Chapter 4

Evolution of J2EE

4.1 The Java 2 platform Enterprise Edition

The Java 2 platform Enterprise Edition reduces the cost and complexity of developing multi-tier services, resulting in services that can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures.

To better control and manage these applications, the business functions to support these various users are conducted in the middle tier. The middle tier is typically run on dedicated server hardware and has access to the full services of the enterprise. In the J2EE platform, middle-tier business functions are implemented as Enterprise JavaBean components, as shown in Figure 4.1. These enterprise beans allow service developers to concentrate on the business logic and let the EJB server handle the complexities of delivering a reliable, scalable service.

![Figure 4.1 EJB Components Implement Business Logic in the Middle Tier](Image)

Java Server Pages technology and servlets present middle-tier functions to the client tier as simple-to-access Internet-style services. Java Server Pages (JSP) technology
makes it easy for user interface developers to present dynamically generated pages to anyone with a browser. Servlets give more sophisticated developers of Java technology-based applications the freedom to implement dynamic presentations completely in the Java programming language [27].

4.1.1 J2EE ARCHITECTURE

Figure 4.2 J2EE ARCHITECTURE

J2EE Containers

 Containers are the interface between a component and the low-level platform specific functionality that supports the component. Before a Web, enterprise bean, or application client component can be executed, it must be assembled into a J2EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize
the underlying support provided by the J2EE server, including services such as security, transaction management, Java Naming and Directory. The container also manages no configurable services such as enterprise bean and servlets life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs [28].

**Container Types**

The deployment process installs J2EE application components in the J2EE containers illustrated in Figure 4.2.

**J2EE server**

The runtime portion of a J2EE product. A J2EE server provides EJB and Web containers. It will be explained in the later part of discussion.

**Enterprise JavaBeans (EJB) container**

Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.

**Web container**

Manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.

**Application client container**

Manages the execution of application client components. Application clients and their container run on the client.

**Applet container**

Manages the execution of applets. Consists of a Web browser and Java Plugin running on the client together.

1. **EJB Container**

Enterprise bean instances run within an EJB container. The container is a runtime environment that controls the enterprise beans and provides them with important system-level services. Since you don't have to develop these services yourself, you are free to concentrate on the business methods in the enterprise beans. The container provides the following services to enterprise beans:
- Transaction Management
- Remote Client Connectivity
- Security
- Life Cycle Management
- Database Connection Pooling

**Transaction Management**
When a client invokes a method in an enterprise bean, the container intervenes in order to manage the transaction. Because the container manages the transaction, you do not have to code transaction boundaries in the enterprise bean. The code required to control distributed transactions can be quite complex. Instead of writing and debugging complex code, you simply declare the enterprise bean’s transactional properties in the deployment descriptor file. The container reads the file and handles the enterprise bean’s transactions for you.

**Security**
The container permits only authorized clients to invoke an enterprise bean’s methods. Each client belongs to a particular role, and each role is permitted to invoke certain methods. You declare the roles and the methods they may invoke in the enterprise bean’s deployment descriptor. Because of this declarative approach, you don’t need to code routines that enforce security.

**Remote Client Connectivity**
The container manages the low-level communications between clients and enterprise beans. After an enterprise bean has been created, a client invokes methods on it as if it were in the same virtual machine.

**Life Cycle Management**
An enterprise bean passes through several states during its lifetime. The container creates the enterprise bean, moves it between a pool of available instances and the active state, and finally, removes it. Although the client calls methods to create and remove an enterprise bean, the container performs these tasks behind the scenes.

**Database Connection Pooling**
A database connection is a costly resource. Obtaining a database connection is time consuming and the number of connections may be limited. To alleviate these problems, the container manages a pool of database connections. An enterprise bean can quickly obtain a connection from the pool. After the bean releases the connection, it may be re-used by another bean.

**Enterprise Beans**

Enterprise beans are server components written in the Java programming language. Enterprise beans contain the business logic for your application. For example, a checkbook client might invoke the debit and credit methods of an account enterprise bean. There are two types of enterprise beans: session beans and entity beans.

2. **Web Container**

The Web container is a runtime environment for JSP files and servlets. Although these Web components are an important part of a J2EE application. It manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.

J2EE Web components are either servlets or pages created using JSP technology (JSP pages). Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Servlet clients allow web browsers to indirectly access enterprise beans. Fig. 4.3 illustrates this access [28].

4.1.2 **J2EE Connector Architecture**

The J2EE Connector architecture is the standard architecture for integrating J2EE products and applications with heterogeneous enterprise information systems. The Connector architecture enables an EIS vendor to provide a standard resource adapter
for its enterprise information system. Because a resource adapter conforms to the Connector architecture specification, it can be plugged into any J2EE-compliant application server to provide the underlying infrastructure for integrating with that vendor's EIS. The EIS vendor is assured that its adapter will work with any J2EE-compliant application server. The J2EE application server, because of its support for the Connector architecture, is assured that it can connect to multiple EISs.

![Diagram of Servlet Client](image)

**Figure 4.3 Servlet Client**

The J2EE application server and EIS resource adapter collaborate to keep all system-level mechanisms—transactions, security, connection management—transparent to the application components. This enables an application component developer to focus on a component's business and presentation logic without getting involved in the system-level issues related to EIS integration. Through its contracts, the J2EE Connector architecture establishes a set of programming design guidelines for EIS access. The J2EE Connector architecture defines two types of contracts: system and application level. The system-level contracts exist between a J2EE application server and a resource adapter. An application-level contract exists between an application component and a resource adapter.

The application-level contract defines the client API that an application component uses for EIS access. The Connector architecture does not require that an application...
component use a specific client API. The client API may be the Common Client Interface (CCI), which is an API for accessing multiple heterogeneous EISs, or it may be an API specific to the particular type of resource adapter and its underlying EIS. There are advantages to using CCI, principally that tool vendors can build their tools on top of this API. Although the CCI is targeted primarily towards application development tools and EAI vendors, it is not intended to discourage vendors from using JDBC APIs. An EAI vendor will typically combine JDBC with CCI by using the JDBC API to access relational databases and using CCI to access other EISs. The system-level contracts define a "pluggability" standard between application servers and EISs. By developing components that adhere to these contracts, an application server and an EIS know that connecting is a straight-forward operation of plugging in the resource adapter. The EIS vendor or resource adapter provider implements its side of the system-level contracts in a resource adapter, which is a system library specific to the EIS. The resource adapter is the component that plugs into an application server. Examples of resource adapters include an adapter that connects to an ERP system and one that connects to a mainframe transaction processing system.

There is also an interface between a resource adapter and its particular EIS. This interface is specific to the EIS, and it may be a native interface or some other type of interface. The Connector architecture does not define this interface.

The Connector architecture defines the services that the J2EE-compliant application server must provide. These services—transaction management, security, and connection pooling—are delineated in the three Connector system-level contracts. The application server may implement these services in its own specific way [29].

4.2 Enterprise JavaBeans

The EJB architecture is a server-side component model for the Java platform. Its purpose is to solve the problems of the distributed environment and to enable enterprises to build scalable, secure, multiplatform, business-critical applications as
reusable, server-side and by allowing the enterprise developer to focus only on writing business logic. EJB technology has removed the need to write "plumbing" code. That is, the enterprise developer no longer needs to write code that handles transactional behavior, security, connection pooling, or threading, because the architecture delegates this task to the server vendor [30].

There are several benefits for the users and implementers of this technology:

- **Productivity**: Enterprise developers will be more productive using this technology. Not only do developers get all the productivity gains of developing on the Java platform, but they also get a boost from the fact that they only have to focus on writing their business logic.

![Diagram](image)

**Figure 4.4 Connector Architecture System and Application Contracts**

- **Industry Support**: Customers attempting to build EJB systems will have a range of solutions to choose from.

- **Protection of Investments**: Enterprise JavaBeans technology builds on top of the systems that exist in the enterprise today. In fact, many of the EJB products coming out will be built on top of established enterprise systems. The systems that
exist in enterprises today will be running Enterprise JavaBeans components tomorrow.

- **Architectural Independence**: Enterprise JavaBeans technology insulates the developer from the underlying middleware. The developer sees only the Java platform. In addition to the cross-platform benefits described below, this allows the EJB server vendor the opportunity to improve and change the middleware layer without disturbing a user's EJB applications.

**Enterprise JavaBeans and extensible application servers**

The basic idea behind Enterprise JavaBeans is to provide a framework for components that may be "plugged in" to a server, thereby extending that server's functionality. Enterprise JavaBeans is similar to the original JavaBeans only in that it uses some similar concept [31]. EJB technology is governed not by the *JavaBeans Component Specification*, but by the entirely different (and massive) *Enterprise JavaBeans Specification*. The *EJB Specification* calls out the various players in the EJB client/server system, describes how EJB interoperates with the client and with existing systems, spells out EJB's compatibility with CORBA, and defines the responsibilities for the various components in the system.

**4.2.1 Types of Enterprise Beans**

**Session Bean**
A session bean represents a single client inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server. As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. Like an interactive session, a session bean is not persistent. (That is, its data is
not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.

State Management Modes
There are two types of session beans: stateless and stateful.

Stateless Session Beans
A stateless session bean does not maintain a conversational state for the client. When a client invokes the method of a stateless bean, the bean’s instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Since stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients. At times, the EJB container may write a stateful session bean to secondary storage. However, stateless session beans are never written to secondary storage. Therefore, stateless beans may offer better performance than stateful beans. A stateless session bean can implement a Web service, but other types of enterprise beans cannot.

Stateful Session Beans
The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the conversational state. The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.
4.2.2 EJB Architecture

The figure 4.5 illustrates architecture of EJB technology. The EJB specification allows for any kind of client. This is because the specification does not mandate any remote object "wire" protocol. This means that a server can support multiple protocols like RMI, IIOP (CORBA), and DCOM. This implies that a client to an EJB server does not have to be written in the Java language.

![Figure 4.5 Architecture of EJB](image)

The EJB server is a collection of services for supporting an EJB installation. These services include management of distributed transactions, management of distributed objects and distributed invocations on these objects, and low-level system services. In short, an EJB server manages the resources needed to support EJB components. An EJB server provider can provide an implementation of a container, (described below) and it can provide an API for third party vendors to plug-in additional EJB containers. The EJB specification allows developers a great deal of freedom in the design and implementation of servers.

An EJB container is just that: a home for EJB components. A container is where a Bean lives, just as a record "lives" in a database. It provides a scalable, secure, transactional environment in which Beans can operate. It is the container that handles
the object life cycle, including creating and destroying an object. The container, among other things, also handles the state management of Beans [32].

4.3 eXtensible Markup Language

XML stands for Extensible Markup Language (often written as eXtensible Markup Language to justify the acronym). XML is a set of rules for defining semantic tags that break a document into parts and identify the different parts of the document. It is a meta-markup language that defines a syntax used to define other domain-specific, semantic, structured markup languages [33].

XML Describes Structure and Semantics, Not Formatting. XML markup describes a document’s structure and meaning. It does not describe the formatting of the elements on the page. Formatting can be added to a document with a Style sheet. The document itself only contains tags that say what is in the document, not what the document looks like.

XML and B2B

With XML, financial information can be exchanged over the Internet. Expect to see a lot about XML and B2B (Business To Business) in the near future. XML is the main language for exchanging financial information between businesses over the Internet. Lot of interesting B2B applications are under development. Other clients and applications can access your XML files as data sources, like they are accessing databases.

XML within J2EE

Java and XML work extremely well together. On the one hand, the abundance of free and well-designed tools is what makes XML so successful. It so happens that most of them are written in java: XML tools are the first kind of system software that has always been written in java first, before C or C++. On the other hand, XML gave java a powerful push that finally propelled it into the family of firmly established languages.
They fit together so well because of two main points: both of these are cross-platform 
and network-ready. It is very natural thing to do to have Java classes exchange XML 
data across components and applications over the network. All these possibilities 
make XML a natural for any distributed Java application, including J2EE 
applications.

XML is an essential component in the J2EE platform. J2EE will provide a framework 
for B2B data interchange using XML. Enterprise JavaBeans component architecture 
uses XML to describe its deployment properties, giving Enterprise JavaBeans data 
portability in addition to its code portability.

**XML for Integrating Legacy Data**
A Significant aspect of Enterprise computing is integrating legacy data. This is where 
XML can play an important role, providing a bridge between a multitude of 
proprietary legacy formats and the formats expected by the application. XML is 
 extremely well suited for the role because, once we put our data in one format, 
switching between XML formats is very easy. So, to integrate legacy data, we only 
have to provide one conversion, from legacy format to an appropriate XML format.

Those J2EE servers that Implement Java Messenger Services (JMS) for asynchronous 
message passing between components will mostly likely use XML for defining 
message formats. This will fit well with the scheme for legacy data integration [34].

**The Promise of XML**
The processing of XML documents is assuming growing importance in modern-day 
IT infrastructures. The most well-known use case is in the implementation of web 
services that rely on XML as the underlying data exchange format. Not only are the 
arguments and attachments involved in web service calls transferred as XML 
messages in the SOAP format, all the associated infrastructure describing web 
services starting from the overall description specified in WSDL (Web Services 
Description Language) down to the message types (XML Schema) are expressed in 
XML.
What has grabbed less attention, yet has become equally prevalent, is the use of XML in seemingly more mundane tasks such as specifying the content of industrial-strength web sites, describing system configurations and even in creating office documents. Now that StarOffice, Microsoft Office and the Lotus suite have decided to store their documents natively in XML format, it is quite possible that the killer apps for XML have finally arrived [35].

**The Rhetorical Principles of XML**

Together, the three technologies – XML, XML Schemas, and XSLT – reflect a unified and specific perspective on the organization and presentation of data: they reflect a particular series of rhetorical propositions.

- The global rules for valid communication of information are basic and minimal.
- The majority of rules and definitions for a successful exchange of information are defined within specific contexts.
- In this system of local sovereignty, there are no universally valid communications; instead, information is structured and communicated according to the needs of a very specific context, and is, if necessary, translated from context to context.

XML is a decentralized system, with a minimal set of central rules and a set of tools that facilitate the local definition and translation of structured data. Its widespread adoption suggests that these rules of design were both well-considered and well-executed; it also provides a strong example of the locality of audience and context in systems of communication, and of the benefits of structuring information for a specific context. The success of XML is a testament to the real-world efficacy of a local perspective on structuring and presenting information, and thus reflects key principles we teach in our Rhetoric and Composition classes [36]. XML Schema should be designed in a fashion so that it places no restrictions on the vocabulary that instance documents employ, and which facilitates the growth of data in a highly distributed fashion [37].
4.4 Java Messaging Service

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is loosely coupled. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application’s methods. Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components [38].

JMS API

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. The JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations. It provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain. The JMS API enables communication that is not only loosely coupled but also

- Asynchronous: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
• Reliable: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

**JMS API with J2EE Platform**

The JMS API in the J2EE platform has the following features:

• Application clients, Enterprise JavaBeans (EJB) components, and Web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)

• Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages. A JMS provider can optionally implement concurrent processing of messages by message-driven beans.

• Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS API enhances the J2EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among J2EE components and legacy systems capable of messaging.

### 4.4.1 JMS API Architecture

A JMS application is composed of the following parts.

• A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the J2EE platform includes a JMS provider.

• JMS clients are the programs or components, written in the Java programming language, that produce and consume messages. Any J2EE application component can act as a JMS client.

• Messages are the objects that communicate information between JMS clients.
Administered objects are pre-configured JMS objects created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories.

4.4.1.1 Point-to-Point Messaging Domain

A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

Figure 4.6 JMS API Architecture

Figure 4.7 Point-to-Point Messaging
Characteristics of PTP Messaging:

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.

The receiver acknowledges the successful processing of a message.

4.4.1.2 Publish/Subscribe Messaging Domain

In a publish/subscribe (pub/sub) product or application, client's address messages to a topic. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

Figure 4.8 illustrates pub/sub messaging:

![Pub/sub messaging diagram]

Figure 4.8 Publish/Subscribe Messaging

Pub/sub messaging has the following characteristics.

- Each message can have multiple consumers.
• Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages [28].