components instead of client requests are load balanced.

Fault Tolerance

Fault tolerance is the capability to resume operations in spite of component failures, network failures, and server crashes. In production environments, usually several servers are configured to work in orchestration with each other. Thus, when one server in the group fails, the other can take up the additional load.

Transaction Monitoring

Application servers should be capable of doing effective business transactions across multiple server instances and networks. Two-phase commit might be important for certain critical enterprise transactions.

How well the application server monitors and ensures transactions is an important feature to be considered.

Availability of Resources

An application server's resources—such as memory, computing power, components, databases, and other backend systems—should always be available to serve client requests. It is the responsibility of the application server to ensure that no client request gets timed out due to the lack of computing power or components to service the request.

Performance

Performance can be generally thought of as the capability of the application server to attend to the client's request within the earliest possible timeslot. For this to succeed, the server must be capable of processing the transactions between the clients and back-end resources efficiently.

Some application server vendors and independent market watch firms have come out with their own benchmarks (for example, ECPerf) for measuring the performance of the application server. A typical benchmark is "so many thousand requests handled per second" under a given configuration.

Platform/OS Support

Java- and J2EE-based servers naturally support more operating systems and platforms than other kinds of application servers. In enterprise server environments, UNIX flavors and Linux dominate, whereas developer platforms are often Windows-based.

Note that Microsoft's .Net architecture (a direct competitor to Java/J2EE) is available only in MS Windows platforms.

Tools and IDE Support

Many application servers come bundled with development tools and integrated development environments (IDEs) for developing and deploying applications with ease. The quality and ruggedness of the tool, as well as its integration with a given application server, are decisive factors for an application server choice.

Examples include IBM's VisualAge, which comes with the IBM Websphere application server; and WebGain Studio, which is the preferred IDE for BEA's Weblogic.

Development and Runtime Licenses

Certain application servers offer free development licenses, charging only for production runtime license. This option is more attractive to novice developers who want to get started with J2EE application servers.

When the size of the project is large, many development teams will be involved, and development-licensing charges could be significant.

Web Server Support

Quality of the HTTP Web server that comes bundled with the application server, and compliance and compatibility with other popular HTTP Web servers in the market is an important factor to be considered.

Security Management

How secure are the transactions between the client and the server and across the tiers? What are the features of the product that specifically support this? What are the security standards supported by the product? Is there any security framework that is bundled with the application server?

These are all questions that should be asked to get an understanding of the security features supported by a given application server.

Back-End Systems Integration

Application servers link different kinds of back-end resources: mainframes, relational databases, ERP, CRM, and other legacy systems. If there is any built-in support available within the application server for a given back-end environment, then it can save the cost and effort required for systems integration.

Administration/Manageability

Application servers should have user-friendly tools for day-to-day administration activities: resource monitoring, application health monitoring, deployment and replacement of components, and so on.

Although the quality and features supported by these tools vary widely, almost all of them have a visual console for performing basic operations. Most of them are empowered with remote administration capabilities so that system administrators need not be physically present in the server room.

One additional positive feature is the capability of the application server to escalate alerts, if any, to enterprise management tools, so that transactions are not adversely affected.

Site Monitoring/Analysis and Reporting

Some application servers produce standardized weekly and monthly reports on the number of hits, resource usage, etc. These are most useful for statistical purposes and site performance reporting.

Availability of Local Technical Support

Depending upon the technical expertise available within the enterprise, this feature also assumes importance. Expert advice that is available from visiting consultants and e-mail is not as good as having a strong local team to support the products. The emphasis is on the problem-solving capability of the local skill set.

Pricing

Depending upon the budget and resources committed, pricing may become a constraint in adopting the right application server for the enterprise. For experimentation and proof-of-concepts, open source servers are better choices.

Conclusions

Application servers are middleware software solutions available from different vendors. The J2EE platform has brought in standardization and compatibility across several competing products in the market. Currently, many companies are offering J2EE-certified application server products.

There are several features—such as scaling, load balancing, security, and administration—that should be considered when selecting a particular product for the enterprise. More than the quality of a particular feature, it is the relevance and importance of it to the given enterprise computing environment that should dictate the choice.