Business Tier Design Considerations and Bad Practices

Topics in This Chapter

- Business Tier Design Considerations
- Business and Integration Tiers Bad Practices

Chapter 3

Business Tier Design Considerations

Business Tier Design Considerations

When you apply the business tier and integration tier patterns in this book, you'll need to know about related design issues, which we cover in this chapter. These issues cover a variety of topics, and can affect many aspects of a system.

The discussions in this chapter simply describe each issue as a design issue that you should consider when implementing systems based on the J2EE pattern catalog.

Using Session Beans

Session beans are distributed business components with the following characteristics, per the EJB specification:

- A session bean is dedicated to a single client or user.
- A session bean lives only for the duration of the client's session.
- A session bean does not survive container crashes.
- A session bean is not a persistent object.
- A session bean can time out.
- A session bean can be transaction aware.
- A session bean can be used to model stateful or stateless conversations between the client and the business tier-components.

Note: In this section, we use the term **workflow** in the context of EJB to represent the logic associated with the enterprise beans communication. For example, workflow encompasses how session bean A calls session bean B, then entity bean C.

Session Bean—Stateless Versus Stateful

Session beans come in two flavors—stateless and stateful. A stateless session bean does not hold any conversational state. Hence, once a client's method invocation on a stateless session beans is completed, the container is free to reuse that session bean instance for another client. This allows the container to maintain a pool of session beans and to reuse session beans among multiple clients. The container pools stateless session beans so that it can reuse them more efficiently by sharing them with multiple clients. The container returns a stateless session bean

to the pool after the client completes its invocation. The container may allocate a different instance from the pool to subsequent client invocations.

A stateful session bean holds conversational state. A stateful session bean may be pooled, but since the session bean is holding state on behalf of a client, the bean cannot simultaneously be shared with and handle requests from another client.

The container does not pool stateful session beans in the same manner as it pools stateless session beans because stateful session beans hold client session state. Stateful session beans are allocated to a client and remain allocated to the client as long as the client session is active. Thus, stateful session beans need more resource overhead than stateless session beans, for the added advantage of maintaining conversational state.

Many designers believe that using stateless session beans is a more viable session bean design strategy for scalable systems. This belief stems from building distributed object systems with older technologies, because without an inherent infrastructure to manage component life cycle, such systems rapidly lost scalability characteristics as resource demands increased. Scalability loss was due to the lack of component life cycle, causing the service to continue to consume resources as the number of clients and objects increased.

An EJB container manages the life cycle of enterprise beans and is responsible for monitoring system resources to best manage enterprise bean instances. The container manages a pool of enterprise beans and brings enterprise beans in and out of memory (called **activation** and **passivation**, respectively) to optimize invocation and resource consumption.

Scalability problems are typically due to the misapplication of stateful and stateless session beans. The choice of using stateful or stateless session beans must depend upon the business process being implemented. A business process that needs only one method call to complete the service is a non-conversational business process. Such processes are suitably implemented using a stateless session bean. A business process that needs multiple method calls to complete the service is a conversational business process. It is suitably implemented using a stateful session bean.

However, some designers choose stateless session beans, hoping to increase scalability, and they may wrongly decide to model all business processes as stateless session beans. When using stateless session beans for conversational business processes, every method invocation requires the state to be passed by the client to the bean, reconstructed at the business tier, or retrieved from a persistent store. These techniques could result in reduced scalability due to the associated overheads in network traffic, reconstruction time, or access time respectively.

Storing State on the Business Tier

Some design considerations for storing state on the Web server are discussed in "Session State in the Presentation Tier" on page 21. Here we continue that discussion to explore when it is appropriate to store state in a stateful session bean instead of in an HttpSession.

One of the considerations is to determine what types of clients access the business services in your system. If the architecture is solely a Web-based application, where all the clients come through a Web server either via a servlet or a JSP, then conversational state may be maintained in an HttpSession in the web tier. This scenario is shown in Figure 3.1.

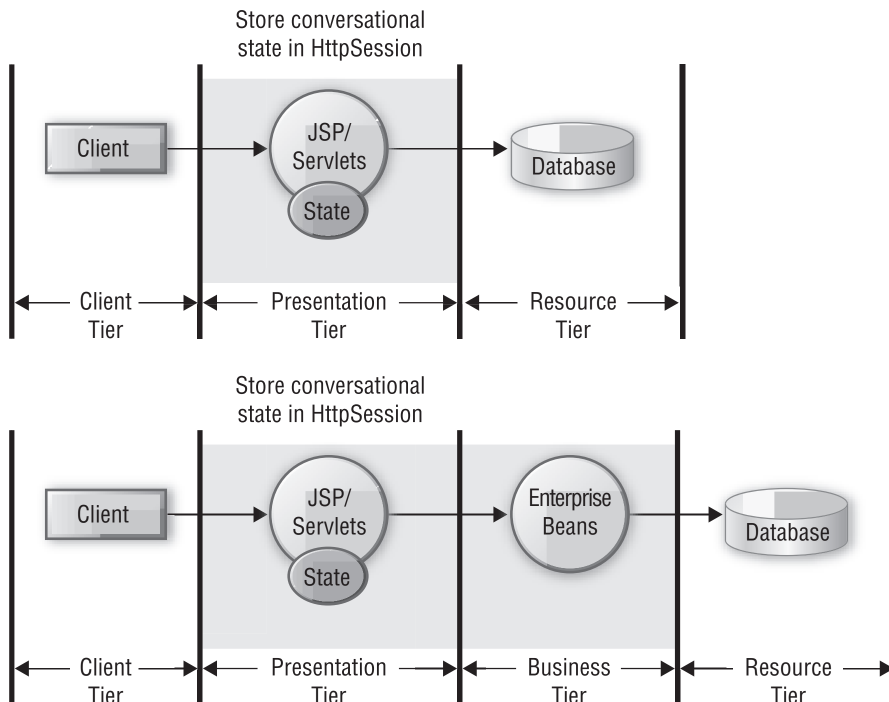

Figure 3.1 Storing State in HttpSession

On the other hand, if your application supports various types of clients, including Web clients, Java applications, other applications, and even other enterprise beans, then conversational state can be maintained in the EJB layer using stateful session beans. This is shown in Figure 3.2.

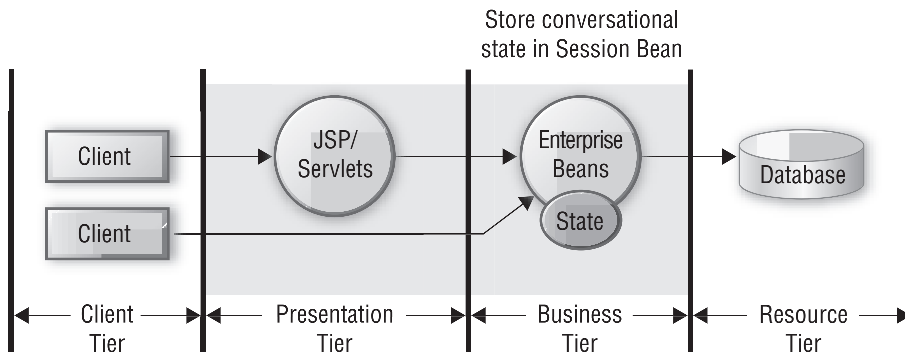

Figure 3.2 Storing State in Session Beans

We have presented some basic discussion on the subject of state management here and in the previous chapter (see "Session State on the Client" on page 20). A full-scale discussion is outside the scope of this book, since the problem is multi-dimensional and depends very much on the deployment environment, including:

- Hardware
- Traffic management
- Clustering of Web container
- Clustering of EJB container
- Server affinity
- Session replication
- Session persistence

We touch on this issue because it is one that should be considered during development and deployment.

Using Entity Beans

Using entity beans appropriately is a question of design heuristics, experience, need, and technology. Entity beans are best suited as coarse-grained business components. Entity beans are distributed objects and have the following characteristics, per the EJB specification:

- Entity beans provide an object view of persistent data.
- Entity beans are transactional.

- Entity beans are multiuser.
- Entity beans are long-lived.
- Entity beans survive container crashes. Such crashes are typically transparent to the clients.

Summarizing this definition, the appropriate use of an entity bean is as a distributed, shared, transactional, and persistent object. In addition, EJB containers provide other infrastructure necessary to support such system qualities as scalability, security, performance, clustering, and so forth. All together, this makes for a very reliable and robust platform to implement and deploy applications with distributed business components.

Entity Bean Primary Keys

Entity beans are uniquely identified by their primary keys. A primary key can be a simple key, made up of a single attribute, or it can be a composite key, made up of a group of attributes from the entity bean. For entity beans with a single-field primary key, where the primary key is a primitive type, it is possible to implement the entity bean without defining an explicit primary key class. The deployer can specify the primary key field in the deployment descriptor for the entity bean. However, when the primary key is a composite key, a separate class for the primary key must be specified. This class must be a simple Java class that implements the serializable interface with the attributes that define the composite key for the entity bean. The attribute names and types in the primary key class must match those in the entity bean, and also must be declared public in both the bean implementation class and primary key class.

As a suggested best practice, the primary key class must implement the optional `java.lang.Object` methods, such as `equals` and `hashCode`.

- Override the `equals()` method to properly evaluate the equality of two primary keys by comparing values for each part of the composite key.
- Override the `Object.hashCode()` method to return a unique number representing the hash code for the primary key instance. Ensure that the hash code is indeed unique when you use your primary key attribute values to compute the hash code.

Business Logic in Entity Beans

A common question in entity bean design is what kind of business logic it should contain. Some designers feel that entity beans should contain only persistence logic and simple methods to get and set data values. They feel that entity beans

should not contain business logic, which is often misunderstood to mean that only code related to getting and setting data must be included in the entity bean.

Business logic generally includes any logic associated with providing some service. For this discussion, consider business logic to include all logic related to processing, workflow, business rules, data, and so forth. The following is a list of sample questions to explore the possible results of adding business logic into an entity:

- Will the business logic introduce entity-entity relationships?
- Will the entity bean become responsible for managing workflow of user interaction?
- Will the entity bean take on the responsibilities that should belong in some other business component?

A "yes" answer to any of these questions helps identify whether introducing business logic into the entity bean can have an adverse impact, especially if you use remote entity beans. It is desirable to investigate the design to avoid inter-entity-bean dependencies as much as possible, since such dependences create overheads that may impede overall application performance.

In general, the entity bean should contain business logic that is self-contained to manage its data and its dependent objects' data. Thus, it may be necessary to identify, extract, and move business logic that introduces entity-bean-to-entity-bean interaction from the entity bean into a session bean by applying the *Session Façade* pattern. The *Composite Entity* (391) pattern and some of the refactorings discuss the issues related to entity bean design.

If any workflow associated with multiple entity beans is identified, then you can implement the workflow in a session bean instead of in an entity bean. Use a *Session Façade* (341) or *Application Service* (357).

- See "Merge Session Beans" on page 96.
- See "Reduce Inter-Entity Bean Communication" on page 98.
- See "Move Business Logic to Session" on page 100.
- See *Session Façade* (341)
- See *Business Object* (374)
- See *Composite Entity* (391)
- See *Application Service* (357)

For bean-managed persistence in entity beans, data access code is best implemented outside entity beans.

- See "Separate Data Access Code" on page 102.
- See *Data Access Object* (462).

Caching Enterprise Bean Remote References and Handles

When clients use an enterprise bean, they might need to cache some reference to an enterprise bean for future use. You will encounter this when using business delegates (see *Business Delegate* (302)), where a delegate connects to a session bean and invokes the necessary business methods on the bean on behalf of the client.

When the client uses the business delegate for the first time, the delegate needs to perform a lookup using the EJB Home object to obtain a remote reference to the session bean. For subsequent requests, the business delegate can avoid lookups by caching a remote reference or its handle as necessary.

The EJB Home handle can also be cached to avoid an additional Java Naming and Directory Interface (JNDI) lookup for the enterprise bean home. For more details on using an EJB Handle or the EJB Home Handle, please refer to the current EJB specification.

Business and Integration Tiers Bad Practices

Mapping the Object Model Directly to the Entity Bean Model

Problem Summary

One of the common practices in designing an EJB application is to map the object model directly into entity beans; that is, each class in the object model is transformed into an entity bean. This results in a large number of fine-grained entity beans.

The container and network overhead increases as the number of enterprise beans increases. Such mapping also transforms object relationships into entity-bean-to-entity-bean relationships. This is best avoided, since entity-bean-to-entity-bean relationships introduce severe performance implications for remote entity beans.

Solution Reference

Identify the parent-dependent object relationships in the object model and design them as coarse-grained entity beans. This results in fewer entity beans, where each entity bean composes a group of related objects from the object model.

- **Refactoring** See “Reduce Inter-Entity Bean Communication” on page 98.
- **Pattern** See *Composite Entity* (391)

Consolidate related workflow operations into session beans to provide a uniform coarse-grained service access layer.

- **Refactoring** See “Merge Session Beans” on page 96.
- **Pattern** See *Session Façade* (341)

Mapping the Relational Model Directly to the Entity Bean Model

Problem Summary

When designing an EJB model, it is bad practice to model each row in a table as an entity bean. While entity beans are best designed as coarse-grained objects,

this mapping results in a large number of fine-grained entity beans, and it affects scalability.

Such mapping also implements inter-table relationships that is, primary key/foreign key relationships) as entity-bean-to-entity-bean relationships.

Solution Reference

Design your enterprise bean application using an object-oriented approach instead of relying on the preexisting relational database design to produce the EJB model.

- **Bad Practice** See solution reference for "Mapping the Object Model Directly to the Entity Bean Model" on page 53.

Avoid inter-entity relationships by designing coarse-grained business objects by identifying parent-dependent objects.

- **Refactoring** See "Reduce Inter-Entity Bean Communication" on page 98.
- **Refactoring** See "Move Business Logic to Session" on page 100.
- **Pattern** See *Composite Entity* (391)

Mapping Each Use Case to a Session Bean

Problem Summary

Some designers implement each use case with its own unique session bean. This creates fine-grained controllers responsible for servicing only one type of interaction. The drawback of this approach is that it can result in a large number of session beans and significantly increase the complexity of the application.

Solution Reference

Apply the *Session Façade* pattern to aggregate a group of the related interactions into a single session bean. This results in fewer session beans for the application, and leverages the advantages of applying the *Session Façade* pattern.

- **Refactoring** See "Merge Session Beans" on page 96.
- **Pattern** See *Session Façade* (341)

Exposing All Enterprise Bean Attributes via Getter/Setter Methods

Problem Summary

Exposing each enterprise bean attribute using getter/setter methods is a bad practice. This forces the client to invoke numerous fine-grained remote invocations and creates the potential to introduce a significant amount of network chattiness across the tiers. Each method call is potentially remote and carries with it a certain network overhead that impacts performance and scalability.

Solution Reference

Use a value object to transfer aggregate data to and from the client instead of exposing the getters and setters for each attribute.

- **Pattern** See *Transfer Object* (415).

Embedding Service Lookup in Clients

Problem Summary

Clients and presentation-tier objects frequently need to look up the enterprise beans. In an EJB environment, the container uses JNDI to provide this service.

Putting the burden of locating services on the application client can introduce a proliferation of lookup code in the application code. Any change to the lookup code propagates to all clients that look up the services. Also, embedding lookup code in clients exposes them to the complexity of the underlying implementation and introduces dependency on the lookup code.

Solution Reference

Encapsulate implementation details of the lookup mechanisms using a *Service Locator* (315).

- **Pattern** See *Service Locator* (315)

Encapsulate the implementation details of business tier-components, such as session and entity beans, using *Business Delegate* (302). This simplifies client code since they no longer deal with enterprise beans and services. *Business Delegate* (302) can in turn use the *Service Locator* (315).

- **Refactoring** See "Introduce Business Delegate" on page 94.
- **Pattern** See *Business Delegate* (302).

Using Entity Beans as Read-Only Objects

Problem Summary

Any entity bean method is subject to transaction semantics based on its transaction isolation levels specified in the deployment descriptor. Using an entity bean as a read-only object simply wastes expensive resources and results in unnecessary update transactions to the persistent store. This is due to the invocation of the ejbStore() methods by the container during the entity bean's life cycle. Since the container has no way of knowing if the data was changed during a method invocation, it must assume that it has and invoke the ejbStore() operation. Thus, the container makes no distinction between read-only and read-write entity beans. However, some containers may provide read-only entity beans, but these are vendor proprietary implementations.

Solution Reference

Encapsulate all access to the data source using *Data Access Object* (462) pattern. This provides a centralized layer of data access code and also simplifies entity bean code.

- **Pattern** See *Data Access Object* (462).

Implement access to read-only functionality using a session bean, typically as a Session Façade that uses a DAO.

- **Pattern** See *Session Façade* (341)

You can implement *Value List Handler* (444) to obtain a list of *Transfer Objects* (415).

- **Pattern** See *Value List Handler* (444).

You can implement *Transfer Objects* (415) to obtain a complex data model from the business tier.

- **Pattern** See *Transfer Object Assembler* (433).

Using Entity Beans as Fine-Grained Objects

Problem Summary

Entity beans are meant to represent coarse-grained transactional persistent business components. Using a remote entity bean to represent fine-grained objects increases the overall network communication and container overhead. This impacts application performance and scalability.

Think of a fine-grained object as an object that has little meaning without its association to another object (typically a coarse-grained parent object). For example, an item object can be thought of as a fined-grained object because it has little value until it is associated with an order object. In this example, the order object is the coarse-grained object and the item object is the fine-grained (dependent) object.

Solution Reference

When designing enterprise beans based on a preexisting RDBMS schema,

- **Bad Practice** See "Mapping the Relational Model Directly to the Entity Bean Model" on page 53.

When designing enterprise beans using an object model,

- **Bad Practice** See "Mapping the Object Model Directly to the Entity Bean Model" on page 53.

Design coarse-grained entity beans and session beans. Apply the following patterns and refactorings that promote coarse-grained enterprise beans design.

- **Pattern** See *Composite Entity* (391).
- **Pattern** See *Session Façade* (341).
- **Refactoring** See "Reduce Inter-Entity Bean Communication" on page 98.
- **Refactoring** See "Move Business Logic to Session" on page 100.
- **Refactoring** See "Business Logic in Entity Beans" on page 50.
- **Refactoring** See "Merge Session Beans" on page 96.

Storing Entire Entity Bean-Dependent Object Graph

Problem Summary

When a complex tree structure of dependent objects is used in an entity bean, performance can degrade rapidly when loading and storing an entire tree of dependent objects. When the container invokes the entity bean's ejbLoad() method, either for the initial load or for reloads to synchronize with the persistent store, loading the entire tree of dependent objects can prove wasteful. Similarly, when the container invokes the entity bean's ejbStore() method at any time, storing the entire tree of objects can be quite expensive and unnecessary.

Solution Reference

Identify the dependent objects that have changed since the previous store operation and store only those objects to the persistent store.

- **Pattern** See *Composite Entity* (391) and *Store Optimization (Dirty Marker) Strategy* (397).

Implement a strategy to load only data that is most accessed and required. Load the remaining dependent objects on demand.

- **Pattern** See *Composite Entity* (391) and *Lazy Loading Strategy* on page 396.

By applying these strategies, it is possible to prevent loading and storing an entire tree of dependent objects.

Exposing EJB-related Exceptions to Non-EJB Clients

Problem Summary

Enterprise beans can throw business application exceptions to clients. When an application throws an application exception, the container simply throws the exception to the client. This allows the client to gracefully handle the exception and possibly take another action. It is reasonable to expect the application developer to understand and handle such application-level exceptions.

However, despite employing such good programming practices as designing and using application exceptions, the clients may still receive EJB-related exceptions, such as a `java.rmi.RemoteException`. This can happen if the enterprise bean or the container encounters a system failure related to the enterprise bean.

The burden is on the application developer, who may not even be aware of or knowledgeable about EJB exceptions and semantics, to understand the implementation details of the non-application exceptions that may be thrown by business tier-components. In addition, non-application exceptions may not provide relevant information to help the user rectify the problem.

Solution Reference

Decouple the clients from the business tier and hide the business-tier implementation details from clients, using business delegates. Business delegates intercept all service exceptions and may throw an application exception. Business delegates are plain Java objects that are local to the client. Typically, business delegates are developed by the EJB developers and provided to the client developers.

- **Refactoring** See “Introduce Business Delegate” on page 94.
- **Pattern** See *Business Delegate* (302).

Using Entity Bean Finder Methods to Return a Large Results Set

Problem Summary

Frequently, applications require the ability to search and obtain a list of values. Using an EJB finder method to look up a large collection of entity beans will return a collection of remote references. Consequently, the client has to invoke a method on each remote reference to get the data. This is a remote call and can become very expensive, especially impacting performance, when the caller invokes remote calls on each entity bean reference in the collection.

Solution Reference

Implement queries using session beans and DAOs to obtain a list of *Transfer Objects* (415) instead of remote references. Use a DAO to perform searches instead of EJB finder methods.

- **Pattern** See *Value List Handler* (444).
- **Pattern** See “Data Access Object” on page 462.

Client Aggregates Data from Business Components

Problem Summary

The application clients (in the client or presentation tier) typically need the data model for the application from the business tier. Since the model is implemented by business components—such as entity beans, session beans, and arbitrary objects in the business tier—the client must locate, interact with, and extract the necessary data from various business components to construct the data model.

These client actions introduce network overhead due to multiple invocations from the client into the business tier. In addition, the client becomes tightly coupled with the application model. In applications where there are various types of clients, this coupling problem multiplies: A change to the model requires changes to all clients that contain code to interact with those model elements comprised of business components.

Solution Reference

Decouple the client from model construction. Implement a business-tier component that is responsible for the construction of the required application model.

- **Pattern** See *Transfer Object Assembler* (433).

Using Enterprise Beans for Long-Lived Transactions

Problem Summary

Enterprise beans (pre-EJB 2.0) are suitable for synchronous processing. Furthermore, enterprise beans do well if each method implemented in a bean produces an outcome within a predictable and acceptable time period.

If an enterprise bean method takes a significant amount of time to process a client request, or if it blocks while processing, this also blocks the container resources, such as memory and threads, used by the bean. This can severely impact performance and deplete system resources.

An enterprise bean transaction that takes a long time to complete potentially locks out resources from other enterprise bean instances that need those resources, resulting in performance bottlenecks.

Solution Reference

Implement asynchronous processing service using a message-oriented middleware (MOM) with a Java Message Service (JMS) API to facilitate long-lived transactions.

- **Pattern** See "Service Activator" on page 496.

Stateless Session Bean Reconstructs Conversational State for Each Invocation

Problem Summary

Some designers choose stateless session beans to increase scalability. They may inadvertently decide to model all business processes as stateless session beans even though the session beans require conversational state. But, since the session bean is stateless, it must rebuild conversational state in every method invocation. The state may have to be rebuilt by retrieving data from a database. This com-

pletely defeats the purpose of using stateless session beans to improve performance and scalability and can severely degrade performance.

Solution Reference

Analyze the interaction model before choosing the stateless session bean mode. The choice of stateful or stateless session bean depends on the need for maintaining conversational state across method invocations in stateful session bean versus the cost of rebuilding the state during each invocation in stateless session bean.

- **Pattern** See *Transfer Object Assembler* (433), *Stateless Session Façade Strategy* on page 345, and *Stateful Session Façade Strategy* on page 345.
- **Design** See "Session Bean—Stateless Versus Stateful" on page 46 and "Storing State on the Business Tier" on page 48.