

***FSA Modernization Partner***  
**United States Department of Education**  
**Federal Student Aid**



# **Integrated Technical Architecture Component Design Guidelines**

***Task Order #69***

Deliverable # 69.1.4

**Version 2.0**

June 27, 2002



Table of Contents

- 1 Executive Summary .....3**
  - 1.1 Purpose ..... 3
- 2 Implementing Distributed Components.....3**
  - 2.1 Reasons to Use EJBs ..... 4
    - 2.1.1 Access to a chatty resource..... 4
    - 2.1.2 Encapsulation of a specialized resource ..... 5
    - 2.1.3 Multi-channel support..... 5
    - 2.1.4 Maintenance and configuration management..... 5
    - 2.1.5 Large memory footprint..... 5
    - 2.1.6 Declarative transaction support ..... 6
  - 2.2 Reasons Not to Use EJBs ..... 6
- 3 Types of Enterprise Java Beans.....7**
  - 3.1 Session Beans..... 7
    - 3.1.1 Stateless Session Beans..... 7
    - 3.1.2 Stateful Session Beans ..... 8
  - 3.2 Entity Beans..... 8
    - 3.2.1 Subtypes of Entity Beans ..... 9
  - 3.3 Transaction Demarcation ..... 9
    - 3.3.1 CMTD Transaction Modes..... 10
    - 3.3.2 BMTD Transaction Mode ..... 10
- 4 General Guidelines..... 12**
  - 4.1 Do Not Model All Components as EJBs..... 12
  - 4.2 Avoid Stateful Session Beans..... 12
  - 4.3 Do Not Access Entity Beans Directly from Client Code ..... 12
  - 4.4 Use Entity Beans Only as a Persistence Mechanism ..... 13
  - 4.5 Use Session Beans to Enforce Business Rules ..... 13
  - 4.6 Pass Business Objects by Value..... 14
  - 4.7 Illustration of Guidelines 4.3 through 4.6 ..... 14
  - 4.8 Return Value Objects (Data Beans) from Entity EJBs..... 15
  - 4.9 Use Container-Managed Transaction Demarcation, with Appropriate Demarcation Modes ..... 16
  - 4.10 Use the EJBHelper Framework ..... 16
- 5 Entity Bean Usage Patterns..... 17**
  - 5.1 Related Objects With Reads And Updates ..... 17
  - 5.2 Object or Group of Objects Mapping to a Relational Join ..... 17
  - 5.3 Read-Only Objects ..... 18
  - 5.4 Write-Only Objects..... 18
  - 5.5 Retrieving and Scrolling through a List of Objects ..... 19
  - 5.6 Optimistic Locking..... 19
- 6 Performance Checklist..... 21**
- 7 References ..... 22**

## 1 Executive Summary

The component design guidelines is a collection of the best practices that have been identified by the Integrated Technical Architecture (ITA) Release 3.0 team for use by FSA application designers with advanced object-oriented background. These guidelines are assembled recommendations conceived by ITA team members as a function of their in-depth experience with E-Commerce and its implementation within Java-based application servers.

In assisting with production releases of previous FSA web applications, the ITA team determined that it was necessary to collect best practice white papers on the design and development of distributed components that run within the FSA ITA environment. This enables FSA to benefit from previous work performed on other web-based applications.

Areas discussed in this document include the appropriate use of Enterprise Java Beans (EJB), the difference between Session and Entity EJBs, EJB usage guidelines, and EJB usage patterns. These guidelines help FSA design and code their applications to best use of Java object and EJBs, and to ensure quality, scalable web sites.

This document will enable technical architects and designers to create Java-based solutions within the FSA ITA environment that are more robust and scalable. The document helps application teams reduce cost through the solution of performance issues and bottlenecks.

### 1.1 Purpose

Although Java offers significant advantages for distributed-application development, it does not transparently handle the system-level complexities associated with multi-tier distributed applications.

When modeling business components, there are a number of decisions to be made. Tradeoffs between complexity (both of development and deployment), the distribution of components (local classes versus EJBs), scalability, and performance must be taken into consideration. In this document, we discuss the balance of these tradeoffs, focusing on development productivity while bearing in mind performance and scalability.

This document is meant for application designers with advanced Java and object-oriented skills, who will refer to this document for architecting their solutions.

## 2 Implementing Distributed Components

Presently, Enterprise Java Beans are a popular way to build distributed components. EJB encompass many capabilities which facilitate the development of distributed applications in Java.

However, the use of EJBs introduces potential development complexity and productivity issues. With EJBs, there is a risk that a project team will spend significant effort dealing with training, methodology, testing, configuration, deployment, and other issues associated with EJBs, detracting from a focus on developing business logic and delivering business value.

When performance is crucial, the appropriateness of EJBs must be carefully assessed. Remote calls involve network round-trips and the serialization (marshaling) / deserialization (unmarshaling) of

parameter data introduce overhead and can result in degraded performance. Even if EJB(s) are accessed from the same JVM as the client module, some marshaling/unmarshaling overhead will be incurred.

Distributed components are not required by every application. Often, developing Web applications with locally linked classes in lieu of distributed components is the correct design approach (provided that good object-oriented design principles are applied). This is especially true for smaller projects and performance-sensitive situations.

The *golden rule* of distributed components: Unless there is a real need for distributed objects/components, don't use them.

## 2.1 Reasons to Use EJBs

There are, nonetheless, several reasons for using distributed components, as discussed below. These reasons need to be weighed against the additional complexity and potential performance issues associated with EJBs. All but the last point apply to distributed component technologies in general, while the last one is more specific to EJBs (and technologies such as COM+ which have comparable capabilities).

**The Atomic, Consistent, Isolated, Durable (ACID) test for the use of distributed components.** Use EJB for logic which cannot be conveniently co-located with all its clients.

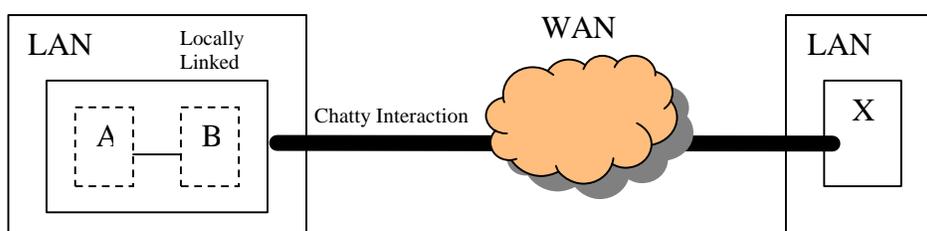
### 2.1.1 Access to a chatty resource

Consider the following scenario:

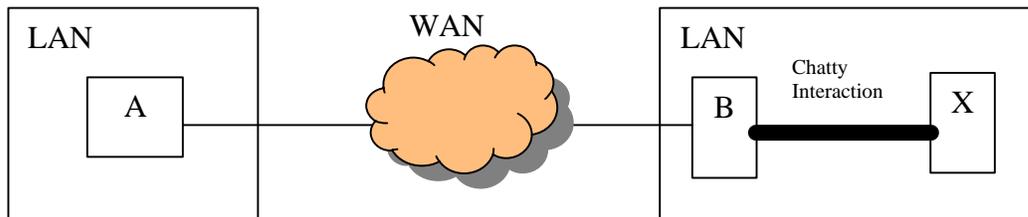
- Component A (e.g., a servlet) calls Component B (e.g., a business logic class or component), which accesses Resource X (e.g., a legacy system or a database)
- Resource X is not co-located with Component A (e.g., they are located in separate data centers, connected by a wide-area network)
- The interactions of Component A with Component B are much less frequent and/or involve much less bandwidth than the interactions between Component B and Resource X. This would be the case, for example, if a user request to a servlet (Component A) triggers a single call to a business component (Component B), which then results in multiple calls to a legacy system (Resource X).

Under this scenario, there are two basic ways to locate components A, B, and X in relation to each other, as depicted below:

Alternative 1:



Alternative 2:



In such a case, the second alternative is preferable from a performance perspective. It makes sense to place Component B close to Resource X and have Component A access Component B remotely.

### 2.1.2 Encapsulation of a specialized resource

If an application needs to access a specialized resource that is not co-located with it, it makes sense to encapsulate the resource with a distributed component. Since remote access needs to be provided anyway, the simplest solution is typically to leverage the component architecture's standard distributed access mechanisms (as opposed to potentially having to deal with lower level protocols).

### 2.1.3 Multi-channel support

If the same business logic needs to be accessed by multiple clients of different types (e.g., servlets, Java applications, Voice Response Unit), it is usually more practical to deploy the business logic as a distributed component than to link the business logic locally into each possible client instance.

### 2.1.4 Maintenance and configuration management

Deploying common business logic as a distributed component simplifies application maintenance and configuration management (including change control and software distribution), as all changes to the common component only need to be applied to one place (or a small number of places) where the component is deployed.

### 2.1.5 Large memory footprint

Components with a large memory footprint (especially code footprint) can be deployed separately from their clients to alleviate memory resource usage.



### 2.1.6 Declarative transaction support

EJB's declarative transaction support contributes clarity to the identification of logical units of work and reduces development complexity related to transaction management.

## 2.2 Reasons Not to Use EJBs

An often-made (but not often valid) argument put forth for the use of EJBs is the security, transaction, and load-balancing services provided by the EJB. These services are also provided by the servlet containers in the major J2EE application servers and are equally as effective. Thus, the decision of whether to use EJBs must be made on a different basis (i.e., for one or more of the reasons discussed in the previous section).

EJB's most touted capabilities is their support for distributed transactions. In practice, this capability has proven to be of limited value. Pragmatic considerations such as manageability and performance often lead to architectures where there is no place for the two-phase commit distributed transaction style supported by EJB.

Resource pooling capabilities: If an application has a resource pooling need then this reason alone should not allow developers to use EJBs. Almost all the major J2EE server vendors allow applications to get connection pooling and thread pooling whether it contains Servlets/JSPs or EJB components.

Clean separation of business logic and presentation logic: EJB enforces a separation of presentation logic (Servlets and JSPs) from business logic (EJB components). However, the developers can achieve the same results with Java classes instead of EJB components as well. The application architect needs to enforce design and coding best-practices about the proper usage of Java classes as a business layer façade.

Performance: There are known performance issues due to incorrect modeling and implementation of EJBs. Modeling fine-grained business domain object as Entity EJBs can hinder performance dramatically.

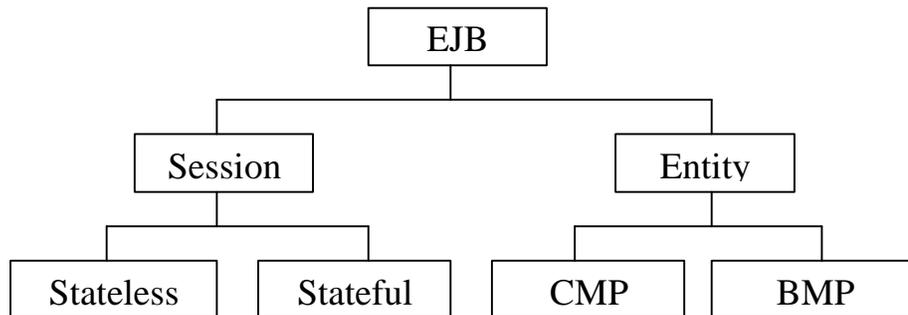
Increased development time: EJB development time in comparison to straight Java class development, not to other distributed environments. EJBs do take longer to develop, and when things go wrong, they can prove more difficult to debug, particularly because the bug might not be in your code but in the application server/container itself.

Added complexity compared to straight Java classes: At a superficial level, three classes are required for every session bean, four for an entity bean, and the developers may additionally employ value objects to reduce network overhead, adding another class.

Continual specification revisions: As EJB technology matures, loose ends in earlier versions are tied up and new features added. Application designers and developers need to move with changes in technology. This potentially can have an impact on code changes.

### 3 Types of Enterprise Java Beans

Enterprise Java Beans come in two fundamentally different types: Session Beans and Entity Beans. The EJB hierarchy is shown in the diagram below:



#### 3.1 Session Beans

Session beans model business functions, processes, or activities, and can be used to control processing flow within an application. For example, “process payment”, “check credit”, or “create reservation”. Session beans may access other EJBs, including entity beans (see below) or other session beans. Session beans keep business logic off the client tier where it might cause performance and maintenance challenges typical of fat-client architectures.

Session beans are transient. Information in a session bean is not automatically stored to a database and is lost in the event of a server crash.

There are two types of session beans: Stateless and Stateful. The two types are explained further in the sections below.

##### 3.1.1 Stateless Session Beans

Stateless session beans are designed to represent pure services. They are anonymous in that they contain no user-specific data. In fact, the EJB architecture provides ways for a single stateless session bean to serve the needs of many clients. This means that all instances of a stateless session bean are equivalent. The term stateless means that it does not have any state information for a specific client. However, stateless session beans can have non-client specific state, for example, an open database connection.

Following are typical uses of stateless session beans:

- Modeling reusable service objects

A business object that provides some set of services to its clients can be modeled as a stateless session bean. Such an object does not need to maintain any client specific state information, so the same bean instance can be reused to service multiple clients (at different points in time). For example, it would be appropriate to model a business object that validates an employee ID against a database as a stateless session bean (or as a method on a stateless session bean). See guideline 4.5 below.

- Operating on multiple entity objects or database rows at a time

An example would be a (read-only) service that retrieves product catalog information. Another example would be a (transactional) service which transfers funds between two bank accounts. This point is a special case of the use mentioned above.

- Providing high performance

Stateless session beans can be more efficient than other EJB types, as they require fewer system resources by the virtue of not being tied to any one client. Stateless session beans typically scale better than stateful session beans. However, in certain situations, this benefit may be offset by the increased complexity of the client application that uses the stateless session beans because the client has to perform the state management functions. (This latter point is normally not an issue with servlet clients due to the Servlet API's session management support).

### 3.1.2 Stateful Session Beans

A stateful session bean contains conversational state on behalf of the client. Stateful session beans do not directly represent data in a persistent data store, but they can access and update data on behalf of the client. As its name suggests, the lifetime of a stateful session bean is subordinate to that of its client.

Following are typical uses of stateful session beans:

- Maintaining client-specific state

Business objects representing client-state-dependent business logic can be modeled as stateful session beans. A shopping cart is a good example of this.

Since stateful session bean instances are tied to a client, system resources held by stateful session beans cannot be shared among multiple clients. As noted earlier, client-specific state management is of limited value to servlet clients due to the Servlet API's session management support.

- Representing work flow

Business objects that represent workflow, managing the interaction of other business objects in a system, are candidates to be modeled as stateful session beans. This is a special case of maintaining client-specific state; the same caveats apply.

## 3.2 Entity Beans

Entity beans model persistent objects in the business domain. For example, “customer”, “order”, “invoice”, and “product” could all be entity beans.

Entity beans are an object representation of persistent data that resides in a database. An entity bean is not lost if the EJB server crashes since it is stored in a database. A simpler way of conceptualizing entity beans is that each entity bean corresponds to a row in a database table, and when clients wish to interact with the data in the table, requests are made of the associated entity bean. This simplified

model facilitates understanding but is not completely accurate since some objects may be stored in more than one table, and since the entity bean can include not only the data but also the behaviors (i.e., methods) associated with the business object.

Since an entity bean approximately correlates to a row in a database table, each entity bean must have an associated *primary key*. This is used by the bean's *finder* method(s) to locate a particular row in the database in response to a client request.

As an entity bean instance can be shared across multiple clients, the EJB server is responsible for concurrency control. The server-provided concurrency control is largely transparent to the developer.

### 3.2.1 Subtypes of Entity Beans

There are two basic subtypes of Entity EJBs: Container-Managed Persistence (CMP) Entity EJBs and Bean-Managed Persistence (BMP) Entity EJBs. CMP entities are those whose persistence (for example, the storing and retrieving of their data from a database) is entirely managed by the EJB container. This means that the container would, for instance, manage both generating and executing SQL code to read and write to the database. On the other hand, Bean-Managed Persistence (BMP) EJBs leave the management of such details as what SQL is executed to the developer of the EJB. Each BMP EJB is responsible for storing and retrieving its own state from a backing store in response to "hook" methods (like *ejbLoad()* and *ejbStore()*) that are called on it at appropriate times during its lifecycle.

### 3.3 Transaction Demarcation

A transaction is an atomic unit of work which may contain multiple operations. When a transaction is completed, all of its operations are either committed or rolled back as a whole. EJB servers monitor access to recoverable resources (e.g., databases) made from EJBs to ensure that a transaction either completely succeeds or is completely rolled back.

The transactional characteristics of EJBs can be controlled declaratively using configuration parameters in the deployment descriptor. (Of course, developers still need to decide and define the nature of the business transactions). This declarative approach allows developers to focus on the business logic instead of system calls required to ensure transactional integrity. This approach to transactional control is called Container-managed Transaction Demarcation (*CMTD*). Both session beans and entity beans can use CMTD.

The EJB specification also allows for session beans to control their own transactions using explicit transaction management. This approach is available for situations where server-managed transactions will not suffice. Explicit transaction control is performed using the Java Transaction API (JTA). This approach to transactional control is called Bean-managed Transaction Demarcation (BMTD). Only session beans may use this demarcation approach.

A third type of transaction demarcation is Client-managed Transaction Demarcation. Under this approach, the client code is responsible for making JTA calls to begin and commit/abort transactions. J2EE application servers must support client-managed transactions for Web (servlet and JSP clients), but not for (non-Web) application clients. For Web applications not using EJBs, client-managed transactions are the only alternative. However, for Web applications using EJBs, it is recommended

that client-managed transactions not be used to avoid clashes with container/bean-managed transactions and confusing transaction semantics.

### 3.3.1 CMTD Transaction Modes

Five container-managed transaction demarcation modes are supported. These modes are specified, in the deployment descriptor, for an EJB or its methods. Each of a CMTD bean's methods is able to implement a different transaction mode or all of the methods in the class can be assigned the same mode with one setting.

- TX\_NOT\_SUPPORTED

The method will not support any transactions and it cannot be used from within a transaction. If a thread calls this method within a transaction, it will remain suspended until the bean completes its operation. Once the bean's operation is complete the client transaction is resumed.

- TX\_REQUIRED

The method must be run within a transaction. If the calling thread is already in a transaction, it is allowed to execute this method. If a thread attempts to enter this method without a transaction, the container will begin a new transaction and terminate it when the method completes.

- TX\_SUPPORTS

If a client is executing a transaction, the method will use the client's transaction. If the client thread does not have a transaction, none will be created by the container.

- TX\_REQUIRES\_NEW

Every time a thread executes this method a new transaction will begin regardless of whether or not the thread is part of a client transaction. If a transaction is under way, the container temporarily suspends it until this new transaction finishes.

- TX\_MANDATORY

Requires that the client have a transaction running before the method is executed. If no transaction is provided, the container will not create a transaction but it will throw an exception.

### 3.3.2 BMTD Transaction Mode

There is only one mode for bean-managed transaction demarcation:



- TX\_BEAN\_MANAGED

The code manages its own transactions by making “begin” and “commit” calls according to the Java Transaction API. A RemoteException is thrown if a thread already associated with a transaction attempts to enter this method.

## 4 General Guidelines

### 4.1 Do Not Model All Components as EJBs

Kyle Brown [Brown 00a] offers the following advice:

Since enterprise beans are remote objects that consume a significant amount of system resources and network bandwidth, it is not appropriate to model all business objects as enterprise beans. Treat an entity EJB as an "OO view" of a database. The key here is that if the entity beans are data gateways (pure data sources), then they have a better chance of being shared between multiple applications. Only the business objects that need to be accessed directly by a client may need to be enterprise beans.

See Section 2 Reasons to Use EJBs for a discussion of situations where the use of distributed components is appropriate and the following sections for additional guidelines on the use of EJBs.

### 4.2 Avoid Stateful Session Beans

The choice between using stateless or stateful session beans can significantly affect system scalability and performance. In certain special situations (e.g., a VRU client), stateful session beans can make an application easier to code since the beans maintain their own conversational state. However, because they hold state, they must be mapped one to one with clients. Thus, if an application has 1,000 concurrent clients, the application must manage 1,000 stateful session beans. As a result, as the number of concurrent clients increases, it would be difficult for the EJB container to manage and maintain acceptable performance. To help with this problem, stateful beans have a complex lifecycle that includes the possibility of the passivation and activation by the container. Passivation and activation are costly, the bean's state information must be committed into and recovered from the database, and the bean must be reinstantiated and populated upon activation.

Stateless session beans shift complexity to the client: If an application must save conversational state, the client code must perform state management functions. For example, the application might save state information into the database or into HTTP session.

The important advantage of stateless beans is that they are effectively anonymous (i.e., any two instances of the same bean are identical) and can be shared among many clients. The basic approach used by EJB servers is to create a pool of the stateless beans. For each method invocation, the EJB container selects a bean from the pool and then returns it to the pool for another client to use. Using this scheme, 1,000 concurrent clients might be served by a few dozen beans. A stateless architecture based on stateless session beans typically has better scalability and performance characteristics than one using stateful session beans.

### 4.3 Do Not Access Entity Beans Directly from Client Code

There are three main reasons for this.

1. Since entity EJBs are remote objects, it is expensive for a client to make multiple method calls on an entity bean to read and/or manipulate the bean's attributes through getter/setter methods.

2. Since entity beans are persistent objects, allowing client code to change the state of entity beans directly may contribute to a violation of business rules and consequent database corruption. It is usually substantially more difficult to ensure the integrity of many client modules than that of a relatively smaller number of server modules.
3. The client process logic view (not to be confused with presentation view) of information may differ from the business modeling perspective which tends to drive the persistent information model. The persistent view of data tends to be more detailed than any single client view. On the other hand, there can be many different client views of the same business object. Tying the client view too closely to the persistent model can sometimes lead to detrimental change impacts. It may be desirable to establish an intermediate layer to insulate the client views from the persistent view.

#### **4.4 Use Entity Beans Only as a Persistence Mechanism**

As recommended by Kyle Brown [Brown 00a], an entity EJB should be treated as an "OO view" of data stored in a database. This is a corollary of the discussion under the previous guideline.

Note, however, that entity beans are not always necessarily the most appropriate means to support persistence. Often, the direct use of JDBC from within session beans (see guideline 4.5 below) to create business data beans (see guideline 4.6 below) is the simplest and most effective way to access persistent data from a J2EE application.

#### **4.5 Use Session Beans to Enforce Business Rules**

The enforcement of business rules, especially those impacting persistent data, should be a primary concern of any application architecture. There are basically three approaches to the problem of enforcing persistent data business rules:

1. Responsibility for business rule enforcement is scattered throughout the application code.
2. A relatively small number of reusable transactional services is responsible for business rule enforcement.
3. The database is responsible for business rule enforcement (e.g., through stored procedures).

Approach 1 tends to result in applications with business rule duplication and inconsistencies.

Approaches 2 and 3 support the effective enforcement and reuse of business rules within and across applications.

Approach 2 is a natural one for J2EE applications. Either locally linked classes or session beans can be used to encapsulate transactions. However, the simplest approach overall is the use of stateless session beans with container-managed transaction demarcation. Approach 2 does not preclude the leveraging of stored-procedures (approach 3) when convenient.

This guideline also provides benefits in terms of a cleaner separation of responsibilities between client and server side development. Server-side developers concentrate on the design and implementation of reusable transactional services, allowing client side developers focus on business process and presentation issues.

### 4.6 Pass Business Objects by Value

Given the previous three guidelines, how can client code access business entity objects?

The answer is to use data beans, serializable objects that encapsulate business entities (their data and immediate validation rules). These objects are used as parameters and return values for session bean methods, and they can be mapped to/from session EJBs for database access. These business data beans (which are not EJBs) should constitute much of the "model" according to the Model-View-Controller (or "Model 2") architecture applied to J2EE applications (see [Sun 00]).

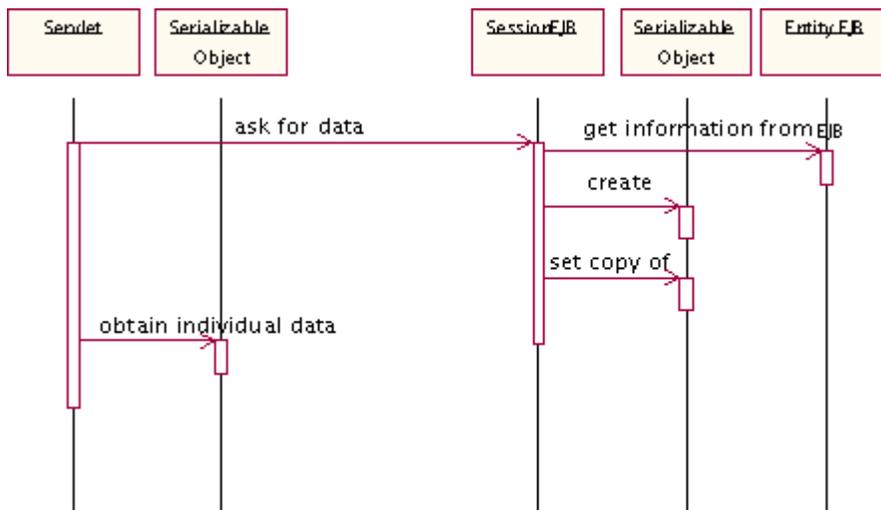
Referring back to point number 3 under guideline 4.3, it may be appropriate to define different business data beans which represent different client process views on the same business object.

Adhering to this guideline, in conjunction with guidelines 4.3 and 4.5, can result in significant performance benefits as multiple remote calls to multiple remote objects (entity EJBs) are replaced with one remote call to one remote object (a session EJB).

### 4.7 Illustration of Guidelines 4.3 through 4.6

The following diagram and explanatory text from [Brown 00a] (see also [IBM 00]) illustrate guidelines 4.3 through 4.6. Note, however, that guideline 4.5 in the present document is stronger than Brown's "EJB Facade" recommendation:

Simply put, our solution is to use a pass-by-value approach to obtain information from our EJBs rather than a pass-by-reference (proxy) approach. We ask a special EJB (which we call a Session Facade) for a serializable java object (which we call a data bean) that contains all the information necessary to display an entire business result or perform a business operation. The data bean is the repository for "business logic" like validation and calculations that do not need to be persistent, and whose results are unique to each instance. Likewise, when we update information contained in an EJB, we send that information as a data bean to the Session Facade, which determines which EJBs to update, so that all updates can happen within the context of a single EJB transaction.



In the sequence diagram above, a servlet is asking a session EJB for a data bean. The session EJB gets the information that makes up the data bean from one or more entity EJBs. It then creates a serializable object (a data bean) and copies the information into the data bean. Finally, the data bean is returned as the result of the message sent to the session EJB. On the client side, the servlet obtains the individual data items from the data bean and displays them to the user, by either embedding the bean directly in a JSP, or by some other mechanism. All of the calls made from the servlet to the data bean are local calls – only the single call made from the servlet to the session EJB is a network call. This avoids the additional overhead of the all-EJB solution, and usually results in a faster overall system, even though the total number of Java methods invoked is greater.

#### 4.8 Return Value Objects (Data Beans) from Entity EJBs

If an entity EJB is used for persistence, then it should implement a pair of getter/setter methods that map the entity bean to/from a value object (data bean) containing all of its state. This facilitates the implementation of guideline 4.6 Pass Business Objects by Value above and minimizes the number of method calls session beans need to make to retrieve the entity bean's state (see guidelines 4.5 Use Session Beans to Enforce Business Rules and 4.7 Illustration of Guidelines 4.3 through 4.6 above).

The following example adapted from [Roman 00] illustrates the point:

```
public class ProductInfo implements java.io.Serializable
{
    protected String m_SKU;
    protected String m_description;

    public String getSKU() {return m_SKU;}
    public void setSKU(String sku) {m_SKU = sku;}
    public String getDescription() {return m_description;}
    public void setDescription(String des) {m_description = desc;}
}

public class Product extends ProductInfo implements
    javax.ejb.EntityBean
{
    // All fields are inherited from the value object!

    public ProductInfo getProductInfo()
    {
        ProductInfo info = new ProductInfo();
        info.setSKU(m_SKU);
        info.setDescription(m_description);
        return info;
    }

    public void setProductInfo(ProductInfo info)
    {
        m_SKU = info.getSKU();
        m_description = info.getDescription();
    }
}
```

Note: While the entity bean in the example above implements (by inheritance) getter and setter methods for all its attributes, that is not necessary. In fact, Ed Roman's original example uses public attributes without getter/setter methods.

#### **4.9 Use Container-Managed Transaction Demarcation, with Appropriate Demarcation Modes**

Declarative, container-managed transaction demarcation should be used wherever possible to simplify coding and avoid errors.

Combining this guideline with guideline 4.5 Use Session Beans to Enforce Business Rules, session beans, using CMTD, should be used to group multiple related entity bean methods into one transaction, representing a single unit of work.

Normally, only the container-managed transaction demarcation modes TX\_REQUIRED, TX\_MANDATORY, and TX\_SUPPORTS should be used. These modes should be used as follows:

- For session beans / methods representing logical units of work, use TX\_REQUIRED
- For session/entity beans / methods which write persistent data but do not represent logical units of work, use TX\_MANDATORY
- For session/entity beans / methods which do not write persistent data, use TX\_SUPPORTS

#### **4.10 Use the EJBHelper Framework**

Before invoking an EJB's business method, a client must create or find an EJB object for that bean. To create or find an instance of a bean's EJB object, the client must make multiple API calls to:

- Locate and obtain an EJB home object for that bean.
- Use the EJB home object to create or find an instance of the bean's EJB object.

The EJBHelper encapsulates these multiple calls to lookup an EJB by logical name. The benefits of using this framework are the insulation of EJB clients from unnecessary complexity and the reduction of duplication of EJB lookup code across the application logic.

## 5 Entity Bean Usage Patterns

The following usage patterns can be useful if it is decided that the use of entity EJBs is the appropriate persistence approach for an application (see guidelines 4.3 Do Not Access Entity Beans Directly from Client Code and 4.4 Use Entity Beans Only as a Persistence Mechanism above). Some of these usage patterns assume the use of VisualAge for Java and/or WebSphere.

Most of these usage patterns are adapted from [Brown 00b], which includes additional details.

### 5.1 Related Objects With Reads And Updates

#### ***Problem Context***

This is a common persistence scenario: an application performs reads and updates to the database with a complex data model. The development environment is assumed to be VisualAge for Java and the application server is WebSphere.

#### ***Recommendation***

Use Container-Managed Persistence (CMP) entity EJBs.

#### ***Discussion***

Visual Age for Java provides significant development productivity support for the use of CMP and WebSphere AE v 3.5.3 has been optimized for the use of CMP. This is particularly the case for 1-1 and 1-N composition relationships. Therefore, CMP should be the default approach when developing entity beans.

### 5.2 Object or Group of Objects Mapping to a Relational Join

#### ***Problem Context***

An object or group of objects maps to a relational join, and performance is a significant consideration (which it often is).

#### ***Recommendation***

Use bean-managed persistence (BMP) entity EJBs.

#### ***Discussion***

Using the default CMP approach will result in multiple SQL statements in this situation. This will occur, for example, if Customer and Address information are kept in two related tables and separate Customer and Address entity beans are defined. An approach which provides better performance is to define an entity bean which maps to the join and retrieve all the related information with a single SQL statement by using bean-managed persistence (BMP). The BMP approach is particularly effective in the case of N-ary relationships involving multiple tables joined by a relationship table.

### 5.3 Read-Only Objects

#### ***Problem Context***

A set of objects is frequently read but rarely updated.

#### ***Recommendation***

Use a stateless session bean that accesses the database directly (without use of entity beans), returns data beans, and caches them.

#### ***Discussion***

As read-only objects do not benefit from the transactional and distributed services inherent in entity beans, modeling read-only objects as entity EJBs incurs unnecessary overhead (internally, each entity EJB instance corresponds to multiple instances of multiple classes). Therefore, a more effective approach would be to avoid entity beans altogether.

The recommended solution involves the creation of a stateless session bean that is responsible for the following:

- Access the database (e.g., using JDBC directly) to retrieve the read-only object states
- Instantiate the read-only objects as data beans (see guidelines 4.6 Pass Business Objects by Value and 4.8 Return Value Objects (Data Beans) from Entity EJBs above). This can be done during application start-up or in a "lazy", just-in-time fashion
- Cache the read-only objects
- Provide one or more methods to return read-only objects by value to clients which need to access them

This solution should save a large number of database calls.

### 5.4 Write-Only Objects

#### ***Problem Context***

Data needs to be saved in a database, but it does not need to be read or updated by the application (e.g., a transaction audit log).

#### ***Recommendation***

Use a stateless session bean that inserts rows directly into the database without using entity beans.

#### ***Discussion***

Just like read-only objects, write-only objects are not subject to contention and do not benefit from the transactional and distributed services inherent in entity beans. Therefore, for similar reasons, an effective approach should avoid entity beans altogether.

The recommended solution involves the creation of a stateless session bean which provides one or more methods, taking data beans as parameters, which access the database (e.g., using JDBC directly) to insert records.

## 5.5 Retrieving and Scrolling through a List of Objects

### ***Problem Context***

Display a list of data in order to let a user select an object from that list. (Caution: this is not a recommended practice for very large list sizes; in such cases, alternative user interface techniques should be considered).

### ***Recommendation***

Use a session bean to return just the subset of information to be displayed to the user. Only instantiate an entity bean for the selected object, not for all the items in the list.

### ***Discussion***

The information that needs to be included in the list displayed to the user is usually a small subset of the attributes of the constituent objects. Therefore, it is wasteful to instantiate all the entity beans corresponding to the list. It is more effective to use direct database access to obtain a result set containing only the information required for the list display, as well as the keys of the corresponding objects. The information required for the list display can be packaged as a hash table or a list of simple data beans (in the case of multi-column lists). Once the user selects an item from the list, the corresponding key can be used to instantiate the corresponding entity bean.

## 5.6 Optimistic Locking

### ***Problem Context***

Ensure the integrity of business transactions that can span multiple user interactions and involve user think time.

### ***Recommendation***

Use an optimistic locking scheme.

### ***Discussion***

System transactions must have a very short duration to avoid unacceptable resource contention (i.e., system transactions and database locks should not span user think time). The use of naive pessimistic locking strategies to ensure the integrity of business transactions results in significant locking overhead and impaired system throughput and performance. Unfortunately, pessimistic locking underlies the concurrency control mechanisms commonly provided by most database systems.

The use of container-managed transaction demarcation or bean-managed transaction demarcation (as opposed to client-managed transaction demarcation) ensures that transactions will be completed

(committed or aborted) and locks will be released at the end of (appropriately configured) EJB method invocations. Container/bean-managed transactions will therefore not span user think time.

However, even with container/bean-managed transactions, the integrity of business transactions is not assured automatically. The following example from [IBM 00] illustrates the problem:

A user needs to update the customer information. For that purpose, he goes through these steps:

- Get a copy of customer data (first transaction)
- Modify this copy
- Send the copy to the server side to make this update permanent (second transaction)

It could happen that another user concurrently accesses the same data for update as described in this scenario:

1. userA requests a copy of customerA's data
2. userB requests a copy of customerA's data
3. userA changes customerA's name from 'Kurt Weiss' to 'Martin Weiss'
4. userB changes customerA's birthdate from 57/05/01 to 55/08/08
5. userA sends the changed copy for server-side update
6. userB sends the changed copy for server-side update

How can we prevent the updates made by userB from overriding those of userA?

The common solution to this problem for traditional (non-EJB) applications is through the use of a time-stamp (or version number) field on each table representing an entity subject to concurrency conflicts. The time-stamp field is updated whenever a record is successfully inserted or updated (e.g., through the use of triggers). Applications are responsible for checking that the time-stamp field has not changed between the time when the record was retrieved and the time the update is attempted. A common way to perform this check is to use a "where" clause in SQL update statements which ensures the update only goes through if the record's time stamp in the database matches the value supplied in the "where" clause.

The above approach can be easily adapted for use with BMP entity EJBs.

A more elaborate approach is described in [IBM 00]. That solution is based on an optimistic lock service implemented as a stateless session bean. The optimistic lock service is a generic, unified session facade for copying and updating all optimistic lockable entities based on a time-stamp. The *updateEntity* method of the *OptimisticLockService* session bean calls a private method *findEntityForUpdate* which attempts to retrieve the database record in question, using an SQL SELECT FOR UPDATE statement with primary key and timestamp as search criteria. If no record is returned, the record has been changed and cannot be updated. If a record is returned, then the record has not been changed, the *findEntityForUpdate* method instantiates the entity bean, and the *updateEntity* method completes the update.

The approach described in [IBM 00] has the advantage of standardizing and factoring out much of the common optimistic locking logic which needs to be implemented by optimistic-lockable objects. On the down side, this approach requires two database accesses (a SELECT FOR UPDATE and an



UPDATE) for each update, while the previously discussed approach only requires one database access (UPDATE WHERE).

## 6 Performance Checklist

Following are performance tips collected from the preceding portions of this document:

- Unless you really need distributed objects/components, don't use them
- Place components close to chatty resources they need to access
- Do not model all components as EJB
- Avoid stateful session beans
- Beware of entity beans
- Do not access entity beans directly from client code
- Access business objects by value through stateless session beans
- Prefer BMP over CMP when using entity beans to implement persistence for entity objects which map to relational joins
- For read-only objects, use a stateless session bean that accesses the database directly (without use of entity beans)
- For write-only objects, use a stateless session bean that inserts rows directly into the database without using entity beans
- Do not instantiate an entire list of entity beans when retrieving and scrolling through a list of objects; instead, use a session bean to return just the subset of information to be displayed to the user, and only instantiate the selected entity bean

## 7 References

[Brown 00] Kyle Brown, “Handling N-Ary Relationships in VisualAge for Java, Enterprise Edition, and WebSphere Advanced Edition”, WebSphere Developer Technical Journal

[Brown 00a] Kyle Brown, Gary Craig, Greg Hester, Jaime Niswonger, David Pitt, and Russell Stinehour, “Building Layered Architectures for EJB Systems”, Enterprise Java Programming with IBM WebSphere

[Brown 00b] Kyle Brown, “Choosing the Right EJB Type: Some Design Criteria”, WebSphere Developer Technical Journal

[IBM 00] Joaquin Picon, et al. “Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server”, IBM RedBooks.

[Res 00] Martijn Res, “Reduce EJB Network Traffic With Astral Clones”, Java World (<http://www.javaworld.com/jw-12-2000/jw-1208-clones.html>). This article describes a technique somewhat similar to that described in section 4.8 (based on [Roman 00]). However, this technique is more complex and potentially confusing.

[Roman 00] Ed Roman, “EJB Design Strategies and Performance Optimization”, JavaOne 2000 Conference

[Sun 99] The Enterprise Java Beans Specification, version 1.1

[Sun 00] “Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition”, version 1.0.1