EJB™ Architecture
Agenda

- Why EJB™?
  - Technologies leading up to EJB™.
  - Goals of EJB.
- What is EJB™?
  - EJB™ architecture.
- How do you develop, assemble, and deploy enterprise beans?
  - Roles.
- Who supports EJB™?
  - Application server providers, bean developers and assemblers.
- An example scenario.
The Enterprise JavaBeans architecture defines the design for the development and deployment of reusable and portable distributed transactional Java Server Components.

The EJB™ Architecture defines a set of standard vendor-independent interfaces for all Java Application Servers (which provide the framework for EJB™’s).
- EJB™’s allow building of distributed applications by combining components developed using tools from different vendors
- EJB™’s free application developers from having to understand low-level transaction and state management details; multi-threading; resource pooling; and other complex low-level APIs.
Evolution of Distributed Transactional Computing

- Overview of technologies leading up to EJB™
- Mainframe -> 2 tier -> 3 tier
- Distributed protocols
- Distributed objects
- Application servers
The history of computing is punctuated with an oscillation between 2 models of computing, centralized and de-centralized. The computing model from the dawn of the computing age until the late 1970’s was clearly one of centralization. It should be noted that these monolithic software applications were difficult to program and maintain.

Single tier, Monolithic application:
- dumb terminal, mainframe, database with TP monitors. All data access, business, and presentation logic are on one computer.

Advantages:
- relatively easy to manage
- data consistency is simple because data is stored in only one place.

Disadvantages:
- single-tier systems have limited scalability
- Availability depends on one machine (if mainframe is down, whole business goes down).
- No separation made between data access, presentation, and business logic. This doesn't facilitate code re-use because all functionality is mixed together.
- Changing data or business logic may affect every part of the application, making changes (new functionality, bug fixes) of software more difficult.
This changed in the 1980s with the advent of client-server computing, which was brought about by the arrival of the PC. With smarter clients, you could have logic running on the client and nicer GUIs. This era of computing marked a shift to the decentralized model of computing. Also, the task of programming became easier, as PC development environments evolved.

Two Tier, Client Server:
- PC Client, Database Server
- "Fat" Client contains GUI Presentation Logic and Business Logic, and Data Access code,
- Client uses database API's for transactions.

Advantages:
- Nice user interface, powerful gui tools

Disadvantages:
- Data Access logic mixed with Business Logic (on client) makes reuse difficult.
- DB connection for every client process causes Scalability issues, DataBase Server can be bottleneck
- Application and database drivers must be installed and configured on each client.
- Versioning, software distribution and maintenance of "fat" clients to lots of PCs very difficult, any change means re-installing on every client.
Disadvantages:
• Fat clients make maintenance and reuse difficult.
  Update and maintenance costs are high, because changes have to be re-deployed to every client. Any change in the logic must be redistributed to all clients.

• Fat Clients directly access the DB using SQL queries, meaning they are tightly bound to the database schema. If the schema changes the client code must be updated and redeployed to all clients.

• Poor network performance because all data is transferred for processing at the client.

• Database connection costs are high because there is one database connection per user with no connection pooling or multiplexing.
3 Tier and RPC

- Thinner client: business data model separated from UI.
- Business logic in middle tier as “services”.
- Standard communication protocol: RPC (but not object-oriented).
- More difficult to program because programmers need to manage multithreaded concurrency, transactions, security... on server.

3 Tier:
- Middle tier responsible for business logic: business methods run on the server, and the client requests that the server execute these methods. The client and server use a protocol that represents a conversation of business “services”, instead of SQL queries and raw data returned.
- Client: user interface logic
- Database: repository of information

**RPC Remote Procedure Call**: standard, transparent way to call procedures remotely. But not object oriented.

Advantages:
- Client Easier to keep up to date, cheaper client deployment and maintenance.
- Separating Presentation tier, business tier, Database tier, makes it easier to change one layer without affecting other layers.

Disadvantages:
- More difficult to program because server programmers need to manage multithreaded concurrency, transactions, security.
3 Tier using Objects:

- **Corba**: Object Request Broker, kind of like object oriented RPC. Allows Client objects to communicate with remote objects. Interoperability with wide variety of software. IIOP is internet inter-orb protocol.

- **RMI**: Java Object request broker. Also like object oriented RPC. Simpler to use than Corba, only for Java to Java, but now there is RMI over IIOP allowing java to non-java.

- **DCOM**: allows binary modules to communicate remotely, like a binary RPC

Advantages:
Same as with the previous slide, with additional re-use possibility provided by object-oriented design model.

DisAdvantages:
Distributed communication, with multiplexing, connection pooling, concurrency, transactions, multithreading is difficult to program!
The early 1990’s saw the popularization of the Internet. In terms of enterprise computing, this had 2 effects, which would seem to be in opposition to each other. First, the Internet allowed “internetworking”, which broke down the geographical barriers of the enterprise LAN. This in turn led to a move toward “browser-based” clients, like Mosaic and its commercial cousin, Netscape. The Internet and the browser made distributed computing accessible to everyone.

The second effect the Internet had on enterprise computing was to reverse the trend of decentralization. This happened because of the emergence of HTTP, HTML and the Java™ platform as standards for client-based computing. This created a huge number of “thin” clients that could be counted on to have the same set of functionality. Because of this defined, but limited, set of functionality, this created the trend of centralizing business logic on the server. The server in this context was no longer a mainframe of mini-computer, but any machine running a web server.

The problem in this model was that there were a myriad of programming and connectivity models. **CGI, NSAPI, and ISAPI** to name a few. Also, the web server tended to mitigate access between the browser-based client and each individual enterprise applications, causing a **programming and maintenance nightmare**.

Advantages:
- Web makes applications available everywhere to any desktop with a browser

Disadvantages:
- Mission critical transaction-oriented applications can not be done easily and maintain-ably with CGI and Perl.
Components

Monolithic:
• 1 binary file
• Recompiled & linked for any changes

Components:
• plug-gable Parts.
• Better design,
• better re-use,
• easier updates
• Implementation separated from Interface, only Interface published

Monolithic application:
• composed of 1 binary file.
• Any change means recompiling and redeploying the application. This makes it difficult to maintain because the requirements and environment usually change with time and updates are then needed.

Component Application:
• software components can be changed or updated without recompiling and replacing the entire application, like hardware components can be changed. Simplifies deployment of updates. Upgrades and bug fixes are easier to make.
• application is separated into logical separate plug-gable parts like presentation parts, business process parts, business logic…
• Component models provide a standard interface and mechanism for the components to interact with each other. Component programming separates the interface from the implementation, meaning that a component’s implementation can be changed without changing the interface or other components using the interface.
• Components are pre-developed pieces of application code that can be assembled to create an application. By only publishing the interface, application assemblers do not have to understand how a component is coded, just how to use the interface, in order to use a component.
• dividing an application into components improves the application’s design. Component programming forces developers to define the application in terms of well structured objects, which have well defined interfaces so that they can interact properly with each other.
• business logic can be reused, multiple instances of the same component can be used in multiple applications.
• Updates are easier because you can change a part without changing the whole.
• Development is easier because it allows you to test and build small parts incrementally (spiral development model), you can also divide development into smaller parts potentially developed by different people.
Lessons Learned:

- **Improvements from 1 tier to n-tier:**
  - One tier can be changed without changing the rest.
  - Lower deployment and maintenance costs.
  - Resources (like connections) can be pooled and re-used.
  - Flexible, scalable, more performant for large # users.
  - Thin clients can be made available on internet on browsers on any desktop.

- **But still a problem with this:**
  - Requires developer to know complex details of distributed protocols, concurrency, transactions, object distribution, load balancing, security, resource management!

- Advantages of n-tier computing:
  - One tier can be changed without changing the rest
  - Lower deployment and maintenance costs
  - Resources (like connections) can be pooled and re-used.
  - More flexible, scalable, and performant
  - Thin clients can be made available on internet on browsers on any desktop

- Distributed n-tier computing has many advantages but it also makes programming to handle multi-user sessions more complex.

- Thin-client multi-tiered applications are hard to write because they involve a lot of complex code to handle transaction management, multi-threading, database connection and resource pooling, performance issues...

- Developing distributed applications is difficult and requires highly skilled and experienced people.
More recently, application servers have allowed the aggregation of enterprise services into a single web-facing display. This had the effect of simplifying the programming process when dealing with multiple enterprise applications.

**Application Servers and Platform Lock**

The problem with this model is that there is a minimum of 25 application servers on the market and each of them (as of late 1997) had a programming model that was product specific. This has the unfortunate effect of locking the application builder into one specific product or platform. This is a damper on the pace of innovation of eCommerce. Applications written to one of these servers will be locked into the server architecture, and will be unable to take advantage of feature and performance enhancements found in the rest of the industry by virtue of this “lock-in.”

The only solution to this problem would be to rewrite the application to a new application server with the desired new features or performance characteristics. This would be an expensive proposition, since an entirely new application model would have to be developed.

**Application Servers with components and TP:**

- With application servers the programmer can focus on business logic by developing components which are deployed in the application server.
- Application Server provides transaction management, multi-threading, database connection and resource pooling, performance…
- **BUT Problem:** until recently there was no standard on how those components had to be developed, and what the API was between the components and the application server.
- **Not Portable:** components developed for one application server couldn't be deployed in another one without a major re-write. Developer was locked to Application Server Vendor.
Why EJB™?

- best of distributed components and application servers:
- Simplifies the complexity of a building n-tier applications.
- Standardizes an API for components and application servers.
- Provides a framework for portable components.

EJB™ specifies a standard component model for application servers. Divide the building of a distributed application into different tasks. EJB™ servers provide services that in the past had to be developed by the application programmer.

Take the best aspect of distributed components and TP application servers:

Simplifies the complexity of a building n-tier component based TP applications.

Standardizes an API for components and application servers. (Standard contract between application server container and EJB™ components).

Provides a framework in which reusable, portable transaction oriented business logic components can be created without programming the infrastructure.
Why EJB™?

- Write your business, not
  - thread/connection pooling
  - threading
  - life cycle
  - management
  - load balance / scalability

EJB™ provides the framework
Taken in this context, the solution to the previous problems would be a single, unified platform that would enable eCommerce in a distributed computing environment across a wide array of competing products and would have a single easy-to-use model for building applications. To be more specific, the solution is the Java 2 platform, Enterprise Edition™.

Java 2 platform, Enterprise Edition (J2EE™) is a platform that enables solutions for developing, deploying and managing eCommerce applications.

As a platform, J2EE™ extends Java computing into the enterprise-class arena. This platform is the natural evolution of the Java 2 platform, Standard Edition. J2EE extends a company's existing Java investment to address the needs of server-centric IT development. As an open standard, the J2EE platform ensures open competition for Java development to avoid proprietary server platform lock.
Prior to J2EE™ there were a bunch of enterprise java apis. This platform weaves them together into a cohesive unifying platform.

Enterprise Java APIs provide programming interfaces for diverse middleware implementations, such as naming, security, transactions, messaging and database.

**Enterprise JavaBeans (EJB™) are the cornerstone of these APIs.**

J2EE™ provides a component-based, server-centric multi-tier application architecture.

By defining the J2EE™ platform, developers are:

- ensured consistent, integrated runtime environment
- which enables application portability and interoperability.
The J2EE™ runtime environment is made up of components and containers. Components are pieces of application code that can be assembled into working applications. They run within an environment called a container.

**Containers** provide a J2SE™ runtime environment, and provide the implementations of J2EE™ APIs for services like transactions and messaging. Examples of J2EE™ tools:

- create components (IDE, HTML editors, content editors)
- application assembly tool
- deployment tools for deployment
- management tools of J2EE™ applications.

**Connectors**: standard way to extend J2EE services to other non-J2EE™ application systems, such as mainframe systems and ERP systems. Will be defined in a future release.

**APM**: Application Programming Model, models an e-commerce application architecture using J2EE™ technology.
J2EE™ Deliverables

- **J2EE™ Reference Implementation**: an SDK operational framework providing all of the J2EE™ APIs
  - helps developers prototype with the J2EE™ technology
  - Provides vendors with a standard for comparing their implementations
- **J2EE™ Specification**: Defines a standard for implementing the J2EE™ platform
- **J2EE™ Compatibility Test Suite**: Test suite for testing J2EE™ compliance of vendor products.
- **J2EE™ APM**: describes and implements a sample application showing “best practices” for developing and deploying a thin client J2EE™ e-commerce application.
Let’s view in more detailed diagram, the relationship of components and containers.

Developers are responsible for the implementation of only the components, shown in green.

The containers provide runtime support for the components. Eg: container is the car and component is the driver.

The containers are responsible for providing the enterprise APIs, shown in gold, and the distributed communication protocols, shown in brown.

3 categories of components:

- components that are deployed, managed & executed on a J2EE™ server: JSP™, servlets™ & EJB™
- components that are deployed & managed on a J2EE™ server but are loaded & executed on a client machine: applet
- components whose deployment and management are not completely defined in this spec: application client

Platform specifies a container for each component type. Each container type must support a set of standard services as defined in the spec.
The container is the car
The component is the driver

The container is the platform
The component is your application
EJB™ Provides:

• Easier development
• open industry standard support
• Multi-tiered architecture
• Scalability
• Distributed transaction support
• Portable cross platform component model, Platform independence

EJB™ Provides
- EJB™ Components provide reusable Data and Business Logic
- EJB™ Servers provide transactions and security support
- n-tier Architecture provides scalability and performance
- component container contract provides cross platform portability
Multi-tier Architecture: Client - EJB™ Application Server – Database

Client:
• Can be GUI written in any language, Servlets, JSP. Clients can access EJBs using either RMI, Corba or DCOM bridge

Bean:
• Responsible for the "business logic". Enterprise JavaBeans execute within an EJB™ container, which in turn executes within an EJB™ server

Containers:
• The container provides a portability layer that lets anyone’s beans operate in anyone else’s container.
• The container intercepts all calls between the client and the bean. This gives it the opportunity to implement services for the enterprise bean.
• Provides the environment in which the beans run (Actually combination of classes/interfaces). The EJB™ container provides services such as transaction coordination, resource management, life cycle maintenance, persistence, and security role verification… to the EJB™ components it contains.

Application Server:
• Provides EJB™ Infrastructure, provides underlying system services required by EJB™ container and manages resources.

Layers of Services: The Application Server provides services to the container, which in turn provides services to the bean.
**EJB™ Server:**

provides **EJB™ Infrastructure**, provides underlying system services required by Container and EJB.

*Provides a **Container Runtime Environment**.

**Containers:**

provides the **environment in which the beans run** (Actually combination of classes/interfaces).

*transparently provides **services by intercepting all method calls to the EJB™ component**.

*provides management and control services:

  • Naming services: **JNDI™ registration** of EJB™ when loaded
  • Lifecycle, state mgmt: **creates and removes bean** instances, Instance pooling, **activation and passivation**
  • security checks - performs authentication & access control. Declarative & programmatic security.
  • serializing method calls
  • resource pooling
  • **Transaction coordination**: declarative transaction mgmt
  • **Persistence management**: storing of beans
  • bean runtime context information (meta data)

The container **simplifies nontrivial aspects of a distributed application** such as security, transaction coordination, and data persistence. The container and service providers implement the EJB™ infrastructure. This infrastructure deals with distribution aspects, transaction management, and security aspects of an application. The EJB™ specification defines the Java APIs for the infrastructure.

Developers focus on their area of expertise: the business logic and leave the infrastructure to the platform specialists.
For each enterprise bean, the container is responsible for:

- registering the object.
- providing a remote interface for the object.
- creating and destroying object instances.
- checking security for the object.
- managing the active state for the object (activation/passivation).
- coordinating distributed transactions.
- Persisting CMP entity beans, and saving stateful session beans attributes when passivated.

When a client calls an EJB™’s remote interface, it actually connects to the container. The container verifies that the call conforms to the semantics specified in the deployment descriptor before dispatching the call to the EJB™ itself. This mechanism enables the container to control all aspects of an EJB™’s execution, including:

- **Lifecycle.** The ejb-jar file includes the home interface of the EJB. This interface includes one or more create and one or more destroy methods. The container is responsible for implementing this interface so that a client may create or destroy instances of an EJB. The reason for having the container as opposed to the EJB™ object implement this interface is so the container may enforce security, transaction, and persistence requirements when executing them. The container is also responsible for putting the home interface in the naming or directory service so that it is accessible from JNDI.

- **Security.** The deployment descriptor object contains declarations about the access control for an EJB’s methods. When a client attempts to call the method of an EJB, the container is responsible for looking up the user's permission in an access control list (ACL) to verify he has the right to invoke that method.

- **Transactions.** The deployment descriptor object contains declarations about the transaction restrictions on an EJB’s methods. When a client attempts to call the method of an EJB, the container is responsible for verifying that the call obeys the transaction restrictions on that method. These restrictions include: must not execute within a transaction context, may execute within a transaction context, must execute within a transaction context, and must execute within a new transaction context. The container is responsible for issuing the appropriate calls to the transaction service so that, if appropriate, an EJB™ participates in the transaction.

- **Persistence.** An Entity Bean may choose either bean-managed or container-managed persistence. In the case of bean-managed persistence, the bean implements its own data access code. In the case of container-managed persistence, the bean delegates data access to the container. For EJ Bs under container-managed persistence, the container is obviously responsible for implementing the appropriate data access functions.
What features must a server offer?

Transaction Management

Connection Pooling

Security

Load Balancing

EJB™ Servers

Servers are primarily resource managers in the EJB™ architecture. Each server manages the allocation of resources to the containers it controls. At the coarsest level of granularity, this resource management includes determining which containers and how many of each may run within the server. At a finer level of granularity, this resource management occurs at three levels. At the client level, the server manages incoming client requests. This management includes providing connection for clients, dispatching client requests to containers, and routing responses back to clients. At the container level, the server manages the processing resources available to the container. This management includes allocating available worker processes and threads to running containers. At the service level, the server manages container access to shared services. Such services may include a centralized security manager, a transaction service implementation, a pool of JDBC drivers, a global cache manager, and an asynchronous messaging service.

An EJB™ application may run on more than one server. So in addition to allocating resources to their managed containers, servers must cooperate as a group to allocate resources efficiently across all the servers in a cluster. This cooperation involves publishing load information to a load balancing process that then assigns client requests to the least-loaded server that can provide the requested services. It also includes system management infrastructure such as demons that automatically start server processes, alerts to a management console when server load exceeds a certain level, and management interfaces for controlling the resource allocation policies of the server.
Open Standard Solution:

- Java™ is platform independent
- EJB™ components are portable between EJB™ application servers (API or contract between container and component)
- Communication protocol is RMI or RMI over IIOP, integrates with CORBA, different clients
- Portable across databases (JDBC™)
- Portable across assembly tools (XML)

components are portable between application servers.
Communication protocol is RMI or RMI over IIOP, integrates with CORBA, different clients
• The Bean Remote Interface provides the client-side view of the Bean: It specifies all the methods or public interfaces of the Bean that clients can call.

• The Home Interface specifies methods to create an actual instance of a Bean.

• EJBHome object plays role of “factory” object design pattern. Provides methods to Create, remove and find enterprise beans.

• EJBOBJECT “proxy” object design pattern: The Client does not have direct access to the EJB. The container provides an EJBOBJECT as mediator or “proxy” between the client, the container, and the EJB™ Instance. The EJBOBJECT allows the container to intercept calls to provide transaction management, security, persistence, or activation/passivation related tasks. EJBOBJECT is kind of like “Remote Control” for the bean (like VCR remote control).

• The Bean is the actual code that implements the business logic of the bean.

• EJBContext is implemented by the container and provides the bean with information about the client, the container, and the bean itself. The bean can use this information while processing requests.

• The Deployment Descriptor: is a xml file which specifies Bean properties, this allows the bean provider, application assembler and the deployer to set attributes for security, transaction, and environment variables.
Enterprise Java Beans
Fundamentals

EJB™'s Three types:

• An object that represents a *stateless service*.

• An object that represents a *conversational session* with a particular client. Such session objects automatically maintain their conversational state across multiple client-invoked methods.

• A *persistent entity object* that is shared among multiple clients.
- **View** - The UI view involves the **presentation layout** (the actual HTML output), for the results of the interaction. For HTML clients, this can be done using JSP™.

- **Controller** - presentation logic (the logical connection between the user and EJB™ services), necessary to process the HTTP request, **control the business logic**, and select an appropriate response. This can be provided by Java Servlets or Java Beans which call EJB’s for the business logic, and then call JSP’s to display the result.

- **Business logic (Model)** - the business logic (the core business transactions and processing), necessary to accomplish the goal of the interaction (calculations, queries, and/or updates to a database, etc). Business logic should be separated from UI logic. This makes it possible to have different clients (for example browser and GUI) using the same beans. Separating the 2 also makes the business components easier to re-use, maintain, and update or change.
  
  - **Session Beans** represent the **business rules or process** part of Business Logic. Session Beans manage information relating to a conversation between the client and the server.
  
  - **Entity Beans** represent the **business data** (stored in database) part of Business Logic. Entity beans represent and manipulate persistent application domain data.

Separating an application into layers like this allows you to change one layer without changing the other layers. For example you can change the business rules without changing the business data, or change the UI without changing the backend.
Session Beans

Represent business rules or process.

- Session beans are process, workflow or control objects
- 2 types of session EJBs:
  - **stateful** - maintains client specific state (E.G. ShoppingCart).
  - **Stateless** - provides services without storing client specific information
- Session beans are often a client of multiple entity beans: control workflow, process, tasks between entity beans.

**Session Beans:**

Session beans are **process or control objects** specific to a particular client. Responsible for **controlling workflow, managing processes** or tasks, managing activities (make reservation, purchase...) **coordinate processes** between entity beans. Control interactions of objects, transient not stored in database. Do not represent data in database, but they can access it. Life span is less than or equal to the client.

- Executes on behalf of a single client
- can be transaction aware
- can update shared data in an underlying database but does not represent data that should be stored in a database
- Is relatively short-lived (life typically is that of its client)
- Is removed when the EJB™ server crashes

**Stateless**

- execute a request and return a result without saving any state information

**Stateful**

- extension of client. Performs tasks for client and maintain state on behalf of client.
Entity Beans

**Represents business model data.**

- Represent **business domain objects** often expressed as **nouns**: customer, order, product.
- Maps business data in the database to a java class.
- Model permanent data: gives in memory view and manipulation of data.

Entity Beans

- **persistent** object: wraps "object data" stored in a database and provides operations to manipulate this data.
- **shared** among multiple clients.
- Model **permanent business “entities” or concepts** that can be expressed as nouns (i.e. customer inventory item)
  - usually persistent records in database
- If the App Server crashes entity beans can be re-instantiated from the database entity data.
## Session vs. Entity Bean

<table>
<thead>
<tr>
<th></th>
<th>Session Bean</th>
<th>Entity Bean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Represents Process. Performs a task, or process for a client.</td>
<td>Represents Data: represents a business entity object that exists in persistent storage.</td>
</tr>
<tr>
<td><strong>Shared Access</strong></td>
<td>One instance per client</td>
<td>Shared instance for multiple clients</td>
</tr>
<tr>
<td><strong>Persistence</strong></td>
<td>Not persistent. When the client terminates its session bean is no longer available.</td>
<td>Persistent. Even when the EJB™ container terminates, the entity state remains in a database.</td>
</tr>
</tbody>
</table>
If an entity bean represents a row in a database then the relationship between entity bean instance and row is 1-to-1. Since multiple clients may access this row, entity beans are shared among multiple clients.

Stateful Session beans can maintain per-client state information because the relationship between the client and the session instance is always 1-to-1. This is why Session Beans can manage information relating to a conversation between the client and the server.
Example Scenario: Use Case

Use Case: ATM customer transfers money from checking to savings account
Example scenario: Classes

ATM
  transfer()

Account
  withdraw()
  deposit()

Checking Account
Savings Account

0...*

accesses

0...*
Example Scenario: Sequence Diagram

1: transfer()
2: debit()
3: credit()
Example scenario: EJB™

Client → ATM Session bean → Account Entity Bean

Transfer:
- debit

Account Entity Bean
- account1

Account Entity Bean
- account2

Transfer:
- credit
EJBs are packaged in a Jar file along with the EJBHome (“factory”) interface, the EJBOjbject (“proxy”) interface and the XML deployment descriptor.
Enterprise Bean Provider

The EJB™ provider is typically a domain expert who develops enterprise beans. The Bean Provider is the producer of enterprise beans. His or her output is an ejb-jar file that contains one or more enterprise bean(s).

The Bean Provider is responsible for:
- the Java classes that implement the enterprise bean’s business methods;
- the definition of the bean’s remote and home interfaces;
- and the bean’s deployment descriptor. The deployment descriptor includes the structural information (e.g. the name of the enterprise bean class) of the enterprise bean and declares all the enterprise bean’s external dependencies (e.g. the names and types of resources that the enterprise bean uses).

The Bean Provider develops reusable enterprise beans which implement business tasks or business entities. The Bean Provider is not required to be an expert at system-level programming. Therefore, the Bean Provider usually does not program transactions, concurrency, security, distribution, or other services into the enterprise Beans. The Bean Provider relies on the EJB™ Container for these services.

component creation:
- create components (EJBs) and EJBHome and EJBOBJECT interfaces using tools like IDEs
- create an XML component deployment descriptor
- package components into EJB™ jar
**Application Assembler**

The application assembler is a domain expert who composes applications by combining existing beans.

The Application Assembler combines enterprise beans into larger deployable application units.

The **input** to the Application Assembler:

- one or more **ejb-jar** files produced by the Bean Provider(s). The Application Assembler works with the enterprise Bean’s **deployment descriptor** and the **enterprise Bean’s client-view contract**. Although the Assembler must be familiar with the functionality provided by the enterprise Beans’ remote and home interfaces, he or she does not need to have any knowledge of the enterprise Beans’ implementation.

The Application Assembler **outputs**:

- one or more **application ejb-jar** files that contain the enterprise beans along with their assembly instructions. The Application Assembler has inserted the **application assembly instruction into the deployment descriptors**.

- The Application Assembler can also **combine** enterprise beans with web components (e.g. Java ServerPages™ or Servlets) when composing an application.

**Assembling an application:**

- take a set of components & assemble them into complete application.
- Reconcile deployment parameters:
  - link internal dependencies of all components, such as EJB™ names
  - synchronize security role-names across components in the application
- create an XML application deployment descriptor
  - must be named application.xml
- package the application as an Enterprise Archive format (.ear)

- This step will typically be done via wizards and tools, and developers will usually not be editing the XML deployment descriptors by hand
Deployer

- The deployer is an expert at a specific operational environment and is responsible for deployment of a EJB™ application in its container.
- The Deployer takes one or more ejb-jar files produced by a Bean Provider or Application Assembler and deploys the enterprise beans contained in the ejb-jar files in a specific operational environment. The operational environment includes a specific EJB™ Server and Container.
- The Deployer must resolve all the external environment dependencies declared by the Bean Provider, and must follow the application assembly instructions defined by the Application Assembler.
- The Deployer uses tools supplied by the EJB™ Container Provider to perform the deployment tasks.
- The Deployer’s output is enterprise beans (or an assembled application that includes enterprise beans) that have been customized for the target operational environment, and that are deployed in a specific EJB™ Container.

Deployment:

- installation: copy all material to server
- configuration: resolve all external dependencies:
- execution: start up new application

Configure environment:

- resolve external references
  - to EJBs
  - to resource factories, like databases
  - to URLs
- set application config parameters

assign security:

- map security roles to user groups and accounts that exist in the enterprise
- set up authorization
- choose authentication mechanisms
- provide authentication data for resource access
**EJB™ Server Provider**

The EJB™ Server Provider is a specialist in the area of **distributed transaction management**, **distributed objects**, and other lower-level system-level services. A typical EJB™ Server Provider is an OS vendor, middleware vendor, or database vendor. The current EJB™ architecture assumes that the EJB™ Server Provider and the EJB™ Container Provider roles are the same vendor. Therefore, it does not define any interface requirements for the EJB™ Server Provider.

**EJB™ Container Provider**

The EJB™ Container Provider (Container Provider for short) provides

- The **deployment tools** necessary for the deployment of enterprise beans.
- The runtime support for the deployed enterprise beans’ instances.

The focus of a Container Provider is on the development of a scalable, secure, transaction-enabled container that is integrated with an EJB™ Server. The Container Provider **insulates the enterprise Bean from the specifics of an underlying EJB™ Server** by providing a **simple, standard API between the enterprise Bean and the container**. This API is the Enterprise JavaBeans component contract.
Lifecycle Illustration

Creation
- J2EE Modules
  - Created by Component Developer

Assembly
- Assembled and Augmented by Application Assembler
  - J2EE APP

Deployment
- Processed by Deployer
  - Deploy
  - J2EE Container
    - Enterprise Components
Declarative Programming

- Instead of programming functionality in the Bean:
  - The bean provider/assembler/deployer specifies bean properties for transaction, security, persistence... in the deployment descriptor.
  - The container's deployment tools generate EJBHome EJBOBJECT classes and code to provide the specified functionality.
  - Container/EJBOBJECT intercepts all calls to the bean at runtime
    - Provides transactional, security, persistence behavior
    - Then delegates the method to the object

Declarative Programming simplifies the development process, and makes the components Portable, Scalable, Customizable.

The deployment descriptor allows the bean provider to specify bean properties which can be set and modified using deployment tools without changing code, recompiling... These settings guide the EJB™ container for managing and controlling the enterprise bean. This allows “declarative programming”, meaning modifications can be made without programming.

Allowing the user to specify the Bean’s transactional and security attributes declaratively simplifies the process of building transactional applications and permits better portability.

The deployment descriptor contains the following information:

- Environment properties
- Transaction attributes
- Security attributes: access control and identification
- EJB™ class names
- Ejbhome, ejbobject interface names
- Type of enterprise bean, session or entity
Bean Provider Example

Hertz Bean Provider
Creates Car Reservation EJB(s)
Bean Provider Example

Delta Bean Provider
Creates flight Reservation EJB(s)
Application Assembler

AAA
Application Assembler
Creates Travel agency
Application

Travel Agency
Session, JSPs

Car
Reservation
EJB

Flight
Reservation
EJB
Deployment in a Target Container

- Client
- Travel Agency
- EJB™ Container
  - Car Reservation
  - Flight Reservation
- Deployment Descriptor
- Car Database
- Flight Database
### Broad Industry Adoption

- BEA/Weblogic
- Bluestone
- Bull
- Forte
- Fujitsu
- Gemstone
- Haht Software
- IBM
- Information Builders
- Informix
- Inline
- Inprise
- Iona
- Luna
- NetBeans
- NetDynamics
- Netscape
- Novasoft
- Novell
- Novera
- ObjectShare
- Oracle
- Persistence
- Progress
- SCT
- Secant
- Siemens
- SilverStream
- Sun Microsystems
- Sybase
- Symantec
- Tendril
- TradeeX
- Valto
- Versant
- Vision
Enterprise Deployment

- Air Canada
- Celera Genomics
- Countrywide
- Covad Communications
- Electric Boat
- FAA
- FedEx
- GE Aircraft Engines
- GTE
- Impresse Corporation
- Instinet (Reuters)
- NationsBank
- Qwest Telecommunications
- Raytheon
- Rorke Data
- Scottish Equitable
- Sparks.com
- Trip.com
- University of Ottawa Heart Institute
- Visa International
EJB™ Links & References:

O’REILLY: Enterprise JavaBeans By Richard Monson-Haefel
http://www.oreilly.com

Wiley: Mastering Enterprise JavaBeans By Ed Roman
Prentice Hall: Enterprise JavaBeans by Example
http://www.prentice-hall.com/ptrbooks/ptr_0130224758.html
Programming with Enterprise JavaBeans, JTS, and OTS
by Andreas Vogel, Madhavan Rangarao
Enterprise JavaBeans: Developing Component-Based Distributed Applications by Thomas C. Valesky http://adams.patriot.net/~tvalesky.ejb.html

Java EJB™ site:
http://java.sun.com/ejb
DEV-x javaBeans:
http://www.javabeans-zone.com/

Special Interest Group - Enterprise JavaBeans
http://www.mgm-edv.de/ejbsig/ejbsig.html

Links to EJB™ articles
http://www.ejbnow.com

Cetus Distributed Objects & Components: Enterprise JavaBeans
http://www.rheinneckar.de/~cetus/oo_javabeans.html
http://www.ejbportal.com/
JavaWorld

© Copyright 2000 Sun Microsystems, Inc., All rights reserved.
**EJB™ Characteristics Summary**

- **Contained & managed at runtime** by an EJB™ container within an EJB™ server
- **Customized at deployment time** using deployment descriptor
- **Declarative properties** (deployment descriptors) separated from compiled code
- **Client only accesses EJB™ via container/server**
- **Portable** across EJB™ servers, client view (interface) is unaffected
- **Can be assembled without source code**

**Portable across EJB™ Application servers**

- **Contained & managed at runtime** by an EJB™ container within an EJB™ server
- **Bean** provides the **business logic**
- **Simplifies the complexity of** a building n-tier applications.
- **Scalable, distributable, performant, easier to update and maintain**
First Exercise

• Think about how an online store could be designed using session and entity EJBs.
Example Solution: Online Shopping

1. Register or Login to online Customer account
2. Browse Catalog
3. Add item to Shopping cart
4. Purchase/Order items in Shopping cart