

University of Cooperative Education Mannheim



Independent Research Project

Sun Enterprise JavaBeans Specification, Version 2.1
Concepts and Architecture

Mentor: Jens Peter Misch

Steffen Grunwald, TIM01ANC
Matriculation No. 130748

June 20, 2004

Declaration of Academic Honesty

I hereby declare to have written this independent research project paper on my own, using only the listed resources.

Location, Date

Steffen Grunwald, Matr. No.: 130748

Abstract

In contemporary business the building of new systems upon existent and proven structures has become a major key to success for a company. The information technology experiences constantly growing challenges. Thus, it is difficult to develop software that provides a rock solid basis for today's business applications.

The idea of Sun Microsystems was that developers implement portable components, the Enterprise JavaBeans, running on a mighty framework which provides the most important functions for enterprise applications. The developer is able to focus on the business functions rather than the system services. This supports and shortens the whole process and reduces the risk of mistakes in design.

The Enterprise JavaBeans specification has to react to the constantly emerging demands of the industry. This document introduces the recently released version 2.1 and provides a background to the Enterprise JavaBeans architecture and concepts.



Contents

1	Introduction	1
1.1	Definition	1
1.2	Purpose of the EJB technology	1
1.3	History of EJB	2
2	EJB Architecture and Concepts	2
2.1	EJB Environment	2
2.1.1	The EJB Server	2
2.1.2	The EJB Container	4
2.2	Enterprise Bean Components	5
2.2.1	The Bean Class	5
2.2.2	The Bean Interfaces	6
2.2.3	The Primary Key Class as a unique Bean Identifier	7
2.2.4	The XML Deployment Descriptor	7
2.3	Interfaces to the Bean	7
2.3.1	Local and Remote Interface	7
2.3.2	Comparison of Advantages	8
2.3.3	The Service Endpoint Interface	9
3	Bean Types	9
3.1	Entity Beans	9
3.1.1	Purpose and Structure	9
3.1.2	Container Managed- or Bean Managed Persistence	10
3.2	Session Beans	10
3.2.1	Purpose and Structure	10
3.2.2	Stateful or Stateless Session Beans	11
3.3	Message Driven Beans (MDB)	11
4	New Features in the Specification	11
4.1	The Service Endpoint Interface to Stateless Session Beans	12



4.1.1	Problems with previous Specifications	12
4.1.2	Previous Solutions and Workarounds	12
4.1.3	The Service Endpoint Interface defined by the Specification	13
4.2	The enhanced Language Elements of EJB QL	14
4.2.1	Problems with previous Specifications	14
4.2.2	Previous Solutions and Workarounds	14
4.2.3	The new EJB QL Elements defined by the Specification	15
4.3	The Timer Service for scheduled Events	15
4.3.1	Problems with previous Specifications	15
4.3.2	Previous Solutions and Workarounds	15
4.3.3	The new Timer Service defined by the Specification	16
4.4	The enhanced Message Driven Bean	16
4.4.1	Previous Solutions and Workarounds	16
4.4.2	The new Messages to access Message Driven Beans	17
4.5	New Features in EJB 3.0	18
5	Conclusion	18
A	Appendix	20
A.1	Alternative Clients for JMS	20

List of Figures

1	The Components in the EJB Architecture	4
2	The Usage of the Remote Interfaces	7
3	The Usage of the Local Interfaces	8
4	The Service Endpoint Interface for JAX-RPC	13
5	The Timer Service Interface TimedObject!	17

List of Tables

1	The EJB History	3
---	---------------------------	---



2 Local Bean Interfaces vs. Remote Bean Interfaces 8

Abbreviations and Acronyms

API	Application Programming Interface
CMP	Container Managed Persistence
CMR	Container Managed Relation
CMT	Container Managed Transaction
CORBA	Common Object Request Broker Architecture
EJB	Enterprise JavaBeans
EJB QL	EJB Query Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IDL	Interface Definition Language
IIOP	Internet- Inter ORB Protocol
JAXM	Java Api for XML Messaging
JAX-RPC	Java API for XML-Based RPC
JVM	Java Virtual Machine
ORB	Object Request Broker
RMI	Remote Method Invocation
RMI-IIOP	RMI over IIOP
RPC	Remote Procedure Call
SEI	Service Endpoint Interface
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
XML	eXtensible Markup Language

1 Introduction

1.1 Definition

The Enterprise JavaBeans (EJB) Specification defines an architecture for developing component-based applications. The original definition in the preamble of the specification is:

“The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.” [SUN03]

The specification constantly changes to reflect the emerging requirements of the industry and is issued in approximately 2 year intervals. The fourth and latest final version 2.1 (EJB 2.1) of the specification describes in 646 pages the components, the tasks, the implementation and the deployment of an Enterprise JavaBean on a Java 2 Enterprise Edition (J2EE) server.

Although Java and Enterprise JavaBeans is a registered trademark by Sun Microsystems, the specification is a result of proposals by several J2EE related software companies like IBM, BEA, Borland and Oracle, to only name a few. It is a standard, not a product. That is why there are several vendors of EJB compliant servers for many operating system platforms.

1.2 Purpose of the EJB technology

The main purpose of the EJB specification is to support the development of software components which deal with enterprise data. Due their rich functionality and the operating system independence the EJB developer is able to focus on the business logic of an application. He or she does not have to bother about system services or lower level related issues like database structures, rollbacks and concurrent access to components.

The component architecture of EJBs allow developers to assemble an enterprise application of existing EJBs. The architecture supports this with the use of deployment

descriptors – eXtensible Markup Language (XML) files that store information regarding the environment. These let a bean deployer modify a bean without the need to recompile it.

A typical application where beans can be reused is a internet shop. A shopping cart is part of almost every shopping application. This can be implemented as a bean. Further reusable components can be a stock watcher or an invoice sender. Due to the EJB architecture these components are neither dependent on the visualization nor on operating system specifics and thus highly reusable.

1.3 History of EJB

As mentioned previously, it is already the fourth release of the specification. The ongoing development process is the response to the growing claims of today's enterprise applications. Although changes from the early versions to version 2.1 significantly increased the functionality and efficiency of development, one is still most likely to meet EJB 1.1 applications in productive J2EE systems. Thus, table 1 shows a list of EJB specification versions, the J2EE version which support it (where applicable) and a brief summary of the major changes. The version 3.0 status is yet declared as draft but the tendencies are mentioned here for a complete overview. The important features starting from EJB 2.1 are explained in a greater detail later (section 4, page 11).

2 EJB Architecture and Concepts

2.1 EJB Environment

2.1.1 The EJB Server

The EJB server is that part of the so called application server or J2EE server that runs most closely to the operating system. It hosts at least one EJB container. In most cases, an J2EE server is also capable of hosting servlets and JSPs in further containers. The EJB server hides the system services from the upper levels to provide platform independence.

Although the EJB container and the EJB server are defined as two different parts of

EJB	J2EE	Date	Introduced Features
1.0	—	1997	Definition of the EJB server tasks
			Definition of the client's and developer's view of an EJB
			Description of the EJB JAR file, the deployment archive for an EJB
1.1	1.0	1999	A round up for the features of EJB 1.0
			Support for a better EJB deployment
			More detailed description to roles affected in the EJB environment
2.0	1.3	2001	Introduction of container managed persistence (CMP)
			Introduction of container managed relations (CMR)
			Creation of EJB QL to make queries in CMP beans
			Introduction of a local client view: pass by reference as performant alternative to pass by value
			Introduction of the message driven bean (MDB) for asynchronous communication
2.1	1.4	2003	Introduction of the service endpoint interface to stateless session beans
			Language enhancements of EJB QL: ORDER BY, AVG, COUNT . . .
			Enhancement of message driven bean messaging technologies (more than only JMS)
			Introduction of the timer service to support notifications or alerts
3.0	1.5	~2005	Focus on simplifying rather than adding new features to the specification
			Support of the development process and reduction of EJB complexity

Table 1: A brief changelog of all EJB versions with their respective J2EE releases

the application server, the tasks and interfaces of these parts are not clearly defined by EJB 2.1. Thus, dependent on the server vendor some tasks are located in the server, whereas others are implemented by the container. This has no implications to the EJB itself but it prevents a mix of container and servers of different vendors.

2.1.2 The EJB Container

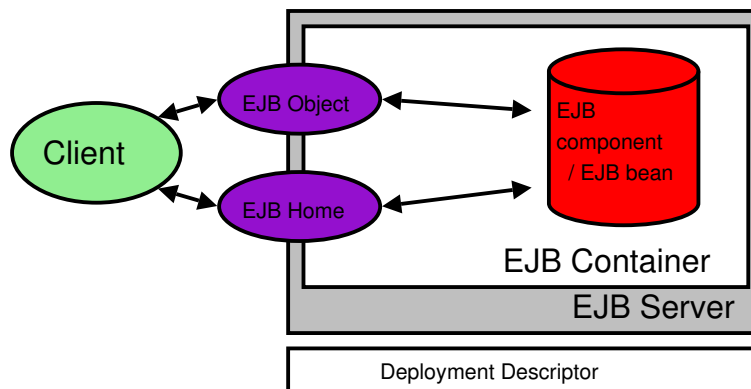


Figure 1: The components in the EJB architecture (source:[IBM03])

The container acts as a runtime environment to the EJBs. Please see figure 1 for a simple client-ejb constellation with all components. The container provides the following services to one or more EJBs:

- **Naming:** Each EJB is registered in a central naming directory at deploy time. An hierarchical name is registered in the Java Naming Directory Interface and bound to an EJB home factory¹. If the application is distributed over more than one server, a single common JNDI² can be used to register and retrieve references.
- **Transaction:** Transactions consist of one or more tasks that have to be executed as a whole or not at all. If transactions have to be used in the application, they can be defined in the deployment descriptor of the bean. The container takes care of the transactions, commits them or initializes rollbacks.

¹Basically an intermediate class which caches and initializes the EJB home interface

²defined by the `javax.naming.Context.PROVIDER_URL` variable in the server environment, vendor dependent

- **Security:** Using access control lists (ACL), a container permits or denies functions to certain users and groups based on a role schema.
- **Persistence:** The container manages the persistence of all entity beans by synchronizing the bean state with the respective record(s) in the database.
- **Concurrency:** If two clients connect to the same entity, there can occur inconsistencies. The container prevents that by concurrency control.
- **Life cycle:** The container pools, reuses, destroys or instantiates beans. Most of this can be done by the container, but in some cases the bean developer has to support this with certain bean methods that can serialize references for database storage (e.g. `passivate`).
- **Messaging:** The container accepts messages on behalf of the message driven bean (MDB). These are sent to the MDB to execute asynchronous methods.

2.2 Enterprise Bean Components

A bean developer has to provide the bean component, that is the sum of all parts described in this section. These are packaged in the java archive file (EJB JAR file). The bean is then supposed to run on any vendor's server that supports the EJB 2.1 implementation.

As EJB has never defined every detail of the EJB implementation, each vendor had to develop additional XML descriptors which have been added to the EJB JAR. This prevents real vendor independence and requires vendor specific training.

2.2.1 The Bean Class

The bean class represents the actual implementation of the enterprise bean. It implements the (local) home interface, the (local) component interface and eventually the web service endpoint interface. The type of bean determines the implementation of the bean. This is a brief list of all enterprise beans:

- **Entity beans:** The entity bean represents entities or nouns in a system (e.g. customer, car). It is persistent, i.e. its state is stored in the database. After a termination of server or client, the bean is still available.
- **Session beans:** Business logic, workflows or functions are modelled in a session bean. It deals with the entity beans and provides its methods to the client. Either it is implemented as exclusively available to just one client and can store a transient state (stateful), or it is shared among all clients (stateless). However, after a server or client termination, a bean and a possible state is lost.
- **Message Driven Beans:** The message driven bean gets messages from clients (via the container) and processes methods³. This asynchronous invocation is an alternative to the synchronous request-response communication, which stays in a waiting position during the process until a response is sent to the client.

2.2.2 The Bean Interfaces

The bean interfaces are used by the client to access the bean. Their methods have to be implemented in the bean class. The following interfaces exist:

- **EJB component interface / EJB object:** The component interface defines the business methods of the bean that are available to the client. This interface can be local and remote.
- **EJB home interface / EJB home:** The methods of the home interface are also available to the client but do not refer to a concrete bean object. The home interface allows to create, find or to remove a bean. It can be local or remote as well.
- **Service endpoint interface (SEI):** The SEI is basically the same as the component interface but it extends another class and it can only be implemented by a stateless session bean. Details about the SEI are found in the introduction of the new features in EJB 2.1.

³e.g. starting a very time-consuming process like an ftp distribution

2.2.3 The Primary Key Class as a unique Bean Identifier

Every bean has to have a single class as a primary key by which it is found or identified during creation. Either a an existent class from the Java API or a custom serializable class is used. It is not possible to use a primitive type, such as `java.lang.int`.

2.2.4 The XML Deployment Descriptor

The deployment descriptor is a human readable XML file. It defines parameters how a container has to handle the EJB. If container management handles relations (CMR), transactions (CMT) or persistence (CMP), this is configured here. Further configurations are the type of state handling⁴ and the interface classes to the bean. If EJB QL is used to access the database from CMP beans, the queries are also stored in this file.

2.3 Interfaces to the Bean

2.3.1 Local and Remote Interface

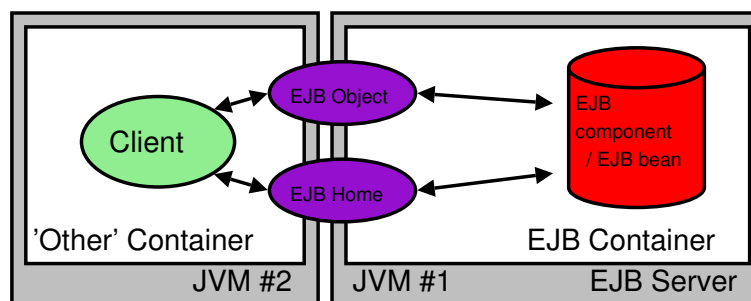


Figure 2: The usage of the remote interfaces (source:[IBM03])

As previously mentioned a client communicates with the server either via the EJB home interface to find, create or remove a bean or it uses the EJB object interface to invoke business methods. Until the EJB specification 2.0 was released this was done over remote interfaces, which enabled a client to access the bean from within the same Java virtual machine (JVM) or even from other server hardware (see figure 2). Thus the actual location of the bean was transparent to the client.

⁴stateful or stateless

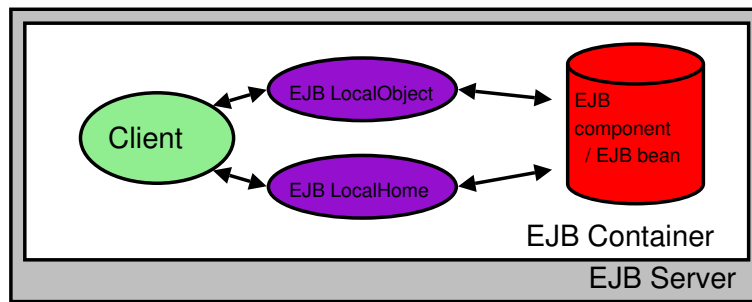


Figure 3: The usage of the local interfaces (source:[IBM03])

In many applications using the EJB technology this lead to a performance problem: By sending the beans serialized to the another bean within the same JVM, the process produced more overhead then necessary. The EJB specification 2.0 addressed this problem with the introduction of local interfaces as an alternative to the remote interfaces (see figure 3).

2.3.2 Comparison of Advantages

Local interfaces cause an improvement from the performance perspective but also imply restrictions to the architecture of the application. Table 2 shows a comparison and side effects.

Remote Interfaces	Local Interfaces
< EJB 2.0	≥ EJB 2.0
EJB Object extends EJBObject	EJB Object extends EJBLocalObject
EJB Home extends EJBHome	EJB Home extends EJBLocalHome
casting by narrow function	casting by regular Java class casts
Call by value → serialization	Call by reference
Fair performance	Better performance
EJB location transparent to the client	Client and EJB resides in the same JVM
EJBs distributable among several application servers	EJBs deployed on the same server
Method throws RemoteException	Method throws regular Java exception
Not supported by CMR	Supported by CMR
All options open for future changes in architecture	Stuck to one JVM

Table 2: A comparison of local- and remote interfaces of a bean

A good rule is to use the local interfaces unless it is really necessary to call bean

methods from outside of the EJB container. This must happen somewhere for a client or servlet to gain access to the beans' values and methods.

Dependent on the application's size it is worth to think about a distributed architecture with remote interfaces and more than one application server. Several servers even make sense on one physical server. [ARN01] states that in most cases the JVM is not able to use all resources of the server hardware, especially in multiprocessor systems.

2.3.3 The Service Endpoint Interface

A stateless session bean can have an additional service endpoint interface (SEI), which defines the methods that are available over a web service, i.e. a remote procedure call over the simple object access protocol (SOAP). For details to this feature that is new in EJB 2.1 see section 4.1 (page 12).

3 Bean Types

3.1 Entity Beans

3.1.1 Purpose and Structure

Entity beans are objects that are stored persistently in a data source. They are used to model the nouns in the system (e.g. customer). For the bean consumer the actual storage is completely transparent and hidden.

The Bean is similar to the value object pattern (see [BIE02]). Typically they do not have business methods but only the getter and setter methods for the attributes of the object they represent (e.g. customer→name). It has to implement the (local) home and the (local) object interface. Furthermore it can implement the timer interface to schedule methods (see section 4.3, page 15). The home interface is used to find, create or remove bean instances. To identify a bean, an entity bean has to have a primary key class that is unique to its bean class.

3.1.2 Container Managed- or Bean Managed Persistence

If a relational database is used, each bean is represented by one table and each bean instance is mapped to one row in that table. Whether this is done automatically by the EJB container or implemented in the bean class itself is decided by the bean developer. There are two types of persistence.

Using **container managed persistence** (CMP), a bean developer relies on the EJB container to implement the mapping of a set or get method to the database. He does not have to bother with writing SQL queries. What he has to do is to define abstract getter and setter methods for every field in the bean class and to propagate this fields in the deployment descriptor. He only has to deal with EJB QL, that is to code custom finder methods (e.g. `findCustomerByAddress`).

With **bean managed persistence** (BMP), the bean developer has to implement this on his own. He implements everything from the opening to the closing of the database connection which means much effort for the developer. There are some reasons for choosing BMP over CMP which are all related to incompatibilities of the application server with the data source or a complex data structure. If this is not the case in the environment, [IBM03] recommends: “Unless you have a really good reason, stick with CMPs”.

3.2 Session Beans

3.2.1 Purpose and Structure

A session bean provides business methods. In implementations of the Value Object Assembler or the Session Façade Pattern (see [BIE02]), the client does not have to deal directly with the entity beans but rather calls the methods of the session bean.

The session bean has a (local) home and a (local) object interface and an optional service endpoint interface as a web service (see section 4.1, page 12). Like a bean, it can also facilitate the timer service to schedule processes (see section 4.3, page 15).

3.2.2 Stateful or Stateless Session Beans

An instance of a session can be used by only one client at a time. After the invocation of a method, the client can either stay in a conversational relation with the bean or the bean is released and made available to other clients.

Stateful session beans can store data related to the session with one single client (e.g. shopping cart bean). For the duration of the session the bean is used exclusively by this client. After the bean is released, the state has to be emptied.

Stateless session beans are not bound to a concrete client and do not store client data. In this case every time a client invokes a method, it might get another bean from the bean pool. Thus, storing data in a stateless session bean is not allowed.

3.3 Message Driven Beans (MDB)

Message driven beans are endpoints for asynchronous messaging. Asynchronous messaging consists only of one message compared to the usual request-response communication (like HTTP or RPC). This can be used to send notifications to the system or to trigger actions that take much time.

Message Driven Beans must not have a home interface. They can not hold a conversational state, hence concrete MDBs do not have to be found by a client. For messaging, they have to implement a messaging interface (see 4.4, page 16).

4 New Features in the Specification

The EJB specification is constantly developed in the Java Community Process (JCP). Everyone can participate in that community. The influence on the so called Java Specification Requests (JSR), that are not restricted to just the EJB specification, may vary because they are mainly driven by the companies. However, after a JSR is finished Sun accepts it as official paper and gives the Sun owned Java and Enterprise JavaBeans trademark to it.

An important aspect of the specification's evolution is that the big vendors have already implemented features as proprietary extension to the specification in advance

and have gained great popularity among the users with these. Thus, the JCP is primarily concerned about which of these extensions are integrated as a consent.

The following describes the JSR No. 153 finished at 12th of November 2003 by the JCP. After a discussions of the four major changes opposed to previous releases, the expected features of EJB 3.0 are reflected.

4.1 The Service Endpoint Interface to Stateless Session Beans

4.1.1 Problems with previous Specifications

According to [IBM03] the communication to and from Enterprise JavaBeans is done over RMI-IIOP. This is actually the mix of the CORBA architecture for distributed applications and the java remote method invocation (RMI). That made the EJB architecture open for client applications that either used RMI-IIOP (e.g. other EJBs, Java clients), or these that were implemented to use the CORBA technology, independent of the programming language.

Today, the interoperability with common technologies like CORBA is not enough. Microsoft's .NET framework was one of the first products that pushed the idea of using an XML based language as a message protocol. In spite of the expensive additional parsing of data into XML valid structures, this simple object access protocol (SOAP) submitted over HTTP became very popular. [LOM01] states that the aggressive leverage of web services implies an increase of efficiency by 30% of the IT development by 2005. Although this is not exactly right today, the assumption is not too far away from reality: ebay.com, amazon.com and google.com all provide an interface to their services that can be consumed by any program with SOAP capabilities.

4.1.2 Previous Solutions and Workarounds

To adapt this technology and to offer the SOAP access to and from beans, there are three possibilities to invoke methods over SOAP in EJB 2.0:

- A servlet *directly* handles the SOAP message received via the post method. This is on the one hand very flexible and can also be used with other protocols than HTTP,

but is very challenging for the developer on the other hand. After the SOAP message is processed, the respective EJB method is invoked and the results are returned to the client via SOAP.

- A servlet container uses a *service endpoint interface* (SEI) that maps to correspondent method of a servlet. This is implemented with JAX-RPC, an implementation of SOAP. Thus, a developer does not have to handle the XML itself.
- The *Apache AXIS* JAX-RPC implementation (see [LAN03]) takes a regular Java source file with a *.jws extension, compiles it, and deploys a web service automatically. This is very comfortable but not part of the J2EE specification, as Apache AXIS is a product developed independently from the JCP.

To consume a web service, of course either of these three options can be utilized. There is no difference to the implementation in a non EJB environment (a regular Java class).

4.1.3 The Service Endpoint Interface defined by the Specification

The stateless session bean of the EJB 2.1 specification can be exposed to a web service client by adding an additional service endpoint interface to the EJB. Similar to the (local) component interface, it describes the business methods that are available to the client (see figure 4).

```
1 public interface StockWS extends java.rmi.Remote{  
2     public String retrieveStockPrice(java.lang.String stockSymbol)  
3         throws java.rmi.RemoteException;  
4 }
```

Figure 4: The service endpoint interface that describes a method in a bean class that retrieves stock prices. The interface is available over SOAP

A web service can only be offered by a stateless session bean, because, like typical HTTP requests, SOAP can not handle states. This also implies that a home interface, which helps to create or find a special bean, is not necessary: Every stateless session bean is equal to the other. The beans can be pooled by the container. Instead of creating the

bean over the home interface, the client simply invokes the business functions and the bean is created by the container automatically if needed.

If a bean is required to consume a web service, this has to be defined in the deployment descriptor: Each reference is mapped with the respective uniform resource identifier (URI) to a JNDI name⁵. During runtime it can lookup that service and use it according to the JAX-RPC specification.

4.2 The enhanced Language Elements of EJB QL

4.2.1 Problems with previous Specifications

As previously mentioned, developing container managed beans, a bean provider does not have to bother about persistence and how to actually store the bean in a database. With the introduction of this feature (\geq EJB 2.0), the EJB query language (EJB QL) emerged to address the issue of finding beans and to select their data from business methods. It is a language that is similar to the regular structured query language (SQL) but in EJB 2.0 some SQL elements are missing.

The functions `AVG`, `MAX`, `MIN`, `SUM`, and `COUNT` and the `ORDER BY` element are not available but required operations in most business applications. The availability of this is would reduce the communication between database and client.

This has either been done to keep certain logic out of the database layer, or because some data sources do not support these elements.

4.2.2 Previous Solutions and Workarounds

A workaround has been created rather by the application server provider than the bean provider. The previously mentioned vendor specific extensions allowed these elements in EJB QL. One obvious drawback is the dependence on one vendor, which is one of the major problems that EJBs want to address.

The `ORDER BY` logic can also be included into the bean client itself, as a sorting of the unordered collection that is returned in the result of EJB QL. The other functions

⁵under `java:comp/env/services`

like `MAX` can also be applied to the result. The effort for this implementation is high, compared to the simple inclusion of an `ORDER BY` clause into the EJB QL query.

Another option is to implement the bean as BMP bean. The price paid for the EJB 2.0 compliance is the the manual implementation of all container comforts.

4.2.3 The new EJB QL Elements defined by the Specification

EJB 2.1 reflects the calls by the JCP for this functionality. The functions mentioned above have been added to the specification. But some restrictions still apply to EJB QL: One example is the IBM WebSphere Application Server, which provides in its EJB 2.0 implementation subselects, `GROUP BY` and `HAVING`. This is still not available in EJB yet and states the need for the workarounds mentioned above.

4.3 The Timer Service for scheduled Events

4.3.1 Problems with previous Specifications

The initial design of EJBs does not offer the possibility to delay the execution of methods – after a business method is invoked (whether via messages or a method call) it is executed immediately. Furthermore, actions that have to take place in intervals can not be implemented with beans, too. For example daily notifications or a delayed automatic expiration of a document is impossible.

The reason for this is that the EJBs are not allowed to use threads to implement this waiting – a rule to prevent a strong coupling from system-services.

4.3.2 Previous Solutions and Workarounds

The missing feature is obviously a daemon like the popular cron from the UNIX world. Whether this is implemented in a Java application or system provided services like cron are used, it always represents a thread that runs continuously, maintains a list of scheduled jobs and runs them accordingly at the specified time.

4.3.3 The new Timer Service defined by the Specification

With EJB 2.1 a timer service was introduced that can be consumed by entity beans, stateless session beans and message driven beans. Similar to cron, it runs a service at a single point in time (single action) or starting from one point in time with a defined interval (interval action). The maximum duration of the existence of a timer can be specified at timer creation.

The prerequisite for a bean that wants to utilize the timer service is the implementation of the `javax.ejb.TimedObject` interface (see figure 5). This is only the `ejbTimeout` method that contains the business logic, which

can be scheduled. To schedule different behaviors of the same bean, an additional argument (`java.io.Serializable info`) is passed. Each bean has to register itself to the timer service on its own with the `createTimer` method.

The advantage to other custom made implementations of a timer is that the fundamental rules to beans apply: A timer is persistently stored and recovered after a server crash and if a transaction is rolled back in which a timer was created, the timer creation is rolled back as well.

Drawbacks of the implementations is the quite simple structure compared to cron, and that only the initiator of the timer itself, the bean, is able cancel the timer.

4.4 The enhanced Message Driven Bean

As it was mentioned in the service endpoint interface description above, the interoperability to other systems is an important feature of EJBs. When message driven beans emerged, the following was added to the EJB 2.0 specification:

“While the EJB 2.0 specification requires support for only JMS-based messaging, a future goal of the message-driven bean model is to provide support for other types of messaging in addition to JMS [...]”[SUN01]

4.4.1 Previous Solutions and Workarounds

The Java Message Service (JMS) was the only way to access the beans. Since the JMS system was widely accepted, most vendors did not implement proprietary extensions to the

```

1  public class InvoiceBean implements javax.ejb.TimedObject,
2      javax.ejb.EntityBean{
3      ...
4      public void ejbPostCreate(){
5          timerService timerService = ejbContext.getTimerService();
6          timerService.createTimer(expirationDate, Expiration);
7      }
8      public void ejbTimeout(Timer timer){
9          String info = (String) timer.getInfo();
10         if (info != null &&
11             info.equals("Expiration")){
12             setStatus("Expired");
13         }
14     }
15 }
    
```

Figure 5: An implementation of the Timer Interface. At creation of an invoice, the expiration date is set. At expiration the timeout method is called and the entity bean field status is set to "Expired"

specification which made the MDBs available to other programming languages. However, third parties offer tools for a great range of programming languages to facilitate JMS messages from a non-Java client (see appendix A.1).

4.4.2 The new Messages to access Message Driven Beans

EJB 2.1 does not explicitly mention a special alternative communication technology to MDBs. A common interpretation according to [SHI03] is the usage of the Java API for XML Messaging (JAXM), which is basically a SOAP implementation that is more flexible but also operating on a lower level than JAX-RPC. The main difference is that JAXM does not imply *mapping* of messages to EJB methods, but it sends XML *documents* (SOAPmessage) that can be interpreted by the `onMessage` method in a custom way.

Depending on the messaging standard and the conversation type, the MDB has to implement the following interfaces:

- **Asynchronous / JMS:** `javax.jms.MessageListener` interface
- **Asynchronous / JAXM:** `javax.xml.messaging.OnewayListener`
- **Synchronous / JAXM:** `javax.xml.messaging ReqRespListener`

The synchronous JAXM does not match to the justification of message driven beans ([SUN03]), as they are intended to model asynchronous business logic. Having said that, it is worth to mention the benefit of this technology concurrent the SEI of stateless session beans, since “compared to the RPC style web services, document-based web services make a better case for synchronous exchange of documents” [PAT03]

4.5 New Features in EJB 3.0

Currently the JSR No. 220 for the Enterprise JavaBeans 3.0 specification (EJB 3.0) is subject to be reviewed and to be accepted by the supporting companies. The main objective is rather optimizing the development than to add new technical features. This reflects the calls for an easier development and more functionality in the EJB container.

To reduce the complexity from the developer point of view, the following issues are in [JCP04]:

- Introduction of metadata in the bean class to reduce or abolish the need of a deployment descriptor and the interfaces
- Specification of programmatic defaults, including for the just mentioned metadata, to reduce the need for the developer to specify common, expected behaviors and requirements on the EJB container
- More enhancement of EJB QL elements
- Provision of utility classes that implement common functionalities
- Easier, regular Java class like structure of stateless session beans

5 Conclusion

In its fourth release, the Enterprise JavaBeans technology shows a good path to maturity and to that what was initially defined as one of its greatest features: the disregard of programming, and the focus on the business logic.

In EJB 2.0 the introduction of container managed persistence has been considered as a giant leap towards this goal. And this is constantly continued. As the versioning implies,

EJB 2.1 does only add little more functionality to EJB 2.0, since it specifies features that were either already available as proprietary extensions or could be implemented as workarounds.

The amount of features and flexibility are in most cases anti-proportional to the learning effort and the performance of an application. This is also the case in EJB and to be exact: the nature of Java. In the enterprise application environment there is an interesting competition to be expected between EJB, Microsoft's .NET, and lightweight components that only implement a subset of what is EJB capable to do but fit to customers' needs. With EJB 2.1 and EJB 3.0 Java is a good contender in that fray and promises a good development.

A Appendix

A.1 Alternative Clients for JMS

- **Active JMS:** Python, Visual Basic and C# – <http://active-jms.sourceforge.net>
- **JMS Courier:** C++ – <http://www.codemesh.com/en/JMSCourier.html>

References

- [SUN01] Sun Microsystems Inc., *Enterprise JavaBeans Specification, Version 2.0*, sun.com
– Sun Microsystems Inc., 2001
- [SUN03] Sun Microsystems Inc., *Enterprise JavaBeans Specification, Version 2.1*, sun.com
– Sun Microsystems Inc., 2003
- [SHI03] Sang Shin, *J2EE 1.4 & Web Services*, sun.com – Sun Microsystems Inc., 2003
- [IBM03] Ueli Wahli et al, *EJB 2.0 Development with WebSphere Studio Application Developer*, International Business Machines Corp., 2003
- [LAN03] Thorsten Langner, *Web Services mit Java*, Markt+Technik, 2003
- [BIE02] Adam Bien, *J2EE Patterns*, Addison-Wesley, 2002
- [ARN01] G. Leslie Arnold, *Distinguishing WebSphere clones*, ibm.com – International Business Machines Corp., 2001
- [LOM01] Dean Lombardo, *The Hype Is Right: Web Services Will Deliver Immediate Benefits*, gartner.com – Gartner, Inc., 2001
- [PAT03] Nikhil Patil, *Developing E-Business Interactions with JAXM*, ONJava.com – OReilly Media, Inc., 2003
- [JCP04] Java Community Process, *JSR 220: Enterprise JavaBeans™ 3.0*, jcp.org - Sun Microsystems Inc., 2004