

OCP JavaEE 6 EJB Developer Study Notes

by

Ivan A Krizsan

Version: February 17, 2011

Copyright 2010-2011 Ivan A Krizsan. All Rights Reserved.

Table of Contents

Table of Contents.....	2
Purpose	9
Structure.....	9
Licensing	9
Disclaimers	9
Prerequisites.....	9
1. Plan the Development of a Business Service Model.....	11
1.1.Reasons for Using EJBs.....	11
1.2.Reasons for Not Using EJBs.....	11
1.3.EJB Programming Restrictions.....	12
1.4.EJB 3.1 Lite vs Full EJB 3.1 API.....	14
1.5.Choosing the Type of EJB.....	15
1.6.Session Beans - Stateful, Stateless or Singleton?.....	15
1.6.1.Stateful Session Beans.....	15
1.6.2.Stateless Session Beans.....	16
1.6.3.Singleton Session Beans.....	16
1.7.Designing the Client View of Session Beans.....	16
1.8.Remote or Local Access of Session Beans.....	17
1.8.1.Choosing Client Access Type.....	17
1.8.2.Access Type and Parameters.....	18
1.9.Message Driven Beans.....	19
1.9.1.Queue or Topic.....	19
2. Create a Stateful Session Bean.....	21
2.1.Create a Stateful Session Bean with a Local No-interface View.....	21
2.2.Create a Stateful Session Bean with a Local Business Interface.....	25
2.3.Create a Stateful Session Bean with a Remote Business Interface.....	27
2.4.Create a Stateful Session Bean without Annotations.....	29
2.5.Stateful Session Bean Life-Cycle.....	31
2.6.Operations Allowed in Methods of a Stateful Session Bean.....	32
3. Create a Stateless Session Bean.....	34
3.1.Create a Stateless Session Bean with a Local No-interface View.....	34
3.2.Create a Stateless Session Bean with a Local Business Interface.....	37
3.3.Create a Stateless Session Bean with a Remote Business Interface.....	39
3.4.Create a Stateless Session Bean without Annotations.....	41
3.5.Stateless Session Bean Life-Cycle.....	43
3.6.Operations Allowed in Methods of a Stateless Session Bean.....	44
4. Create a Singleton Session Bean.....	47
4.1.Basic Singleton Session Bean Example.....	48
4.2.Singleton Session Bean Initialization.....	52
4.2.1.Singleton Session Bean Eager Initialization.....	52
4.3.Singleton Session Bean Destruction.....	53
4.4.Singleton Session Bean Initialization and Destruction Dependencies.....	54
4.5.Transactions and Singleton Session Bean Creation and Destruction.....	55
4.6.Singleton Session Bean Concurrency.....	56
4.6.1.Basic Singleton Session Bean Concurrency Example.....	56
4.6.2Concurrency Management.....	61
The @Lock Annotation.....	62
4.7.Singleton Session Bean Life-Cycle.....	66

4.8.Operations Allowed in Methods of a Singleton Session Bean.....	67
5.Create a Session Bean Client.....	69
5.1.Local Session Bean Clients.....	69
5.1.1.Local Session Bean Client with Dependency Injection.....	70
5.1.2.Portable JNDI Names for Session Beans.....	73
5.1.3.Local Session Bean Client with JNDI Lookup.....	75
5.2.Remote Session Bean Clients.....	77
5.2.1.Create the Remotely Accessible EJB.....	77
5.2.2.Remote Java EE Client with Dependency Injection.....	79
5.2.3.Remote Java EE Client with JNDI Lookup.....	81
5.2.4.Remote Java SE Client with JNDI Lookup.....	81
5.2.5.Remote Java SE Client with Dependency Injection.....	83
5.3.Client View of EJB 3.1 Session Beans.....	84
5.3.1.Obtaining References to Session Beans.....	84
5.3.2.Views of Session Beans.....	85
5.3.3.Method Parameters.....	86
5.3.4.Session Bean Identity.....	87
5.4.Exceptions from EJBs.....	92
5.4.1.EJB Clients, Transactions and Exceptions.....	92
5.4.2.Application Exceptions.....	92
5.4.3.System Exceptions.....	92
6.Create Interceptors for an EJB.....	94
6.1.Interceptor Example.....	94
6.1.1.A Session Bean and its Client.....	94
6.1.2.Adding a Logging Interceptor Using Annotations.....	97
6.1.3.Adding a Default Interceptor.....	99
6.1.4.Adding an Interceptor Superclass.....	101
6.1.5.Adding a Life-Cycle Event Interceptor.....	102
6.1.6.Adding a Timeout Method Interceptor.....	104
6.2.Interceptor Types.....	106
6.2.1.Business Method Interceptors.....	106
6.2.2.Timeout Method Interceptors.....	107
6.2.3.Life-Cycle Event Interceptors.....	108
6.3.Interceptor Programming Restrictions.....	109
6.4.Interceptor Life-Cycle and Concurrency Behaviour.....	109
6.5.Interceptors and Exceptions.....	109
6.6.Interceptors and Transactions.....	109
6.7.Interceptors with Annotations.....	110
6.7.1.The @AroundInvoke Annotation.....	110
6.7.2.The @AroundTimeout Annotation.....	110
6.7.3.The @ExcludeClassInterceptors Annotation.....	110
6.7.4.The @ExcludeDefaultInterceptors Annotation.....	110
6.7.5.The @Interceptor Annotation.....	110
6.7.6.The @Interceptors Annotation.....	111
6.7.7.The @InterceptorBinding Annotation.....	111
6.8.Interceptors in the Deployment Descriptor.....	112
6.9.Default Interceptors.....	114
6.10.Interceptor Order of Invocation.....	114
7.Modify a Session Bean to Become Asynchronous.....	115
7.1.Asynchronous Session Bean Example Program.....	115

7.2. Asynchronous Methods in Session Beans	120
7.2.1. Denote Method Asynchronicity	120
7.2.2. Requirements on Methods to be Made Asynchronous	121
7.2.3. Asynchronous Methods and Transactions	121
7.2.4. Asynchronous Methods and Security	121
7.3. Client View of Asynchronous Session Beans	122
7.3.1. Asynchronous Methods and Exceptions	122
7.3.2. Asynchronous Method Return Values	122
8. Create a Message Driven Bean	124
8.1. Common GlassFish Example Preparations	124
8.1.1. Setting Up Physical Destinations	125
8.1.2. Setting Up Connection Factories	125
8.1.3. Setting Up Destination Resources	126
8.2. Topic Message Driven Bean Example	127
8.2.1. Creating the Project and the Message Bean	127
8.2.2. Creating the Message Driven Bean	128
8.2.3. Configuring the ejb-jar.xml Deployment Descriptor	130
8.2.4. Creating a Message Producing Servlet	131
8.2.5. Running the Example	133
8.3. Queue Message Driven Example	134
8.3.1. Creating the Project and the Message Bean	134
8.3.2. Creating the Message Driven Bean	134
8.3.3. Configuring the ejb-jar.xml Deployment Descriptor	136
8.3.4. Creating a Message Producing Servlet	137
8.3.5. Running the Example	140
8.4. Characteristics of Message Driven Beans	141
8.5. Dependency Injection in Message Driven Beans	141
8.6. Message Driven Beans and the Container	142
8.7. Messaging Configuration	143
8.7.1. Messaging Configuration using Annotations	143
8.7.2. Messaging Configuration in the Deployment Descriptor	144
8.8. Message Driven Beans in the Deployment Descriptor	145
8.9. Message Driven Beans and Exceptions	146
8.9.1. Message Listener Method Exception Handling	147
8.9.2. PostConstruct and PreDestroy Method Exception Handling	148
8.9.3. Timeout Callback Method Exception Handling	148
8.9.4. Exception Handling in Other Callback Methods	148
8.10. Message Driven Beans and Security	148
8.11. Message Driven Beans and Transactions	149
8.11.1. Transaction Attributes	149
8.12. Message Driven Bean Life-Cycle	150
8.13. Operations Allowed in Methods of Message Driven Beans	151
9. Create a Client that Sends Messages to a JMS Queue	153
9.1. Creating a Java EE Client that Sends Messages to a JMS Queue	153
9.2. Creating a Java SE Client that Sends Messages to a JMS Queue	156
10. Add Transaction Demarcation Information to an EJB	160
10.1. Transaction Propagation	161
10.2. Transaction Context of EJB Methods	163
10.3. Bean-Managed Transaction Demarcation	163
10.3.1. Example of an EJB with Bean Managed Transactions	166

10.4.	Container-Managed Transaction Demarcation	169
10.4.1.	Session Synchronization	170
10.4.2.	Example of an EJB with Container Managed Transactions	171
10.4.3.	Transaction Attributes	173
10.5.	JMS Messaging and Transactions	177
11.	Add Security Checks to Business Logic	178
11.1.	EJB Security Examples Overview	179
11.2.	Create Users in GlassFish	180
11.3.	Annotations EJB Security Example	180
11.3.1.	Common Session Bean Superclass	180
11.3.2.	First Session Bean	182
11.3.3.	EJB Client Servlet	183
11.3.4.	Mapping Security Roles in GlassFish	184
11.3.5.	Running the Example Program – First Time	185
11.3.6.	Adding a Second Session Bean	186
11.3.7.	Running the Example Program – Second Time	188
11.3.8.	Changing the Invocation Security Role	190
11.4.	Deployment Descriptor EJB Security Example	193
11.5.	Programmatic Security	198
11.6.	Declarative Security	201
11.6.1.	Declarative Security Using Annotations	201
11.6.2.	Declarative Security Using the Deployment Descriptor	203
12.	Schedule Execution of Business Logic	208
12.1.	EJB Business Logic Scheduling Example	208
12.1.1.	Create the Project	208
12.1.2.	Create the Scheduled Stateless Session Bean	208
12.1.3.	Running the Example Program – Declarative Scheduling	211
12.1.4.	Programmatic Timer Creation Example	212
12.1.5.	Running the Example Program – Programmatic Scheduling	215
12.1.6.	Rolling Back Programmatically Scheduled Timers	216
12.2.	Timeout Callback Methods	218
12.3.	The Timer Interface	219
12.3.1.	Timer Life-Cycle	220
12.3.2.	Timers and Persistence	220
12.4.	Declarative Scheduling	221
12.4.1.	Declarative Scheduling with Annotations	221
12.4.2.	Declarative Scheduling in the Deployment Descriptor	223
12.5.	Programmatic Scheduling	226
12.5.1.	Timeout Callback Methods for Programmatically Scheduled Timers	226
12.5.2.	Creating Calendar Timers	228
12.5.3.	Creating Interval Timers	229
12.5.4.	Creating Single-Action Timers	230
12.5.5.	Timer Configuration	230
12.6.	Timers and Transactions	231
13.	Package and Deploy EJB Applications	232
13.1.	Java EE Application Assembly	232
13.2.	Packaging Options	233
13.2.1.	EJB-JAR Files	234
13.2.2.	WAR Files	236
13.2.3.	JAR Files	238

13.2.4. EAR Files	238
13.2.5. EJB-Client JAR Files	239
13.3. Packaging Requirements	240
13.4. Module Visibility	240
13.5. The ejb-jar.xml Deployment Descriptor	240
14. Session Beans	243
14.1. Session Beans in the Deployment Descriptor	244
14.2. Session Bean Types	246
14.3. Session Bean Metadata	247
14.3.1. Session Bean Type Metadata	247
14.3.2. Session Bean View Metadata	248
14.3.3. Session Bean Life-Cycle Annotations	249
14.4. Session Bean Concurrency	250
14.4.1. Stateful Session Bean Concurrency	250
14.4.2. Stateless Session Bean Concurrency	253
14.5. Session Bean Initialization	253
15. EJB Dependency Injection and Resource Lookup	254
15.1. The @Resource Annotation	255
15.2. The @EJB Annotation	256
15.2.1. Declaring JNDI Names with the @EJB Annotation	257
15.2.2. Setter Method Injection with the @EJB Annotation	257
15.2.3. Instance Field Injection with the @EJB Annotation	257
15.3. Injection Using the Deployment Descriptor	258
15.4. Programmatic Lookup of References from EJBs	259
15.4.1. Lookup Using InitialContext	259
15.4.2. Lookup Using EJBContext	260
15.5. Simple Environment Entries	261
15.5.1. Simple Environment Entry Injection Using Annotations	262
15.5.2. Defining Simple Environment Entries in the Deployment Descriptor	262
15.5.3. Programmatic Lookup of Simple Environment Entries	262
15.6. EJB References	263
15.6.1. Reference Injection Using Annotations	263
15.6.2. Defining EJB References in the Deployment Descriptor	264
15.6.3. Programmatic Lookup of EJB References	266
15.6.4. Overriding an @EJB Annotation with the Deployment Descriptor	267
15.7. Web Service References	268
15.8. Resource Manager Connection Factory References	268
15.8.1. Reference Injection Using Annotations	268
15.8.2. Defining Resource Manager Connection Factory References in the Deployment Descriptor	269
15.8.3. Programmatic Access of Resource Manager Connection Factory References	271
15.9. Resource Environment References	272
15.9.1. Injecting a Resource Environment Reference Using Annotations	272
15.9.2. Defining Resource Environment References in the Deployment Descriptor	273
15.9.3. Programmatically Retrieving Resource Environment References	275
15.10. Message Destination References	276
15.10.1. Configuring Message Destination References in the Deployment Descriptor	276
15.10.2. Injecting a Message Destination Reference Using Annotations	279
15.10.3. Programmatically Retrieving Message Destination References	279
15.11. Persistence Unit References	280

15.11.1 Persistence Unit References in the Deployment Descriptor.....	280
15.11.2 Persistence Unit Reference Injection Using Annotations.....	281
15.11.3 Programmatic Retrieval of Persistence Unit References.....	282
15.12 Persistence Context References.....	282
15.12.1 Persistence Context References in the Deployment Descriptor.....	283
15.12.2 Persistence Context Reference Injection Using Annotations.....	285
15.13 UserTransaction Interface.....	286
15.13.1 UserTransaction Object References in the Deployment Descriptor.....	286
15.13.2 UserTransaction Reference Injection Using Annotations.....	287
15.13.3 Programmatic Retrieval of UserTransaction Objects.....	287
15.14 ORB References.....	287
15.14.1 ORB References in the Deployment Descriptor.....	287
15.14.2 ORB Reference Injection Using Annotations.....	288
15.14.3 Programmatic Retrieval of ORB References.....	288
15.15 TimerService References.....	289
15.15.1 TimerService References in the Deployment Descriptor.....	289
15.15.2 TimerService Reference Injection Using Annotations.....	289
15.15.3 Programmatic Retrieval of TimerService References.....	290
15.16 EJBContext References.....	290
15.16.1 EJBContext References in the Deployment Descriptor.....	290
15.16.2 EJBContext Reference Injection Using Annotations.....	291
15.16.3 Programmatic Retrieval of EJBContext References.....	291
16. EJBs and Exceptions.....	292
16.1 Application Exceptions.....	328
16.1.1 Application Exceptions and Transactions.....	328
16.2 System Exceptions.....	329
16.2.1 System Exceptions and Transactions.....	329
16.3 Session Bean Exception Handling.....	329
16.3.1 Business Interface and No-Interface View Method Exception Handling.....	330
16.3.2 Web Service Client View and EJB 2.1 Client View Method Exception Handling.....	331
16.3.3 PostConstruct and PreDestroy Method Exception Handling.....	332
16.4 Timeout Callback Method Exception Handling.....	333
16.5 Exception Handling in Other Callback Methods.....	335
Appendix A – Setting Up a Dynamic Web Project in Eclipse.....	337
Appendix B – Setting Up an EJB Project in Eclipse.....	339
Appendix C – Message Destination Link Example.....	341
Listener Facade EJB.....	342
First Topic Listener EJB.....	344
Second Topic Listener EJB.....	345
EJB Deployment Descriptor.....	346
Appendix D – Configuring Messaging in GlassFish v3.....	348
Create a JMS Physical Destination.....	348
Create a JMS Connection Factory.....	349
Create a JMS Destination Resource.....	350
Appendix E – Defining Resource Environment References in GlassFish v3.....	351
Appendix F – Setting Up JDBC Resources in GlassFish v3.....	352
Create a JDBC Connection Pool.....	352
Create a JDBC Resource.....	354
Configuring Persistence Settings in Web Applications.....	355
Appendix G – GlassFish Security Configuration.....	356

<u>Basic Concepts</u>	356
<u>Creating Users in GlassFish</u>	358
<u>Security Role Mapping</u>	360
<u>Appendix H – Embeddable EJB Container Example</u>	362
<u>Creating the Example Project</u>	362
<u>Updating the Maven pom.xml File</u>	365
<u>Creating the EJB Class</u>	366
<u>Creating the Client Class</u>	366
<u>Running the Example Program in Eclipse</u>	368
<u>Creating the Standalone Application</u>	369
<u>Running the Standalone Application</u>	371

Purpose

These are the notes I made during the preparations for the OCP Java EE 6 EJB Developer certification. The focus of this book is development of Enterprise Java Beans according to the EJB 3.1 (JSR-318) standard.

While it is possible to use, for instance, enterprise Java beans adhering to the EJB 2.1 standard in an EJB 3.1 container, I have deemed this as being out of the scope of this book. Also, the use of JAX-RPC in connection to EJBs has not been considered.

Structure

The structure of this book is as follows:

- Chapters 1 to 13 maps to the certification objectives.
- Chapters 14 to 17 contains common information that, for instance, applies to multiple types of enterprise beans.
- The appendices contain examples related to development of EJBs.
For instance, how to set up the Eclipse development environment, how to configure different aspects of the GlassFish application server and additional examples.

Licensing

This book is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 3.0](#) license. In short this means that:

- You may share this book with others.
- You may not use this book for commercial purposes.
- You may not create derivative works from this book.

Disclaimers

Though I have done my best to avoid it, this book might contain errors. I cannot be held responsible for any effects caused, directly or indirectly, by the information in this book – you are using it on your own risk.

I cannot make any guarantees concerning the completeness of the information contained in this book – if you are preparing for the certification, please do consult additional sources.

Submitting any suggestions, or similar, the information submitted becomes my property and you give me the right to use the information in whatever way I find suitable, without compensating you in any way.

All trademarks are properties of their respective owner and do not imply endorsement of any kind. This book has been written in my spare time and has no connection whatsoever with my employer.

Prerequisites

When developing the examples in this book, I have been using the GlassFish v3 application server and Eclipse Helios as the IDE.

The version of Eclipse I worked with did not find a GlassFish plugin when adding a new server, instead I had to install the GlassFish plugin using the Install New Software... menu alternative in Eclipse using the following update site: <http://download.java.net/glassfish/eclipse/helios>

Part One

Certification Topics

1. Plan the Development of a Business Service Model

References:

JavaEE 6 Tutorial, part IV
EJB 3.1 Specification, chapter 21.

1.1. Reasons for Using EJBs

Some reasons for choosing to implement business services as EJBs are:

- Scalability is required.
EJBs can be distributed across multiple computers while their location remain transparent to clients.
- Transactions are required.
- Different kinds of clients will access the application.
EJBs can be accessed in multiple ways; local or remote EJBs, SOAP or REST web service.
- Management of concurrency is desired.
The EJB container can manage concurrent access to an EJB, allowing only a single item at a time to use the EJB.
- Security is required.
EJBs offer a security model that enables security to be configured declaratively, without affecting the implementation. Programmatic security is also available.
- Portability across different application servers is desired.

Distributed deployment: If required, there must be a remote view of the EJB or the EJB must be exposed as a web service, either RESTful or SOAP. An EJB can also be exposed as a message consumer.

Exposing an EJB as a web service does not require any modifications to the code implementing the business logic in the EJB.

We will look closer at transactions with EJBs in [chapter 10](#) and security with EJBs in [chapter 11](#).

1.2. Reasons for Not Using EJBs

References: EJB 3.1 Specification, chapter 21 and 22.

The following reasons exists that may cause EJBs to be a less optimal solution:

- Use of an EJB container is not desired.
- An EJB container is not available.
- One or more of the EJB programming restrictions cannot be respected.
For details on these programming restrictions, please see next section.

If the full EJB API is considered too heavyweight, there is still the EJB Lite alternative, which is a subset of the EJB 3.1 API. EJB Lite still requires an EJB container.

There is also an embeddable EJB 3.1 container, which is only required to support the EJB Lite subset of the EJB 3.1 API. The embeddable container enables use of the EJB programming model in, for instance, desktop applications.

1.3. EJB Programming Restrictions

References: EJB 3.1 Specification, chapter 21, section 2.

There are a number of programming restrictions that an EJB developer must follow, in order to ensure portability across different containers adhering to the EJB 3.1 standard. The following restrictions apply to both EJBs as well as EJB interceptors:

Programming Restriction	Motivation
An EJB must not use writeable static fields.	Ensure consistency in a distributed environment.
An EJB must not use thread synchronization primitives to synchronize execution of multiple instances, except if the EJB is a Singleton session bean with bean managed concurrency.	Will not work in a distributed environment.
An EJB must not use AWT functionality to generate output to a display or obtain input from a keyboard.	Servers commonly do not allow access to keyboard or screen from an application program.
An EJB must not access files and directories in the file system directly.	Business components should use a resource manager API, such as JDBC, to access persistent data.
An EJB must not act as a network server, listen on a socket, accept connections on a socket or use a socket for multicast. It may act as a network socket client.	Conflicts with the duties of the application server.
An EJB must not attempt to circumvent the security rules of the Java programming language when accessing information about other classes or when accessing other classes.	Could compromise security.
An EJB must not attempt to set the socket factory used by <i>ServerSocket</i> and <i>Socket</i> or attempt to set the stream handler factory used by URL.	Could compromise security. Interferes with the container's ability to manage the runtime environment.
An EJB must not attempt to manage threads or thread groups.	Interferes with the container's ability to manage the runtime environment.
An EJB must not attempt to read or write a file descriptor directly.	Could compromise security.
An EJB must not attempt to obtain the security policy information for a particular code source.	Could compromise security.
An EJB must not attempt to load a native library.	Could compromise security.
An EJB must not attempt to circumvent the rules of the Java programming language when accessing packages and classes.	Could compromise security.
An EJB must not attempt to define a class in a package.	Could compromise security.
An EJB must not attempt to access or modify security configuration objects (<i>Policy</i> , <i>Security</i> ,	Could compromise security.

<i>Provider, Signer and Identity).</i>	
An EJB must not attempt to use the subclass or object substitution features of the Java Serialization Protocol.	Could compromise security.
An EJB must not pass <i>this</i> as a method argument or result. Instead, use the result from one of the <i>getXXXObject</i> methods in the <i>SessionContext</i> or <i>EntityContext</i> classes.	The container may use a proxy mechanism to, for instance, control concurrent access of the EJB instance. The reference injected into clients of the EJB is a reference to the proxy and not to an instance of the EJB implementation class.

1.4. EJB 3.1 Lite vs Full EJB 3.1 API

References: EJB 3.1 Specification, chapter 21, section 1.

EJB Lite is a subset of the full EJB 3.1 API, allowing container developers to implement a EJB container of reduced size and complexity while still being compliant with the EJB 3.1 specification (at least partially).

For example, the embeddable EJB 3.1 container is, as before, only required to support the EJB Lite API.

The following table lists features of the EJB 3.1 API and indicates their availability in the full EJB 3.1 API and the EJB 3.1 Lite API:

Feature	Full EJB 3.1 API	EJB 3.1 Lite API
Java Persistence 2.0	Available	Available
Session beans local/no interface client view.	Available	Available
Session beans 3.0 remote client view.	Available	Not available
Session beans 2.x remote client view.	Available	Not available
Session beans exposed as JAX-WS web service endpoints.	Available	Not available
Session beans exposed as JAX-RPC web service endpoints.	Available	Not available
EJB timer service.	Available	Not available
Asynchronous invocation of session beans.	Available	Not available
Interceptors.	Available	Available
RMI-IIOP interoperability.	Available	Not available
Bean and container managed transactions.	Available	Available
Declarative and programmatic security.	Available	Available
Embeddable API.	Available, but embeddable container only required to support EJB 3.1 Lite.	Available

1.5. Choosing the Type of EJB

References: JavaEE Tutorial, chapter 14.

There are two types of EJBs:

- Session beans.
- Message-driven beans.

Session beans are components that contains business logic that can be invoked over local, remote or web service client views.

Message-driven beans are components that, upon receiving an asynchronous message, executes some business logic. Usually message-driven beans listen for JMS messages.

Different reasons for choosing to use session-driven beans are discussed in the [next](#) section.

Reasons for using message-driven beans are discussed in a [subsequent](#) section.

1.6. Session Beans - Stateful, Stateless or Singleton?

There are three different types of session beans; stateful, stateless and singleton. The following sections summarizes some characteristics of the different types and gives reasons for choosing a certain type.

1.6.1. Stateful Session Beans

Stateful session beans will retain state during the conversation with one single client.

Reasons for choosing stateful session beans are:

- The EJB needs to retain state related to a client across multiple method invocations.
- The EJB mediates between a client and other components of the application, presenting a simplified view to the client.
- Behind the scenes, the EJB manages the workflow of several EJBs.

1.6.2. Stateless Session Beans

Stateless session beans may exist in a pool from which the container may choose any bean to serve a request. A stateless session bean may maintain state in the form of instance variables, but the state cannot be associated with a specific client. The relation between an instance of a stateless session bean and a client only lasts for one single method invocation.

Reasons for choosing stateless session beans are:

- The EJB does not need to retain client-specific state.
- Better scalability is desired.
No need to maintain state and multiple instances being available to server requests gives better performance.
- Better performance is desired.
- The functionality of the EJB is to be exposed as a web service.
- The EJB performs one or more generic tasks, for instance sending an email, that can be finished during one single method invocation.

1.6.3. Singleton Session Beans

Singleton session beans are session beans for which each type there is only one single instance during the lifetime of an application.

Reasons for choosing singleton session beans are:

- Some state needs to be shared throughout the application.
- One single EJB is to be accessed by multiple threads concurrently.
- The application requires an EJB to perform startup or shutdown task(s).
- The functionality of the EJB is to be exposed as a web service.

Singleton Session Bean Concurrent Access

Singleton session beans can be configured to allow multiple threads to execute in an instance concurrently. Under such circumstances, special care have to be taken when designing the singleton session bean in order to, for instance, protect resources that must not be accessed concurrently.

For more details, please refer to [chapter four](#).

1.7. Designing the Client View of Session Beans

Common for all types of session beans is that clients will see a business interface or a no-interface view of the session bean. The latter being comprised by the public methods in the class implementing the session bean.

Care should be taken designing business interfaces and no-interface views, since any future changes in these will require corresponding change in all clients. Properly designed interfaces and no-interface views will also help shielding clients from complexities in the EJB tier.

1.8. Remote or Local Access of Session Beans

References: EJB 3.1 Specification, section 3.2.

One of the first design-decisions that has to be taken when designing a Java EE application is which type of clients you allow to access a session bean; local, remote or web service.

1.8.1. Choosing Client Access Type

As before, the following three types access exist for session beans:

- Local Access

The session beans is accessed from within the same application.

Possible clients are web components and other EJBs.

- Remote Access

The session bean is accessed from the same or from another JVM. The JVM in which the client executes may be deployed on a different computer.

Note that remote access of an EJB also is allowed for a client running in the same JVM as the EJB.

Possible clients are web components, other EJBs and application clients.

- Web Service Access

The session bean is exposed as a web service, SOAP or RESTful, allowing any web service client to access the EJB. Such clients may, for instance, be implemented in another language. Web service access is mainly chosen to provide interoperability with external systems.

It should be noted that local access does not exclude remote access of an EJB. An EJB may allow both local and remote access.

Whether to select local or remote access of an EJB may depend on one or more of the following factors:

- Tight or loose coupling between an EJB and its clients / EJB operation granularity.
If the client closely depends on the EJB to perform its duties and often invoke the EJB, then it is more tightly coupled to the EJB and will benefit from local access, which will improve performance.
- Type of clients.
If the EJB is accessed by application clients, then it needs to provide remote access, since such clients almost always run on other machines.
Web components and other EJBs may access the EJB locally or remotely, depending on how the application is distributed.
- Component distribution.
When using JavaEE, server-side components may be distributed on several different nodes to facilitate better scalability. When distributing web component on one machine and the EJBs implementing the business functionality on another machine, the EJBs need to provide remote access.
- Location independence.
An EJB providing a remote view can be deployed in the same container as the client or on a remote container, while an EJB providing only a local view are only accessible by clients in the same application as the EJB. An EJB with a remote view is more flexible in that it may be moved as desired.
- Parameters and return values must be serializable for EJBs with a remote view.
If passing large amounts of data between an EJB and its clients, serialization of the data may cause overhead. With local access to an EJB, parameters are passed by reference and serialization overhead is thus avoided – the EJB can operate on the data and the client, holding a reference to the same data, will see the modifications.
- Additional error cases due to remote communication.
- Performance
Remote calls of EJBs always incur a performance penalty, since it means access over some network. However, distribution of components on several nodes may improve the overall performance of the system. Measuring the performance of a system in different configurations is the best way of learning about the characteristics of a particular application.

1.8.2. Access Type and Parameters

Depending on which type of clients will access an EJB, as discussed in the previous session, some care may also need to be taken when considering parameters and return data of EJB methods.

Please refer to the section on [Method Parameters](#) in the chapter on [Session Beans](#) for details.

An EJB that is to provide a remote view to clients are typically designed as to be more coarse-grained with fewer interactions between client and EJB, while an EJB with a local view may be designed as to be more fine-grained.

1.9. Message Driven Beans

Message-driven beans listen for messages, usually JMS messages, and, upon receipt of a message, executes some business function. Message-driven EJBs share the following characteristics with stateless session beans:

- Retains no client-specific state.
- The container can assign any instance of an EJB to serve a client request.
Instances may be pooled by the container.
- A single EJB instance can process requests from multiple clients.
Though at most one single thread at a time can execute on a message-driven bean instance.

Reasons for choosing message-driven beans are:

- Business functionality is to be invoked by asynchronous messaging.
- A long-running business process is to be performed.
Waiting for a response from such a process may hold up the client for a longer period of time, which may not be desirable. Synchronous invocation ties up resources for the duration of the call, which may cause degradation in performance and scalability if many, long-running, tasks are to be performed.
- More loose coupling between client and the EJB is desired.
Clients does not “see” the EJB, instead it just sends a message to some message listener.

1.9.1. Queue or Topic

There are two different messaging models that can be employed by queues that message driven beans are listening to; queue or topic.

The queue messaging model is also referred to as the point-to-point messaging model, while the topic messaging model is also referred to as the publish-and-subscribe or pub/sub messaging model.

Common for both these messaging models are:

- Messages from producer to consumer(s) are exchanged through virtual channels.
The type of virtual channel depends on the messaging model.
- Producers and consumers are decoupled.
Producers and consumers of messages does not know about each other, apart from the fact that there are messages from the former delivered to the latter.
- Produces and consumers of messages can be dynamically added at runtime.
- Messages are pushed to message driven beans regardless of the messaging model.
Regular JMS message consumers can pull messages from a queue but message driven beans does not have this option.

Characteristic of the point-to-point messaging model (queues) are:

- The virtual channel used to exchange messages is called a queue.
- A message sent to a queue is delivered to exactly one consumer.
- Messages are delivered in the order in which they were produced.
Using different message priorities may affect the order in which messages are delivered.
- Messages sent to a queue can be persistent or non-persistent.
Persistent messages are retained in the case of JMS provider failures.
- A message sent to a queue may be assigned a priority.

- A message sent to a queue may be assigned an expiration time.

Characteristics of the publish-and-subscribe messaging model (topics) are:

- The virtual channel used to exchange messages is called a topic.
- A message sent to a topic is delivered to all consumers of the topic.
Message filtering may cause some consumers not to receive some messages.

When To Use Queues

Queues are suitable when an application want to, for instance, divide work among a number of workers - the workers all listen to the queue but each message is only received by one of the workers. The producer knows that the unit of work will eventually be received and processed by one of the workers.

Queues are also suitable for reliable asynchronous communication between two parties; the first party sends messages to a queue which are later delivered to the other party. The other party replies by sending another message to another queue, which the first party is a consumer of.

When To Use Topics

Topics are suitable when multiple consumers are to receive a copy of each message produced. Commonly topics are used when the producer of some information want to give the opportunity of interested parties to receive the information. The producer commonly does not care about who receives the information.

A commonly used example of when to use a topic is in a stock quote application.

2. Create a Stateful Session Bean

References: EJB 3.1 Specification, chapter 4.

In this chapter we look at how to create different kinds of stateful session beans.

What characterizes a stateful session bean is that it is tied to one single client and only serves requests from that client. The stateful session bean may, as implied by the name, hold client-related state. Stateful session beans may also, as we will see when looking at the life-cycle of stateful session beans, be temporarily taken out of service (passivated) and later, when need arises, taken back into service (activated).

Detailed information applying to session beans can be found in [chapter 14](#). Clients to session beans will be discussed in [chapter five](#).

2.1. Create a Stateful Session Bean with a Local No-interface View

The stateful session bean has one single method that creates a greeting. The EJB is developed in a web project, since the session bean will only have a local (no-interface) view and can thus only be accessed by clients in the same application. In this example we'll also create a client to the session bean.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “StatefulSession1Web”.
- In the package *com.ivan.scbcd6*, create the session bean implementation class implemented as this:

```
package com.ivan.scbcd6;

import java.util.Date;
import java.util.concurrent.TimeUnit;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.LocalBean;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.ejb.StatefulTimeout;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

/**
 * Simplest possible stateful session bean exposing a local, no-interface view.
 */
@Stateful
@LocalBean
@StatefulTimeout(value=10, unit=TimeUnit.SECONDS)
@TransactionAttribute(TransactionAttributeType.NEVER)
public class StatefulSession1Bean
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatefulSession1Bean " + mInstanceNumber +
                           " created.");
    }

    @PreDestroy
    public void destroy()
    {
        System.out.println("**** StatefulSession1Bean " + mInstanceNumber +
                           " destroyed.");
    }
}
```

```

}

@Remove
public void remove()
{
    System.out.println("**** StatefulSession1Bean " + mInstanceNumber +
    " remove.");
}

/**
 * Creates a greeting to the person with the supplied name.
 *
 * @param inName Name of person to greet.
 * @return Greeting.
 */
public String greeting(final String inName)
{
    Date theCurrentTime = new Date();

    String theMessage = "Hello " + inName + ", I am stateful session bean " +
        mInstanceNumber + ". The time is now: " + theCurrentTime;
    return theMessage;
}
}

```

- Create the package `com.ivan.scbcd6.client` and in it, create the client servlet implemented as follows:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;

import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.StatefulSession1Bean;

/**
 * Servlet implementing a local EJB client.
 *
 * @author Ivan A Krizsan
 */
@WebServlet(name = "StatefulSession1Servlet", urlPatterns = "/test.do")
public class StatefulSession1Servlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB
    private StatefulSession1Bean mStatefulSessionBean;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        PrintWriter theResponseWriter = inResponse.getWriter();

        String theRequestNameParam = inRequest.getParameter("name");
        if (theRequestNameParam == null)
        {
            theRequestNameParam = "Anonymous Coward";
        }
        String theResponse = mStatefulSessionBean.greeting(theRequestNameParam);
        theResponseWriter.println("Response from the EJB: " + theResponse);
    }
}

```

```
}
```

- Deploy the project to the GlassFish application server.
- Read all of the instructions below.
This example depends on the timing between requests so you should be prepared.
If too much time passes between requests then re-deploy the application and start over.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatefulSession1Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- Within 10 seconds, issue another request in the browser. For instance to the following URL:
<http://localhost:8080/StatefulSession1Web/test.do?name=Steven>
The URL may differ if you have chosen a different project name.
- For each request you should see a greeting string from the session bean in the browser.
In the console, there should be one single log entry saying that one instance of the stateful session bean was created.
Regardless of how many times the request to the URL is sent from the browser, there will only be one single stateful session bean created.
- Wait more than 10 seconds.
- In the browser, re-issue the request to the following URL:
<http://localhost:8080/StatefulSession1Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- An error message should occur in the browser.
The cause of the error is a *NoSuchEJBException* exception.
- In the *StatefulSession1Servlet* class, modify the doGet method to look like this:

```
...
protected void doGet(HttpServletRequest inRequest,
                     HttpServletResponse inResponse) throws ServletException, IOException
{
    PrintWriter theResponseWriter = inResponse.getWriter();

    String theRequestNameParam = inRequest.getParameter("name");
    if (theRequestNameParam == null)
    {
        theRequestNameParam = "Anonymous Coward";
    }
    String theResponse = mStatefulSessionBean.greeting(theRequestNameParam);
    theResponseWriter.println("Response from the EJB: " + theResponse);

    mStatefulSessionBean.remove();
}
...
```

- Re-deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatefulSession1Web/test.do?name=Chuck>
The URL may differ if you have chosen a different project name.
- You should see the familiar greeting in the browser, but you should also see the following in the console:

```
INFO: *** StatefulSession1Bean 1 created.
INFO: *** StatefulSession1Bean 1 remove.
INFO: *** StatefulSession1Bean 1 destroyed.
```

- Any subsequent requests, despite being made within 10 seconds of the prior request, will result in a *NoSuchEJBException* being thrown.

Note that:

- The session bean implementation class is annotated with the `@LocalBean` annotation. This annotation is used to annotate session beans that expose a no-interface view. If the session bean does not expose any other view, then the `@LocalBean` annotation is optional and the session bean will expose a no-interface view.
- The session bean implementation class is annotated with the `@Stateful` annotation. This annotation is used to annotate session beans that are to be stateful. For details about the options available with the `@Stateful` annotation, please see the section on [Session Bean Metadata](#) in the chapter on session beans.
- The session bean implementation class is annotated with the `@StatefulTimeout` annotation. This annotation is used to tell the container the idle time after which an instance of the stateful session bean will be eligible for removal. When waiting too long, 10 seconds in the case of this example, the EJB instance is removed by the container. Any subsequent attempts to invoke the EJB instance will cause a *NoSuchEJBException* to be thrown.
- The session bean implementation class is annotated with the `@TransactionAttribute(TransactionAttributeType.NEVER)` annotation. This is to ensure that instances of the EJB is never in the Method Ready in Transaction state, in which case they will never timeout and become eligible for removal (as described above).
- As before, when a session bean has a no-interface view, the business methods are all the public methods in the bean implementation class.
- The remove method in the session bean implementation class is annotated with the `@Remove` annotation.
- When the servlet calls the *remove* method on an instance of the stateful session bean, it signals that it is done with the stateful session bean and the container disposes the EJB instance, as seen in the console log.

2.2. Create a Stateful Session Bean with a Local Business Interface

Compared to the stateful session bean developed in the previous section, the stateful session bean developed in this section, a session bean with a local business interface, differs in the following areas:

- The session bean implementation class is not annotated with the @LocalBean annotation.
- The session bean implementation class implements an interface.

The EJB is developed in a web project, since the session bean will only have a local view, albeit with a local business interface, and can thus only be accessed by clients in the same application.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I'll call my project "StatefulSession2Web".
- In the new project, create the session bean implementation class as below:

```
package com.ivan.scbcd6;

import java.util.Date;

import javax.annotation.PostConstruct;
import javax.ejb.Stateful;

/**
 * Simplest possible stateful session bean exposing a local business interface view.
 */
@Stateful
public class StatefulSession2Bean implements StatefulSession2Local
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatefulSession2Bean " + mInstanceNumber +
                           " created.");
    }

    /* (non-Javadoc)
     * @see com.ivan.scbcd6.StatefulSession2Local#greeting(java.lang.String)
     */
    @Override
    public String greeting(final String inName)
    {
        Date thecurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateful session bean " +
                           mInstanceNumber + ". The time is now: " + thecurrentTime;
        return theMessage;
    }
}
```

- Create the local business interface:

```
package com.ivan.scbcd6;

import javax.ejb.Local;

@Local
public interface StatefulSession2Local
{

    /**
     * Creates a greeting to the person with the supplied name.
    
```

```

/*
 * @param inName Name of person to greet.
 * @return Greeting.
 */
public String greeting(final String inName);
}

```

- Create a client servlet as in the [previous example](#).
The client servlet is to have an instance variable with the type of the local business interface (*StatefulSession2Local*) in which a reference to an instance of the stateful session bean is injected.
- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatefulSession2Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- In the browser, you should see a greeting string from the session bean.
In the console, there should be a log entry saying that one instance of the stateful session bean was created.

Note that:

- The session bean implementation class is annotated with the `@Stateful` annotation.
- The session bean implementation class implements an interface.
This is the local business interface.
- The local business interface does not extend any other interface.
With EJB 2.1 and earlier, the local business interface had to extend the `javax.ejb.EJBLocalObject`. This is no longer necessary.
- The *StatefulSession1Local* interface is annotated with the `@Local` annotation.
The `@Local` annotation can be applied to both the session bean implementation class, in which case it also needs to specify which interface(s) defines the local view, or to the interface(s) which defines the local view, in which case no further elements need to be specified.

2.3. Create a Stateful Session Bean with a Remote Business Interface

Compared to the stateful session bean developed in the previous section, the stateful session bean developed in this section, a session bean with a remote business interface, differs in the following areas:

- The interface implemented by the session bean implementation class is annotated with the `@Remote` annotation.

The session bean with a remote view can be created in an EJB project, since it may be accessed by clients outside of the application in which it is deployed.

- Create an EJB project in Eclipse, as described in [appendix B](#).
- In the new project, create the session bean implementation class as below:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.ejb.Stateful;

/**
 * Simplest possible stateful session bean with a remote business interface view.
 */
@Stateful
public class StatefulSession3Bean implements StatefulSession3Remote
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatefulSession3Bean " + mInstanceNumber +
                           " created.");
    }

    /* (non-Javadoc)
     * @see com.ivan.scbcd6.StatefulSession3Remote#greeting(java.lang.String)
     */
    @Override
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateful session bean " +
                           mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }
}
```

- Create the remote business interface:

```
package com.ivan.scbcd6;

import javax.ejb.Remote;

/**
 * Remote business interface of the stateful session bean.
 */
@Remote
public interface StatefulSession3Remote
{
    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
}
```

```
 */  
public String greeting(final String inName);  
}
```

- Create a client servlet as in the [previous example](#).

The client servlet is to have an instance variable with the type of the remote business interface (*StatefulSession3Remote*) in which a reference to an instance of the stateful session bean is injected.

- Deploy the project to the GlassFish application server.

Remote clients accessing the stateful session bean with the remote business interface will be developed in [the second section of the chapter on session bean clients](#).

Note that:

- The session bean implementation class is annotated with the `@Stateful` annotation.
- The session bean implementation class implements an interface.
This is the remote business interface.
- The remote business interface does not extend any other interface.
With EJB 2.1 and earlier, the remote business interface had to extend the `javax.ejb.EJBOBJECT`. This is no longer necessary.
- The *StatefulSession1Remote* interface is annotated with the `@Remote` annotation.
The `@Remote` annotation can be applied to both the session bean implementation class, in which case it also needs to specify which interface(s) defines the remote view, or to the interface(s) which defines the remote view, in which case no further elements need to be specified.

2.4. Create a Stateful Session Bean without Annotations

In this section we will create a stateful session bean without using annotations, configuring it in the ejb-jar.xml deployment descriptor. The implementation of the session bean is identical to that in the previous sections.

The EJB is developed in a web project, since the session bean will only have a local (no-interface) view and can thus only be accessed by clients in the same application. In [section one in chapter five](#) we'll implement a servlet acting as client to the EJB.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “StatefulSession4Web”.
- In the package *com.ivan.scbcd6*, create the session bean implementation class implemented as below:

```
package com.ivan.scbcd6;

import java.util.Date;

/**
 * Simplest possible stateful session bean without using annotations.
 */
public class StatefulSession4Bean
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatefulSession4Bean " + mInstanceNumber +
            " created.");
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateful session bean " +
            mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }
}
```

- In the WebContent/WEB-INF directory, create a file named “ejb-jar.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <!--
            Minimal configuration to define a stateful session bean
            with a no-interface view using only the deployment
            descriptor and no annotations.
        -->
    
```

```

<session>
    <ejb-name>StatefulSession4Bean</ejb-name>
    <!--
        Indicates the session bean has a no-interface view.
        Equivalent to the @LocalBean annotation.
    -->
    <local-bean/>
    <!-- Specify the session bean implementation class. -->
    <ejb-class>com.ivan.scbcd6.StatefulSession4Bean</ejb-class>
    <!--
        Specifies that the session bean is stateful.
        Equivalent to the @Stateful annotation.
    -->
    <session-type>Stateful</session-type>
</session>
</enterprise-beans>
</ejb-jar>

```

- Create a client servlet in the same way as in the [first section on creating stateful session beans](#) but with the type of the instance variable in which the EJB reference is to be injected being *StatefulSession4Bean*.
- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatefulSession4Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- Again, you will see the response from the session bean in the browser.

Note that:

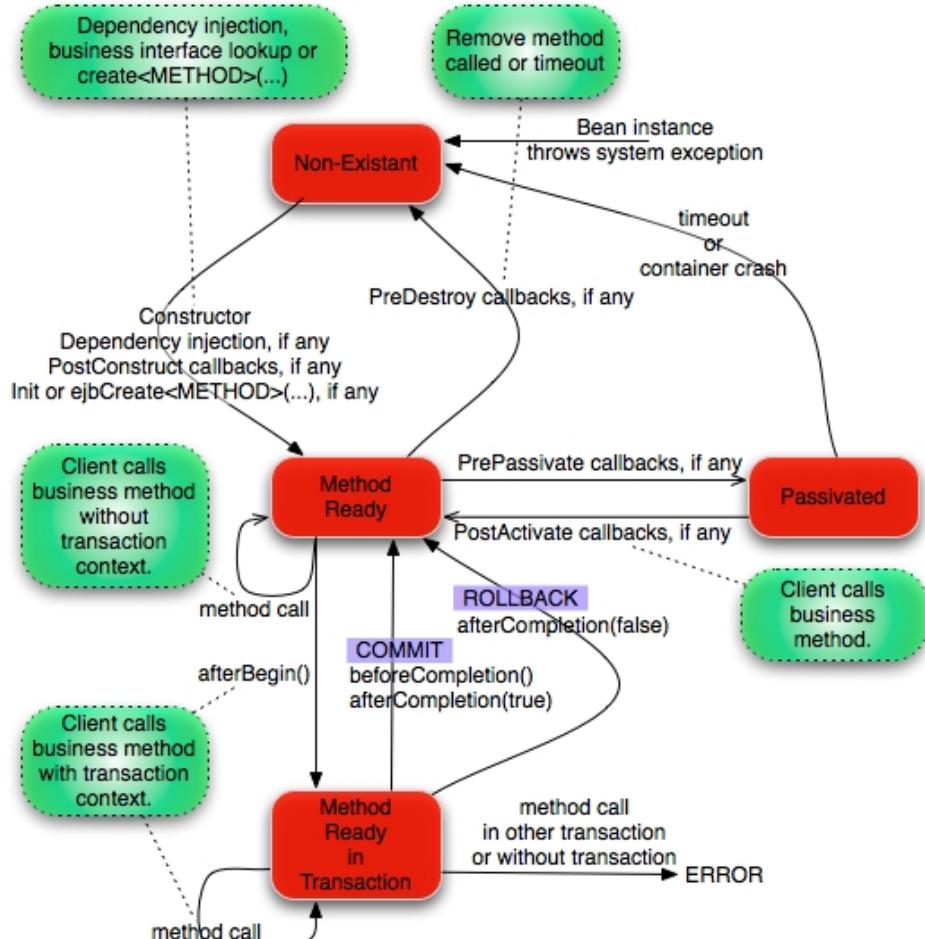
- There are no annotations in the session bean implementation class.
- There are no special interfaces to implement in the session bean implementation class.
- There are no special superclass that the session bean implementation class inherits from.
- The `<local-bean>` element in the deployment descriptor is equivalent to the `@LocalBean` annotation we have seen in an earlier example.
This element indicates that the session bean has a local no-interface view.
- The `<session-type>` element with the value “Stateful” in the deployment descriptor is equal to the `@Stateful` annotation we have seen in earlier examples.
This element is used to specify the type of the session bean – stateful in this case.

More on session beans and the deployment descriptor in [chapter 14](#).

2.5. Stateful Session Bean Life-Cycle

References: EJB 3.1 Specification, section 4.6.

The stateful session bean life-cycle is the most complex among all the EJBs, which can be seen in the following figure:



Life-cycle of a stateful session bean.

The red boxes denote different states of a stateful session bean.

The green boxes denote client events that causes a stateful session bean to change state.

Text that is not boxed and not colored denote (callback) methods that are invoked on a stateful session bean in connection to a state transition.

The time a stateful session bean must be without requests before being considered for removal can be specified using the `@StatefulTimeout` annotation. The timeout may also be configured so that a session bean will never be removed due to timeout. Note that a stateful session bean may not be timed out while being in the Method Ready in Transaction state.

2.6. Operations Allowed in Methods of a Stateful Session Bean

References: EJB 3.1 Specification, section 4.6.1.

There are certain restrictions of what can be done in different kinds of method of a stateful session bean. The following table lists different kinds of methods and what operations they are allowed to perform.

The notes “BMT only” or “CMT only” means that these operations are only available when the bean has been managed, respective container managed, transactions.

JNDI access indicates whether the context “java:comp/env” is accessible.

If a stateful session bean attempts to perform an illegal operation, an *IllegalStateException* will be thrown in the following two cases:

- Invoke a method in the *SessionContext* interface not permitted.
- Invoke a method in the *Timer* interface not permitted.

Bean Method(s)	Allowed Operations
Constructor	None.
Dependency injection methods (setter methods).	SessionContext: getEJBHome, getEJBLocalHome, lookup. JNDI Access: Available
PostConstruct, PreDestroy, PrePassivate, PostActivate methods (lifecycle callback methods).	SessionContext: getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData, getUserTransaction (BMT only). JNDI Access: Available EntityManagerFactory: Accessible.
Business method from any view or business method interceptor method.	SessionContext: getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData, getInvokedBusinessInterface,

	<p>wasCancelled, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only).</p> <p>JNDI Access: Available</p> <p>Resource managers: Accessible.</p> <p>Other EJBs: Accessible.</p> <p>EntityManagerFactory: Accessible.</p> <p>EntityManager: Accessible.</p> <p>Timer methods: Accessible.</p> <p>UserTransaction methods: Accessible (BMT only).</p>
afterBegin and beforeCompletion (when EJB implements the SessionSynchronization interface) or methods in the EJB annotated with @AfterBegin and @BeforeCompletion	<p>ONLY AVAILABLE TO CMT EJBs!</p> <p>SessionContext:</p> <p>getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData, getRollbackOnly, setRollbackOnly.</p> <p>JNDI Access: Available</p> <p>Resource managers: Accessible.</p> <p>Other EJBs: Accessible.</p> <p>EntityManagerFactory: Accessible.</p> <p>EntityManager: Accessible.</p> <p>Timer methods: Accessible.</p>
afterCompletion (when EJB implements the SessionSynchronization interface) or method annotated with @AfterCompletion	<p>ONLY AVAILABLE TO CMT EJBs!</p> <p>SessionContext:</p> <p>getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData.</p> <p>JNDI Access: Available.</p>

In addition, the `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface may only be invoked from within a method executing in a transaction context or else an `IllegalStateException` will be thrown.

3. Create a Stateless Session Bean

References: EJB 3.1 Specification, chapter 4.

In this chapter we look at how to create different kinds of stateless session beans. These are similar to stateful session beans, except that stateless session beans are not tied to one single client, but may serve requests from multiple clients. The stateless session bean does not, as the name implies, hold any state related to a particular client.

Detailed information applying to session beans can be found in [chapter 14](#).

3.1. Create a Stateless Session Bean with a Local No-interface View

The stateless session bean has one single method that creates a greeting. The EJB is developed in a web project, since the session bean will only have a local (no-interface) view and can thus only be accessed by clients in the same application.

In this example, we will also create a client to the stateless session bean.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “StatelessSession1Web”.
- In the package *com.ivan.scbcd6*, create the session bean implementation class implemented as below:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

/**
 * Simplest possible stateless session bean exposing a local, no-interface view.
 */
@Stateless
@LocalBean
public class StatelessSession1Bean
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatelessSession1Bean " + mInstanceNumber +
                           " created.");
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateless session bean " +
                           mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }
}
```

- Create the package `com.ivan.scbcd6.client` and in it, create the client servlet implemented as follows:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.StatelessSession1Bean;

/**
 * Servlet implementing a local EJB client.
 *
 * @author Ivan A Krizsan
 */
@WebServlet(name = "StatelessSession1Servlet", urlPatterns = "/test.do")
public class StatelessSession1Servlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB
    private StatelessSession1Bean mStatelessSessionBean;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        PrintWriter theResponseWriter = inResponse.getWriter();

        String theRequestNameParam = inRequest.getParameter("name");
        if (theRequestNameParam == null)
        {
            theRequestNameParam = "Anonymous Coward";
        }
        String theResponse = mStatelessSessionBean.greeting(theRequestNameParam);
        theResponseWriter.println("Response from the EJB: " + theResponse);
    }
}

```

- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatelessSession1Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- In the console log, you should see the following message:
“INFO: *** StatelessSession1Bean 1 created.”
Regardless of how many times the request to the URL is sent from the browser, there will only be one single stateless session bean. To provoke the creation of more than one bean instance, I had to use Apache JMeter to issue multiple concurrent requests.

Note that:

- The session bean implementation class is annotated with the `@LocalBean` annotation. This annotation is used to annotate session beans that expose a no-interface view. If the session bean does not expose any other view, then the `@LocalBean` annotation is optional and the session bean will expose a no-interface view.
- The session bean implementation class is annotated with the `@Stateless` annotation. This annotation is used to annotate session beans that are to be stateless. For details about the options available with the `@Stateless` annotation, please see the section on [Session Bean Metadata](#) in the chapter on session beans.
- As before, when a session bean has a no-interface view, the business methods are all the public methods in the bean implementation class.

3.2. Create a Stateless Session Bean with a Local Business Interface

Compared to the stateless session bean developed in the previous section, the stateless session bean developed in this section, a session bean with a local business interface, differs in the following areas:

- The session bean implementation class is not annotated with the @LocalBean annotation.
- The session bean implementation class implements an interface.

The EJB is developed in a web project, since the session bean will only have a local view, albeit with a local business interface, and can thus only be accessed by clients in the same application.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “StatelessSession2Web”. A web project is used, since the session bean will only have a local view and can thus only be accessed by clients in the same application.
- In the new project, create the session bean implementation class as below:

```
package com.ivan.scbcd6;

import java.util.Date;

import javax.annotation.PostConstruct;
import javax.ejb.Stateless;

/**
 * Simplest possible stateless session bean exposing a local business interface view.
 */
@Stateless
public class StatelessSession2Bean implements StatelessSession2Local
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatelessSession2Bean " + mInstanceNumber +
                           " created.");
    }

    /* (non-Javadoc)
     * @see com.ivan.scbcd6.StatelessSession2Local#greeting(java.lang.String)
     */
    @Override
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateless session bean " +
                           mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }
}
```

- Create the local business interface:

```
package com.ivan.scbcd6;

import javax.ejb.Local;

@Local
public interface StatelessSession2Local
{
    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName);
}
```

- Create a client servlet in the same way as in the [previous](#) section.
- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatelessSession2Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- In the browser, you should see a greeting string from the session bean.
In the console, there should be a log entry saying that one instance of the stateless session bean was created.

Note that:

- The session bean implementation class is annotated with the `@Stateless` annotation.
- The session bean implementation class implements an interface.
This is the local business interface.
- The local business interface does not extend any other interface.
With EJB 2.1 and earlier, the local business interface had to extend the `javax.ejb.EJBLocalObject`. This is no longer necessary.
- The `StatelessSession2Local` interface is annotated with the `@Local` annotation.
The `@Local` annotation can be applied to both the session bean implementation class, in which case it also needs to specify which interface(s) defines the local view, or to the interface(s) which defines the local view, in which case no further elements need to be specified.

3.3. Create a Stateless Session Bean with a Remote Business Interface

Compared to the stateless session bean developed in the previous section, the stateless session bean developed in this section, a session bean with a remote business interface, differs in the following areas:

- The interface implemented by the session bean implementation class is annotated with the `@Remote` annotation.

The session bean with a remote view can be created in an EJB project, since it may be accessed by clients outside of the application in which it is deployed.

- Create an EJB project in Eclipse, as described in [appendix B](#).
- In the new project, create the session bean implementation class as below:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.ejb.Stateless;

/**
 * Simplest possible stateless session bean with a remote business interface view.
 */
@Stateless
public class StatelessSession3Bean implements StatelessSession3Remote
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatelessSession3Bean " + mInstanceNumber +
                           " created.");
    }

    /* (non-Javadoc)
     * @see com.ivan.scbcd6.StatelessSession3Remote#greeting(java.lang.String)
     */
    @Override
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateless session bean " +
                           mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }
}
```

- Create the remote business interface:

```
package com.ivan.scbcd6;

import javax.ejb.Remote;

/**
 * Remote business interface of the stateless session bean.
 */
@Remote
public interface StatelessSession3Remote
{
    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
}
```

```
 */
public String greeting(final String inName);
}
```

- Deploy the project to the GlassFish application server.

Remote clients accessing the stateful session bean with the remote business interface will be developed in [chapter five, section 2](#).

Note that:

- The session bean implementation class is annotated with the `@Stateless` annotation.
- The session bean implementation class implements an interface.
This is the remote business interface.
- The remote business interface does not extend any other interface.
With EJB 2.1 and earlier, the remote business interface had to extend the `javax.ejb.EJBObject`. This is no longer necessary.
- The `StatelessSession3Remote` interface is annotated with the `@Remote` annotation.
The `@Remote` annotation can be applied to both the session bean implementation class, in which case it also needs to specify which interface(s) defines the remote view, or to the interface(s) which defines the remote view, in which case no further elements need to be specified.

3.4. Create a Stateless Session Bean without Annotations

In this section we will create a stateless session bean without using annotations, configuring it in the ejb-jar.xml deployment descriptor. The implementation of the session bean is identical to that in the previous sections.

The EJB is developed in a web project, since the session bean will only have a local (no-interface) view and can thus only be accessed by clients in the same application. In [section one in chapter five](#) we'll implement a servlet acting as client to the EJB.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “StatelessSession4Web”.
- In the package *com.ivan.scbcd6*, create the session bean implementation class implemented as below:

```
package com.ivan.scbcd6;

import java.util.Date;

/**
 * Simplest possible stateless session bean without using annotations.
 */
public class StatelessSession4Bean
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatelessSession4Bean " + mInstanceNumber +
                           " created.");
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateless session bean " +
                           mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }
}
```

- In the WebContent/WEB-INF directory, create a file named “ejb-jar.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="
               http://java.sun.com/xml/ns/javaee
               http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <!--
            Minimal configuration to define a stateless session bean
            with a no-interface view using only the deployment
            descriptor and no annotations.
        -->
    
```

```

<session>
    <ejb-name>StatefulSession1Bean</ejb-name>
    <!--
        Indicates the session bean has a no-interface view.
        Equivalent to the @LocalBean annotation.
    -->
    <local-bean/>
    <!-- Specify the session bean implementation class. -->
    <ejb-class>com.ivan.scbcd6.StatelessSession4Bean</ejb-class>
    <!--
        Specifies that the session bean is stateless.
        Equivalent to the @Stateless annotation.
    -->
    <session-type>Stateless</session-type>
</session>
</enterprise-beans>
</ejb-jar>

```

- Create a client servlet in the same way as in the [first section on creating stateless session beans](#).
- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/StatelessSession4Web/test.do?name=Ivan>
The URL may differ if you have chosen a different project name.
- Again, you will see the response from the session bean in the browser.

Note that:

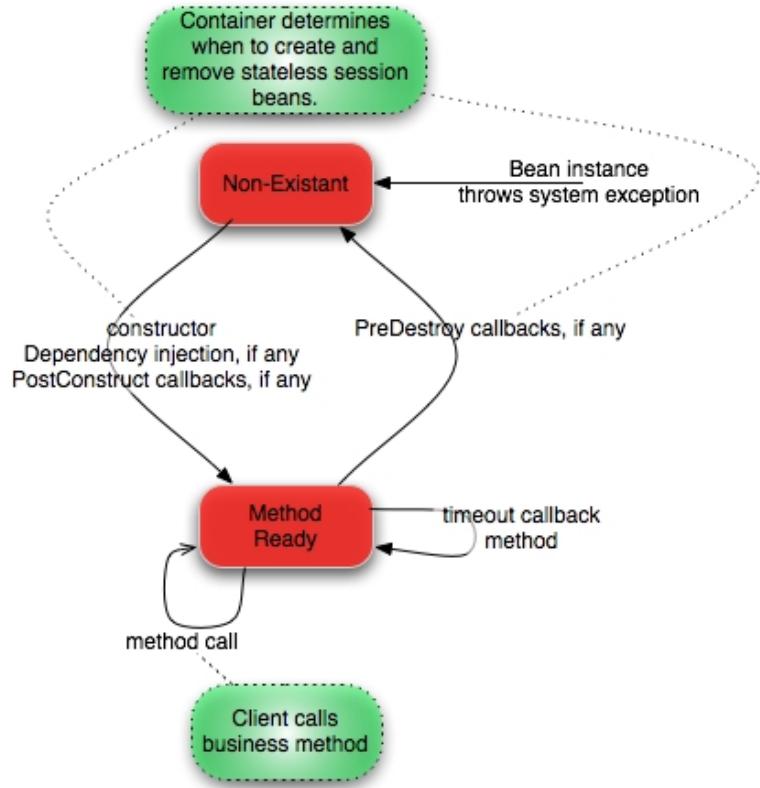
- There are no annotations in the session bean implementation class.
- There are no special interfaces to implement in the session bean implementation class.
- There are no special superclass that the session bean implementation class inherits from.
- The <local-bean> element in the deployment descriptor is equivalent to the @LocalBean annotation we have seen in an earlier example.
This element indicates that the session bean has a local no-interface view.
- The <session-type> element with the value “Stateless” in the deployment descriptor is equal to the @Stateless annotation we have seen in earlier examples.
This element is used to specify the type of the session bean – stateless in this case.

More on session beans and the deployment descriptor in [chapter 14](#).

3.5. Stateless Session Bean Life-Cycle

References: EJB 3.1 Specification, section 4.7.1.

The stateless session bean has a significantly less complex life-cycle, compared to the stateful session bean.



Life-cycle of a stateless session bean.

The red boxes denote different states of a stateless session bean.

The green boxes denote client events that causes a stateless session bean to change state.

Text that is not boxed and not colored denote (callback) methods that are invoked on a stateless session bean in connection to a state transition.

3.6. Operations Allowed in Methods of a Stateless Session Bean

References: EJB 3.1 Specification, section 4.7.2.

There are certain restrictions of what can be done in different kinds of method of a stateless session bean. The following table lists different kinds of methods and what operations they are allowed to perform.

The notes “BMT only” or “CMT only” means that these operations are only available when the bean has bean managed, respective container managed, transactions.

JNDI access indicates whether the context “java:comp/env” is accessible.

If a stateless session bean attempts to perform an illegal operation, an *IllegalStateException* will be thrown in the following two cases:

- Invoke a method in the *SessionContext* interface not permitted.
- Invoke a method in the *Timer* or *TimerService* interfaces not permitted.

Bean Method(s)	Allowed Operations
Constructor	None.
Dependency injection methods (setter methods).	SessionContext: getEJBHome, getEJBLocalHome, lookup. JNDI Access: Available
PostConstruct, PreDestroy methods (lifecycle callback methods).	SessionContext: getBusinessObject, getEJBHome, getEJBLocalHome, getEJBObject, getEJBLocalObject, lookup, getContextData, getTimerService, getUserTransaction (BMT only). JNDI Access: Available EntityManagerFactory: Accessible.
Business method from any view or business method interceptor method.	SessionContext: getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData, getInvokedBusinessInterface, wasCancelCalled, getTimerService,

	<p>getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only).</p> <p>JNDI Access: Available</p> <p>Resource managers: Accessible.</p> <p>Other EJBs: Accessible.</p> <p>EntityManagerFactory: Accessible.</p> <p>EntityManager: Accessible.</p> <p>Timer and TimerService methods: Accessible.</p> <p>UserTransaction methods: Accessible (BMT only).</p>
Business methods from web service endpoint.	<p>SessionContext:</p> <p>getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData, getTimerService, getMessageContext, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only).</p> <p>MessageContext methods: Available</p> <p>JNDI Access: Available</p> <p>Resource managers: Accessible.</p> <p>Other EJBs: Accessible.</p> <p>EntityManagerFactory: Accessible.</p> <p>EntityManager: Accessible.</p> <p>Timer and TimerService methods: Accessible.</p> <p>UserTransaction methods: Accessible (BMT only).</p>
Timeout callback method.	<p>SessionContext:</p> <p>getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup, getContextData, getTimerService, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only).</p>

	<p>JNDI Access: Available</p> <p>Resource managers: Accessible.</p> <p>Other EJBs: Accessible.</p> <p>EntityManagerFactory: Accessible.</p> <p>EntityManager: Accessible.</p> <p>Timer and TimerService methods: Accessible.</p> <p>UserTransaction methods: Accessible (BMT only).</p>
--	--

In addition to the constraints specified in the above table, the *getRollbackOnly* and *setRollbackOnly* methods may only be invoked from within a transaction, otherwise an *IllegalStateException* will be thrown.

4. Create a Singleton Session Bean

References: EJB 3.1 Specification, section 4.8.

Singleton session beans are session beans for which at most one instance per application exists. Such session beans may be used to implement, for instance, a central repository or application initialization that needs to be performed when an application starts up.

Singleton session bean instances has the following characteristics:

- One single instance of a singleton session bean exists per application.
If the application is distributed, then there is one instance per JVM in which the application containing the singleton session bean is running.
- Can be shared.
- Supports concurrent access.
- An instance of a singleton session bean is not destroyed until the application shuts down.
The instance even survives system exceptions from business methods and callback methods invoked after creation and initialization of the bean has completed.
- May hold state.
Can hold instance state that is preserved over multiple calls, but any state of a singleton session bean is not retained in the case of container shutdown or crash.

As with stateless and stateful session beans, singleton session beans can also have three different kinds of views; local no-interface view, local business interface view and a remote business interface view. Examples on how to configure the different kinds of views has been shown in [chapters two](#) and [three](#) and will not be repeated here.

4.1. Basic Singleton Session Bean Example

In this section we will create the first incarnation of the singleton session bean example program. This program will be extended and modified in subsequent sections, as to demonstrate different capabilities of singleton session beans.

The example program is developed in a web project, since the session beans will only have a local (no-interface) view and can thus only be accessed by clients in the same application.

In this example, we will also create a client to the singleton session beans.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “SingletonSession1Web”.
- In the package *com.ivan.scbcd6*, create the first session bean implementation class according to the following listing:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;

/**
 * This class implements the first singleton session bean of this
 * example program.
 * This singleton session bean has the ability to hold state.
 */
@Singleton
@LocalBean
public class SingletonSessionBeanA
{
    /* Constant(s): */
    private final static String BEAN_NAME = "SingletonSessionBeanA";

    /* Instance variable(s): */
    private String mStoredMessage = "[no message set]";

    @PostConstruct
    public void intialize()
    {
        System.out.println("**** " + BEAN_NAME + " - Initialized");
    }

    @PreDestroy
    public void cleanUp()
    {
        System.out.println("**** " + BEAN_NAME + " - Destroyed");
    }

    public String retrieveMessage()
    {
        Date theCurrentTime = new Date();

        return "Message from " + BEAN_NAME + " - " + mStoredMessage + " "
            + theCurrentTime;
    }

    public void storeMessage(final String inStoredMessage)
    {
        mStoredMessage = inStoredMessage;
    }
}
```

- In the same package, create the second singleton session bean implementation class:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;

/**
 * This class implements the second singleton session bean of this
 * example program.
 */
@Singleton
@LocalBean
public class SingletonSessionBeanB
{
    private final static String BEAN_NAME = "SingletonSessionBeanB";

    @PostConstruct
    public void intialize()
    {
        System.out.println("**** " + BEAN_NAME + " - Initialized");
    }

    @PreDestroy
    public void cleanUp()
    {
        System.out.println("**** " + BEAN_NAME + " - Destroyed");
    }

    public String retrieveMessage()
    {
        Date theCurrentTime = new Date();

        return "Message from " + BEAN_NAME + " - " + theCurrentTime;
    }
}
```

- In the *com.ivan.scbcd6.client* package, create the servlet that is to be the client of the two singleton session beans we just implemented:

```
package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.SingletonSessionBeanA;
import com.ivan.scbcd6.SingletonSessionBeanB;

/**
 * Servlet acting as a client of the two singleton session beans.
 */
@WebServlet(name = "SingletonClientServlet", urlPatterns = "/test.do")
public class SingletonClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;
    private final static String STORE_ACTION = "store";
    private final static String CLEAR_ACTION = "clear";

    /* Instance variable(s): */
    @EJB
    private SingletonSessionBeanA mSingletonBeanA;
    @EJB
    private SingletonSessionBeanB mSingletonBeanB;
```

```

@Override
protected void doGet(HttpServletRequest inRequest,
                      HttpServletResponse inResponse) throws ServletException, IOException
{
    System.out.println("**** Entering SingletonClientServlet");

    String theRequestNameParam = inRequest.getParameter("name");
    String theRequestActionParam = inRequest.getParameter("action");

    /* Set default name if none provided. */
    if (theRequestNameParam == null || theRequestNameParam.equals(""))
    {
        theRequestNameParam = "Anonymous Coward";
    }

    /* Generate output from servlet using session beans. */
    PrintWriter theResponseWriter = inResponse.getWriter();
    String theMessage;

    theMessage = mSingletonBeanA.retrieveMessage();
    theResponseWriter.println(theMessage);
    theMessage = mSingletonBeanB.retrieveMessage();
    theResponseWriter.println(theMessage);

    /*
     * Store or clear data in one of the singleton session beans
     * if the supplied action so specifies.
     */
    if (theRequestActionParam != null)
    {
        if (STORE_ACTION.equals(theRequestActionParam))
        {
            mSingletonBeanA.storeMessage(theRequestNameParam);
        }
        if (CLEAR_ACTION.equals(theRequestActionParam))
        {
            mSingletonBeanA.storeMessage("[Cleared]");
        }
    }

    System.out.println("**** Exiting SingletonClientServlet");
    theResponseWriter.println("Finished invoking singleton session beans!");
}
}

```

- Deploy the project to the GlassFish application server.

- In a browser, issue a request to the following URL:

<http://localhost:8080/SingletonSession1Web/test.do>

You should see three messages in the browser; one from the first singleton session bean, one from the second session bean and a message saying that the invocation of the singleton session beans finished:

```

Message from SingletonSessionBeanA - [no message set] Mon Aug 23 07:05:27 CEST 2010
Message from SingletonSessionBeanB - Mon Aug 23 07:05:27 CEST 2010
Finished invoking singleton session beans!

```

- In the browser, issue a request to the following URL:

<http://localhost:8080/SingletonSession1Web/test.do?name=Ivan&action=store>

The output does not change.

- Finally, re-issue the request to the first URL:
<http://localhost:8080/SingletonSession1Web/test.do>

You should now see that the first singleton session bean retrieves the name supplied in the URL earlier:

```
Message from SingletonSessionBeanA - Ivan Mon Aug 23 07:10:05 CEST 2010
Message from SingletonSessionBeanB - Mon Aug 23 07:10:05 CEST 2010
Finished invoking singleton session beans!
```

- Any subsequent requests to the last URL will cause the same name to be retrieved.

Note that:

- Both the singleton session bean implementation classes are annotated using the `@Singleton` annotation.
- The singleton session beans are also annotated with the `@LocalBean` annotation.
We recognize this annotation from earlier examples as declaring that the bean has a local, no-interface view.
- As before, when a session bean has a no-interface view, the business methods are all the public methods in the bean implementation class.
- Both the singleton session beans have initialization methods annotated with the `@PostConstruct` annotation.
Such methods are invoked after construction of an EJB instance, immediately prior to the instance being taken into use.
- Both the singleton session beans also have methods annotated with the `@PreDestroy` annotation.
These methods are invoked immediately prior to an instance being taken out of service.
- Singleton session beans can hold state between requests.

4.2. Singleton Session Bean Initialization

References: EJB 3.1 Specification, section 4.8.1.

Without any special configuration, it is up to the container to decide when to initialize singleton session beans. Apart from being initialized on demand, when the first call to the singleton session bean occurs, it is possible to configure eager initialization as well as initialization ordering.

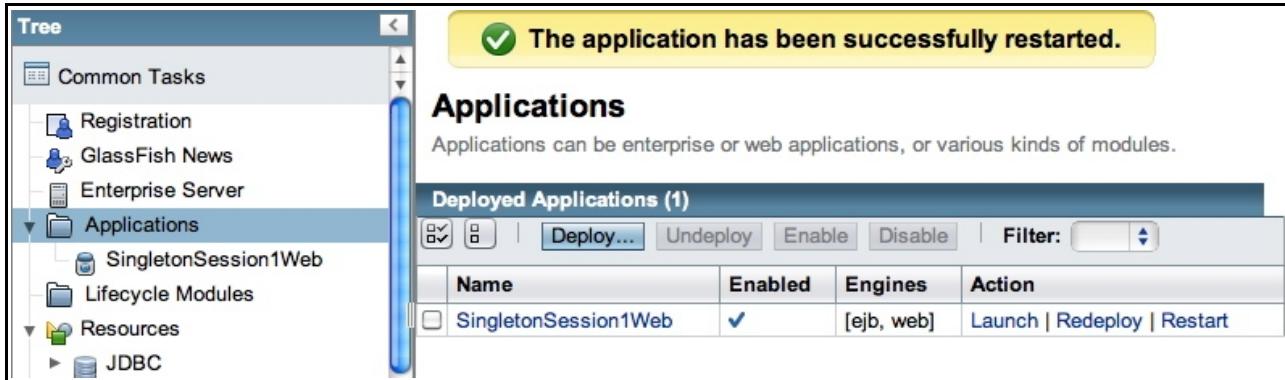
4.2.1. Singleton Session Bean Eager Initialization

To configure a singleton session bean so that it is (eagerly) initialized prior to the application receives any requests, we can use the `@Startup` annotation or setting the value of the `<init-on-startup>` element in the deployment descriptor to true.

- In the example program, modify the `SingletonSessionBeanA` class adding the `@Startup` annotation:

```
...  
@Singleton  
@LocalBean  
@Startup  
public class SingletonSessionBeanA  
{  
...  
}
```

- To see what happens when our application is shut down and restarted, navigate to the Applications node in the GlassFish v3 administration console and click Restart in the Action column for our application. A message in the administration console should confirm that the application has been restarted:



- Having restarted the application, you should see the following output in the GlassFish log file or in the console:

```
INFO: *** SingletonSessionBeanB - Destroyed  
INFO: *** SingletonSessionBeanA - Destroyed  
  
INFO: Portable JNDI names for EJB SingletonSessionBeanA :  
[ java:global/SingletonSession1Web/SingletonSessionBeanA!  
com.ivan.scbcd6.SingletonSessionBeanA,  
java:global/SingletonSession1Web/SingletonSessionBeanA]  
INFO: Portable JNDI names for EJB SingletonSessionBeanB :  
[ java:global/SingletonSession1Web/SingletonSessionBeanB,  
java:global/SingletonSession1Web/SingletonSessionBeanB!  
com.ivan.scbcd6.SingletonSessionBeanB]  
INFO: *** SingletonSessionBeanA - Initialized  
INFO: Loading application SingletonSession1Web at /SingletonSession1Web  
INFO: Loading application SingletonSession1Web at /SingletonSession1Web
```

```
INFO: Loading application SingletonSession1Web at /SingletonSession1Web
```

Note that:

- If the application has not been run, then you will only see a message about SingletonSessionBeanA having been destroyed, not SingletonSessionBeanB, since the latter has not been created.
- There is a message saying that SingletonSessionBeanA is initialized immediately on application startup – this is what we wanted to accomplish using the @Startup annotation.

4.3. Singleton Session Bean Destruction

References: EJB 3.1 Specification, section 4.8.2.

As we saw in the previous section, singleton session beans are destroyed when the application in which they are deployed is destroyed.

4.4. Singleton Session Bean Initialization and Destruction Dependencies

Not only can we tell the container that we want a singleton session bean eagerly initialized, we can also tell the container that one singleton session bean needs to be initialized prior to the initialization of another singleton session bean. This is accomplished using the `@DependsOn` annotation or the `<depends-on>` element in the `ejb-jar.xml` deployment descriptor.

- In the example program, modify the `SingletonSessionBeanA` class adding the `@DependsOn` annotation:

```
...
@Singleton
@LocalBean
@Startup
@DependsOn( "SingletonSessionBeanB" )
public class SingletonSessionBeanA
{
...
}
```

- As after having added the `@Startup` annotation, navigate to the Applications node in the GlassFish v3 administration console and click Restart in the Action column for our application.
- Having restarted the application, you should see the following output in the GlassFish log file or in the console.

Note that there are now messages from both the singleton session beans saying that they have been initialized and that `SingletonSessionBeanB` is initialized before `SingletonSessionBeanA` is initialized – this is what we wanted to accomplish using the `@DependsOn` annotation.

```
INFO: *** SingletonSessionBeanA - Destroyed
INFO: *** SingletonSessionBeanB - Destroyed
INFO: Portable JNDI names for EJB SingletonSessionBeanA :
[ java:global/SingletonSession1Web/SingletonSessionBeanA!
com.ivan.scbcd6.SingletonSessionBeanA,
java:global/SingletonSession1Web/SingletonSessionBeanA ]
INFO: Portable JNDI names for EJB SingletonSessionBeanB :
[ java:global/SingletonSession1Web/SingletonSessionBeanB,
java:global/SingletonSession1Web/SingletonSessionBeanB!
com.ivan.scbcd6.SingletonSessionBeanB ]
INFO: *** SingletonSessionBeanB - Initialized
INFO: *** SingletonSessionBeanA - Initialized
INFO: Loading application SingletonSession1Web at /SingletonSession1Web
INFO: Loading application SingletonSession1Web at /SingletonSession1Web
INFO: Loading application SingletonSession1Web at /SingletonSession1Web
```

- If we restart the application again from the GlassFish administration console, we can also note that `SingletonSessionBeanA` is destroyed prior to `SingletonSessionBeanB`. This is also caused by `SingletonSessionBeanA` having a dependency on `SingletonSessionBeanB`. The container ensures that all singleton session beans which a singleton session beans depend on are still present during destruction.

If `SingletonSessionBeanA` simply wanted to invoke `SingletonSessionBeanB`, then regular dependency injection or JNDI lookup would have been sufficient and any `@DependsOn` annotations are not necessary.

Cyclic dependencies are not allowed and may cause deployment errors.

4.5. Transactions and Singleton Session Bean Creation and Destruction

References: EJB 3.1 Specification, section 4.8.3.

For details on transaction behaviour common to the different kinds of EJBs, please refer to the [chapter on transactions](#).

Methods annotated with either @PostConstruct or @PreDestroy in singleton session beans with container managed transactions can have one of the following transaction attributes:

- REQUIRED
- REQUIRES_NEW
- NOT_SUPPORTED

See the [section on transaction attributes](#) in the chapter on transactions for details on transaction attributes.

4.6. Singleton Session Bean Concurrency

References: EJB 3.1 Specification, section 4.8.5.

While concurrent access of stateful and stateless session beans is always controlled by the container, singleton session beans allow for more control in this area. First, singleton session bean concurrency-management can be configured to be one of two types:

- Container-managed concurrency.
This is the default concurrency-management type.
- Bean-managed concurrency.

This can be configured using either annotations or the ejb-jar.xml deployment descriptor.

The EJB 3.1 specification prohibits overriding either default or explicitly configured concurrency-management configuration because the implementation of a singleton session bean can be very dependent of its concurrency-management configuration and any changes may cause the bean to malfunction.

4.6.1. Basic Singleton Session Bean Concurrency Example

In this section we will look at an example program demonstrating both container-managed as well as bean-managed concurrency. The example program is developed in a web project, since the session beans will only have a local (no-interface) view and can thus only be accessed by clients in the same application. In the example program, there are two singleton session beans; one with container-managed concurrency and one with bean-managed concurrency. The client of the two beans is a servlet which will call two methods in one of the singleton session beans from two different threads.

- Create a dynamic web project in Eclipse, as described in [appendix A](#).
I call my project “SingletonSessionBeanConcurrency”.
- In the package *com.ivan.scbcd6*, create the first session bean implementation class according to the following listing:

```
package com.ivan.scbcd6;

import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;

/**
 * This class implements a singleton session bean with container-managed
 * concurrency.
 */
@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class SingletonSessionBeanA
{
    public void slowMethod()
    {
        System.out.println("SingletonSessionBeanA - Entering slowMethod");
        waitSomeTime(10);
        System.out.println("SingletonSessionBeanA - Exiting slowMethod");
    }
}
```

```

public void fastMethod()
{
    System.out.println("SingletonSessionBeanA - Entering fastMethod");
    waitSomeTime(1);
    System.out.println("SingletonSessionBeanA - Exiting fastMethod");
}

private void waitSomeTime(final long inSecondsDelay)
{
    try
    {
        Thread.sleep(1000L * inSecondsDelay);
    } catch (InterruptedException e)
    {
        // Ignore exceptions.
    }
}

```

- In the same package, create the second singleton session bean implementation class:

```

package com.ivan.scbcd6;

import javax.ejbConcurrencyManagement;
import javax.ejbConcurrencyManagementType;
import javax.ejbLocalBean;
import javax.ejbSingleton;

/**
 * This class implements a singleton session bean with bean-managed
 * concurrency.
 */
@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class SingletonSessionBeanB
{
    public void slowMethod()
    {
        System.out.println("SingletonSessionBeanB - Entering slowMethod");
        waitSomeTime(10);
        System.out.println("SingletonSessionBeanB - Exiting slowMethod");
    }

    public void fastMethod()
    {
        System.out.println("SingletonSessionBeanB - Entering fastMethod");
        waitSomeTime(1);
        System.out.println("SingletonSessionBeanB - Exiting fastMethod");
    }

    private void waitSomeTime(final long inSecondsDelay)
    {
        try
        {
            Thread.sleep(1000L * inSecondsDelay);
        } catch (InterruptedException e)
        {
            // Ignore exceptions.
        }
    }
}

```

- In the *com.ivan.scbcd6.client* package, create the servlet that is to be the client of the two singleton session beans we just implemented:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.SingletonSessionBeanA;
import com.ivan.scbcd6.SingletonSessionBeanB;

/**
 * Servlet acting as a client of the two singleton session beans.
 */
@WebServlet(name = "SingletonClientServlet", urlPatterns = "/test.do")
public class SingletonClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB
    private SingletonSessionBeanA mSingletonBeanA;
    @EJB
    private SingletonSessionBeanB mSingletonBeanB;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        System.out.println("**** Entering SingletonClientServlet");

        /* Get parameter specifying which test to run. */
        String theSelectorString = inRequest.getParameter("selector");
        if (theSelectorString == null || theSelectorString.equals(""))
        {
            theSelectorString = "1";
        }
        int theSelector = Integer.parseInt(theSelectorString);

        switch (theSelector)
        {
            case 1: testContainerManagedConcurrency(); break;
            case 2: testBeanManagedConcurrency(); break;
            default: break;
        }

        System.out.println("**** Exiting SingletonClientServlet");

        /* Display a message on the web page. */
        PrintWriter theResponseWriter = inResponse.getWriter();
        theResponseWriter.println("Finished invoking singleton session bean " +
                               "concurrency test " + theSelector);
    }

    private void testContainerManagedConcurrency()
    {
        System.out.println("**** Entering testContainerManagedConcurrency");

        /*
         * Call first the slow method and then the fast method from separate
         * threads.
         */
        System.out.println("      Calling slowMethod... ");
    }
}

```

```

        new Thread()
    {
        @Override
        public void run()
        {
            mSingletonBeanA.slowMethod();
        }
    }.start();

    System.out.println("      Calling fastMethod... ");
    new Thread()
    {
        @Override
        public void run()
        {
            mSingletonBeanA.fastMethod();
        }
    }.start();

    System.out.println("*** Exiting testContainerManagedConcurrency");
}

private void testBeanManagedConcurrency()
{
    System.out.println("*** Entering testBeanManagedConcurrency");

    /*
     * Call first the slow method and then the fast method from separate
     * threads.
     */
    System.out.println("      Calling slowMethod... ");
    new Thread()
    {
        @Override
        public void run()
        {
            mSingletonBeanB.slowMethod();
        }
    }.start();

    System.out.println("      Calling fastMethod... ");
    new Thread()
    {
        @Override
        public void run()
        {
            mSingletonBeanB.fastMethod();
        }
    }.start();

    System.out.println("*** Exiting testBeanManagedConcurrency");
}
}

```

- Deploy the project to the GlassFish application server.
- In a browser, first issue a request to the following URL:
<http://localhost:8080/SingletonSessionBeanConcurrency/test.do?selector=1>

You should see the following messages being printed to the console:

```

INFO: *** Entering SingletonClientServlet
INFO: *** Entering testContainerManagedConcurrency
INFO:      Calling slowMethod...
INFO:      Calling fastMethod...
INFO: SingletonSessionBeanA - Entering slowMethod
INFO: *** Exiting testContainerManagedConcurrency
INFO: *** Exiting SingletonClientServlet

INFO: SingletonSessionBeanA - Exiting slowMethod
INFO: SingletonSessionBeanA - Entering fastMethod

INFO: SingletonSessionBeanA - Exiting fastMethod

```

- Observe the following and the order of the events:
 - The SingletonClientServlet is entered.
 - The *testContainerManagedConcurrency* method is entered.
 - The two methods *slowMethod* and *fastMethod* are called on the singleton session bean in that order.
 - The execution of the *slowMethod* is started.
 - The *testContainerManagedConcurrency* method is exited.
 - The SingletonClientServlet is exited.
 - After some time, the execution of the *slowMethod* is finished.
 - The execution of the *fastMethod* is started.
 - Finally, the execution of *fastMethod* is finished.
 - The execution of the *slowMethod* and the *fastMethod* methods does not overlap.
- In a browser, issue another request to the following URL:
<http://localhost:8080/SingletonSessionBeanConcurrency/test.do?selector=2>

You should see the following messages being printed to the console:

```
INFO: *** Entering SingletonClientServlet
INFO: *** Entering testBeanManagedConcurrency
INFO: Calling slowMethod...
INFO: Calling fastMethod...
INFO: SingletonSessionBeanB - Entering slowMethod
INFO: *** Exiting testBeanManagedConcurrency
INFO: *** Exiting SingletonClientServlet
INFO: SingletonSessionBeanB - Entering fastMethod
INFO: SingletonSessionBeanB - Exiting fastMethod
INFO: SingletonSessionBeanB - Exiting slowMethod
INFO: SingletonSessionBeanB - Exiting slowMethod
```

- Observe the following and the order of the events:
 - The SingletonClientServlet is entered.
 - The *testContainerManagedConcurrency* method is entered.
 - The two methods *slowMethod* and *fastMethod* are called on the singleton session bean in that order.
 - The execution of the *slowMethod* is started.
 - The *testContainerManagedConcurrency* method is exited.
 - The SingletonClientServlet is exited.
 - The execution of the *fastMethod* is started.
 - The execution of *fastMethod* is finished.
 - Finally, the execution of the *slowMethod* is finished.
 - The execution of the *slowMethod* and the *fastMethod* methods does overlap – the *fastMethod* is executed while *slowMethod* is executed.

Note that:

- The *SingletonSessionBeanA* class is annotated with the `@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)` annotation. Using this annotation or by omitting the `@ConcurrencyManagement` annotation altogether, instances of the singleton session bean will have container-managed concurrency.
- The *SingletonSessionBeanB* class is annotated with the `@ConcurrencyManagement(ConcurrencyManagementType.BEAN)` annotation. Using this annotation, instances of the singleton session bean will have bean-managed concurrency.

4.6.2. Concurrency Management

As we have seen, there are two different types of concurrency management for singleton session beans:

- Container-managed concurrency.
- Bean-managed concurrency.

The type of concurrency management is configured using either an annotation or in the ejb-jar.xml deployment descriptor.

The @ConcurrencyManagement Annotation

The concurrency management mode of a singleton session bean can be configured using the `@ConcurrencyManagement` annotation or the `<concurrency-management-type>` element in the ejb-jar.xml deployment descriptor.

The `@ConcurrencyManagement` annotation has one single element:

Element	Description
value	Chooses between bean- or container-managed concurrency management using one of the following values: <code>ConcurrencyManagementType.BEAN</code> <code>ConcurrencyManagementType.CONTAINER</code> Default is container-managed concurrency.

While container-managed concurrency is, as the name hints, container-managed the developer still has some opportunities to influence the concurrent behaviour of the singleton session bean, as we will see in subsequent sections.

With bean-managed concurrency, the container allows full concurrent access and it is left up to the bean developer to impose any restrictions, as will be discussed further in the section on bean-managed concurrency.

Container-Managed Concurrency

Container-managed concurrency means that the container controls the concurrent access to bean instances. The bean developer is able to further configure container-managed concurrency using the `@Lock` annotation.

The @Lock Annotation

The bean developer can control the concurrent access by applying the @Lock annotation to either the class or to individual methods in the bean using one of two values:

- LockType.READ
Any number of threads may simultaneously execute methods with read-locks.
- LockType.WRITE
Default value.
One single thread may execute in a method with write-lock. No other threads may execute in the same instance.

The following rules apply regarding inheritance of the @Lock annotation:

- A class-level @Lock annotation in the superclass applies to the (inherited) methods defined in the superclass.
- A method-level @Lock annotation in the superclass applies to the (inherited) method which it annotates.
- A method defined in a subclass does not inherit the @Lock annotation, neither that from the superclass nor from a method it overrides.
- The default annotation @Lock(LockType.WRITE) is implied.
This applies both at class and method level.
- Subclasses may annotate overridden methods using the @Lock annotation.
If overridden methods are not annotated, then @Lock(LockType.WRITE) is implied.

Threads that are not allowed to execute a method due to a lock being held are blocked until the lock is released, except when a timeout has been specified using the @AccessTimeout annotation.

The @AccessTimeout annotation can be specified at class-level, method-level or both. In the latter case, the method-level annotation takes precedence. The @AccessTimeout annotation has the following elements:

Element	Description
value	Long integer specifying the timeout time to wait for the lock allowing entry to a business method. Zero means concurrent access is not allowed and results in a <i>ConcurrentAccessException</i> if concurrent access is attempted. -1 means no timeout.
unit	Time unit of the above value. Default is TimeUnit.MILLISECONDS.

The @AccessTimeout annotation on methods in the superclass is inherited to methods defined in the superclass but is not inherited to overriding methods (defined in a subclass).

Container-Managed Concurrency Examples

To see how a change of concurrency locking attribute affects our example program introduced in the previous section, we make one single modification to the singleton session bean with container managed concurrency:

- Annotate the *SingletonSessionBeanA* class with an @Lock annotation:

```
...
@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Lock(LockType.READ)
public class SingletonSessionBeanA
{
...
}
```

- Run the example program using the URL
<http://localhost:8080/SingletonSessionBeanConcurrency/test.do?selector=1>
We should see the following console output:

```
INFO: *** Entering SingletonClientServlet
INFO: *** Entering testContainerManagedConcurrency
INFO: Calling slowMethod...
INFO: Calling fastMethod...
INFO: SingletonSessionBeanA - Entering slowMethod
INFO: *** Exiting testContainerManagedConcurrency
INFO: *** Exiting SingletonClientServlet
INFO: SingletonSessionBeanA - Entering fastMethod
INFO: SingletonSessionBeanA - Exiting fastMethod
INFO: SingletonSessionBeanA - Exiting slowMethod
```

Note that:

- The execution of the methods *slowMethod* and *fastMethod* now overlaps.
This is due to the change in the concurrency locking attribute of the methods in the *SingletonSessionBeanA* from LockType.WRITE, which was the default, to LockType.READ.

To see an example on how to add timeout to a business method in a singleton session bean, we modify the *SingletonSessionBeanA* class again:

- Change the lock type back to LockType.WRITE:

```
...
@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Lock(LockType.WRITE)
public class SingletonSessionBeanA
...
```

- Annotate the *fastMethod* with the @AccessTimeout annotation to have a timeout of 100 milliseconds:

```
...
@AccessTimeout(100)
public void fastMethod()
...
```

- Again, run the example program using the URL
<http://localhost:8080/SingletonSessionBeanConcurrency/test.do?selector=1>
We should see the following console output:

```

INFO: *** Entering SingletonClientServlet
INFO: *** Entering testContainerManagedConcurrency
INFO: Calling slowMethod...
INFO: Calling fastMethod...
INFO: SingletonSessionBeanA - Entering slowMethod
INFO: *** Exiting testContainerManagedConcurrency
INFO: *** Exiting SingletonClientServlet

WARNING: A system exception occurred during an invocation on EJB SingletonSessionBeanA
method public void com.ivan.scbcd6.SingletonSessionBeanA.fastMethod()
javax.ejb.ConcurrentAccessTimeoutException: Couldn't acquire a lock within 100
MILLISECONDS
    at
com.sun.ejb.containers.CMCSingletonContainer._getContext(CMCSingletonContainer.java:112)
...
INFO: SingletonSessionBeanA - Exiting slowMethod

```

Note that:

- The *fastMethod* never executes.
The thread wanting to invoke the method times out waiting to access the method and a *ConcurrentAccessTimeoutException* is thrown.
- The *slowMethod* executes normally.

Reentrant Locking

The following semantics applies to methods in singleton session beans being called by another method in the singleton session bean:

- If the calling method holds a WRITE lock on the same thread and calls a method has the lock type READ, then the call will proceed immediately, without releasing the original WRITE lock.
- If the calling method holds a WRITE lock on the same thread and calls a method has the lock type WRITE, then the call will proceed immediately, without releasing the original WRITE lock.
- If the calling method holds only a READ lock on the same thread and calls a method has the lock type READ, then the call will proceed immediately, without releasing the original READ lock.
- If the calling method holds only a READ lock on the same thread and calls a method has the lock type WRITE, then the an *IllegalLoopbackException* will be thrown.

Bean-Managed Concurrency

It is also possible for a singleton session bean to instruct the container to relinquish all concurrency-management by annotating a singleton session bean with the `@ConcurrencyManagement(ConcurrencyManagementType.BEAN)` annotation or using the `ejb-jar.xml` deployment descriptor.

With bean-managed concurrency, it is up to the bean developer to use standard Java mechanisms, such as the `synchronized` and `volatile` keywords as well as the concurrent utilities in the Java API. As the final example on singleton session bean concurrency, we'll show that by synchronizing the two business methods in the `SingletonSessionBeanB` of our previous example, it will display the same behaviour as the bean with container-managed concurrency.

- In the `SingletonSessionBeanB` class, modify the `slowMethod` and `fastMethod` methods to become synchronized:

```
...
@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class SingletonSessionBeanB
{
    public synchronized void slowMethod()
    {
        System.out.println("SingletonSessionBeanB - Entering slowMethod");
        waitSomeTime(10);
        System.out.println("SingletonSessionBeanB - Exiting slowMethod");
    }

    public synchronized void fastMethod()
    {
        System.out.println("SingletonSessionBeanB - Entering fastMethod");
        waitSomeTime(1);
        System.out.println("SingletonSessionBeanB - Exiting fastMethod");
    }
...
}
```

- Run the example program using the URL
<http://localhost:8080/SingletonSessionBeanConcurrency/test.do?selector=2>
We should see the following console output:

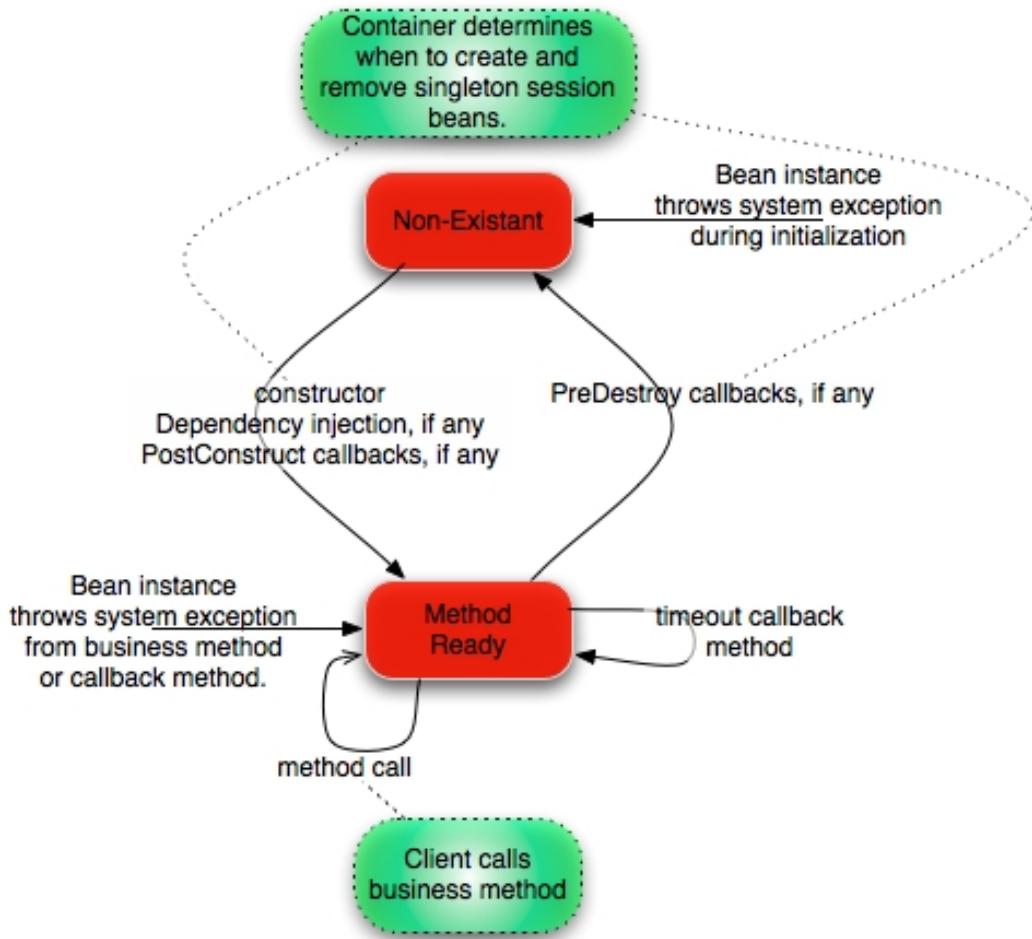
```
INFO: *** Entering SingletonClientServlet
INFO: *** Entering testBeanManagedConcurrency
INFO: Calling slowMethod...
INFO: Calling fastMethod...
INFO: SingletonSessionBeanB - Entering slowMethod
INFO: *** Exiting testBeanManagedConcurrency
INFO: *** Exiting SingletonClientServlet
INFO: SingletonSessionBeanB - Exiting slowMethod
INFO: SingletonSessionBeanB - Entering fastMethod
INFO: SingletonSessionBeanB - Exiting fastMethod
```

Note that:

- The `synchronized` keyword is used to control concurrent access to the two methods.
- The execution of the `fastMethod` and `slowMethod` no longer overlaps.

4.7. Singleton Session Bean Life-Cycle

The singleton session bean life-cycle is almost identical to the stateless session bean's life-cycle, except with the important difference that an instance of a singleton session bean is not destroyed if a system exception is thrown from a business method or from a callback method invoked when the bean is in the Method Ready state:



Life-cycle of a singleton session bean.

The red boxes denote different states of a singleton session bean.

The green boxes denote client events that causes a singleton session bean to change state.

Text that is not boxed and not colored denote (callback) methods that are invoked on a singleton session bean in connection to a state transition.

4.8. Operations Allowed in Methods of a Singleton Session Bean

References: EJB 3.1 Specification, section 4.8.6.

There are certain restrictions of what can be done in different kinds of method of a singleton session bean. The following table lists different kinds of methods and what operations they are allowed to perform.

The notes “BMT only” or “CMT only” means that these operations are only available when the bean has been managed, respective container managed, transactions.

JNDI access indicates whether the context “java:comp/env” is accessible.

If a singleton session bean attempts to perform an illegal operation, an *IllegalStateException* will be thrown in the following two cases:

- Invoke a method in the *SessionContext* interface not permitted.
- Invoke a method in the *Timer* or *TimerService* interfaces not permitted.

Bean Method(s)	Allowed Operations
Constructor	None.
Dependency injection methods (setter methods).	SessionContext: lookup. JNDI Access: Available
PostConstruct, PreDestroy, PrePassivate, PostActivate methods (lifecycle callback methods).	SessionContext: getBusinessObject, lookup, getContextData, getTimerService, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only). JNDI Access: Available Resource managers: Accessible. Other EJBs: Accessible. EntityManagerFactory: Accessible. EntityManager: Accessible. Timer service & methods: Accessible. UserTransaction methods: Accessible (BMT only).
Business method from any view or business method interceptor method.	SessionContext: getBusinessObject, getCallerPrincipal, isCallerInRole, lookup, getContextData, getInvokedBusinessInterface, wasCancelCalled, getTimerService, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only). JNDI Access: Available Resource managers: Accessible. Other EJBs: Accessible.

	EntityManagerFactory: Accessible. EntityManager: Accessible. Timer service & methods: Accessible. UserTransaction methods: Accessible (BMT only).
Business methods from web service endpoint.	SessionContext: getBusinessObject, getCallerPrincipal, isCallerInRole, lookup, getContextData, getTimerService, getMessageContext, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only). MessageContext methods: Available JNDI Access: Available Resource managers: Accessible. Other EJBs: Accessible. EntityManagerFactory: Accessible. EntityManager: Accessible. Timer service & methods: Accessible. UserTransaction methods: Accessible (BMT only).
Timeout callback method.	SessionContext: getBusinessObject, getCallerPrincipal, isCallerInRole, lookup, getContextData, getTimerService, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only). JNDI Access: Available Resource managers: Accessible. Other EJBs: Accessible. EntityManagerFactory: Accessible. EntityManager: Accessible. Timer service & methods: Accessible. UserTransaction methods: Accessible (BMT only).

In addition to the constraints specified in the above table, the *getRollbackOnly* and *setRollbackOnly* methods may only be invoked from within a transaction, otherwise an *IllegalStateException* will be thrown.

5. Create a Session Bean Client

References: EJB 3.1 Specification, chapter 3.

There are three different types of clients to session beans, namely:

- Local clients.
- Remote clients.
- Web service clients.

The different types of clients never access session bean instances directly. All access is done through the session bean's client view.

It is possible for a session bean to have multiple client views but usually only one is supplied.

In this chapter we will look at local and remote clients, but since web service clients of EJBs does not differ from regular web service clients, there will be no examples of web service clients in this book.

5.1. Local Session Bean Clients

Session beans with a local view can be accessed by local clients, which are clients packed in the same application and running in the same JVM as the session bean. Examples of local session bean clients are:

- Other EJBs.
- Servlets.
- Web services.

A local session bean is accessed in the same manner regardless of the type of client. The difference is whether the client obtains the reference to the session bean using dependency injection or whether the reference is looked up using JNDI.

Parameters and return values in a local view of an EJB are passed by reference – see the section on [Session Bean Method Parameters](#) for more information.

5.1.1. Local Session Bean Client with Dependency Injection

In this section we will create a local client, a servlet, to the Greeting EJB created [earlier](#).

- Create a Dynamic Web Project in Eclipse, as described in [appendix A](#). I'll call my project "LocalSessionBeanClient".
- Create the package `com.ivan.scbcd6`.
- Create the EJB class `StatefulSession1Bean` in the package created above. The EJB is implemented as follows:

```
package com.ivan.scbcd6;

import java.util.Date;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Stateful;

/**
 * Simplest possible stateful session bean exposing a local, no-interface view.
 */
@Stateful
@LocalBean
public class StatefulSession1Bean
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatefulSession1Bean " + mInstanceNumber +
            " created.");
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateful session bean " +
            mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }

    /**
     * Processes the supplied list.
     * The purpose of this method is to illustrate the difference
     * in parameter passing semantics between local and remote EJBs.
     *
     * @param inList List to process.
     */
    public void processList(List<String> inList)
    {
        String theListStrings = "";

        for (String theString : inList)
        {
            theListStrings += theString + ", ";
        }
        System.out.println("\nStatefulSession1Bean.processList: " +
            "The list contains: [" + theListStrings + "]");
    }
}
```

```

        * Add string to list.
        * If parameter passing is by reference, the client will also be able
        * to see this string, but if parameter passing is by value, then
        * this modification to the list is local only.
        */
        inList.add("String added in EJB");
    }
}

```

- Create the package `com.ivan.scbcd6.client`.
- Create the client class `LocalEJBClientServlet` in the newly created package.
The servlet is implemented as follows:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.StatefulSession1Bean;

/**
 * Servlet implementing a local EJB client.
 */
@WebServlet(name = "LocalEJBClientServlet", urlPatterns = "/test.do")
public class LocalEJBClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB
    private StatefulSession1Bean mLocalSessionBean;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        PrintWriter theResponseWriter = inResponse.getWriter();

        String theResponse = mLocalSessionBean.greeting("Anonymous");
        theResponseWriter.println("Response from the EJB: " + theResponse);

        /* Process a list to examine parameter passing semantics. */
        List<String> theList = new ArrayList<String>();
        theList.add("string 1");
        theList.add("string 2");
        theList.add("last string");
        mLocalSessionBean.processList(theList);

        /* Output list after EJB invocation. */
        String theListStrings = "";

        for (String theString : theList)
        {
            theListStrings += theString + ", ";
        }
        System.out.println("\nList after having invoked EJB processList: [" +
                           theListStrings + "]");
    }
}

```

- In the Package or Project Explorer in Eclipse, right-click the project and select Run As -> Run on Server...
When being asked about which server to run the web application on, select the GlassFish v3 server.
- In a browser, enter the following URL (or click the link):
<http://localhost:8080/LocalSessionBeanClient/test.do>
Note that the URL may differ due to having chosen another name for the project.
- A message similar to this should appear in the browser:

```
Response from the EJB: Hello Anonymous, the time is now: Wed Jul 07 18:42:23 CEST 2010
```

- In the console, the following output should be generated:

```
...
INFO: *** StatefulSession1Bean 1 created.
INFO: StatefulSession1Bean.processList: The list contains: [string 1, string 2, last
string, ]
INFO: List after having invoked EJB processList: [string 1, string 2, last string,
String added in EJB, ]
...
```

Note that the string added to the list passed as parameter to the *processMethod* method of the EJB in the EJB is output by the servlet after the method has finished executing. We can thus confirm that method parameters are passed by reference to local EJBs.

In the servlet implementation class, note that:

- The servlet implementation class is annotated with the @WebServlet annotation, which is part of the Servlet 3.0 standard. A JavaEE 6 container is thus required when deploying this web project.
- There is an instance field of the type *StatefulSession1Bean* that is annotated with an @EJB annotation.
The EJB that the servlet is the client of does not have an interface, thus the servlet has a no-interface view of the EJB. In a case like this, we must use the actual EJB implementation class when declaring the field holding the reference to the EJB.
In this case, the @EJB annotation annotates an instance field in which an EJB reference is to be injected. More about injection of EJB references in [the section on Dependency Injection and Resource Lookup in chapter 14](#).
- All the requests from the servlet will be done to one and the same EJB, despite the fact that the EJB is a stateful session bean which is said to only have one single client.
However, there will be only one single instance of the servlet class serving all requests and thus only one single client of the EJB. The container will manage the requests to the session bean as to allow only one single thread at a time to execute in the bean.

An alternative way of performing dependency injection into EJB components using only the ejb-jar.xml deployment descriptor. Please refer to the section on [dependency injection](#) in the chapter on [session beans](#) for more details.

5.1.2. Portable JNDI Names for Session Beans

References: EJB 3.1 Specification, section 4.4, JavaEE 6 Specification EE5.2.2.

The Java EE specification defines a number of standardized JNDI names by which a session bean is required to register. Being standardized means that these JNDI name are portable and thus available in any EJB container adhering to the EJB 3.1 specifications. The syntax of the different kinds of names are:

JNDI Name Syntax	Scope	Comments
<code>java:global[<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]</code>	Global	<p>Global namespace. The app-name part is only required if the EJB is packed in an EAR file. The module of an EJB is the WAR or JAR file it is packaged in. The bean-name is the name of the EJB, as can be specified with the @EJB annotation. The fully-qualified-interface-name must only be specified if the session bean has two or more of the interfaces/views listed below.</p>
<code>java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]</code>	Application. Client and session bean deployed in the same application.	<p>Application-specific namespace. The module-name part is always required.</p>
<code>java:module/<bean-name>[!<fully-qualified-interface-name>]</code>	Module. Client and session bean deployed in the same module of an application.	Module-specific namespace.

In addition, there also is the `java:comp` JNDI namespace. All components, except for components contained in web modules, have a `java:comp` namespace of their own.

All components contained in a web module share one and the same component namespace.

Depending on the different views presented by a session bean, the container registers one JNDI name for each of the following views:

- Local business interface.
- Remote business interface.
- No-interface view.
- EJB 2.x local home interface.
- EJB 2.x remote home interface.

The existence of a global JNDI name for a local business interface or a no-interface view does not mean that access to those entries are allowed from other applications.

In the example program in the next section, in which a servlet accesses a stateful session bean, the session bean can be looked up using any of the following JNDI names:

- java:global/LocalSessionBeanClient/StatefulSession1Bean
- java:global/LocalSessionBeanClient/StatefulSession1Bean!
com.ivan.scbcd6.StatefulSession1Bean
- java:app/LocalSessionBeanClient/StatefulSession1Bean
- java:app/LocalSessionBeanClient/StatefulSession1Bean!
com.ivan.scbcd6.StatefulSession1Bean
- java:module/StatefulSession1Bean
- java:module/StatefulSession1Bean!com.ivan.scbcd6.StatefulSession1Bean

The JNDI name under which a session bean can be accessed may be altered by using a deployment descriptor. All the above JNDI names assume that the EJB is deployed without a deployment descriptor altering its JNDI name(s).

5.1.3. Local Session Bean Client with JNDI Lookup

Using the same project setup as when looking at local session bean clients with dependency injection [above](#), we create a new servlet implementation replacing the one that used dependency injection:

```
package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.StatefulSession1Bean;

/**
 * Servlet implementing a local EJB client.
 */
@WebServlet(name = "LocalEJBClientServlet", urlPatterns = "/test.do")
public class LocalEJBClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    private StatefulSession1Bean mLocalSessionBean;

    /**
     * Initializes the servlet after creation.
     */
    @PostConstruct
    public void intialize()
    {
        /*
         * Use the JNDI API to look up a reference to the session bean
         * the servlet is a client of.
         * Any of the following JNDI names can be used:
         * java:global/LocalSessionBeanClient/StatefulSession1Bean
         * java:global/LocalSessionBeanClient/StatefulSession1Bean!
         */
        com.ivan.scbcd6.StatefulSession1Bean
            * java:app/LocalSessionBeanClient/StatefulSession1Bean
            * java:app/LocalSessionBeanClient/StatefulSession1Bean!
        com.ivan.scbcd6.StatefulSession1Bean
            * java:module/StatefulSession1Bean
            * java:module/StatefulSession1Bean!com.ivan.scbcd6.StatefulSession1Bean
        */
        try
        {
            InitialContext theInitialContext = new InitialContext();
            mLocalSessionBean =
                (StatefulSession1Bean)theInitialContext
                    .lookup("java:module/StatefulSession1Bean");
        } catch (NamingException theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
```

```

System.out.println("EJB reference: " + mLocalSessionBean);
System.out.println("EJB reference type: "
    + mLocalSessionBean.getClass());
PrintWriter theResponseWriter = inResponse.getWriter();

String theResponse = mLocalSessionBean.greeting("Anonymous");
theResponseWriter.println("Response from the EJB: " + theResponse);

/* Process a list to examine parameter passing semantics. */
List<String> theList = new ArrayList<String>();
theList.add("string 1");
theList.add("string 2");
theList.add("last string");
mLocalSessionBean.processList(theList);

/* Output list after EJB invocation. */
String theListStrings = "";

for (String theString : theList)
{
    theListStrings += theString + ", ";
}
System.out.println("\nList after having invoked EJB processList: [" +
    theListStrings + "]");
}
}

```

When running the example program as described [earlier](#), it should produce result identical to that of the previous example program with dependency injection.

In the servlet implementation class, note that:

- The instance variable *mLocalSessionBean* is no longer annotated.
- There is a method named *initialize* which is annotated with the `@PostConstruct` annotation. This method will be invoked by the servlet container immediately after the servlet instance has been created and before it starts to serve requests.
In this method, a reference to the stateful session bean is looked up using the JNDI API.
- As in the previously, any of the listed JNDI names can be used to retrieve a reference to the stateful session bean.
You are encouraged to try some different JNDI names.

5.2. Remote Session Bean Clients

In order to be accessible by remote clients, a session bean must provide a remote view. A remote view is location independent – that is, it is accessed using the same API, regardless of whether the client is a local client (for instance, another EJB in the same application) or a remote client (for instance, a standalone application). The following types of remote session bean clients exists:

- JavaEE clients deployed in the same or in another container.
Examples of such clients are EJBs, servlets and web services.
- JavaSE clients, such as applets or standalone applications.
- Non-Java clients.
The client view of a session bean can be mapped to for instance CORBA, allowing for clients implemented in other languages.

Parameters and return values in the remote business interface of an EJB are passed by value – see the section on [Session Bean Method Parameters](#) for more information.

5.2.1. Create the Remotely Accessible EJB

Before looking at the different kinds of clients accessing EJBs with a remote view, we will create an EJB that can be accessed remotely:

The session bean with a remote view is created in a project of its own, since it has a remote view and may thus be accessed by clients external to the application in which it is deployed.

- Create an EJB project in Eclipse, as described in [appendix B](#).
I name my project “RemoteSessionBean”.
- In the new project, create the session bean implementation class as below:

```
package com.ivan.scbcd6;

import java.util.Date;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.Stateful;

/**
 * Simplest possible stateful session bean exposing a remote business
 * interface.
 */
@Stateful
public class StatefulSession1Bean implements StatefulSession1Remote
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatefulSession1Bean " + mInstanceNumber +
                           " created.");
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
```

```

        Date theCurrentTime = new Date();

        String theMessage = "Hello " + inName + ", I am stateful session bean " +
            mInstanceNumber + ". The time is now: " + theCurrentTime;
        return theMessage;
    }

    /**
     * Processes the supplied list.
     * The purpose of this method is to illustrate the difference
     * in parameter passing semantics between local and remote EJBs.
     *
     * @param inList List to process.
     */
    public void processList(List<String> inList)
    {
        String theListStrings = "";

        for (String theString : inList)
        {
            theListStrings += theString + ", ";
        }
        System.out.println("\nStatefulSession1Bean.processList: " +
            "The list contains: [" + theListStrings + "]");

        /*
         * Add string to list.
         * If parameter passing is by reference, the client will also be able
         * to see this string, but if parameter passing is by value, then
         * this modification to the list is local only.
         */
        inList.add("String added in EJB");
    }
}

```

- Create the remote business interface:

```

package com.ivan.scbcd6;

import java.util.List;
import javax.ejb.Remote;

/**
 * Remote business interface of the StatefulSession1 EJB.
 */
@Remote
public interface StatefulSession1Remote
{

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName);

    /**
     * Processes the supplied list.
     * The purpose of this method is to illustrate the difference
     * in parameter passing semantics between local and remote EJBs.
     *
     * @param inList List to process.
     */
    public void processList(List<String> inList);
}

```

- Deploy the project to the GlassFish application server.

Having deployed the project, we can in the server log see that the session bean can be accessed using any of the following portable JNDI names:

```

...
INFO: Portable JNDI names for EJB StatefulSession1Bean :
[java:global/RemoteSessionBean/StatefulSession1Bean!
com.ivan.scbcd6.StatefulSession1Remote,
java:global/RemoteSessionBean/StatefulSession1Bean]
...

```

There are also a number of non-portable JNDI names, but we will not use those since they are specific to GlassFish.

5.2.2. Remote Java EE Client with Dependency Injection

In this example we will develop a servlet in a web application project that acts as a client of a session bean with a remote interface deployed in an application of its own. We will develop the session bean in a project of its own, an EJB Project, and the client in another project, a Dynamic Web Project.

- Create a dynamic web project as described in [appendix A](#). I call my project “RemoteJavaEESessionBeanClient”.
- Copy the session bean's remote interface to the package *com.ivan.scbcd6*. A better way is to have a library containing the interface(s) etc shared by the different projects, but since this is only a small example program, the interface is copied.
- Implement the client servlet class:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.StatefulSession1Remote;

/**
 * Servlet implementing a remote EJB client.
 */
@WebServlet(name = "RemoteEJBClientServlet", urlPatterns = "/test.do")
public class RemoteEJBClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB(lookup = "java:global/RemoteSessionBean/StatefulSession1Bean")
    private StatefulSession1Remote mRemoteSessionBean;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        PrintWriter theResponseWriter = inResponse.getWriter();

        String theRequestNameParam = inRequest.getParameter("name");
        if (theRequestNameParam == null)
        {
            theRequestNameParam = "Anonymous";
        }
    }
}

```

```

        }
        String theResponse = mRemoteSessionBean.greeting(theRequestNameParam);
        theResponseWriter.println("Response from the EJB: " + theResponse);

        /* Process a list to examine parameter passing semantics. */
        List<String> theList = new ArrayList<String>();
        theList.add("string 1");
        theList.add("string 2");
        theList.add("last string");
        mRemoteSessionBean.processList(theList);

        /* Output list after EJB invocation. */
        String theListStrings = "";

        for (String theString : theList)
        {
            theListStrings += theString + ", ";
        }
        System.out.println("\nList after having invoked EJB processList: [ " +
            theListStrings + " ]");
    }
}

```

- In the Package or Project Explorer in Eclipse, right-click the project and select Run As -> Run on Server...
When being asked about which server to run the web application on, select GlassFish v3.
- In a browser, enter the following URL (or click the link):
<http://localhost:8080/RemoteJavaEESessionBeanClient/test.do>
Note that the URL may differ if you have chosen a different name for the project.
- A message similar to this should appear in the browser:

```
Response from the EJB: Hello Anonymous, I am stateful session bean 1. The time is now:
Mon Nov 08 17:17:43 CET 2010
```

- In the console, the following output should be generated:

```
...
INFO: *** StatefulSession1Bean 1 created.
INFO: StatefulSession1Bean.processList: The list contains: [string 1, string 2, last
string, ]
INFO: List after having invoked EJB processList: [string 1, string 2, last string, ]
```

Note that:

- There is an instance field of the type *StatefulSession1Remote* that is annotated with an @EJB annotation.
The type of the field is an interface type which is annotated with the @Remote annotation – the remote business view of the EJB.
In this case, the @EJB annotation annotates an instance field in which an EJB reference is to be injected. More about injection of EJB references in [the section on Dependency Injection and Resource Lookup in chapter 14](#).
- The @EJB annotation specifies a value for the *lookup* element.
As in the section on the [@EJB annotation](#), this is the JNDI name used to resolve the dependency to inject.
If both applications run in the same GlassFish v3 server, specifying the JNDI name is not necessary.
- The contents of the list passed as a parameter to the *processList* method of the EJB is the same before and after the method call.
This confirms that parameters passed to the remote business view of an EJB is passed by value, rather than by reference.
Thus the list is copied (rather: serialized and unserialized) when the method is invoked.

5.2.3. Remote Java EE Client with JNDI Lookup

Using the example in the previous section, but replacing the dependency injection with JNDI lookup, as seen in the section [Local Session Bean Client with JNDI Lookup](#) and choosing the appropriate, remotely available, JNDI name it becomes trivial to develop a remote Java EE session bean client that uses JNDI lookup to obtain reference(s) to the EJB(s) it uses.

Writing such a client is left as an exercise for the reader.

5.2.4. Remote Java SE Client with JNDI Lookup

Deploying the EJB that is to be remotely accessible to GlassFish v3, as we have done, and using the appropriate client runtime library makes the development of a standalone client application that can invoke the EJB through its remote business view quite simple.

- Create a new Java project.
I call my project `RemoteJavaSESessionBeanClient`, though the name does not matter.
- Include the GlassFish v3 client runtime library JAR on the classpath.
The library can be found in: `$GLASSFISH_HOME/modules/gf-client.jar`
- Create the package `com.ivan.scbcd6`.
- Copy the EJB's remote interface `StatefulSession1Remote` from the [remote EJB project](#) to the package we just created.
- Create the package `com.ivan.scbcd6.seclient`.
- Implement the Java SE client class:

```
package com.ivan.scbcd6.seclient;

import java.util.ArrayList;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.ivan.scbcd6.StatefulSession1Remote;

/**
 * Remote Java SE EJB client that uses JNDI lookup to obtain references
 * to EJBs.
 *
 * Note that the following JAR must be included on the client's class path:
 * GlassFish 3: $GLASSFISH_HOME/modules/gf-client.jar
 */
public class RemoteSESessionBeanClient
{
    /* Constant(s): */
    private final static String REMOTE_EJB_JNDI =
        "java:global/RemoteSessionBean/StatefulSession1Bean";

    /* Instance variable(s): */
    /** Remotely accessed EJB. */
    private StatefulSession1Remote mRemoteSessionBean;

    /**
     * Looks up the remote EJB using its JNDI name.
     *
     * @throws NamingException If error occurs during lookup.
     */
    private void lookupEJB() throws NamingException
    {
        InitialContext theContext = new InitialContext();

        System.out.println("**** Starting Remote EJB Lookup...");

        mRemoteSessionBean = (StatefulSession1Remote) theContext
            .lookup(REMOTE_EJB_JNDI);

        System.out.println("    Remote EJB Lookup Finished.");
    }
}
```

```

}

/**
 * Invokes the remote EJB and outputs results.
 */
private void invokedRemoteEJB()
{
    String theResponse = mRemoteSessionBean.greeting("Java SE");
    System.out.println("*** Response from the EJB: " +
        theResponse);

    /* Process a list to examine parameter passing semantics. */
    List<String> theList = new ArrayList<String>();
    theList.add("string 1");
    theList.add("string 2");
    theList.add("last string");
    mRemoteSessionBean.processList(theList);

    /* Output list after EJB invocation. */
    String theListStrings = "";

    for (String theString : theList)
    {
        theListStrings += theString + ", ";
    }
    System.out.println("*** List after having invoked EJB processList: [ " +
        theListStrings + " ]");
}

public static void main(String[] args)
{
    RemoteSESessionBeanClient theClient = new RemoteSESessionBeanClient();
    try
    {
        theClient.lookupEJB();
        theClient.invokedRemoteEJB();
    } catch (Exception theException)
    {
        theException.printStackTrace();
    }
}
}

```

- Launch the *RemoteSESessionBeanClient* class.
In the Package Explorer, right-click it and select Run As -> Java Application.
- If everything worked as expected, console output similar to this should be printed:

```

*** Starting Remote EJB Lookup...
Nov 8, 2010 5:52:45 PM com.sun.enterprise.transaction.JavaEETransactionManagerSimplified
initDelegates
INFO: Using com.sun.enterprise.transaction.jts.JavaEETransactionManagerJTSDelegate as
the delegate
    Remote EJB Lookup Finished.
*** Response from the EJB: Hello Java SE, I am stateful session bean 5. The time is now:
Mon Nov 08 17:52:47 CET 2010
*** List after having invoked EJB processList: [string 1, string 2, last string, ]

```

Note that:

- The GlassFish v3 client runtime library gf-client.jar must be included on the classpath.
- Creating the InitialContext used to look up a reference to the remote EJB requires no parameters.
This is possible as long as the client runs on the same computer as the GlassFish v3 server.
- The JNDI lookup code is similar to that of the [Local Session Bean Client with JNDI Lookup](#) seen earlier.
The only difference is that the type of the variable in which the reference to the EJB is stored is the remote interface type.
- The location of the EJB is completely transparent to the client.
The only clue to the location of the EJB we get from looking at the code is that an interface type annotated with @Remote is used to reference the EJB.

5.2.5. Remote Java SE Client with Dependency Injection

It is possible to develop a Java SE client which take advantage of dependency injection. Such a client application is deployed to the GlassFish server and launched using Java Web Start.

Details developing application clients are left as an exercise to the reader – the NetBeans IDE includes wizards that makes this task significantly easier.

5.3. Client View of EJB 3.1 Session Beans

References: EJB 3.1 Specification, chapter 3.

In this slightly more theoretical section we will look at how clients of session beans developed according to the 3.1 version of the EJB specification interact with such EJBs. Some comparisons will be made with EJB 2.1 session beans.

Some examples of clients of session beans are:

- Servlets.
- Web services.
- Java SE applications.
- Other EJBs.

5.3.1. Obtaining References to Session Beans

Before a client can use a session bean, it has to obtain a reference to the appropriate view of the session bean. The mechanism for obtaining a reference to a session bean is the same, regardless of the kind of view that is referenced. More about the different kinds of views of session beans in a [subsequent](#) section.

The two ways to obtain a reference to a session bean are:

- Dependency injection.
An instance field or setter-method in the client is annotated and the container ensures that the appropriate reference is injected at runtime.
- JNDI lookup.
Using the JNDI name of the bean, a reference to the bean is looked up programmatically from within the client's code.

Please refer to the section on [Dependency Injection and Resource Lookup](#) in chapter 14 for more information!

5.3.2. Views of Session Beans

Session beans are accessed through the no-interface view or through their business interface view. The no-interface view is always local while the business interface view may be remote or local.

Clients never access session bean instances directly, but always through the view of a session bean. The view can be thought of as a proxy which the client must use to be able to use the session bean. This mechanism is transparent to the client.

Clients of session beans may be one of the following:

Client Type	Comments
Remote	<p>The client view is the remote business interface of the session bean. The remote business interface is a regular Java interface that is not required to extend any other interface.</p> <p>The bean may be located in the same or in a different JVM as the client.</p> <p>Parameters are passed by value. Parameters and return values must be serializable.</p>
Local	<p>The client view is either the local business interface or the no-interface view of the session bean. The no-interface view contains all the public methods of the bean implementation class. The local business interface is a regular Java interface that is not required to extend any other interface.</p> <p>The bean is located in the same JVM as the client and packaged in the same application as the client.</p>
Web Service	<p>Only stateless session beans and singleton session beans can provide a web service client view. The remote business interface may also be used as the web service endpoint interface.</p> <p>This kind of clients will not be discussed in this book.</p>

Compare the above with EJB 2.1 session beans, which also has remote and local home interfaces, in addition to local and remote interfaces of the session beans.

It is possible for a session bean to have more than one client view; for instance it can give local clients access to the no-interface view and expose its functionality as a web service.

5.3.3. Method Parameters

The developer of an EJB, as well as the developer of the client of the EJB, must take into consideration the method parameters in the EJB. Whether an EJB is accessed remotely or locally decides the parameter-passing semantics.

- Local EJB Access
Parameters and return values are passed by reference.
- Remote EJB Access
Parameters and return values are passed by value.

Local Access Method Parameters

As mentioned earlier, parameters and return values of methods in the local client view are passed by reference and will thus be shared between the client and the session bean. The following should thus be avoided:

- Objects being part of the state of a session bean must not be part of the return data.
Objects being part of the state of an EJB that are to contribute to the return data must be copied (deep copy) and the copy included in the return data.
- Objects being part of parameter data received when the session bean is invoked must not become part of the state of a session bean.
Again, such objects must be copied (deep copy) and the copies may then be assigned to the state of the EJB.

Remote Access Method Parameters

The risk of sharing references to objects between a client and an EJB does not exist with remotely accessed EJBs, as the parameters and return values are passed by value.

However, the types of all parameters and return values of an EJB with a remote view must be serializable.

EJB Reference Parameters and Return Values

References: EJB 3.1 Specification, sections 3.4.3 and 3.4.4.

References to the remote business interface of a session bean may be passed as a parameter or return value to methods in the local view, no-interface view and remote view of other session beans. References to the local view or the no-interface view of a session bean may only be passed as a parameter or return value of methods in the local or no-interface view of other session beans.

5.3.4. Session Bean Identity

References: EJB 3.1 Specification, section 3.4.7.

Clients of session beans can compare references to session beans. Instead of using the *isIdentical* methods in the *EJBLocalObject* or *EJBObject* interfaces to compare references to session beans, it is now possible to test EJB references for identity using the *equals* and *hashCode* methods that originate from the *Object* class.

To examine the characteristics when comparing references of session beans, we develop a small sample web application:

- Create a dynamic web project in Eclipse, as described in [appendix A](#).
- In the package *com.ivan.scbcd6*, create the following four session bean implementation classes:

```
package com.ivan.scbcd6;

import javax.ejb.EJB;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class IdentityCheckingBean
{
    @EJB
    private StatelessSessionBean mStateless1;
    @EJB
    private StatelessSessionBean mStateless2;
    @EJB
    private StatefulSessionBean mStatefull1;
    @EJB
    private StatefulSessionBean mStatefull2;
    @EJB
    private SingletonSessionBean mSingleton1;
    @EJB
    private SingletonSessionBean mSingleton2;
    @EJB
    private StatelessSessionBeanLocalBusiness mStatelessLocalBusiness;

    public void checkBeanIdentities()
    {
        /* Compare stateless bean with stateless bean. */
        System.out.println("\n***** Compare stateless bean with stateless bean:");
        doIdentityCheck(mStateless1, mStateless1, "Stateless1 equals stateless1");
        doIdentityCheck(mStateless1, mStateless2, "Stateless1 equals stateless2");
        doIdentityCheck(mStateless2, mStateless1, "Stateless2 equals stateless1");
        doIdentityCheck(mStateless1, mStatelessLocalBusiness,
                        "Stateless1 equals stateless local business");

        /* Compare stateless bean with other kinds of beans. */
        System.out.println("\n***** Compare stateless bean with other kinds of beans:");
        doIdentityCheck(mStateless1, mStatefull1, "Stateless1 equals statefull1");
        doIdentityCheck(mStateless1, mSingleton1, "Stateless1 equals singleton1");

        /* Compare stateful bean with stateful bean. */
        System.out.println("\n***** Compare stateful bean with stateful bean:");
        doIdentityCheck(mStatefull1, mStatefull1, "Statefull1 equals statefull1");
        doIdentityCheck(mStatefull1, mStatefull2, "Statefull1 equals statefull2");
        doIdentityCheck(mStatefull2, mStatefull1, "Statefull2 equals statefull1");

        /* Compare stateful bean with other kinds of beans. */
        System.out.println("\n***** Compare stateless bean with other kinds of beans:");
        doIdentityCheck(mStatefull1, mSingleton1, "Statefull1 equals singleton1");

        /* Compare singleton bean with singleton bean. */
        System.out.println("\n***** Compare singelton bean with singleton bean:");
        doIdentityCheck(mSingleton1, mSingleton1, "Singleton1 equals singleton1");
        doIdentityCheck(mSingleton1, mSingleton2, "Singleton1 equals singleton2");
        doIdentityCheck(mSingleton2, mSingleton1, "Singleton2 equals singelton1");
```

```

}

public void checkBeanHashCodes()
{
    /* Hash codes of stateless beans. */
    System.out.println("\n***** Stateless bean hash codes:");
    doBeanHashCode(mStateless1, "Stateless1");
    doBeanHashCode(mStateless2, "Stateless2");
    doBeanHashCode(mStatelessLocalBusiness, "Stateless local business");

    /* Hash codes of stateful beans. */
    System.out.println("\n***** Stateful bean hash codes:");
    doBeanHashCode(mStatefull1, "Statefull1");
    doBeanHashCode(mStatefull2, "Statefull2");

    /* Hash codes of singleton beans. */
    System.out.println("\n***** Singelton bean hash codes:");
    doBeanHashCode(mSingleton1, "Singleton1");
    doBeanHashCode(mSingleton2, "Singleton2");
}

private void doIdentityCheck(Object inBean1, Object inBean2,
    String inMessage)
{
    boolean theIdentityFlag;

    theIdentityFlag = inBean1.equals(inBean2);
    System.out.println("      " + inMessage + " " + theIdentityFlag);
}

private void doBeanHashCode(Object inBean, String inMessage)
{
    int theHashCode = inBean.hashCode();

    System.out.println("      " + inMessage + " hash code: " + theHashCode);
}
}

```

```

package com.ivan.scbcd6;

import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class StatelessSessionBean implements StatelessSessionBeanLocalBusiness
{
    public StatelessSessionBean()
    {
        System.out.println("***** StatelessSessionBean created");
    }

    @PostConstruct
    public void initialize()
    {
        System.out.println("***** StatelessSessionBean initialized");
    }
}

```

```

package com.ivan.scbcd6;

import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Stateful;

@Stateful
@LocalBean
public class StatefulSessionBean
{
    public StatefulSessionBean()
    {
        System.out.println("***** StatefulSessionBean created");
    }
}

```

```

    }

    @PostConstruct
    public void initialize()
    {
        System.out.println("***** StatefulSessionBean initialized");
    }
}

```

```

package com.ivan.scbcd6;

import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;

@Singleton
@LocalBean
public class SingletonSessionBean
{
    public SingletonSessionBean()
    {
        System.out.println("***** SingletonSessionBean created");
    }

    @PostConstruct
    public void initialize()
    {
        System.out.println("***** SingletonSessionBean initialized");
    }
}

```

- Implement the local business interface of the stateless session bean.
This bean will have two different views; a no-interface view and a local business interface view.

```

package com.ivan.scbcd6;

import javax.ejb.Local;

@Local
public interface StatelessSessionBeanLocalBusiness
{
}

```

- In the package *com.ivan.scbcd6.client*, implement the following servlet:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.IdentityCheckingBean;

@WebServlet(name = "IdentityCheckingServlet", urlPatterns = "/checkidentity.do")
public class IdentityCheckingServlet extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    @EJB
    private IdentityCheckingBean mIdentityChecker;

    @Override
    protected void doGet(HttpServletRequest inRequest,

```

```

        HttpServletResponse inResponse) throws ServletException, IOException
    {
        PrintWriter theResponseWriter = inResponse.getWriter();

        mIdentityChecker.checkBeanIdentities();
        mIdentityChecker.checkBeanHashCodes();
        theResponseWriter
            .println("Identity checks and hash code checks result printed to console.");
    }
}

```

- Deploy the project to the GlassFish application server.
- In a web browser, go to the URL
<http://localhost:8080/StatefulSession1Web/checkidentity.do>
- In the web browser, you should see the message from the above servlet.
- In the GlassFish log you should see the following output, or similar, among other messages:

```

INFO: ***** StatefulSessionBean created
INFO: ***** StatefulSessionBean created
INFO: ***** StatefulSessionBean initialized
INFO: ***** StatefulSessionBean created
INFO: ***** StatefulSessionBean created
INFO: ***** StatefulSessionBean initialized
INFO: ***** Compare stateless bean with stateless bean:
INFO:     Stateless1 equals stateless1 true
INFO:     Stateless1 equals stateless2 true
INFO:     Stateless2 equals stateless1 true
INFO:     Stateless1 equals stateless local business: false
INFO: ***** Compare stateless bean with other kinds of beans:
INFO:     Stateless1 equals statefull false
INFO:     Stateless1 equals singleton1 false
INFO: ***** Compare stateful bean with stateful bean:
INFO:     Statefull1 equals statefull1 true
INFO:     Statefull1 equals stateful2 false
INFO:     Stateful2 equals statefull1 false
INFO: ***** Compare stateless bean with other kinds of beans:
INFO:     Statefull1 equals singleton1 false
INFO: ***** Compare singelton bean with singleton bean:
INFO:     Singleton1 equals singleton1 true
INFO:     Singleton1 equals singleton2 true
INFO:     Singleton2 equals singleton1 true
INFO: ***** Stateless bean hash codes:
INFO:     Stateless1 hash code: 364806942
INFO:     Stateless2 hash code: 364806942
INFO:     Stateless local business hash code: 1157496838
INFO: ***** Stateful bean hash codes:
INFO:     Statefull1 hash code: 2444462009
INFO:     Stateful2 hash code: 536768552
INFO: ***** Singelton bean hash codes:
INFO:     Singleton1 hash code: 2037736064
INFO:     Singleton2 hash code: 2037736064

```

The conclusions to draw from this example, which match the behaviour stipulated by the EJB 3.1 specification, are:

- A reference to a session bean is equal to another, or the same, reference to the same session bean, provided the references are to views of the same type.
This is true for all three types of session beans.
- Two references to session beans to different views are never equal.
This is true for all three types of session beans.
- References to any two stateless session beans of the same type and to the same type of view are always equal.

- Two references to different types of session beans are never equal.
- References to any two singleton session beans of the same type and to the same type of view are always equal.
- References to two different stateful session beans are never equal.
This even if the two beans are of the same type and the references are to the same type of view.
- Hash codes of references to any two stateless session beans of the same type and to the same type of view are always equal.
- Hash codes of references to any two session beans to different type of views are always different.
- Hash codes of references to two different stateful session beans of the same type are always different.
- Hash codes of references to any two singleton session beans of the same type and to the same type of view are always equal.

5.4. Exceptions from EJBs

References: EJB 3.1 Specification, sections 4.7.3, 14.3 and 14.4.

Hopefully, the invocations from a client of a session bean will proceed without problems.

Occasionally, however, there are problems of some kind and an exception will be thrown.

In this section we'll look at the types of exceptions that the client of a session bean will see under different circumstances.

5.4.1. EJB Clients, Transactions and Exceptions

A client of an EJB receiving an application or system exception should note that the following cases exists in connection to transactions:

- A system exception was thrown due to an error on the client side trying to call an EJB.
Any transaction will not be marked for rollback or rolled back.
- A system exception was thrown due to an error occurring executing the EJB.
Any transaction will be marked for rollback by the container.
- An application exception was thrown during the execution of the EJB.
Any transaction may or may not have been marked for rollback by the EJB throwing the exception..
The application exception may have been configured to cause the container to roll back any transaction when the exception is being thrown – see the `@ApplicationException` annotation.

A recommendation that applies both in connection to application and system exceptions and transactions is that an EJB client that receives an exception, be it an application or a system exception, queries the transaction status prior to performing any lengthy, or in any other ways costly, processing. This prevents doing unnecessary work that is later rolled back.

5.4.2. Application Exceptions

A client to an EJB that receives an application exception may continue to call the EJB.

Application exceptions from web service methods of stateless session beans are mapped to SOAP faults specified in the WSDL. For details on how SOAP faults are mapped to exceptions on the client side, please consult the JAX-WS specification.

5.4.3. System Exceptions

System exceptions may be thrown for the following reasons:

- An error occurred on the client side issuing a request to an EJB.
A problem on the client side prevented the request to the EJB to be sent.
Any transaction will not be marked for rollback.
- An error occurred executing the request to the EJB.
An error prevented the EJB from completing the request.
Any transaction will be marked for rollback.

The following table lists common system exceptions.

Exception	Transaction Status	Notes
The following exceptions are the basic system exceptions.		
EJBException	May or may not have been marked for rollback.	Not possible to determine whether EJB method has executed or not. Error may have occurred during communication with, or processing in, the EJB.
RemoteException	May or may not have been marked for rollback.	See EJBException.
The following exceptions indicate that the transaction has been marked for rollback and will never commit.		
EJBTransactionRolledbackException	Rolled back or marked for rollback.	Subclass of EJBException. EJB invoked using EJB 3.1 view and EJB executing in client's transaction.
TransactionRolledbackLocalException	Rolled back or marked for rollback.	Subclass of EJBException. EJB invoked using EJB 2.1 local client view.
TransactionRolledbackException	Rolled back or marked for rollback.	Subclass of RemoteException. JTA standard exception. EJB invoked using EJB 2.1 remote view or web service view.
The following exceptions indicate that an attempt was made to invoke a method that require a client transaction without the client having a transaction context. These exceptions usually indicate an error in method transaction requirement declaration(s).		
EJBTransactionRequiredException	No transaction.	Subclass of EJBException. EJB invoked using EJB 3.1 view.
TransactionRequiredLocalException	No transaction.	Subclass of EJBException. EJB invoked using EJB 2.1 local client view.
TransactionRequiredException	No transaction.	Subclass of RemoteException. JTA standard exception. EJB invoked using EJB 2.1 remote view or web service view.
The following exceptions indicate that an attempt was made to invoke a method on an EJB object that does not exist.		
NoSuchEJBException	Rolled back or marked for rollback.	Subclass of EJBException. EJB invoked using EJB 3.1 view.
NoSuchObjectLocalException	Rolled back or marked for rollback.	Subclass of EJBException. EJB invoked using EJB 2.1 local client view.
NoSuchObjectException	Rolled back or marked for rollback.	Subclass of RemoteException. EJB invoked using EJB 2.1 remote client view.

RemoteException

The *RemoteException* occurs in the following places related to EJBs:

- EJB 2.1 Remote Home Interfaces.
- EJB 2.1 Remote Component Interfaces.
- JAX-RPC Web Service Endpoint Interfaces.

Local EJB 2.1 interfaces and EJB 3.1 client view methods may not declare *RemoteException*.

6. Create Interceptors for an EJB

References: EJB 3.1 Specification, chapter 12. The Interceptors 1.1 Specification.

For those familiar with [AOP \(Aspect Oriented Programming\)](#), interceptors are similar to AOP for EJBs, enabling us to apply around-advice to certain kinds of methods of session beans and message driven beans.

For those not familiar with AOP, interceptors can be likened to filters inserted and executed when certain methods are invoked on a session bean or a message driven bean. The filter can perform additional processing before and after a method is being invoked. The filter may even select not to invoke the intercepted method.

6.1. Interceptor Example

Before going on to details of EJB interceptors, we are going to look at an example. A classic AOP example is to add logging that prints a message just before entering a method and immediately after having exited the method. This section will demonstrate how to use an EJB interceptor to accomplish this.

The example program may, with all right, seem familiar – the singleton session bean example from chapter four has been reused with some modifications. All the parts of the project is present in this section, so there is no need to refer to the earlier example.

6.1.1.A Session Bean and its Client

First we'll develop a singleton session bean and its client, a servlet, in a web project.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “BusinessMethodInterceptors”.
- In the package *com.ivan.scbcd6*, create the session bean implementation class according to the following listing:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.annotation.PostConstruct;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;

/**
 * This class implements a singleton session bean with the ability
 * to hold state.
 */
@Singleton
@LocalBean
public class SingletonSessionBeanA
{
    /* Constant(s): */
    private final static String BEAN_NAME = "SingletonSessionBeanA";

    /* Instance variable(s): */
    private String mStoredMessage = "[no message set]";

    @PostConstruct
    public void intialize()
    {
        System.out.println("**** " + BEAN_NAME + " - Initialized");
    }

    @PreDestroy
    public void cleanUp()
    {
        System.out.println("**** " + BEAN_NAME + " - Destroyed");
    }
}
```

```

    }

    public String retrieveMessage()
    {
        Date theCurrentTime = new Date();

        return "Message from " + BEAN_NAME + " - " + mStoredMessage + " "
               + theCurrentTime;
    }

    public void storeMessage(final String inStoredMessage)
    {
        mStoredMessage = inStoredMessage;
    }
}

```

- In the *com.ivan.scbcd6.client* package, create the servlet that is to be the client of the singleton session beans we just implemented:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.SingletonSessionBeanA;

/**
 * Servlet acting as a client of the singleton session bean.
 */
@WebServlet(name = "SingletonClientServlet", urlPatterns = "/test.do")
public class SingletonClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;
    private final static String STORE_ACTION = "store";
    private final static String CLEAR_ACTION = "clear";

    /* Instance variable(s): */
    @EJB
    private SingletonSessionBeanA mSingletonBeanA;

    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        System.out.println("***** Entering SingletonClientServlet");

        String theRequestNameParam = inRequest.getParameter("name");
        String theRequestActionParam = inRequest.getParameter("action");

        /* Set default name if none provided. */
        if (theRequestNameParam == null || theRequestNameParam.equals(""))
        {
            theRequestNameParam = "Anonymous Coward";
        }

        /* Generate output from servlet using session beans. */
        PrintWriter theResponseWriter = inResponse.getWriter();
        String theMessage;

        theMessage = mSingletonBeanA.retrieveMessage();
        theResponseWriter.println(theMessage);

        /*
         * Store or clear data in one of the singleton session beans
         * if the supplied action so specifies.
         */
        if (theRequestActionParam != null)

```

```

    {
        if (STORE_ACTION.equals(theRequestActionParam))
        {
            mSingletonBeanA.storeMessage(theRequestNameParam);
        }
        if (CLEAR_ACTION.equals(theRequestActionParam))
        {
            mSingletonBeanA.storeMessage("[Cleared]");
        }
    }

    System.out.println("***** Exiting SingletonClientServlet");
    theResponseWriter.println("Finished invoking singleton session beans!");
}
}

```

- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/BusinessMethodInterceptors/test.do>
You should see the following messages printed to the console:

```

INFO: ***** Entering SingletonClientServlet
INFO: *** SingletonSessionBeanA - Initialized
INFO: ***** Exiting SingletonClientServlet

```

If it is not the first time a request is issued to the above URL then there will be no initialization message printed from the SingletonSessionBeanA.

All in all the program does what we expect it to do, but we cannot see any log messages.

6.1.2. Adding a Logging Interceptor Using Annotations

In order to have log messages printed to the console, we will add an EJB interceptor. In this section annotations will be used to specify the interceptor class and which EJB it is to be intercepted.

- In the `com.ivan.scbcd6` package create a class named `LogInterceptor` implemented as below. Note the `@Interceptor` annotation that tells the EJB container that this is an interceptor. Also note the `@AroundInvoke` annotation. This annotation tells the EJB container that the annotated method is an interceptor method that is to be applied to business methods.

```
package com.ivan.scbcd6;

import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class LogInterceptor
{
    public LogInterceptor()
    {
        System.out.println("LogInterceptor - Constructor");
    }

    @AroundInvoke
    public Object logMethodEntryExit(InvocationContext inInvocationContext) throws
Exception
    {
        System.out.println("  LogInterceptor - Entering method: " +
inInvocationContext.getMethod().getName());

        /* Invoke the intercepted method on the EJB and save the result. */
        Object theResult = inInvocationContext.proceed();

        System.out.println("  LogInterceptor - Exiting method: " +
inInvocationContext.getMethod().getName());

        /* Return the result from the intercepted method. */
        return theResult;
    }
}
```

- Add the annotation `@Interceptors` to the `SingletonSessionBeanA` class as shown in the listing below. This annotation tells the EJB container that we want to apply one or more interceptors to the annotated class.

```
...
@Singleton
@LocalBean
@Interceptors(LogInterceptor.class)
public class SingletonSessionBeanA
{
    /* Constant(s): */
...
```

- Deploy the project to the GlassFish application server.

- In a browser, issue a request to the following URL:
<http://localhost:8080/BusinessMethodInterceptors/test.do>
 You should now see the following messages printed to the console:

```
INFO: ***** Entering SingletonClientServlet
INFO: LogInterceptor - Constructor
INFO: *** SingletonSessionBeanA - Initialized
INFO: LogInterceptor - Entering method: retrieveMessage
INFO: LogInterceptor - Exiting method: retrieveMessage
INFO: ***** Exiting SingletonClientServlet
```

- In a browser, issue a request to the following URL:
<http://localhost:8080/BusinessMethodInterceptors/test.do?name=Ivan&action=store>
 You should then see the following messages printed to the console:

```
INFO: ***** Entering SingletonClientServlet
INFO: LogInterceptor - Entering method: retrieveMessage
INFO: LogInterceptor - Exiting method: retrieveMessage
INFO: LogInterceptor - Entering method: storeMessage
INFO: LogInterceptor - Exiting method: storeMessage
INFO: ***** Exiting SingletonClientServlet
```

Note that:

- Business methods, like *retrieveMessage* and *storeMessage*, of the *SingletonSessionBeanA* are intercepted.
- PreDestroy and PostConstruct methods of the *SingletonSessionBeanA* are not intercepted.

6.1.3. Adding a Default Interceptor

Default interceptors are interceptors that intercept calls to all the EJBs within the ejb-jar file or .war file that the default interceptors are declared. In this section we will add a default interceptor to our example program. Such interceptors can only be specified using the ejb-jar.xml deployment descriptor, so we will also get an opportunity to see an example of how interceptors are specified in the deployment descriptor.

- In the package *com.ivan.scbcd6*, add the Java class that implements the default interceptor.

```
package com.ivan.scbcd6;

import javax.interceptor.InvocationContext;

public class MyDefaultInterceptor
{
    public Object aroundInvoke(InvocationContext inInvocationContext) throws Exception
    {
        System.out.println("  MyDefaultInterceptor intercepting: " +
                           inInvocationContext.getTarget().getClass().getSimpleName() +
                           "." + inInvocationContext.getMethod().getName());

        return inInvocationContext.proceed();
    }
}
```

- In the WebContent/WEB-INF directory add, if not already present, the ejb-jar.xml deployment descriptor and make sure it has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

  <!--
      The <interceptors> method contains the specification of all
      the interceptor classes in the module.
  -->
  <interceptors>
    <!--
        In an <interceptor> element a class is specified as being
        an interceptor class and the different interceptor method(s)
        are also specified.
        The <interceptor> element corresponds, at least partially,
        to the @Interceptor annotation.
    -->
    <interceptor>
      <interceptor-class>com.ivan.scbcd6.MyDefaultInterceptor</interceptor-class>
      <!--
          The <around-invoke> element corresponds to the @AroundInvoke
          annotation, specifying at most one method in the interceptor
          class that is to intercept business method calls on EJBs.
      -->
      <around-invoke>
        <method-name>aroundInvoke</method-name>
      </around-invoke>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <!--
        This element is used to specify the target(s) to which
        a specified interceptor is to be applied.
        In the case of a default interceptor, we use the wildcard *
        whe specifying the name of the EJB.
    -->
    <interceptor-binding>
      <ejb-name>*</ejb-name>
    </interceptor-binding>
  </assembly-descriptor>

```

```

<interceptor-class>com.ivan.scbcd6.MyDefaultInterceptor</interceptor-class>
</interceptor-binding>
</assembly-descriptor>
</ejb-jar>

```

- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/BusinessMethodInterceptors/test.do?name=Ivan&action=store>
 You should then see the following messages being printed to the console:

```

INFO: ***** Entering SingletonClientServlet
INFO: LogInterceptor - Constructor
INFO: *** SingletonSessionBeanA - Initialized
INFO: MyDefaultInterceptor intercepting: SingletonSessionBeanA.retrieveMessage
INFO: LogInterceptor - Entering method: retrieveMessage
INFO: LogInterceptor - Exiting method: retrieveMessage
INFO: MyDefaultInterceptor intercepting: SingletonSessionBeanA.storeMessage
INFO: LogInterceptor - Entering method: storeMessage
INFO: LogInterceptor - Exiting method: storeMessage
INFO: ***** Exiting SingletonClientServlet

```

Note that:

- The default interceptor class does not contain any annotations.
- The <interceptor> element in the ejb-jar.xml deployment descriptor specifies a class as being an interceptor class and which method is to be the around-invoke interceptor method.
- One <interceptor-binding> element is used to link an interceptor class to the target EJB(s) which method calls are to be intercepted.
- When specifying the name of the target EJB in the <interceptor-binding> element, we used the wildcard character “*”.
 This means that all the EJBs in the same module (ejb-jar file or .war file) are to be intercepted.
- Looking at the console output, we can see that the default interceptor is invoked before the log interceptor.
 We will look more closely at the order in which interceptors are invoked later in this chapter.
- We see that it is possible to retrieve the name of the target object and the method that is intercepted.

6.1.4. Adding an Interceptor Superclass

An interceptor class may inherit from a parent class that itself is an interceptor class. Any interceptor methods in the superclass will also be invoked.

In this section we'll add a superclass to the log interceptor and observe the behaviour of the example program.

- In the package `com.ivan.scbcd6`, add the following class:

```
package com.ivan.scbcd6;

import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class LogInterceptorSuperclass
{
    @AroundInvoke
    public Object logSuper(InvocationContext inInvocationContext) throws Exception
    {
        System.out.println("  LogInterceptorSuperclass intercepting: " +
                           inInvocationContext.getTarget().getClass().getSimpleName() +
                           "." + inInvocationContext.getMethod().getName());

        return inInvocationContext.proceed();
    }
}
```

- Modify the `LogInterceptor` class to inherit from the `LogInterceptorSuperclass` we just added.

```
...
@Interceptor
public class LogInterceptor extends LogInterceptorSuperclass
{
...
}
```

- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/BusinessMethodInterceptors/test.do?name=Ivan&action=store>
You should then see the following messages being printed to the console:

```
INFO: ***** Entering SingletonClientServlet
INFO: LogInterceptor - Constructor
INFO: MyDefaultInterceptor.postConstruct
INFO: *** SingletonSessionBeanA - Initialized
INFO: MyDefaultInterceptor intercepting: SingletonSessionBeanA.retrieveMessage
INFO: LogInterceptorSuperclass intercepting: SingletonSessionBeanA.retrieveMessage
INFO: LogInterceptor - Entering method: retrieveMessage
INFO: LogInterceptor - Exiting method: retrieveMessage
INFO: MyDefaultInterceptor intercepting: SingletonSessionBeanA.storeMessage
INFO: LogInterceptorSuperclass intercepting: SingletonSessionBeanA.storeMessage
INFO: LogInterceptor - Entering method: storeMessage
INFO: LogInterceptor - Exiting method: storeMessage
INFO: ***** Exiting SingletonClientServlet
```

Note that:

- The *LogInterceptorSuperclass* looks, and indeed is, a regular interceptor class.
- We did not modify the list of interceptor classes in the `@Interceptors` annotation annotating the *SingletonSessionBeanA* class.
- The *LogInterceptorSuperclass* is not included in the list of interceptors specified in the `@Interceptors` annotation in the *SingletonSessionBeanA* class.
- In the console output, we can see that the *LogInterceptorSuperclass* interceptor is invoked after the *MyDefaultInterceptor* interceptor and before the *LogInterceptor* interceptor.
We will look more closely at the order in which interceptors are invoked [later](#) in this chapter.

6.1.5. Adding a Life-Cycle Event Interceptor

Life-cycle event interceptor methods are not intercepting any methods being called on the target EJB, instead these are notification methods that are called at certain points in the life of an EJB. We have already seen life-cycle event interceptor methods – one example is methods annotated with the `@PostConstruct` annotation.

In this section we will add a life-cycle event interceptor to the default interceptor.

- In the *MyDefaultInterceptor* class, add a method named *postConstruct* implemented as below. Note that the method should also be annotated with the `@PostConstruct` annotation.

```
...
public class MyDefaultInterceptor
{
    public Object aroundInvoke(InvocationContext inInvocationContext) throws Exception
    {
        ...
    }

    @PostConstruct
    public void postConstruct(InvocationContext inInvocationContext)
        throws Exception
    {
        System.out.println("    MyDefaultInterceptor.postConstruct");

        /*
         * Important!
         * Must call proceed, in order for the other interceptor methods
         * in the interceptor chain to become invoked.
         */
        inInvocationContext.proceed();
    }
}
```

- Deploy the project to the GlassFish application server.

- In a browser, issue a request to the following URL:

<http://localhost:8080/BusinessMethodInterceptors/test.do>

You should then see the following messages being printed to the console:

```
INFO: **** Entering SingletonClientServlet
INFO: LogInterceptor - Constructor
INFO: MyDefaultInterceptor.postConstruct
INFO: *** SingletonSessionBeanA - Initialized
INFO: MyDefaultInterceptor intercepting: SingletonSessionBeanA.retrieveMessage
INFO: LogInterceptorSuperclass intercepting: SingletonSessionBeanA.retrieveMessage
INFO: LogInterceptor - Entering method: retrieveMessage
INFO: LogInterceptor - Exiting method: retrieveMessage
INFO: **** Exiting SingletonClientServlet
```

Note that:

- The post-construct life-cycle event interceptor method in the default interceptor class is annotated with the `@PostConstruct` annotation.
Life-cycle event interceptor methods, as all interceptor methods, can be located either in separate interceptor classes or in the target class (or both).
- The post-construct life-cycle event interceptor in the default interceptor class is called the first time a request is issued to the URL, but not for subsequent requests.
This matches the expected behaviour of our application – the singleton session bean in our example application is created once and once only.
- The post-construct life-cycle event interceptor method in the default interceptor class invokes the `proceed()` method on the `InvocationContext` object supplied as a parameter.
This is to ensure that post-construct life-cycle event interceptor methods in other interceptor classes are also invoked.
- The post-construct life-cycle event interceptor method in the EJB bean class *SingeltonSessionBeanA* is called once, after the post-construct life-cycle event interceptor method in the default interceptor class has been invoked.
We'll examine interceptor invocation order in a [subsequent section of this chapter](#).

6.1.6. Adding a Timeout Method Interceptor

A timeout method is a method called when an EJB timer is triggered. We will examine timeout methods and scheduling of business logic in the chapter [Scheduling Execution of Business Logic](#), so no details will be given here regarding the timer.

We start by adding a timeout callback method to our singleton session bean.

- In the *SingletonSessionBeanA* class, add a timeout callback method as below. Remember to make sure all the necessary import-statements are in the class.

```
...  
    @Schedule(second="*/5", minute="*", hour="*")  
    public void doPeriodic()  
    {  
        System.out.println("**** Do periodic: " + (new Date()));  
    }  
...
```

- Deploy the project to the GlassFish application server.
- The timeout callback method should start printing messages to the console every five seconds. If it does not, access the following URL to start it up:
<http://localhost:8080/BusinessMethodInterceptors/test.do>

```
...  
INFO: **** Do periodic: Mon Sep 13 07:05:45 CEST 2010  
INFO: *** Do periodic: Mon Sep 13 07:05:50 CEST 2010  
INFO: *** Do periodic: Mon Sep 13 07:05:55 CEST 2010  
INFO: *** Do periodic: Mon Sep 13 07:06:00 CEST 2010  
...
```

- In the *LogInterceptor* class, add the following timeout method interceptor. Again, make sure that the necessary import-statements are in the class.

```
...  
    @AroundTimeout  
    public Object logTimeout(InvocationContext inInvocationContext)  
        throws Exception  
    {  
        System.out.println(" LogInterceptor - Entering timeout: "  
            + inInvocationContext.getMethod().getName());  
  
        Object theResult = inInvocationContext.proceed();  
  
        System.out.println(" LogInterceptor - Exiting timeout: "  
            + inInvocationContext.getMethod().getName());  
  
        return theResult;  
    }  
...
```

- Deploy the project to the GlassFish application server.
- The messages from the timeout callback method are now surrounded with the messages from the timeout method interceptor:

```
...  
INFO: LogInterceptor - Entering timeout: doPeriodic  
INFO: *** Do periodic: Mon Sep 13 07:11:35 CEST 2010  
INFO: LogInterceptor - Exiting timeout: doPeriodic  
  
INFO: LogInterceptor - Entering timeout: doPeriodic  
INFO: *** Do periodic: Mon Sep 13 07:11:40 CEST 2010  
INFO: LogInterceptor - Exiting timeout: doPeriodic  
  
INFO: LogInterceptor - Entering timeout: doPeriodic  
INFO: *** Do periodic: Mon Sep 13 07:11:45 CEST 2010  
INFO: LogInterceptor - Exiting timeout: doPeriodic  
...
```

Note that:

- The timeout method interceptor is annotated with the `@AroundTimeout` annotation.
- The signature of the timeout method interceptor is identical to the signature of business method interceptors, that is:
`Object someMethodName(InvocationContext) throws Exception`
- The timeout method interceptor uses the `InvocationContext.proceed` method to invoke the timeout callback method.

6.2. Interceptor Types

This section lists the different types of interceptors available to an EJB developer and some details on each of the different interceptor types. First, the following interceptor types are available:

- Business method interceptors.
These apply to business methods of session beans as well as message listener methods of message driven beans.
- Timeout method interceptors.
- Life-cycle event interceptors.

All the different interceptor methods have the following common properties:

- Method visibility may be public, private, protected or package.
- Must not be declared as final or static.
- Can access the same components and resources as the method being intercepted.
For life-cycle event interceptors, see the different sections on operations allowed in the different bean types; [stateful session beans](#), [stateless session beans](#), [singleton session beans](#), [message driven beans](#).

6.2.1. Business Method Interceptors

The following table lists some properties of business method interceptors.

Question	Answer
May be defined on which EJB Type(s).	Session beans and message driven beans.
Intercept which method types.	Session bean business methods and message driven bean's message listener methods.
Location of interceptor methods.	In special interceptor class, interceptor superclass, or in the target class.
Restrictions on interceptor methods.	At most one per class.
Annotation(s) use to define interceptor method(s).	@AroundInvoke
Interceptor method security context.	Same security context as intercepted business method.
Intercept method transaction context.	Same transaction context as intercepted business method.
Exceptions interceptor methods may throw.	Exceptions declared in the throws-clause of the intercepted business method and system exceptions.
Interceptor method signature.	Object someMethodName(InvocationContext) throws Exception
Interceptor method visibility.	Public, protected, private or package visibility

6.2.2. Timeout Method Interceptors

The following table lists some properties of timeout method interceptors.

Question	Answer
May be defined on which EJB Type(s).	Session beans and message driven beans.
Intercept which method types.	EJB timer timeout methods of session bean and message driven bean.
Location of interceptor methods.	In special interceptor class, interceptor superclass, or in the target class.
Restrictions on interceptor methods.	At most one per class.
Annotation(s) use to define interceptor method(s).	@AroundTimeout
Interceptor method security context.	Same security context as intercepted timeout method.
Interceptor method transaction context.	Same transaction context as intercepted timeout method.
Exceptions interceptor methods may throw.	Exceptions declared in the throws-clause of the intercepted timeout method and system exceptions.
Interceptor method signature.	Object someMethodName(InvocationContext) throws Exception
Interceptor method visibility.	Public, protected, private or package visibility

6.2.3. Life-Cycle Event Interceptors

The following table lists some properties of life-cycle event interceptors.

Question	Answer
May be defined on which EJB Type(s).	Session beans and message driven beans.
Intercept which method types.	EJB timer timeout methods of session bean and message driven bean.
Location of interceptor methods.	In special interceptor class, interceptor superclass, or in the target class.
Restrictions on interceptor methods.	At most one of each kind per class.
Annotation(s) use to define interceptor method(s).	@PostConstruct, @PreDestroy, @PostActivate and @PrePassivate. Multiple annotations may be applied to one method.
Interceptor method security context.	Unspecified.
Interceptor method transaction context.	Unspecified except with singleton session beans @PostConstruct and @PreDestroy methods, for which transaction attribute may be specified.
Exceptions interceptor methods may throw.	System exceptions.
Interceptor method signature.	In interceptor class: void someMethodName(InvocationContext) In target class: void someMethodName()

6.3. Interceptor Programming Restrictions

The programming restrictions listed in the section [EJB Programming Restrictions](#) also applies to EJB interceptors.

6.4. Interceptor Life-Cycle and Concurrency Behaviour

The life-cycle of an interceptor is the same as the life-cycle of the EJB that it is associated with. This includes passivation and activation of an interceptor associated with a stateful session bean that is being passivated and later activated.

The lifecycle and concurrency behaviour of an interceptor associated with a singleton session bean has the same lifecycle and concurrency behaviour as the singleton session bean. Care should also be taken when considering storing state in an interceptor that is to be applied to a singleton session bean – please refer to the section on [Singleton Session Bean Concurrency](#) for detailed information.

6.5. Interceptors and Exceptions

As discussed in the section on [Interceptor Types](#) above, different interceptor methods are allowed to throw different kinds of exceptions.

If a system exception propagates out of the chain of interceptors, then the EJB instance in question and all its interceptors are destroyed except in the case when the EJB is a singleton session bean. If the EJB instance is destroyed, any `@PreDestroy` callback method, as well as any interceptors of the `@PreDestroy` method, will not be invoked.

All interceptors are allowed to suppress exceptions by catching them and recover by calling `proceed()`.

Note that an exception caught by an interceptor was not necessarily thrown by the intercepted method, but may be thrown by an interceptor method further down the chain of interceptor methods.

6.6. Interceptors and Transactions

An interceptor method executes in the same transaction context as the method it is intercepting. Any transactions can be marked for rollback by throwing system exception or calling `EJBContext.setRollbackOnly()`.

Transaction can be marked for rollback before or after invoking the intercepted business method

6.7. Interceptors with Annotations

Interceptors can be defined using annotations, except for default interceptors that have to be defined in the ejb-jar.xml deployment descriptor. In this section we'll take a look at the different annotations available to us when working with interceptors.

6.7.1. The @AroundInvoke Annotation

The @AroundInvoke method annotates at most one method in an interceptor class, specifying the interceptor method that will be invoked when business methods are intercepted. The signature of a method annotated by the @AroundInvoke annotation is:

```
@AroundInvoke  
Object arbitraryMethodName(InvocationContext inInvocationContext) throws Exception  
{  
    ...  
}
```

Business method interceptor method properties have been discussed [earlier](#).

6.7.2. The @AroundTimeout Annotation

The @AroundTimeout method annotates at most one method in an interceptor class, specifying the interceptor method that will be invoked when timer timeout methods are intercepted. The signature of a method annotated by the @AroundTimeout annotation is:

```
@AroundTimeout  
Object arbitraryMethodName(InvocationContext inInvocationContext) throws Exception  
{  
    ...  
}
```

Timeout method interceptor method properties have been discussed [earlier](#).

6.7.3. The @ExcludeClassInterceptors Annotation

Annotates a business method or timer timeout method of an EJB to prevent any interceptors defined on the EJB class from being invoked when the annotated method is being called.

6.7.4. The @ExcludeDefaultInterceptors Annotation

Annotates an EJB class, a business method or a timer timeout method of an EJB to prevent any default interceptors from being invoked when the method in the class or the annotated method is being called.

6.7.5. The @Interceptor Annotation

Annotates a class specifying it as an interceptor class. This annotation is optional if the class is listed in a @Interceptors annotation on the target EJB class or if the ejb-jar.xml deployment descriptor is used to associate the interceptor class with a target EJB class.
Required when an interceptor binding is used (see @InterceptorBinding).

6.7.6. The @Interceptors Annotation

Annotates an EJB class or a business method or timer timeout method of an EJB class specifying the interceptor class(es) that are to intercept the methods in the class or the annotated method. When annotating a single method, then the annotation can only be used to specify business method or timer timeout method interceptors.

6.7.7. The @InterceptorBinding Annotation

Annotates an annotation, specifying it as being an interceptor binding type. The EJB 3.1 specification does not mandate support for this annotation.

6.8. Interceptors in the Deployment Descriptor

Not only can interceptors be declared using annotations, as we have seen, but they can also be specified in the ejb-jar.xml deployment descriptor. In addition to an interceptor that is being applied to one single EJB, the deployment descriptor allows for specification of so-called default interceptors that apply to all the EJBs in the containing module.

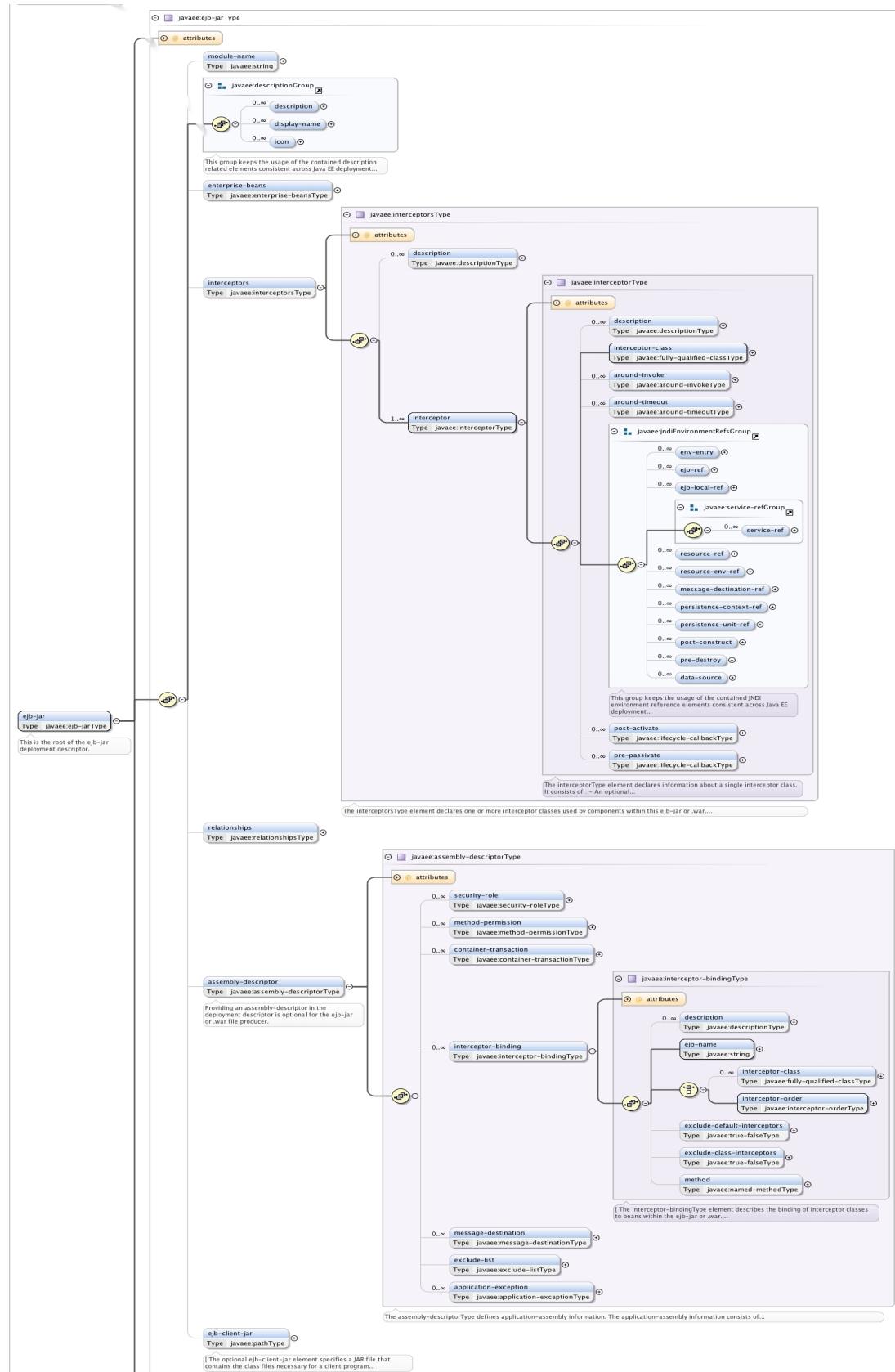
The following is an example of an interceptor specification in a deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <!--
        The <interceptors> method contains the specification of all
        the interceptor classes in the module.
    -->
    <interceptors>
        <!--
            In an <interceptor> element a class is specified as being
            an interceptor class and the different interceptor method(s)
            are also specified.
            The <interceptor> element corresponds, at least partially,
            to the @Interceptor annotation.
        -->
        <interceptor>
            <interceptor-class>com.ivan.scbcd6.MyDefaultInterceptor</interceptor-class>
            <!--
                The <around-invoke> element corresponds to the @AroundInvoke
                annotation, specifying at most one method in the interceptor
                class that is to intercept business method calls on EJBs.
            -->
            <around-invoke>
                <method-name>aroundInvoke</method-name>
            </around-invoke>
        </interceptor>
    </interceptors>

    <assembly-descriptor>
        <!--
            This element is used to specify the target(s) to which
            a specified interceptor is to be applied.
            In the case of a default interceptor, we use the wildcard *
            when specifying the name of the EJB.
        -->
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-class>com.ivan.scbcd6.MyDefaultInterceptor</interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

This figure shows the parts of the ejb-jar.xml deployment descriptors relevant to specifying interceptors:



Elements in the ejb-jar.xml deployment descriptor relevant to configuration of interceptors (`<interceptor>` and `<interceptor-binding>` elements).

6.9. Default Interceptors

Default interceptors are interceptors that can be specified to intercept multiple EJB classes. As before, default interceptors can only be specified using the ejb-jar.xml deployment descriptor. Default interceptors are invoked before any other type of interceptors, as we will see in the next section.

Please see the section [Adding a Default Interceptor](#) in the interceptor example above for an example how a default interceptor is defined.

6.10. Interceptor Order of Invocation

The following specifies the order in which interceptors are invoked. This applies to all the different interceptor types with the corresponding interceptor methods:

- Business methods interceptors – around-invoke methods.
- Timeout method interceptors – around-timeout methods.
- Lifecycle event interceptors – PostConstruct, PreDestroy, PrePassivate, PostActivate methods.

The order in which interceptors are invoked is as follows:

- Default interceptors.
Any default interceptors are invoked first in the order in which they are declared in the ejb-jar.xml deployment descriptor.
- Interceptors defined in @Interceptor annotated interceptor classes listed in an @Interceptors annotation on the target EJB class.
Such interceptor methods are invoked in the order they are declared in the @Interceptors annotation. Interceptor methods of superclasses to the interceptor classes are invoked first, starting with the most general superclass and traversing down the inheritance hierarchy.
- Interceptors defined in @Interceptor annotated interceptor classes listed in an @Interceptors annotation on the target method in an EJB.
Such interceptor methods are invoked in the order they are declared in the @Interceptors annotation annotating the EJB method to be invoked.
Interceptor methods of superclasses to the method-level interceptor classes are invoked first, starting with the most general superclass and traversing down the inheritance hierarchy.
- Interceptor methods defined in the target EJB class.
Such interceptor methods are invoked, starting with those defined in the most general superclass and traversing down the inheritance hierarchy until finally invoking the interceptor methods in the target EJB class itself.

Note that:

- The invocation order specified by annotations may be overridden using the ejb-jar.xml deployment descriptor.
- Interceptor methods overridden in a subclass will not be invoked, regardless of whether the overriding method is an interceptor method or not.

Important note!

When there are multiple interceptors of the same kind, for instance for post-construct life-cycle events, a interceptor chain is built and the first interceptor method in the chain is invoked by the container.

If an interceptor method takes an *InvocationContext* object parameter, then the *proceed()* method must be invoked in this object, in order for the other interceptors in the chain to be invoked. This applies to all kinds of interceptor methods!

7. Modify a Session Bean to Become Asynchronous

References: EJB 3.1 Specification, section 4.5.

This section describes how method(s) of a session bean can be modified to become asynchronous, the implications this have and in what way this affects the behaviour of the session bean.

7.1. Asynchronous Session Bean Example Program

We will first look at an example program that consists of one stateful session bean and one servlet, acting as a client to the session bean:

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project AsynchEJBInvocation.
- In the package *com.ivan.scbcd6.beans*, create the session bean implementation class implemented as below:

```
package com.ivan.scbcd6.beans;

import java.util.Date;
import java.util.concurrent.Future;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.LocalBean;
import javax.ejb.SessionContext;
import javax.ejb.Stateful;

/**
 * Stateful session bean with asynchronous methods.
 */
@Stateful
@LocalBean
public class AsynchStatefulSessionBean
{
    @Resource
    private SessionContext mSessionContext;

    @PostConstruct
    public void initialize()
    {
        System.out.println("***** AsynchStatefulSessionBean initialized");
    }
}
```

```

@Asynchronous
public Future<String> asynchWithException() throws Exception
{
    /* Trick the compiler to accept the method always throws an exception. */
    int i = 1;
    if (i < 2)
    {
        System.out
            .println("***** AsynchStatefulSessionBean asynchWithException throwing
exception");
        throw new Exception("Exception from AsynchStatefulSessionBean");
    }

    return new AsyncResult<String>("Never happens");
}

@Asynchronous
public Future<String> slowAsynch()
{
    String theResult = (new Date()).toString();

    waitSomeTime(5000L);

    System.out
        .println("***** AsynchStatefulSessionBean Exiting slowAsynch");
    return new AsyncResult<String>(theResult);
}

@Asynchronous
public void slowOneWayAsynch()
{
    waitSomeTime(5000L);

    System.out
        .println("***** AsynchStatefulSessionBean Exiting slowOneWayAsynch");
}

@Asynchronous
public Future<String> canBeCancelled()
{
    String theResult = "Not cancelled " + new Date();

    for (int i = 1; i < 100; i++)
    {
        waitSomeTime(100L);

        System.out
            .println("***** AsynchStatefulSessionBean canBeCancelled waited "
+ i);

        /* Check if client attempted to cancel the method. */
        if (mSessionContext.wasCancelCalled())
        {
            theResult = "Cancelled " + new Date();
            break;
        }
    }

    return new AsyncResult<String>(theResult);
}

private void waitSomeTime(long inDelayInMillisec)
{
    try
    {
        Thread.sleep(inDelayInMillisec);
    } catch (InterruptedException theException)
    {
        // Ignore exceptions.
    }
}

```

- Create the package *com.ivan.scbcd6.client*.

- Create the client class *AsynchClientServlet* in the package.

The servlet is implemented as follows:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.beans.AsynchStatefulSessionBean;

@WebServlet(name = "AsynchClientServlet", urlPatterns = "/asynch.do")
public class AsynchClientServlet extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    @EJB
    private AsynchStatefulSessionBean mAsynchBean;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        System.out.println("***** Entering AsynchClientServlet");

        Future<String> theSlowAsynchResult = null;
        Future<String> theAsynchWithExceptionResult = null;
        Future<String> theCanBeCancelledResult = null;

        PrintWriter theResponseWriter = inResponse.getWriter();

        try
        {
            /* Call void asynchronous method. */
            System.out
                .println("***** AsynchClientServlet - About to call slowOneWayAsynch");
            mAsynchBean.slowOneWayAsynch();
            System.out
                .println("***** AsynchClientServlet - Finished calling
slowOneWayAsynch");

            /* Call slow asynchronous method. */
            System.out
                .println("***** AsynchClientServlet - About to call slowAsynch");
            theSlowAsynchResult = mAsynchBean.slowAsynch();
            System.out
                .println("***** AsynchClientServlet - Finished calling slowAsynch");

            /* Call asynchronous method that will throw an exception. */
            System.out
                .println("***** AsynchClientServlet - About to call
asynchWithException");
            theAsynchWithExceptionResult = mAsynchBean.asynchWithException();
            System.out
                .println("***** AsynchClientServlet - Finished calling
asynchWithException");

            /* Call asynchronous method that can be canceled and cancel it. */
            System.out
                .println("***** AsynchClientServlet - About to call canBeCancelled");
            theCanBeCancelledResult = mAsynchBean.canBeCancelled();
            System.out
                .println("***** AsynchClientServlet - Finished calling canBeCancelled");
            waitSomeTime((long)(Math.random() * 1000) + 1000L);
            System.out
                .println("***** AsynchClientServlet - About to cancel canBeCancelled");
        }
    }
}

```

```

        theCanBeCancelledResult.cancel(true);
        System.out
            .println("***** AsynchClientServlet - Cancelled canBeCancelled");
    } catch (Exception theException)
    {
        System.out
            .println("***** AsynchClientServlet - An exception was thrown: "
                    + theException.getMessage());
    }

    try
    {
        /* Retrieve results from asynchronous invocations. */
        System.out
            .println("\n***** AsynchClientServlet - slowAsynch result: "
                    + theSlowAsynchResult.get());
        System.out
            .println("***** AsynchClientServlet - canBeCancelled result: "
                    + theCanBeCancelledResult.get());

        /* Wait for asynchWithException to complete. */
        while (!theAsynchWithExceptionResult.isDone())
        {
            System.out.println("      Waiting... ");
        }

        System.out
            .println("***** AsynchClientServlet - asynchWithException result: "
                    + theAsynchWithExceptionResult.get());
        theAsynchWithExceptionResult.get();
    } catch (InterruptedException e)
    {
        System.out.println("***** AsynchClientServlet - "
                + "An InterruptedException was thrown");
    } catch (ExecutionException e)
    {
        System.out.println("***** AsynchClientServlet - "
                + "An ExecutionException was thrown: " + e);
    }

    System.out.println("**** Exiting AsynchClientServlet");

    theResponseWriter
        .println("Finished invoking asynchronous session bean!");
    }

private void waitSomeTime(long inDelayInMillisec)
{
    try
    {
        Thread.sleep(inDelayInMillisec);
    } catch (InterruptedException theException)
    {
        // Ignore exceptions.
    }
}
}

```

- Deploy the project to GlassFish.
- Navigate to the URL <http://localhost:8080/AsynchEJBInvocation/asynch.do> in a browser. If your project has a different name, you have to use the appropriate URL and append “/asynch.do”.
- Wait until the view in the browser reads “Finished invoking asynchronous session bean!”.
- In the GlassFish log you should see the following output, or similar, among other messages:

```

INFO: **** Entering AsynchClientServlet
INFO: ***** AsynchClientServlet - About to call slowOneWayAsynch
INFO: ***** AsynchClientServlet - Finished calling slowOneWayAsynch
INFO: ***** AsynchClientServlet - About to call slowAsynch
INFO: ***** AsynchClientServlet - Finished calling slowAsynch

```

```

INFO: ***** AsynchClientServlet - About to call asynchWithException
INFO: ***** AsynchClientServlet - Finished calling asynchWithException
INFO: ***** AsynchClientServlet - About to call canBeCancelled
INFO: ***** AsynchClientServlet - Finished calling canBeCancelled

INFO: ***** AsynchClientServlet - About to cancel canBeCancelled
INFO: ***** AsynchClientServlet - Cancelled canBeCancelled

INFO: ***** AsynchStatefulSessionBean Exiting slowOneWayAsynch

INFO: ***** AsynchStatefulSessionBean Exiting slowAsynch
INFO: ***** AsynchClientServlet - slowAsynch result: Mon Aug 16 18:09:58 CEST 2010
INFO: ***** AsynchStatefulSessionBean asynchWithException throwing exception
INFO: ***** AsynchStatefulSessionBean canBeCancelled waited 1
INFO: ***** AsynchClientServlet - canBeCancelled result: Cancelled Mon Aug 16 18:10:03
CEST 2010
INFO: ***** AsynchClientServlet - asynchWithException result: EjbFutureTask
taskId=15,	cancelCalled=false,complete=true,resultException=java.util.concurrent.ExecutionException: java.lang.Exception: Exception from AsynchStatefulSessionBean
INFO: ***** AsynchClientServlet - An ExecutionException was thrown:
java.util.concurrent.ExecutionException: java.lang.Exception: Exception from
AsynchStatefulSessionBean
INFO: ***** Exiting AsynchClientServlet

```

Note that:

- A number of methods in the session bean implementation class are annotated with the `@Asynchronous` annotation.
This annotation demarcates an asynchronous method in an EJB.
- All the methods annotated with `@Asynchronous` has a return type `void` or `Future<T>`, where `T` is the type of the result produced by the method.
- All the asynchronous methods that has a return type `Future<T>` creates an object of the type `AsyncResult<T>`, where `T` is the type of the result produced by the method.
- The method `asynchWithException` declares and throws an exception of the type `Exception`.
The slightly superfluous `if`-clause in this method is there in order to trick the compiler into accepting a method that always throws an exception.
- The method `canBeCancelled` uses the `SessionContext.wasCancelCalled` method to check if the call to the method has been cancelled by the client.
- The client calling the `canBeCancelled` method calls the `Future.cancel(true)` method in order to attempt to cancel the method call.
If the parameter value had been false, then any call already in progress would have been allowed to continue until completion.
If the session bean had not checked the cancel-flag in the `SessionContext` using the `wasCancelCalled` method, then the method would have continued until completion.
- We can see from the console output that all asynchronous method invocations immediately returns and the client is free to continue with its duties. The methods later completes and prints an exit-message.
This is of course the normal and expected behaviour when invoking asynchronous operations.
- The results from asynchronous methods that produce a result is retrieved by the client by invoking the `Future.get` method.
- Calling `get` on the `Future<String>` object returned by the `asynchWithException` method causes an `ExecutionException` to be thrown on the client side.

- The *ExecutionException* thrown as the result of the asynchronous method in the session bean thrown an exception wraps the exception thrown by the session bean.
- The exception thrown as a result of the asynchronous invocation of the *asynchWithException* method is not thrown when the *Future<String>* object is printed (the method *toString* is called on the object). Instead, the exception is thrown when the *get* method is called on the *Future<String>* object.

This concludes this example. In subsequent sections we will look at what the EJB specification says about asynchronous session beans. Finally, we will summarize the chapter by listing the modifications that are required in order for method(s) of a session bean to become asynchronous.

7.2. Asynchronous Methods in Session Beans

References: EJB 3.1 Specification, section 4.5.

This section discusses the requirements on asynchronous methods in session beans.

7.2.1. Denote Method Asynchronicity

One or more methods in a session bean can be denoted as being asynchronous using one of the following options:

- Annotate individual methods with the *@Asynchronous* annotation.
This causes a single business method in the session bean to become asynchronous.
- Annotate the session bean class with the *@Asynchronous* annotation.
This causes all the business methods in the session bean to become asynchronous.
- Using the *<async-method>* element when declaring a session bean in the deployment descriptor.
This causes one or more business methods in the session bean to become asynchronous.

The *@Asynchronous* annotation has no additional elements.

Below follows an example of how the method *myAsynchMethod* in the stateless session bean *DDConfigedStatelessBean* has been made asynchronous using the ejb-jar.xml deployment descriptor.

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

  <enterprise-beans>

    <session>
      <ejb-name>DDConfigedStatelessBean</ejb-name>
      <local-bean/>
      <ejb-class>com.ivan.scbcd6.DDConfigedStatelessBean</ejb-class>
      <session-type>Stateless</session-type>
      <async-method>
        <!--
          Specifies the name of the asynchronous method(s).
          Wildcards, such as *, may also be used.
        -->
        <method-name>myAsynchMethod</method-name>
      <!--
    
```

```

Optionally the method parameter types can be listed
if we need to separate two methods with the same
name and make one of them asynchronous.

-->
<method-params>
    <method-param>java.lang.String</method-param>
</method-params>
</async-method>
</session>

```

Denoting a method as being asynchronous is not enough, as we will see in the next section.

7.2.2. Requirements on Methods to be Made Asynchronous

Only methods in the no-interface view, local business interface view or remote business interface view can be made asynchronous. Methods in any EJB 2.x views may not be made asynchronous.

The return type of asynchronous methods can be either *void* or *Future<T>*, where *T* is the type of the result produced by a synchronous version of the method. This regardless of whether the method is declared in a session bean implementation class or in a business interface.

The EJB 3.1 API supplies the *AsyncResult<T>* class that implements the *Future<T>* interface. This class can be used when returning results from asynchronous methods, as we saw in the example program [above](#).

Asynchronous methods with the return type *void* may not declare any exceptions. Asynchronous methods with the return type *Future<T>* may declare application, checked, exceptions.

7.2.3. Asynchronous Methods and Transactions

Transaction context does not propagate over asynchronous method invocations. The following table lists the different transaction attributes and the corresponding transaction in an asynchronous method:

Transaction Attribute	Transaction in Asynchronous Method
NOT_SUPPORTED	No transaction context exists in the method.
REQUIRED	A new transaction context will be created.
SUPPORTS	No transaction context exists in the method.
REQUIRES_NEW	A new transaction context will be created.
MANDATORY	An exception will be thrown, indicating that a transaction context is required.
NEVER	No transaction context exists in the method.

A method without a transaction context will throw an exception if an attempt is made to, for instance, invoke the *SessionContext.getRollbackOnly()* method or other methods that assume the presence of a transaction context.

7.2.4. Asynchronous Methods and Security

A caller security principal propagates, in the same way as with synchronous methods, with the invocation of an asynchronous method.

7.3. Client View of Asynchronous Session Beans

This section discusses special behaviour that clients may expect from asynchronous methods of session beans.

7.3.1. Asynchronous Methods and Exceptions

References: EJB 3.1 Specification, sections 3.4.8 and 4.5.5.

When a client of a session bean invokes an asynchronous method, the following exceptions may be thrown at different stages of the invocation:

Stage	Exception	Comments
Preparation of asynchronous method dispatch.	EJBException or RemoteException (if business interface extends Remote)	Probable cause is problems allocating internal resources for the method call.
Execution of asynchronous method.	Application (checked) exceptions.	Methods with void return type are not allowed to declare any checked exceptions. No exceptions propagate to the client.
Execution of asynchronous method.	System (unchecked) exceptions.	Unchecked exceptions cause the bean to be taken out of service – see Session Bean Life-Cycles . If the method has a non-void return type, exceptions are propagated to the client and an ExecutionException is thrown when the client attempts to retrieve the result from the Future<T> object.

7.3.2. Asynchronous Method Return Values

References: EJB 3.1 Specification, section 3.4.8.

Clients of asynchronous session bean methods receives a *Future<T>*, where T is the type of the result produced by the method. The *Future<V>* object which can be used to:

- Determine whether the operation has completed.
- Retrieve the result value of the operation.
- Determine whether the operation completed abnormally and any associated exception.
- Attempt to cancel an ongoing invocation.

The container may throw a system exception of the type *EJBException* as a result of calling the asynchronous method if there are any problems when preparing to dispatch the call.

The table below lists the methods of a *Future<V>* object and describes their semantics in when used in connection with asynchronous invocation of session beans:

Method Signature	Description
boolean cancel(boolean mayInterruptIfRunning)	Container will attempt to cancel the method invocation only if it has not already been dispatched. Returns true if invocation successfully cancelled, false otherwise. Setting the mayInterruptIfRunning flag to true

	will, if the call has already been dispatched, allow the session bean, through the <i>SessionContext.wasCancelCalled</i> method, to know about the cancellation attempt.
V get()	<p>Waits for the method invocation to complete and retrieves the resulting value or exception of the asynchronous method invocation.</p> <p>Throws:</p> <ul style="list-style-type: none"> CancellationException – If invocation was cancelled. ExecutionException – If asynchronous method threw an exception. InterruptedException – If client thread was interrupted.
V get(long timeout, TimeUnit unit)	<p>Waits at most for the supplied amount of time for the method invocation to complete and retrieves the resulting value or exception of the asynchronous method invocation.</p> <p>Exceptions as the above get() method and additionally:</p> <ul style="list-style-type: none"> TimeoutException – If waiting timed out.
boolean isCancelled()	Returns true if method invocation was cancelled and did not complete normally.
boolean isDone()	Returns true if method invocation has completed. Completion may be due to normal termination, an exception or cancellation.

8. Create a Message Driven Bean

References: EJB 3.1 Specification, chapter 5.

Message driven beans is an EJB that asynchronously consume messages sent to a destination. Such a destination can be either a point-to-point destination or a publish/subscribe destination.

Different kinds of destinations are theoretically possible, but in this book we will only use JMS destinations.

- Point-to-point destination.

A queue destination. Retains messages until they are consumed by a (single) message consumer. Note that several consumers may be registered on the same queue, but in such a case the consumer that will receive the message is randomly chosen.

- Publish/subscribe destination.

A topic destination. Immediately forwards messages sent by the message producer to all the consumers registered for the topic. In default configuration, the topic does not retain messages – thus messages sent to a topic that has no consumers will be lost.

JMS allows for durable subscription to topic destinations, which means that messages will be retained until the consumer will become active.

Before going deeper into detailed information on message driven beans, we'll first look at two example programs. The first one shows message driven beans listening to a topic and the second one shows message driven beans listening to a queue.

A word of caution:

When developing message driven beans, be clear about what beans are listening to which queues or topics.

Example: Application A that produces messages to a topic will cause message driven beans in application B listening to the same topic to receive messages.

This may have unwanted side effects.

8.1. Common GlassFish Example Preparations

Before we can start develop the message driven bean example programs, we need to make some preparations in the GlassFish application server to which the examples will be deployed by setting up the resources necessary for JMS communication.

Below follows a more terse description of the setup. Please consult [appendix D](#) for details on the different steps of the preparation.

8.1.1. Setting Up Physical Destinations

Set up two physical destinations as described in the [first section in appendix D](#); one for a topic and one for a queue. I name the physical destinations “MyTopicPhysicalDestination” and “MyQueuePhysicalDestination” with the types *javax.jms.Topic* and *javax.jms.Queue* respectively. All additional parameters uses default values.

When finished, there should be two new physical destinations in the list in the GlassFish administration console:

JMS Physical Destinations

Java Message Service (JMS) physical destination objects are maintained by Message Queue brokers. The queue named mq.sys.dmq is the system destination, to which expired and undeliverable messages are redirected. Click New to create a new physical destination.

Destinations (4)			
		New...	Delete
	Name	Type	Statistics
<input type="checkbox"/>	MyTopicPhysicalDestination	topic	View
<input type="checkbox"/>	MyQueuePhysicalDestination	queue	View
<input type="checkbox"/>	DSQueueDest	queue	View

Having created two physical destinations in GlassFish for the message driven bean example programs.

8.1.2. Setting Up Connection Factories

Create two connection factories as described in the [second section in appendix D](#); again one for a topic and one for a queue. I name the connection factories “jms/TopicConnectionFactory” and “jms/QueueConnectionFactory” respectively. Additionally, configure the Resource Type to be *javax.jms.TopicConnectionFactory* for the topic connection factory and *javax.jms.QueueConnectionFactory* for the queue connection factory. Set Connection Validation to be required for both of the connection factories and leave the remaining parameters at their default values.

When finished, there should be two new connection factories in the list in the GlassFish administration console:

JMS Connection Factories

Java Message Service (JMS) connection factories are objects that allow an application to create other JMS objects programmatically. Click New to create a new connection factory. Click the name of a connection factory to modify its properties.

Connection Factories (2)				
	New...	Delete	Enable	Disable
	Pool Name	JNDI Name	Enabled	Resource Type
<input type="checkbox"/>	jms/QueueConnectionFactory	jms/QueueConnectionFactory	true	javax.jms.QueueConnectionFactory
<input type="checkbox"/>	jms/TopicConnectionFactory	jms/TopicConnectionFactory	true	javax.jms.TopicConnectionFactory

Having created two JMS connection factories in GlassFish for the message driven bean example programs.

8.1.3. Setting Up Destination Resources

The final step of the preparations is to create two JMS Destination Resources, again one for a topic and one for a queue. The procedure for these resources is described in the [final section of appendix D](#). The JNDI names of the destination resources are “jms/TopicDestination” for the topic and “jms/QueueDestination” for the queue. The topic destination resource uses the topic physical destination “MyTopicPhysicalDestination” created earlier and the queue destination resource uses the queue physical destination “MyQueuePhysicalDestination”. Resource Type is *javax.jms.Topic* and *javax.jms.Queue* respectively.

Finally, when finished, there should be two new JMS connection factories in the list in the GlassFish administration console:

Destination Resources (2)					
	JNDI Name	Enabled	Resource Type	Description	
<input type="checkbox"/>	jms/TopicDestination	true	javax.jms.Topic		
<input type="checkbox"/>	jms/QueueDestination	true	javax.jms.Queue		

Having created two JMS destination resources in GlassFish for the message driven bean example programs.

8.2. Topic Message Driven Bean Example

The first message driven bean example will consist of three instances of a message driven bean that are all listening to the same topic. In addition, there will also be a servlet acting as a message producer and a class which represents the messages being sent from the producer to the topic.

8.2.1. Creating the Project and the Message Bean

First we'll create the project and the class implementing the message bean.

- Create a dynamic web project in Eclipse, as described in [appendix A](#).
I call my project “TopicMessageDrivenExample”.
- In the package *com.ivan.scbcd6.entity*, create the message bean class. A message holds a message string, a message number and a timestamp.

```
package com.ivan.scbcd6.entity;

import java.io.Serializable;
import java.util.Date;

/**
 * Represents a message sent by a producer to the topic.
 */
public class MyMessage implements Serializable
{
    /* Constant(s): */
    private static final long serialVersionUID = -4682924711829199796L;

    /* Instance variable(s): */
    private String mMessageString = "";
    private long mMessageNumber;
    private Date mMessageTime;

    public String getMessageString()
    {
        return mMessageString;
    }

    public void setMessageString(final String inMessageString)
    {
        mMessageString = inMessageString;
    }

    public long getMessageNumber()
    {
        return mMessageNumber;
    }

    public void setMessageNumber(final long inMessageNumber)
    {
        mMessageNumber = inMessageNumber;
    }

    public Date getMessageTime()
    {
        return mMessageTime;
    }

    public void setMessageTime(final Date inMessageTime)
    {
        mMessageTime = inMessageTime;
    }
}
```

Note that:

- The message bean class implements *Serializable* and have a *serialVersionUID*.
Objects transferred over JMS must be serializable.

8.2.2.Creating the Message Driven Bean

Each instance of the message driven bean keeps track of a number identifying the bean. When receiving a message from the topic it listens to, it prints a log statement telling us which bean received the message and the contents of the message bean in the message.

- In the package `com.ivan.scbcd6.ejb`, create a class named `TopicListenerEJB` which is implemented like this:

```
package com.ivan.scbcd6.ejb;

import java.util.concurrent.atomic.AtomicInteger;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

import com.ivan.scbcd6.entity.MyMessage;

/**
 * Message driven bean listening to a topic.
 */
@MessageDriven(activationConfig =
{
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Topic")
}, mappedName = "jms/TopicDestination", name="TopicListener1")
public class TopicListenerEJB implements MessageListener
{
    /* Constant(s): */

    /* Class variable(s): */
    private static AtomicInteger mCurrentBeanNumber = new AtomicInteger(0);

    /* Instance variable(s): */
    private int mBeanNumber = mCurrentBeanNumber.incrementAndGet();

    /**
     * Default constructor.
     */
    public TopicListenerEJB()
    {
        System.out.println("**** TopicListenerEJB created: " + mBeanNumber);
    }

    /**
     * @see MessageListener#onMessage(Message)
     */
    @Override
    public void onMessage(Message inMessage)
    {
        System.out.println("**** Bean " + mBeanNumber + " received message: "
+ inMessage);

        extractMessagePayload(inMessage);
    }

    private void extractMessagePayload(Message inMessage)
    {
        /* Extract the message payload, if any. */
        if (inMessage instanceof ObjectMessage)
        {
            try
            {
                ObjectMessage theObjMsg = (ObjectMessage)inMessage;
                MyMessage theMsgPayload = (MyMessage)theObjMsg.getObject();

                System.out.println("Received message with number: " +
theMsgPayload.getMessageNumber());
                System.out.println("    Message string: " +
theMsgPayload.getMessageString());
            }
        }
    }
}
```

```
        System.out.println("  Message time: " +
                           theMsgPayload.getMessageTime());
    } catch (JMSEException theException)
    {
        System.out.println(
            "An error occurred retrieving message payload: " +
            theException);
    }
}
}
```

Note that:

- The class is annotated by the `@MessageDriven` annotation.
This annotation denotes the class as being an implementation of a message driven EJB.
- The `@MessageDriven` annotation contains an *activationConfig* element.
This element is used to specify configuration of the message driven bean in its operational environment such as acknowledgement mode, message selectors specifying which messages the message driven bean is to receive and the type of destination (topic or queue) the message driven bean is to be associated with.
- The `@MessageDriven` annotation contains a *mappedName* element.
The value of this element is the global JNDI name of a queue or topic that the message driven bean is to receive messages from. The use of this element is product-specific.
- The `@MessageDriven` annotation contains a *name* element.
As seen with other types of EJBs, this element is used to specify the EJB-name of the EJB. The name of the message driven bean is “TopicListener1”.
- The `TopicListenerEJB` class implements the *MessageListener* interface.
This interface tells the container that the message driven bean is a JMS message driven bean that is listening for JMS messages. Theoretically there can be message driven beans listening for other kinds of messages, but in practice only JMS is used.
- The `TopicListenerEJB` class contains a default, no-arguments constructor.
This constructor is used to log the creation of instances of the message driven bean for the sake of this example and is not required.
- The `TopicListenerEJB` class contains a method named *onMessage*.
This method is defined by the *MessageListener* interface and is invoked every time there is a message for the message driven bean.
- Before extracting the payload of the message, the JMS message is first examined to determine if it is of the type *ObjectMessage*.
There are a number of different types of payload that can be enclosed with JMS messages, please refer to the API documentation of `javax.jms.Message` for more information.
- Nowhere is the *acknowledge* method called on the JMS message instance received.
The EJB 3.1 specification explicitly states that message driven beans should not use the JMS API for message acknowledgement. This is a duty that is to be handled by the EJB container.

8.2.3. Configuring the ejb-jar.xml Deployment Descriptor

In order to have multiple message driven beans of the same kind listening to the same topic, we use the ejb-jar.xml deployment descriptor to create three EJBs from the same class.

- Create a file named ejb-jar.xml in the WebContent/WEB-INF directory with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:ejb="http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
    version="3.1">

    <enterprise-beans>
        <message-driven>
            <ejb-name>TopicListener1</ejb-name>
            <ejb-class>com.ivan.scbcd6.ejbs.TopicListenerEJB</ejb-class>
        </message-driven>

        <message-driven>
            <ejb-name>TopicListener2</ejb-name>
            <ejb-class>com.ivan.scbcd6.ejbs.TopicListenerEJB</ejb-class>
        </message-driven>

        <message-driven>
            <ejb-name>TopicListener3</ejb-name>
            <ejb-class>com.ivan.scbcd6.ejbs.TopicListenerEJB</ejb-class>
        </message-driven>
    </enterprise-beans>
</ejb-jar>
```

Note that:

- The ejb-jar.xml deployment descriptor contains definitions of three message driven beans. The name of the EJBs are “TopicListener1”, “TopicListener2” and “TopicListener3”.
- All the EJB definitions in the ejb-jar.xml deployment descriptor uses the same implementation class.
- The first message driven bean definition in the ejb-jar.xml deployment descriptor has the same name as the name specified in the @MessageDriven annotation in the EJB implementation class.
This EJB definition is not strictly necessary, since the EJB is already defined using annotations. The definition in the deployment descriptor can be used to override the configuration in the annotations or to add additional configuration parameters.

8.2.4. Creating a Message Producing Servlet

For the sake of completeness of the example program, a servlet that sends messages to the topic will also be created. Further details on JavaEE message producing clients will be given in the section on [Java EE clients sending messages to a JMS queue in chapter 9](#).

The servlet will, when being accessed, send a numbered message to a JMS topic.

- In the package *com.ivan.scbcd6.producer* create a class named *MessageProducerServlet*.

```
package com.ivan.scbcd6.producer;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.concurrent.atomic.AtomicLong;
import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.jms.Topic;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.entity.MyMessage;

/**
 * Servlet producing JMS messages when accessed.
 */
@WebServlet(name="MessageProducerServlet", urlPatterns="/sendmsg.do")
public class MessageProducerServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1647640647915937983L;

    /* Instance variable(s): */
    /** Connection factory for topic. */
    @Resource(mappedName = "jms/TopicConnectionFactory")
    private ConnectionFactory mTopicConnectionFactory;
    /** Topic destination. */
    @Resource(mappedName = "jms/TopicDestination")
    private Topic mTopicDestination;
    /** MyMessage number counter. */
    private AtomicLong mMessageNumber = new AtomicLong(0);

    /**
     * @see HttpServlet#HttpServlet()
     */
    public MessageProducerServlet()
    {
        super();
        System.out.println("**** MessageProducerServlet created");
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        sendJmsMessage();

        PrintWriter theResponseWriter = inResponse.getWriter();

        theResponseWriter.println("A message was sent at " + new Date());
    }
}
```

```

private void sendJmsMessage()
{
    MessageProducer theJMSMessageProducer = null;
    Connection theJMSConnection = null;

    try
    {
        /* Retrieve a JMS connection from the topic connection factory. */
        theJMSConnection = mTopicConnectionFactory.createConnection();

        /*
         * Create the JMS session; not transacted and with auto-
         * acknowledge.
         */
        Session theJMSSession =
            theJMSConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        /* Create a JMS message producer for the topic destination. */
        theJMSMessageProducer =
            theJMSSession.createProducer(mTopicDestination);

        /* Create the object to be sent in the message created above. */
        MyMessage theObjectToSend = new MyMessage();
        theObjectToSend.setMessageNumber(mMessageNumber.incrementAndGet());
        theObjectToSend.setMessageString("Hello Message Driven Beans");
        theObjectToSend.setMessageTime(new Date());

        /* Create message used to send a Java object. */
        ObjectMessage theJmsObjectMessage =
            theJMSSession.createObjectMessage();
        theJmsObjectMessage.setObject(theObjectToSend);

        /* Send the message. */
        theJMSMessageProducer.send(theJmsObjectMessage);
    } catch (JMSException theException)
    {
        theException.printStackTrace();
    } finally
    {
        if (theJMSMessageProducer != null)
        {
            try
            {
                theJMSMessageProducer.close();
            } catch (JMSException theException)
            {
                // Ignore exceptions.
            }
        }
        if (theJMSConnection != null)
        {
            try
            {
                theJMSConnection.close();
            } catch (JMSException theException)
            {
                // Ignore exceptions.
            }
        }
    }
}

```

8.2.5. Running the Example

With the servlet in place, we are ready to deploy and run the topic example program.

- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/TopicMessageDrivenExample/sendmsg.do>
The URL may differ if you have chosen a different project name.
- In the browser, there will be a message saying that a message was sent and the time at which it was sent:

```
A message was sent at Tue Oct 19 06:47:18 CEST 2010
```

- Looking in the console, you can see the log output from the example program:

```
INFO: *** MessageProducerServlet created  
  
INFO: *** TopicListenerEJB created: 2  
INFO: *** TopicListenerEJB created: 3  
INFO: *** TopicListenerEJB created: 1  
  
INFO: *** Bean 1 received message: com.sun.messaging.jms.ra.DirectObjectPacket@8f4fe3d  
INFO: *** Bean 2 received message: com.sun.messaging.jms.ra.DirectObjectPacket@c5f5b63  
INFO: Received message with number: 1  
INFO:     Message string: Hello Message Driven Beans  
INFO:     Message time: Tue Oct 19 06:47:18 CEST 2010  
INFO: *** Bean 3 received message: com.sun.messaging.jms.ra.DirectObjectPacket@434aa410  
INFO: Received message with number: 1  
INFO:     Message string: Hello Message Driven Beans  
INFO:     Message time: Tue Oct 19 06:47:18 CEST 2010  
INFO: Received message with number: 1  
INFO:     Message string: Hello Message Driven Beans  
INFO:     Message time: Tue Oct 19 06:47:18 CEST 2010
```

Note that:

- An instance of the message-producing servlet is created.
- Three instances of the message driven bean listening to the topic are created.
- Each of the three message driven beans receive a copy of the message sent to the topic.

This concludes the topic message driven example.

8.3. Queue Message Driven Example

In this second message driven example there will also be three instances of one and the same message driven bean, however they will be listening to the same queue. The servlet producing messages from the previous example will be slightly modified to send messages to a queue instead of to a topic.

8.3.1. Creating the Project and the Message Bean

First create the project and the class implementing the message bean.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “QueueMessageDrivenExample”.
- In the package `com.ivan.scbcd6.entity`, create the message bean class `MyMessage`. The class is identical to that used in the [previous example](#), so no code listing is provided here.

8.3.2. Creating the Message Driven Bean

The only difference of the message driven bean implementation below compared to the one in the topic example program is that this one listens to a queue instead of a topic.

- In the package `com.ivan.scbcd6.ejb`, create a class named `QueueListenerEJB` which is implemented as follows:

```
package com.ivan.scbcd6.ejb;

import java.util.concurrent.atomic.AtomicInteger;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

import com.ivan.scbcd6.entity.MyMessage;

/**
 * Message driven bean listening to a queue.
 */
@MessageDriven(activationConfig =
{
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue")
}, mappedName = "jms/QueueDestination", name="QueueListener1")
public class QueueListenerEJB implements MessageListener
{
    /* Constant(s): */

    /* Class variable(s): */
    private static AtomicInteger mCurrentBeanNumber = new AtomicInteger(0);

    /* Instance variable(s): */
    private int mBeanNumber = mCurrentBeanNumber.incrementAndGet();

    /**
     * Default constructor.
     */
    public QueueListenerEJB()
    {
        System.out.println("**** QueueListenerEJB created: " + mBeanNumber);
    }

    /**
     * @see MessageListener#onMessage(Message)
     */
    @Override
```

```

public void onMessage(Message inMessage)
{
    System.out.println("**** Bean " + mBeanNumber + " received message: "
        + inMessage);

    extractMessagePayload(inMessage);
}

private void extractMessagePayload(Message inMessage)
{
    /* Extract the message payload, if any. */
    if (inMessage instanceof ObjectMessage)
    {
        try
        {
            ObjectMessage theObjMsg = (ObjectMessage)inMessage;
            MyMessage theMsgPayload = (MyMessage)theObjMsg.getObject();

            System.out.println("Received message with number: " +
                theMsgPayload.getMessageNumber());
            System.out.println("    Message string: " +
                theMsgPayload.getMessageString());
            System.out.println("    Message time: " +
                theMsgPayload.getMessageTime());
        } catch (JMSEException theException)
        {
            System.out.println(
                "An error occurred retrieving message payload: " +
                theException);
        }
    }
}
}

```

The following notes are almost identical to those of the message driven bean in the topic example [above](#).

Note that:

- The class is annotated by the `@MessageDriven` annotation.
This annotation denotes the class as being an implementation of a message driven EJB.
- The `@MessageDriven` annotation contains an *activationConfig* element.
This element is used to specify configuration of the message driven bean in its operational environment such as acknowledgement mode, message selectors specifying which messages the message driven bean is to receive and the type of destination (topic or queue) the message driven bean is to be associated with.
- The `@MessageDriven` annotation contains a *mappedName* element.
The value of this element is the global JNDI name of a queue or topic that the message driven bean is to receive messages from. The use of this element is product-specific.
- The `@MessageDriven` annotation contains a *name* element.
As seen with other types of EJBs, this element is used to specify the EJB-name of the EJB.
The name of the message driven bean is “QueueListener1”.
- The `QueueListenerEJB` class implements the `MessageListener` interface.
This interface tells the container that the message driven bean is a JMS message driven bean that is listening for JMS messages. Theoretically there can be message driven beans listening for other kinds of messages, but in practice only JMS is used.
- The `QueueListenerEJB` class contains a default, no-arguments constructor.
This constructor is used to log the creation of instances of the message driven bean for the sake of this example and is not required.

- The *QueueListenerEJB* class contains a method named *onMessage*. This method is defined by the *MessageListener* interface and is invoked every time there is a message for the message driven bean.
- Before extracting the payload of the message, the JMS message is first examined to determine if it is of the type *ObjectMessage*. There are a number of different types of payload that can be enclosed with JMS messages, please refer to the API documentation of *javax.jms.Message* for more information.
- Nowhere is the *acknowledge* method called on the JMS message instance received. The EJB 3.1 specification explicitly states that message driven beans should not use the JMS API for message acknowledgement. This is a duty that is to be handled by the EJB container.

8.3.3. Configuring the ejb-jar.xml Deployment Descriptor

The ejb-jar.xml deployment descriptor is also very similar to that of the topic example and has the same purpose; to have multiple message driven beans of the same kind listening to the same queue, we use the ejb-jar.xml deployment descriptor to create three EJBs from the same class.

- Create a file named ejb-jar.xml in the WebContent/WEB-INF directory with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:ejb="http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
    version="3.1">

    <enterprise-beans>
        <message-driven>
            <ejb-name>QueueListener1</ejb-name>
            <ejb-class>com.ivan.scbcd6.ejbs.QueueListenerEJB</ejb-class>
        </message-driven>

        <message-driven>
            <ejb-name>QueueListener2</ejb-name>
            <ejb-class>com.ivan.scbcd6.ejbs.QueueListenerEJB</ejb-class>
        </message-driven>

        <message-driven>
            <ejb-name>QueueListener3</ejb-name>
            <ejb-class>com.ivan.scbcd6.ejbs.QueueListenerEJB</ejb-class>
        </message-driven>
    </enterprise-beans>
</ejb-jar>
```

Note that:

- The ejb-jar.xml deployment descriptor contains definitions of three message driven beans. The name of the EJBs are “QueueListener1”, “QueueListener2” and “QueueListener3”.
- All the EJB definitions in the ejb-jar.xml deployment descriptor uses the same implementation class.
- The first message driven bean definition in the ejb-jar.xml deployment descriptor has the same name as the name specified in the @MessageDriven annotation in the EJB implementation class.
This EJB definition is not strictly necessary, since the EJB is already defined using annotations. The definition in the deployment descriptor can be used to override the configuration in the annotations or to add additional configuration parameters.

8.3.4. Creating a Message Producing Servlet

For the sake of completeness of the example program, a servlet that sends messages to the queue will also be created. Further details on JavaEE message producing clients will be given in the section on [Java EE clients sending messages to a JMS queue in chapter 9](#).

The servlet will, when being accessed, send a numbered message to a JMS queue.

- In the package *com.ivan.scbcd6.producer* create a class named *MessageProducerServlet*.

```
package com.ivan.scbcd6.producer;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.concurrent.atomic.AtomicLong;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Queue;
import javax.jms.Session;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.entity.MyMessage;

/**
 * Servlet producing JMS messages when accessed.
 */
@WebServlet(name="MessageProducerServlet", urlPatterns="/sendmsg.do")
public class MessageProducerServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = -4364474814559146703L;

    /* Instance variable(s): */
    /** Connection factory for queue. */
    @Resource(mappedName = "jms/QueueConnectionFactory")
    private ConnectionFactory mQueueConnectionFactory;
    /** Queue destination. */
    @Resource(mappedName = "jms/QueueDestination")
    private Queue mQueueDestination;
    /** MyMessage number counter. */
    private AtomicLong mMessageNumber = new AtomicLong(0);

    /**
     * @see HttpServlet#HttpServlet()
    
```

```

/*
public MessageProducerServlet()
{
    super();
    System.out.println("**** MessageProducerServlet created");
}

/**
 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
 */
@Override
protected void doGet(HttpServletRequest inRequest,
                      HttpServletResponse inResponse) throws ServletException, IOException
{
    PrintWriter theResponseWriter = inResponse.getWriter();

    try
    {
        sendJmsMessage();
        theResponseWriter.println("A message was sent at " + new Date());
    } catch (JMSException theException)
    {
        theResponseWriter.println("An error occurred sending message: " +
                                  theException);
    }
}

private void sendJmsMessage() throws JMSException
{
    MessageProducer theJMSMessageProducer = null;
    Connection theJMSConnection = null;

    try
    {
        /* Retrieve a JMS connection from the queue connection factory. */
        theJMSConnection = mQueueConnectionFactory.createConnection();

        /*
         * Create the JMS session; not transacted and with auto-
         * acknowledge.
         */
        Session theJMSSession =
            theJMSConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        /* Create a JMS message producer for the queue destination. */
        theJMSMessageProducer =
            theJMSSession.createProducer(mQueueDestination);

        /* Create the object to be sent in the message created above. */
        MyMessage theObjectToSend = new MyMessage();
        theObjectToSend.setMessageNumber(mMessageNumber.incrementAndGet());
        theObjectToSend.setMessageString("Hello Message Driven Beans");
        theObjectToSend.setMessageTime(new Date());

        /* Create message used to send a Java object. */
        ObjectMessage theJmsObjectMessage =
            theJMSSession.createObjectMessage();
        theJmsObjectMessage.setObject(theObjectToSend);

        /* Send the message. */
        theJMSMessageProducer.send(theJmsObjectMessage);
    } finally
    {
        closeJmsResources(theJMSMessageProducer, theJMSConnection);
    }
}

/**
 * Closes the supplied JMS resources if they are not null.
 * If a supplied resource is null, then do nothing.
 *
 * @param inJMSMessageProducer JMS message producer to close.
 * @param inJMSConnection JMS connection to close.
 */
private void closeJmsResources(MessageProducer inJMSMessageProducer,
                               Connection inJMSConnection)
{
}

```

```
if (inJMSSMessageProducer != null)
{
    try
    {
        inJMSSMessageProducer.close();
    } catch (JMSEException theException)
    {
        // Ignore exceptions.
    }
}
if (inJMSSConnection != null)
{
    try
    {
        inJMSSConnection.close();
    } catch (JMSEException theException)
    {
        // Ignore exceptions.
    }
}
```

8.3.5. Running the Example

With the servlet in place, we are ready to deploy and run the queue example program.

- Deploy the project to the GlassFish application server.
- In a browser, issue a request to the following URL:
<http://localhost:8080/QueueMessageDrivenExample/sendmsg.do>
The URL may differ if you have chosen a different project name.
- In the browser, there will be a message saying that a message was sent and the time at which it was sent:

```
A message was sent at Tue Oct 19 17:31:52 CEST 2010
```

- Re-issue the request to the above URL two more times.
This is to send two more messages to the queue.
- Looking in the console, you can see the log output from the example program.
The log may differ in that a different message driven bean instance may handle an individual message.

```
INFO: *** MessageProducerServlet created
INFO: *** QueueListenerEJB created: 1
INFO: *** Bean 1 received message: com.sun.messaging.jms.ra.DirectObjectPacket@5f4c1615
INFO: Received message with number: 1
INFO:     Message string: Hello Message Driven Beans
INFO:     Message time: Tue Oct 19 17:31:52 CEST 2010

INFO: *** QueueListenerEJB created: 2
INFO: *** Bean 2 received message: com.sun.messaging.jms.ra.DirectObjectPacket@7b044837
INFO: Received message with number: 2
INFO:     Message string: Hello Message Driven Beans
INFO:     Message time: Tue Oct 19 17:31:54 CEST 2010

INFO: *** QueueListenerEJB created: 3
INFO: *** Bean 3 received message: com.sun.messaging.jms.ra.DirectObjectPacket@e0a1ba9
INFO: Received message with number: 3
INFO:     Message string: Hello Message Driven Beans
INFO:     Message time: Tue Oct 19 17:31:55 CEST 2010
```

Note that:

- An instance of the message-producing servlet is created.
- Instances of the message driven beans listening to the queue are created as needed, that is, immediately prior to being dispatched a message.
- Each message sent to the queue are received and handled by one single message driven bean instance only.
In the example above, the first message is handled by bean number one, the second message is handled by bean number two and the third message is handled by bean number three.

This concludes the queue message driven example.

8.4. Characteristics of Message Driven Beans

The following are some characteristics of message driven beans:

- Message driven beans are invisible to message producing clients.
In the case of using JMS, a client only knows about a connection factory and a destination queue or topic. It has no knowledge of the receiver of the messages it produces.
- Message driven beans has no conversational state related to a specific client.
State information may be stored in instance variables, but it is not related to a specific client.
- Asynchronous processing of messages.
Clients send messages to a queue or topic and continues without waiting for responses.
Messages are processed sometime in the future by one or more instances of message driven beans. The normal message-processing mechanisms does not generate any response.
- Multiple instances of the same message driven bean can process messages in parallel.

Some examples of drawbacks with using message driven beans as message consumers:

- Requires an EJB container.
- Cannot “act” unless an incoming message is received.
- Can only listen for messages to a single destination.
- No way of putting a received message back on the queue unless causing the transaction in which the message driven bean executes to roll back.
- Cannot send exceptions back to clients.
- Cannot hold client-related state.
A message driven bean may receive a message from any client sent to the destination the bean listens to.
- There is no standardized means of enclosing security information in messages to message driven beans.
- Message ordering is not guaranteed.

8.5. Dependency Injection in Message Driven Beans

References: EJB 3.1 Specification, sections 5.4.3 and 5.4.4.

The rules described in [chapter 15](#) also applies to dependency injection and lookup of references to miscellaneous resources.

8.6. Message Driven Beans and the Container

The following are some properties of message driven beans and their relation to the container in which they run in:

- A client message can be delivered to any instance of a message driven bean.
- A class implementing a message driven bean must be annotated with the `@MessageDriven` annotation or defined in the `ejb-jar.xml` deployment descriptor.
- A message driven bean must use one of the following means to specify what type of messaging it uses. In the case of JMS, this is the `javax.jms.MessageListener` interface.
 - Implement appropriate message listener interface.
 - Use the `messageListenerElement` of the `@MessageDriven` annotation.
 - Use the `<messaging-type>` element in the `ejb-jar.xml` deployment descriptor.
- If a class implementing a message driven bean implements more than one interface other than `Serializable`, `Externalizable` and the interfaces in the `javax.ejb` package, then the message listener interface must be specified using either the `@MessageDriven` annotation or the `ejb-jar.xml` deployment descriptor.
- The class in which a message driven bean is implemented must:
 - Be public, but must not be final or abstract.
 - Have no constructor or a no-args constructor.
 - Must not define the `finalize` method.
- Message listener methods of message driven beans must be public, but must not be final or static.
- Message driven beans may use the dependency injection and lookup mechanisms described in [chapter 15](#) to obtain references to miscellaneous resources.
- Only the `PostConstruct` and `PreDestroy` life-cycle callback methods are supported for message driven beans. For details, please refer to the chapter on [EJB interceptors](#).
- Business method, timeout method and life-cycle event interceptors can be applied to message driven beans. Please refer to the [chapter on EJB interceptors](#) for details.
- Invocations of a message driven bean is serialized by the container.
In addition, multiple instances of a message driven bean class may exist handling messages in parallel. Message driven beans do not need to consider reentrant invocations.
- The order in which messages are delivered to message driven beans is not guaranteed. Thus message driven beans must consider that messages may be delivered out of order.
- Message driven beans using JMS must not use the JMS API for message acknowledgement.

8.7. Messaging Configuration

References: EJB 3.1 Specification, sections 5.4.14 to 5.4.17.

The messaging configuration of a message driven bean determine how the message driven bean will interact with the messaging provider from which it receives messages. In the EJB specification, this configuration is referred to as Activation Configuration Properties.

Configuration is done using either annotations or the ejb-jar.xml deployment descriptor.

8.7.1. Messaging Configuration using Annotations

Messaging configuration with annotations is done in the @MessageDriven annotation, as we have seen in the message driven examples:

```
...
@MessageDriven(activationConfig =
{
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue")
}, mappedName = "jms/QueueDestination", name="QueueListener1")
public class QueueListenerEJB implements MessageListener
{
...
}
```

In the *activationConfiguration* element of the @MessageDriven annotation one or more @ActivationConfigProperty annotations can be listed, each naming a property and supplying a value for the property.

Examples of properties that can be set this way are:

Property Name	Description
destinationType	Message destination from which the message driven bean receives messages. Value must be either javax.jms.Queue or javax.jms.Topic for a JMS message driven bean.
acknowledgeMode	Message acknowledgement mode for JMS message driven beans. Possible values: Auto-acknowledge or Dups-ok-acknowledge. Default: Auto-acknowledge.
messageSelector	JMS message selector acting as a filter determining which messages the message driven bean will receive.
subscriptionDurability	Indicates whether the topic subscription is durable or not. A durable subscription allow a message driven bean to receive the messages published while the bean was disconnected. Possible values: Durable or NonDurable. Default: NonDurable.

8.7.2. Messaging Configuration in the Deployment Descriptor

The following is an example showing how messaging configuration is accomplished in the ejb-jar.xml deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:ejb="http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1">

  <enterprise-beans>
    <message-driven>
      <ejb-name>QueueListener3</ejb-name>
      <ejb-class>com.ivan.scbcd6.ejbs.QueueListenerEJB</ejb-class>

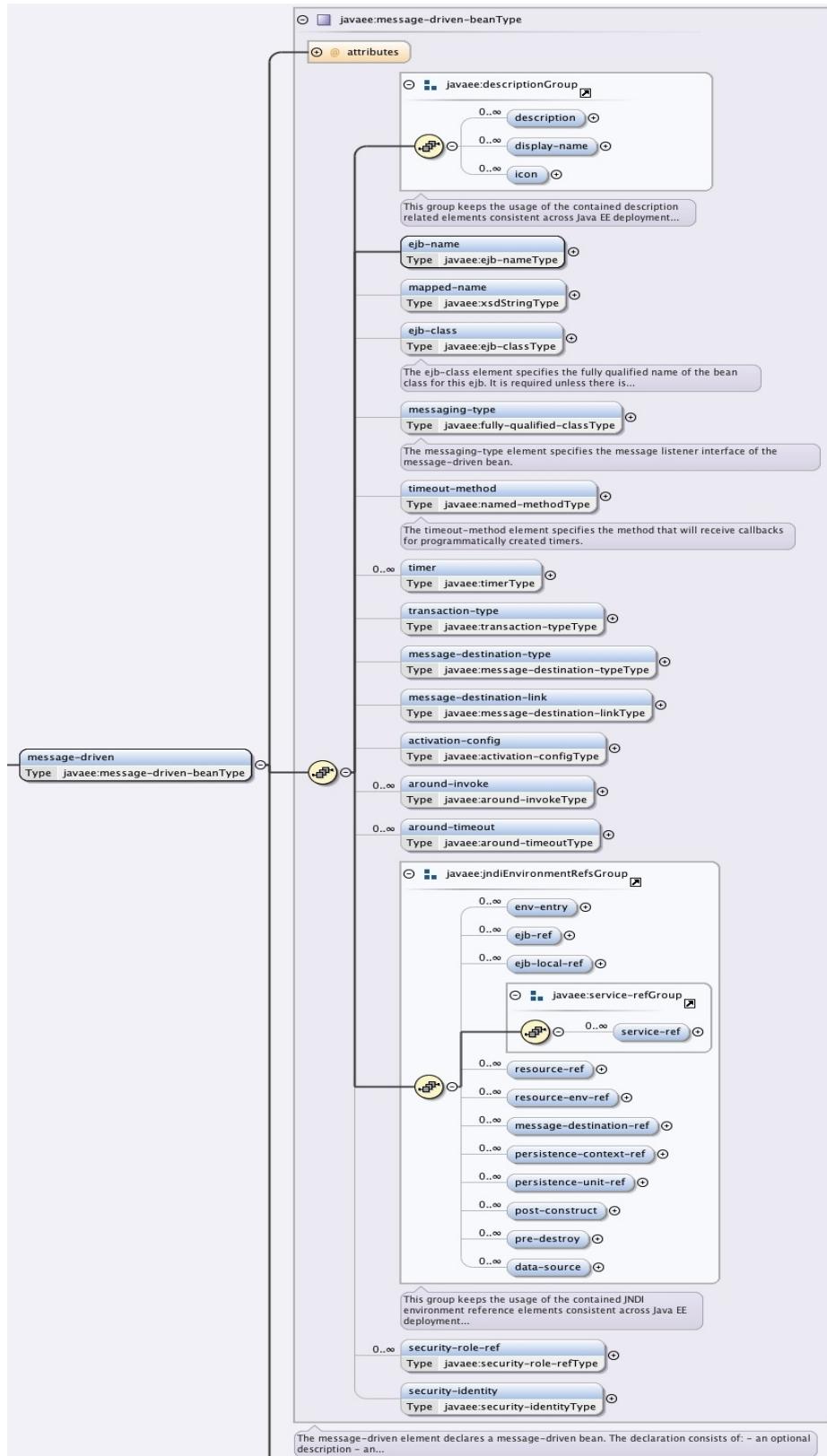
      <activation-config>
        <!--
          Set the destinationType property to javax.jms.Queue.
        -->
        <activation-config-property>
          <activation-config-property-name>
            destinationType
          </activation-config-property-name>
          <activation-config-property-value>
            javax.jms.Queue
          </activation-config-property-value>
        </activation-config-property>
      </activation-config>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Note that:

- The <activation-config> element in which the messaging configuration is specified appears in the <message-driven> element that defines the message driven to which the messaging configuration applies.

8.8. Message Driven Beans in the Deployment Descriptor

The following XML schema fragment describes configuration options available when defining a message driven bean in the ejb-jar.xml deployment descriptor:



8.9. Message Driven Beans and Exceptions

References: EJB 3.1 Specification, sections 5.4.18, 14.3.4 and 14.3.5.

This section contains information specific to message driven beans. Classification of exceptions as well as other exception-related semantics that applies to all kinds of EJBs can be found in the chapter on [EJBs and Exceptions](#).

The EJB 3.1 specification makes the following recommendations concerning exceptions and message driven beans:

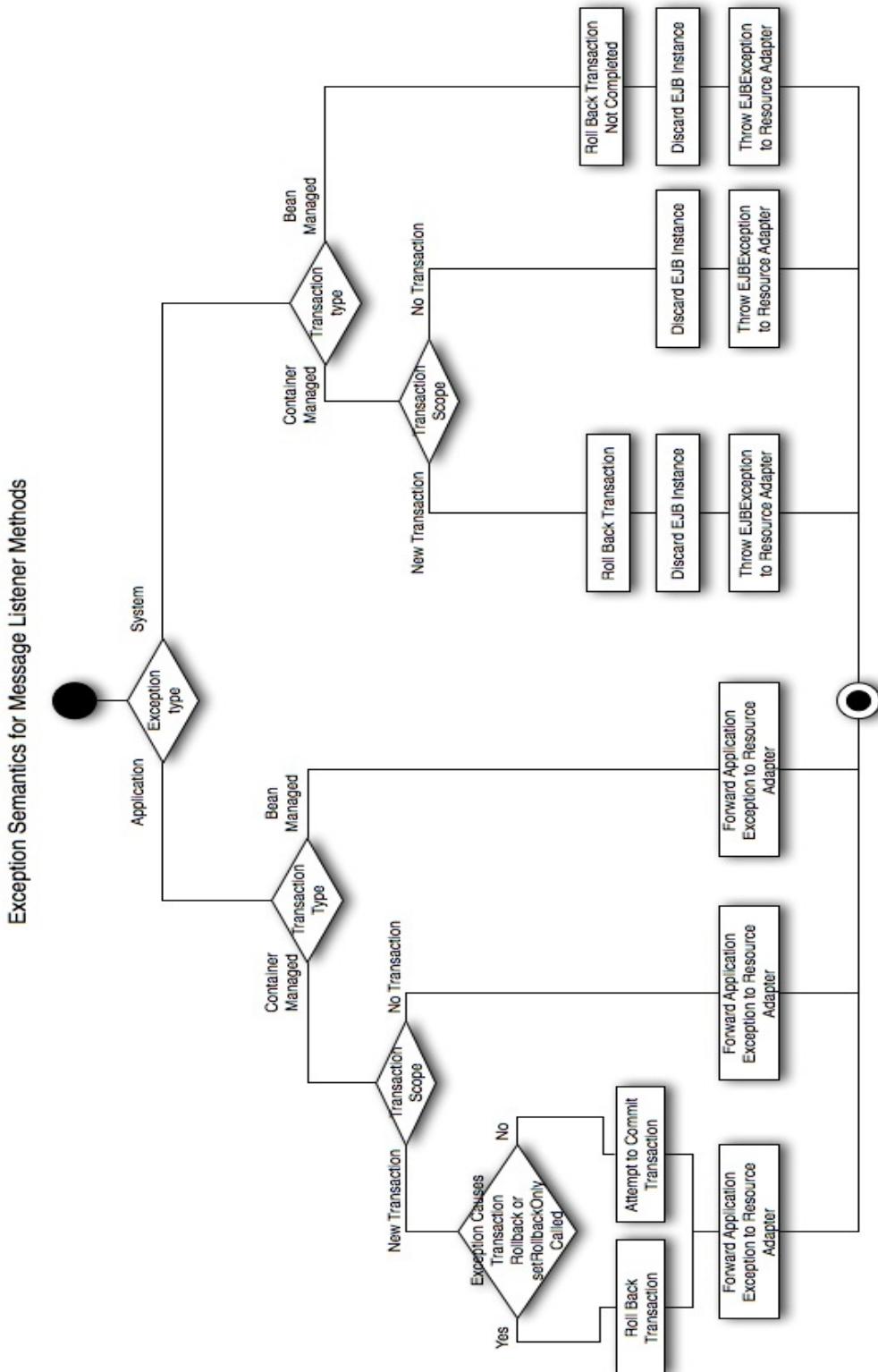
- Message listener methods must now throw the *RemoteException*.
- Message driven beans should not throw *RuntimeException*-s from any method.
Runtime exceptions will cause the message driven bean to be destroyed by the container.
If the message driven bean throwing a runtime exception uses bean-managed transactions, the container should not acknowledge the message.
- Message listener methods may throw application exceptions.
Such exceptions are propagated to the resource adapter.

The figures in this section show how exceptions from message driven bean methods are handled, how transactions are affected and what exceptions clients receives.

Omitted from the diagrams are logging to the container's log, which always takes place in the case of system exceptions.

8.9.1. Message Listener Method Exception Handling

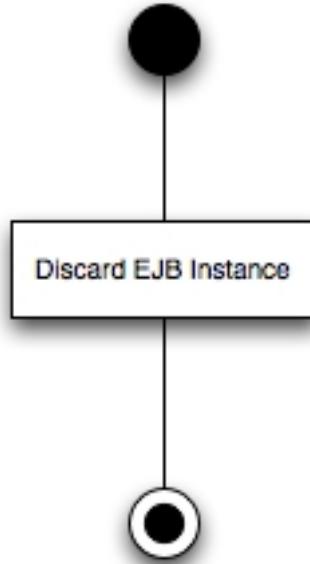
The following figure shows the exception handling for message listener methods, which are the business methods, of session beans:



Exception handling for message listener methods of message driven beans.

8.9.2. PostConstruct and PreDestroy Method Exception Handling

The following figure shows how system exceptions thrown by a PostConstruct or PreDestroy method of a message driven bean are handled. These methods are not allowed to throw application exceptions.



PostConstruct and PreDestroy message driven bean method exception handling.

8.9.3. Timeout Callback Method Exception Handling

The way the container handles exceptions from timeout callback methods is common to all kinds of EJBs. Please refer to the [corresponding section](#) in the chapter on exceptions.

8.9.4. Exception Handling in Other Callback Methods

Exceptions from other callback methods are handled in the same way, regardless of the type of EJB from which the exception is thrown. For a detailed description and a list of the different callback methods this applies to, please see the [corresponding section](#) in the chapter on exceptions.

8.10. Message Driven Beans and Security

References: EJB 3.1 Specification, section 5.4.13

The EJB 3.1 specification does not specify whether or not a caller principal will be available when the message listener methods of a message driven bean are invoked.

Applying the `@RunAs` annotation to a message driven bean will cause message listener methods and timeout methods to be executed with a specific security role. See [chapter 11](#) for details on security.

Different interceptor methods execute in different security contexts, please refer to the section on [interceptor types](#) in chapter 6 for details.

8.11. Message Driven Beans and Transactions

References: EJB 3.1 Specification, section 5.4.12.

There are usually two different transactions associated with each message being sent to a message driven bean; one transaction in which the producer sends the message to the destination and one transaction in which the message driven bean receives the message. In this section we will primarily concern ourselves with the latter transaction.

The methods of message driven beans are invoked in different transaction contexts, as specified in the following table:

Method	Transaction Context
PostConstruct life-cycle methods	Undefined
PreDestroy life-cycle methods	Undefined
Listener method(s)	As specified by annotations or in the ejb-jar.xml deployment descriptor.
Timeout callback methods	As specified by annotations or in the ejb-jar.xml deployment descriptor.
Constructor	Undefined
Dependency injection setter methods	Undefined
MessageDrivenContext setter method	Undefined

8.11.1. Transaction Attributes

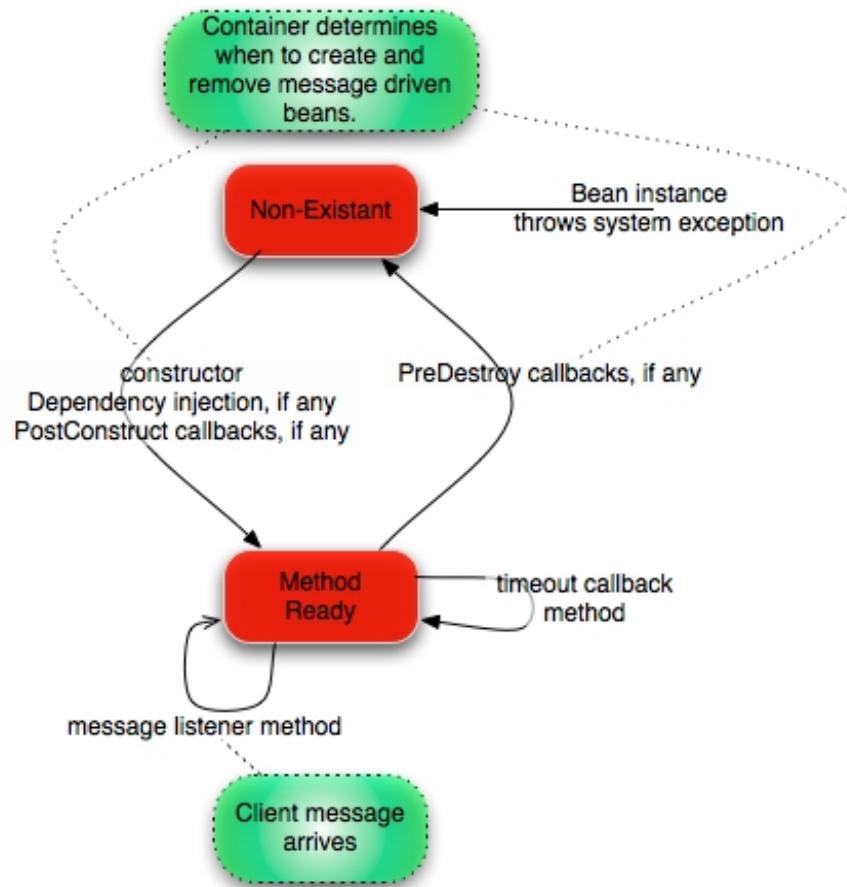
In message driven beans with container managed transactions, methods that support transactions must use transactions attributes as follows:

- Listener method(s)
REQUIRED or NOT_SUPPORTED
- Timeout callback methods
REQUIRED, REQUIRES_NEW or NOT_SUPPORTED

8.12. Message Driven Bean Life-Cycle

References: EJB 3.1 Specification, section 5.5.

The following figure illustrates the life-cycle of a message driven bean:



Life-cycle of a message driven bean.

The red boxes denote different states of a message driven bean.

The green boxes denote client events that causes the message driven bean to change state.

Text that is not boxed and not colored denote (callback) methods that are invoked on a message driven bean in connection to a state transition.

8.13. Operations Allowed in Methods of Message Driven Beans

References: EJB 3.1 Specification, section 5.5.1.

As with session beans, there are certain restrictions of what is allowed to do in different kinds of method of a message driven bean. The following table lists different kinds of methods and what operations they are allowed to perform.

The notes “BMT only” or “CMT only” means that these operations are only available when the bean has been managed, respective container managed, transactions.

JNDI access indicates whether the context “java:comp/env” is accessible.

If a message driven bean attempts to perform an illegal operation, an *IllegalStateException* will be thrown in the following two cases:

- Invoke a method in the *MessageDrivenContext* interface not permitted.
- Invoke a method in the *Timer* or *TimerService* interfaces not permitted.

Bean Method(s)	Allowed Operations
Constructor	None.
Dependency injection methods (setter methods).	MessageDrivenContext: lookup. JNDI Access: Available
PostConstruct and PreDestroy methods (lifecycle callback methods).	MessageDrivenContext: lookup, getContextData, getTimerService, getUserTransaction (BMT only). JNDI Access: Available EntityManagerFactory: Accessible.
Message listener methods or business method interceptor method.	MessageDrivenContext: getCallerPrincipal, isCallerInRole, lookup, getContextData, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only). JNDI Access: Available Resource managers: Accessible. Other EJBs: Accessible. EntityManagerFactory: Accessible. EntityManager: Accessible. Timer and TimerService methods: Accessible. UserTransaction methods: Accessible (BMT only).
Timeout callback methods.	MessageDrivenContext: getCallerPrincipal, lookup,

	<p>getContextData, getUserTransaction (BMT only), getRollbackOnly (CMT only), setRollbackOnly (CMT only).</p> <p>JNDI Access: Available</p> <p>Resource managers: Accessible.</p> <p>Other EJBs: Accessible.</p> <p>EntityManagerFactory: Accessible.</p> <p>EntityManager: Accessible.</p> <p>Timer and TimerService methods: Accessible.</p> <p>UserTransaction methods: Accessible (BMT only).</p>
--	--

In addition, the *getRollbackOnly* and *setRollbackOnly* methods in the *MessageDrivenContext* interface may only be called in methods of message driven beans that execute in a transaction context.

9. Create a Client that Sends Messages to a JMS Queue

References: EJB 3.1 Specification, section 5.3.

In order to send messages to a message driven bean, a client sends a message to a destination or an endpoint. In this chapter we will see how to implement clients sending messages to a JMS queue.

9.1. Creating a Java EE Client that Sends Messages to a JMS Queue

Java EE clients run in the managed environment of the container. This gives the advantage of facilities such as dependency injection being available.

In addition, not shown in this example, the sending of messages from other EJBs can occur in a transaction ensuring that either all or no messages are sent.

In [chapter 8](#) an example with a servlet sending messages to a message driven bean was shown. Instead of developing a new example, we'll take a closer look at the servlet developed in that example. For convenience, the servlet source-code has been reproduced here.

```
package com.ivan.scbcd6.producer;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.concurrent.atomic.AtomicLong;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Queue;
import javax.jms.Session;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.entity.MyMessage;

/**
 * Servlet producing JMS messages when accessed.
 */
@WebServlet(name="MessageProducerServlet", urlPatterns="/sendmsg.do")
public class MessageProducerServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = -4364474814559146703L;

    /* Instance variable(s): */
    /** Connection factory for queue. */
    @Resource(mappedName = "jms/QueueConnectionFactory")
    private ConnectionFactory mQueueConnectionFactory;
    /** Queue destination. */
    @Resource(mappedName = "jms/QueueDestination")
    private Queue mQueueDestination;
    /** MyMessage number counter. */
    private AtomicLong mMessageNumber = new AtomicLong(0);

    /**
     * @see HttpServlet#HttpServlet()
     */
    public MessageProducerServlet()
    {
        super();
        System.out.println("**** MessageProducerServlet created");
    }
}
```

```

 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
 */
@Override
protected void doGet(HttpServletRequest inRequest,
                      HttpServletResponse inResponse) throws ServletException, IOException
{
    PrintWriter theResponseWriter = inResponse.getWriter();

    try
    {
        sendJmsMessage();
        theResponseWriter.println("A message was sent at " + new Date());
    } catch (JMSException theException)
    {
        theResponseWriter.println("An error occurred sending message: " +
                                  theException);
    }
}

private void sendJmsMessage() throws JMSEException
{
    MessageProducer theJMSMessageProducer = null;
    Connection theJMSConnection = null;

    try
    {
        /* Retrieve a JMS connection from the queue connection factory. */
        theJMSConnection = mQueueConnectionFactory.createConnection();

        /*
         * Create the JMS session; not transacted and with auto-
         * acknowledge.
         */
        Session theJMSSession =
            theJMSConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        /* Create a JMS message producer for the queue destination. */
        theJMSMessageProducer =
            theJMSSession.createProducer(mQueueDestination);

        /* Create the object to be sent in the message created above. */
        MyMessage theObjectToSend = new MyMessage();
        theObjectToSend.setMessageNumber(mMessageNumber.incrementAndGet());
        theObjectToSend.setMessageString("Hello Message Driven Beans");
        theObjectToSend.setMessageTime(new Date());

        /* Create message used to send a Java object. */
        ObjectMessage theJmsObjectMessage =
            theJMSSession.createObjectMessage();
        theJmsObjectMessage.setObject(theObjectToSend);

        /* Send the message. */
        theJMSMessageProducer.send(theJmsObjectMessage);
    } finally
    {
        closeJmsResources(theJMSMessageProducer, theJMSConnection);
    }
}

/**
 * Closes the supplied JMS resources if they are not null.
 * If a supplied resource is null, then do nothing.
 *
 * @param inJMSMessageProducer JMS message producer to close.
 * @param inJMSConnection JMS connection to close.
 */
private void closeJmsResources(MessageProducer inJMSMessageProducer,
                               Connection inJMSConnection)
{
    if (inJMSMessageProducer != null)
    {
        try
        {
            inJMSMessageProducer.close();
        } catch (JMSEException theException)
        {
            // Ignore exceptions.
        }
    }
}

```

```

        }
    }
    if (inJMSConnection != null)
    {
        try
        {
            inJMSConnection.close();
        } catch (JMSEException theException)
        {
            // Ignore exceptions.
        }
    }
}

```

Note that:

- There is an instance variable named *mQueueConnectionFactory*.
The variable holds a reference to a JMS connection factory which is used to retrieve JMS connections.
- The *mQueueConnectionFactory* instance variable is annotated with the [@Resource annotation](#).
The *mappedName* element of the [@Resource annotation](#) specifies the JNDI name of the reference to be injected into the instance variable.
- There is an instance variable named *mQueueDestination*.
The variable holds a reference to a JMS queue destination, to which the client will send messages.
- The *mQueueDestination* instance variable is annotated with the [@Resource annotation](#).
The *mappedName* element of the [@Resource annotation](#) specifies the JNDI name of the reference to the queue destination to be injected into the instance variable.
- The *sendJmsMessage* method is responsible for sending JMS messages.
The messages are sent using the connection factory and destination which references are contained in the instance variables discussed above.
- The procedure for sending a JMS message is as follows:
 - Retrieve a JMS connection.
 - Create a JMS session in which the message will be sent using the JMS connection.
 - Create a JMS message producer that will be used to send the message to its destination.
 - Create the Java object to be enclosed in the JMS message and set its properties.
 - Create the JMS message to be sent.
 - Set the object to be enclosed in the JMS message.
 - Send the JMS message.
 - Close the JMS message producer and the JMS connection.
- The *closeJmsResources* method is responsible for closing the JMS resources used when sending a JMS message.

9.2. Creating a Java SE Client that Sends Messages to a JMS Queue

In this section we will create a standalone Java SE client that sends messages to a JMS queue. Prior to developing the example program, make sure that the GlassFish server is started and that only the example program from the section [Queue Message Driven Example](#) above is deployed.

The client is developed in a regular Java project in Eclipse.

- Create a new Java project.
I call my project MessageDrivenBeanSEClient, though the name does not matter.
- If using GlassFish v3, include the appropriate runtime library JAR on the classpath:
\$GLASSFISH_HOME/modules/gf-client.jar
- In the package *com.ivan.scbcd6.entity*, create the message bean class.
Under normal circumstances, classes such as this should be placed in a common library shared by the message driven bean application and the client application.

```
package com.ivan.scbcd6.entity;

import java.io.Serializable;
import java.util.Date;

/**
 * Represents a message sent by a producer to the topic.
 */
public class MyMessage implements Serializable
{
    /* Constant(s): */
    private static final long serialVersionUID = -4682924711829199796L;

    /* Instance variable(s): */
    private String mMessageString = "";
    private long mMessageNumber;
    private Date mMessageTime;

    public String getMessageString()
    {
        return mMessageString;
    }

    public void setMessageString(final String inMessageString)
    {
        mMessageString = inMessageString;
    }

    public long getMessageNumber()
    {
        return mMessageNumber;
    }

    public void setMessageNumber(final long inMessageNumber)
    {
        mMessageNumber = inMessageNumber;
    }

    public Date getMessageTime()
    {
        return mMessageTime;
    }

    public void setMessageTime(final Date inMessageTime)
    {
        mMessageTime = inMessageTime;
    }
}
```

- In the package `com.ivan.scbcd6.seclient`, create the `MessageDrivenBeanSEClient` class:

```

/**
 *
 */
package com.ivan.scbcd6.seclient;

import java.util.Date;
import java.util.concurrent.atomic.AtomicLong;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Queue;
import javax.jms.Session;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import com.ivan.scbcd6.entity.MyMessage;

/**
 * Java SE message driven bean client that sends messages to the
 * JMS queue that one or more message driven beans listen to.
 *
 * Note that the following JAR must be included on the client's class path:
 * GlassFish 3: $GLASSFISH_HOME/modules/gf-client.jar
 * GlassFish 2: $GLASSFISH_HOME/lib/appserv-rt.jar and $APS_HOME/lib/javaee.jar
 *
 * @author Ivan A Krizsan
 */
public class MessageDrivenBeanSEClient
{
    /* Constant(s): */
    private final static String JMS_CONNECTIONFACTORY_JNDI =
        "jms/QueueConnectionFactory";
    private final static String JMS_JMS_QUEUEDESTINATION_JNDI =
        "jms/QueueDestination";

    /* Instance variable(s): */
    /** Connection factory for queue. */
    private ConnectionFactory mQueueConnectionFactory;
    /** Queue destination. */
    private Queue mQueueDestination;
    /** MyMessage number counter. */
    private AtomicLong mMessageNumber = new AtomicLong(0);

    /**
     * Looks up the JMS resources required by the client to send
     * JMS messages.
     *
     * @throws NamingException If error occurs during lookup.
     */
    private void lookupJmsResources() throws NamingException
    {
        InitialContext theContext = new InitialContext();

        System.out.println("**** Starting JMS Resource Lookup...");

        mQueueConnectionFactory = (ConnectionFactory) theContext
            .lookup(JMS_CONNECTIONFACTORY_JNDI);
        mQueueDestination = (Queue) theContext
            .lookup(JMS_JMS_QUEUEDESTINATION_JNDI);

        System.out.println("      JMS Resource Loopup Finished.");
    }

    /**
     * Sends a JMS message with the next message number and increments the
     * message counter.
     *
     * @throws JMSException If error occurred sending message.
     */
    private void sendJmsMessage() throws JMSException
    {
}

```

```

MessageProducer theJMSMessageProducer = null;
Connection theJMSConnection = null;

try
{
    /* Retrieve a JMS connection from the queue connection factory. */
    theJMSConnection = mQueueConnectionFactory.createConnection();

    /*
     * Create the JMS session; not transacted and with auto-
     * acknowledge.
     */
    Session theJMSSession =
        theJMSConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);

    /* Create a JMS message producer for the queue destination. */
    theJMSMessageProducer =
        theJMSSession.createProducer(mQueueDestination);

    /* Create the object to be sent in the message created above. */
    MyMessage theObjectToSend = new MyMessage();
    theObjectToSend.setMessageNumber(mMessageNumber.incrementAndGet());
    theObjectToSend.setMessageString("Hello Message Driven Beans");
    theObjectToSend.setMessageTime(new Date());

    /* Create message used to send a Java object. */
    ObjectMessage theJmsObjectMessage =
        theJMSSession.createObjectMessage();
    theJmsObjectMessage.setObject(theObjectToSend);

    /* Send the message. */
    theJMSMessageProducer.send(theJmsObjectMessage);
} finally
{
    closeJmsResources(theJMSMessageProducer, theJMSConnection);
}

/**
 * Closes the supplied JMS resources if they are not null.
 * If a supplied resource is null, then do nothing.
 *
 * @param inJMSMessageProducer JMS message producer to close.
 * @param inJMSConnection JMS connection to close.
 */
private void closeJmsResources(MessageProducer inJMSMessageProducer,
                               Connection inJMSConnection)
{
    if (inJMSMessageProducer != null)
    {
        try
        {
            inJMSMessageProducer.close();
        } catch (JMSEException theException)
        {
            // Ignore exceptions.
        }
    }
    if (inJMSConnection != null)
    {
        try
        {
            inJMSConnection.close();
        } catch (JMSEException theException)
        {
            // Ignore exceptions.
        }
    }
}

/**
 * Main entry point of the Java SE message driven bean example
 * program.
 *
 * @param args Command line arguments.
 */
public static void main(String[] args)

```

```

{
    MessageDrivenBeanSEClient theClient = new MessageDrivenBeanSEClient();
    try
    {
        theClient.lookupJmsResources();

        for (int i = 0; i < 10; i++)
        {
            theClient.sendJmsMessage();
            System.out.println("### Sent message: " + (i + 1));
        }
    } catch (Exception theException)
    {
        theException.printStackTrace();
    }

    System.out.println("**** Java SE JMS Client finished.");
    System.exit(0);
}

```

Note that:

- When using GlassFish running on the same computer as the client, no parameters are needed when creating the *InitialContext* in the *lookupJmsResources* method.
- There is an instance variable named *mQueueConnectionFactory*.
The variable holds a reference to a JMS connection factory which is used to retrieve JMS connections.
- There is an instance variable named *mQueueDestination*.
The variable holds a reference to a JMS queue destination, to which the client will send messages.
- None of the two instance variables above are annotated.
- Instead of using dependency injection, as we saw in the Java EE client example earlier, the references to be stored in the two instance variables are looked up using JNDI in the method *lookupJmsResources*.
- The method *sendJmsMessage* is identical to the method with the same name in the Java EE client example above.
Please refer to the Java EE example for a detailed notes on this method.
- The *closeJmsResources* method is responsible for closing the JMS resources used when sending a JMS message.
The method is identical to the corresponding method in the Java EE client example above.
- We can conclude that the only difference between the Java EE and the Java SE clients is resource reference retrieval; in the Java EE client references are dependency-injected while they are looked-up in the Java SE client.

10. Add Transaction Demarcation Information to an EJB

References: EJB 3.1 Specification, chapter 13, section 15.6.

A transaction is a unit of work that consists of a number of operations that are either all performed successfully or not performed at all.

EJBs have support for distributed transactions meaning that the transaction can involve operations on EJBs that are deployed in different containers, for instance performing updates on a number of different databases.

There are two styles of transaction demarcation available in EJBs:

- Bean-managed transaction demarcation.
- Container-managed transaction demarcation.

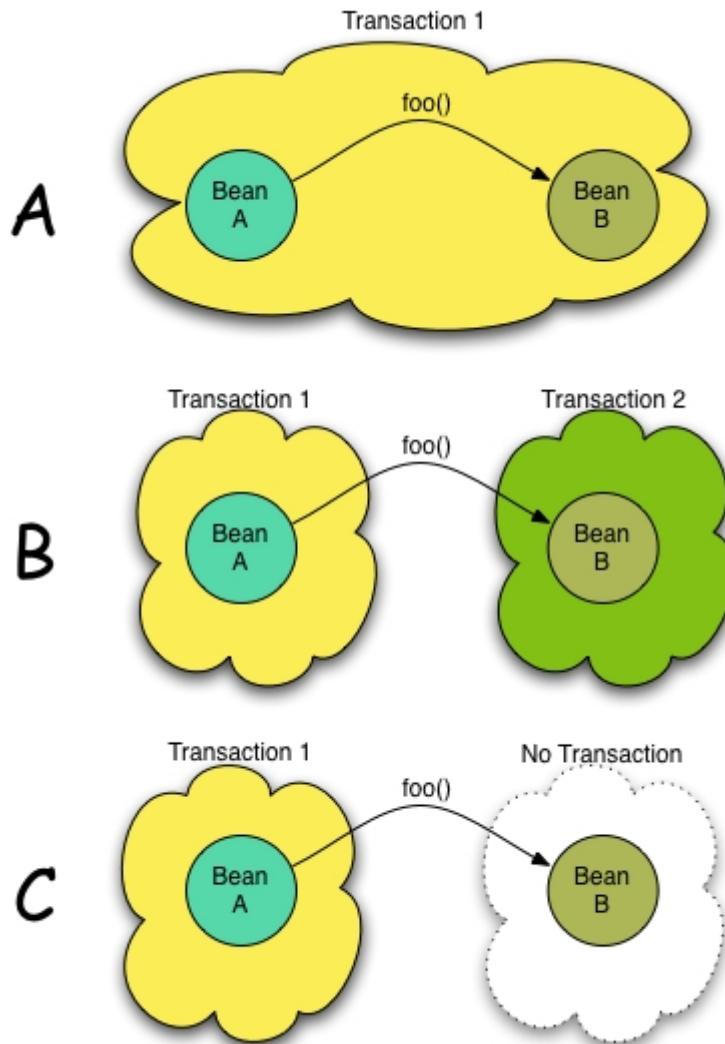
The EJB container is responsible for implementing low-level transaction management, such as the protocol between the transaction manager and miscellaneous resources such as databases and messaging providers, transaction context propagation and distributed commits.

The EJB specification only require support for flat transactions, meaning that there may not be transactions that exist within the scope of another transaction.

Please also see the chapter [EJBs and Exceptions](#) on how transactions are handled in connection to different kinds of exceptions occurring.

10.1. Transaction Propagation

A transaction context can be shared by multiple EJBs. There are three different ways that a transaction context can propagate from an EJB that invokes the services of another EJB.

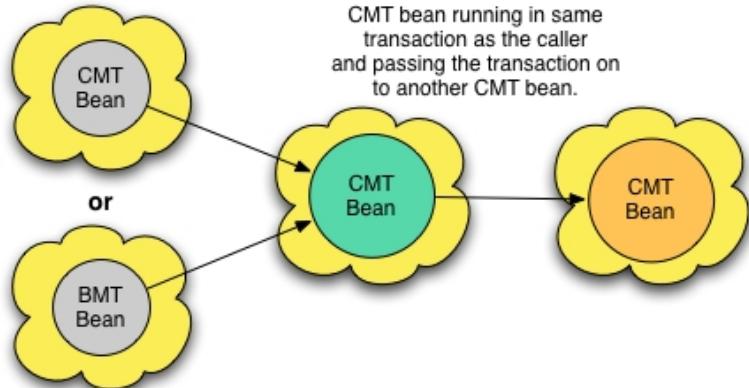


The three different ways in which transactions may propagate between EJBs.

- In figure A, the transaction context in which bean A executes propagates to bean B.
- In figure B, a new transaction context is created in which bean B executes.
- In figure C, the transaction context in which bean A executes is suspended and bean B executes outside of any transaction context.

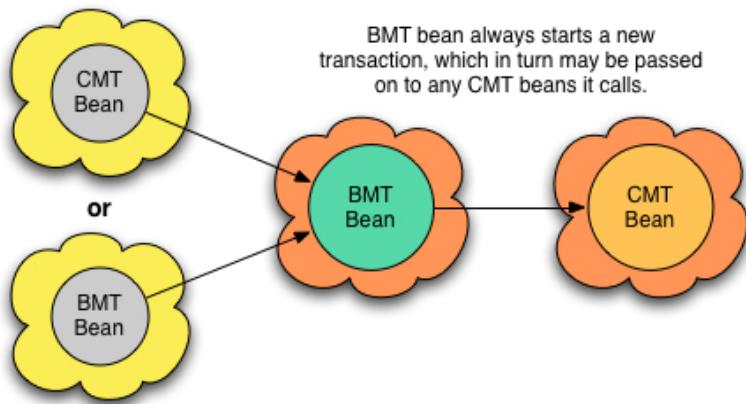
For EJBs with bean managed transactions (BMT), only options B and C applies – transactions from other EJBs never propagate into an EJB with BMT.

Transactions can propagate between EJBs, that is a business method of bean A can call a business method of bean B, both which may or may not execute in the same transaction. The first figure below illustrates transaction context propagation from EJBs with container managed transactions (CMT):



Possible transactions propagations into CMT EJBs.

The second figure shows transaction context propagation from EJBs with bean managed transactions (BMT):



Possible transaction propagations into BMT EJBs.

10.2. Transaction Context of EJB Methods

References: EJB 3.1 Specification sections 4.3.15 and 13.6.5.

A transaction context is not available in all methods of EJBs. Examples of methods that execute in an unspecified transaction context include:

- PostConstruct and PreDestroy methods in EJBs with container managed transactions.
PostConstruct and PreDestroy methods of [singleton session beans](#).
- PrePassivate and PostActivate methods in EJBs with container managed transactions.
- Methods in EJBs with container managed transactions that have the transaction attributes NEVER or NOT_SUPPORTED.
- Methods in EJBs with container manage transactions that have the transaction attribute SUPPORTS when the client executes in an unspecified transaction context.

In the above cases, the EJB should not rely on a transaction context being present. In addition, resource manager access, such as access to a database, that fails in a method with an unspecified transaction context may result in the resource in an unknown state.

10.3. Bean-Managed Transaction Demarcation

Bean-managed transaction demarcation means that the developer uses code in the EJB to start, commit and rollback transactions. The following is specific to EJBs with bean managed transactions (BMT):

- BMT, is coded in the bean using the *UserTransaction* interface.
- An EJB using BMT must not use the *getRollbackOnly* and *setRollbackOnly* method in the *EJBContext* interface.
Instead the *getStatus* and *rollback* methods in *UserTransaction* should be used.
- Only session beans and message driven beans may use bean managed transactions.
- Message driven EJBs and stateless session EJBs must end a BMT before the end of the method (business method or timeout callback method).
- Stateful session EJBs may keep a bean managed transaction open over multiple business method calls from client.
- Stateless session beans and message driven beans must commit a transaction before the method in which the transaction was started returns.
- An EJB with BMT will only run in a transaction created by the EJB itself.
Client transactions never propagate into an EJB with BMT.
- Updates to resource managers between *UserTransaction.begin()* and *UserTransaction.commit()* are performed in a transaction.
- While in a transaction, resource-manager transaction demarcation APIs must not be used.
- With BMT, the call to *UserTransaction.begin()* is not considered as inside the transaction while the call to *UserTransaction.commit()* (or *UserTransaction.rollback()*) is considered to be inside the transaction.

Reasons for using bean managed transactions:

- If there is a need to keep the transaction open over several business method calls when using a stateful session bean.

- Transaction scope can be reduced to a scope smaller than a business methods, to improve performance.
- Can be used with message driven beans, in order to be able to acknowledge a message even though the transaction rolls back.

Transaction management is done, as before, by invoking methods on an object that implements the *javax.transaction.UserTransaction* interface, which contains the following methods:

Method Name	Description
begin	Creates a new transaction and associates it with the current thread.
commit	Attempts to commit the transaction by performing all the operations that have been performed in the scope of the transaction. Disassociates the current thread and the transaction.
getStatus	Retrieves the current status of the transaction. Transaction status codes are declared in the <i>javax.transaction.Status</i> interface.
rollback	Roll back the transaction and ensures that none of the operations that have been performed in the scope of the transaction are performed. Disassociates the current thread and the transaction.
setRollbackOnly	Sets the only possible outcome of the transaction to rollback, ensuring that none of the operations that have been performed in the scope of the transaction will be performed.
setTransactionTimeout	Sets the time in seconds counting from the point in time when the <i>UserTransaction.start</i> method was called after which the transaction will timeout. Supplying a zero value restores the default timeout, which is implementation specific.

10.3.1. Example of an EJB with Bean Managed Transactions

The following is an example of a stateless session bean that uses bean managed transactions. Using transactions in this bean is quite unnecessary, as it does not use any transactional resources – it is just meant as an example of how to write bean managed transaction code in an EJB.

```
package com.ivan.scbcd6;

import java.util.Date;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.NotSupportedException;
import javax.transaction.RollbackException;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;

/**
 * Simplest possible stateless session bean exposing a local, no-interface
 * view.
 * The EJB has bean managed transactions.
 */
@Stateless
@LocalBean
@TransactionManagement(TransactionManagementType.BEAN)
public class StatelessSession1Bean
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;
    private int mCallCounter;
    @Resource
    private SessionContext mBeanContext;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatelessSession1Bean " + mInstanceNumber +
                           " created: " + new Date());
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();
        String theMessage = "";
        mCallCounter++;

        try
        {
            /*
             * Retrieve the UserTransaction object and start the transaction.
             */
            UserTransaction theUserTransaction = mBeanContext.getUserTransaction();
            System.out.println("**** Transaction begin...");
            theUserTransaction.begin();

            /*
             * Do what is to be done inside the transaction, accessing any
             * transactional resources, other EJBs etc.
             */
        }
    }
}
```

```

theMessage = "Hello " + inName + ", I am stateless session bean " +
mInstanceNumber + ". The time is now: " + thecurrentTime;

/*
 * End the transaction.
 * Rollback the transaction every third call to one and the
 * same session bean instance.
 */
if (mCallCounter % 3 == 0)
{
    System.out.println("**** Transaction rollback.");
    theUserTransaction.rollback();
} else
{
    System.out.println("**** Transaction commit.");
    theUserTransaction.commit();
}
} catch (NotSupportedException theException)
{
/*
 * Current thread is already associated with a transaction
 * and nested transactions are not supported.
 */
theException.printStackTrace();
} catch (SystemException theException)
{
/* Transaction manager encountered unexpected problem. */
theException.printStackTrace();
} catch (SecurityException theException)
{
/* Thread is not allowed to commit the transaction. */
theException.printStackTrace();
} catch (IllegalStateException theException)
{
/* The current tread is not associated with the transaction. */
theException.printStackTrace();
} catch (RollbackException theException)
{
/* Transaction has been rolled back instead of committed. */
theException.printStackTrace();
} catch (HeuristicMixedException theException)
{
/* Some updates has been committed, some rolled back. */
theException.printStackTrace();
} catch (HeuristicRollbackException theException)
{
/* All updates have been rolled back. */
theException.printStackTrace();
}

return theMessage;
}
}

```

Note that:

- The session bean implementation class is annotated with the `@TransactionManagement(TransactionManagementType.BEAN)` annotation to configure the session bean to use bean-managed transactions.
- A reference to the *SessionContext* is injected into the EJB.
- Using the *SessionContext* reference, a reference to an *UserTransaction* is retrieved.
- The transaction is started by invoking the *begin* method on the *UserTransaction* reference.
- The transaction can be rolled back by invoking *rollback* on the *UserTransaction* reference.
- The transaction can be committed by invoking the *commit* method on the *UserTransaction* reference.
- There are a number of exceptions that may be thrown when using bean-managed transactions, as can be seen in the different catch-clauses.
Especially note *HeuristicMixedException* and *HeuristicRollbackException*.
For further information on these exceptions, please consult the API documentation.

10.4. Container-Managed Transaction Demarcation

References: EJB 3.1 Specification, section 4.3.7.

With container-managed transactions, annotations or the ejb-jar.xml deployment descriptor is used to specify transaction boundaries. This metadata can instruct the EJB container to:

- Consider an EJB method as an operation to be included in the current transaction.
- Start a new transaction and consider an EJB method as an operation to be included in the new transaction.
- Consider an EJB method as an operation that is not included in any transaction.

The following is specific to beans with container managed transactions (CMT):

- Container managed transactions, CMT, are declared using annotations or in the ejb-jar.xml deployment descriptor.
- In the following types of methods of an EJB with CMT resource-manager transaction demarcation APIs must not be used, neither must they attempt to obtain or use the *UserTransaction* interface:
 - Business methods
 - Message listener methods
 - Business method interceptor methods
 - Lifecycle callback interceptor methods
 - Timeout callback methods
- Transactions can propagate into an EJB with CMT.
- Stateful session beans with CMT can optionally implement the *javax.ejb.SessionSynchronization* interface or use the @AfterBegin, @BeforeCompletion and @AfterCompletion annotations, in order to receive notification of transaction boundaries. See subsequent section for details!
- A CMT transaction ends when the method that started the transaction completes.

Reasons for using container managed transactions:

- EJB transactional behaviour can be changed without having to change the bean code.
- Simplified development.
- Makes EJB reuse easier.
- The business methods of an EJB using CMT can run in the transaction context of another bean, which beans with BMT cannot.

10.4.1. Session Synchronization

Stateful session beans with container-managed transactions can optionally implement the *SessionSynchronization* interface or use one or more of the `@AfterBegin`, `@BeforeCompletion` and `@AfterCompletion` annotations to receive notification when a transaction has begun, is about to end and after it has ended.

The *SessionSynchronization* interface contains the following methods:

Method	Comments
<code>void afterBegin()</code>	Notifies a session bean that a new transaction has started and the subsequent business methods on that instance will run in that transaction. Invoked before first business method in transaction is invoked, which is not necessarily immediately after the transaction has been started.
<code>void afterCompletion(boolean committed)</code>	Notifies a session bean the transaction has finished and whether the transaction was committed or rolled back.
<code>void beforeCompletion()</code>	Notifies a session bean that the current transaction is about to be ended.

- The *afterBegin* method gives the session bean an opportunity to load any data it needs during a transaction.
The method executes inside a transaction, so “everything” is available; resource managers, EJB environment, other EJBs, the *SessionContext* (except for, of course, BMT related methods).
Corresponding annotation: `@AfterBegin`
- The *beforeCompletion* method gives the session bean an opportunity to update persistent storage prior to the transaction end.
The method also executes inside a transaction, so “everything” is available; resource managers, EJB environment, other EJBs, the *SessionContext* (except for BMT related methods).
The bean is also given a last chance to roll back a transaction that is about to be committed.
Corresponding annotation: `@BeforeCompletion`
- The *afterCompletion* method gives the session bean an opportunity to learn the result of a transaction and act accordingly.
This method executes outside of a transaction context, so resource managers and other EJBs may no longer be accessed. Also, any transaction-related methods in the *SessionContext* interface are now unavailable.
Corresponding annotation: `@AfterCompletion`

In addition to implementing the *SessionSynchronization* interface and using annotations, it is also possible to configure session synchronization methods in the ejb-jar.xml deployment descriptor. Session synchronization methods can have any visibility modifier; public, package, protected or private but must not be final or static.

The methods in an EJB implementing the *SessionSynchronization* interface or using one or more of the corresponding annotations are restricted to using the REQUIRED, REQUIRES_NEW or MANDATORY transaction attributes.

10.4.2. Example of an EJB with Container Managed Transactions

The following is an example of a stateful session bean that uses container managed transactions as well as implements the *SessionSynchronization* interface. Using transactions in this bean is quite unnecessary, as it does not use any transactional resources – it is just meant as an example of how to write bean managed transaction code in an EJB.

```
package com.ivan.scbcd6;

import java.rmi.RemoteException;
import java.util.Date;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.EJBException;
import javax.ejb.LocalBean;
import javax.ejb.SessionContext;
import javax.ejb.SessionSynchronization;
import javax.ejb.Stateful;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

/**
 * Simplest possible stateless session bean exposing a local, no-interface
 * view.
 * The EJB has container managed transactions.
 */
@Stateful
@LocalBean
@TransactionManagement(TransactionManagementType.CONTAINER)
public class StatefulSessionBean implements SessionSynchronization
{
    private static int sCurrentInstanceNumber = 1;

    private int mInstanceNumber;
    private int mCallCounter;
    @Resource
    private SessionContext mBeanContext;

    @PostConstruct
    public void initialize()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
        System.out.println("**** StatelessSession1Bean " + mInstanceNumber +
                           " created: " + new Date());
    }

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public String greeting(final String inName)
    {
        Date theCurrentTime = new Date();
        String theMessage = "";
        mCallCounter++;

        /* Do what is to be done inside the transaction. */
        theMessage = "Hello " + inName + ", I am stateless session bean " +
                     mInstanceNumber + ". The time is now: " + theCurrentTime;

        /*
         * Every third call to the same session bean instance
         * the transaction will be marked for rollback.
         * Marking a transaction for rollback should be done before
         * throwing application exceptions that have not been
         * marked as causing transaction rollback when being thrown
         * using the @ApplicationException annotation or corresponding
         * ejb-jar.xml deployment descriptor element.
         */
    }
}
```

```

        if (mCallCounter % 3 == 0)
        {
            System.out.println("**** Transaction rollback.");
            mBeanContext.setRollbackOnly();
        }

        System.out.println("**** Transaction marked for rollback: " +
            mBeanContext.getRollbackOnly());

        return theMessage;
    }

    /**
     * Notifies the stateful session bean that a new transaction
     * has begun.
     * This method executes in a transaction context.
     *
     * @throws EJBException If error occurred.
     * @throws RemoteException Not used.
     */
    @Override
    public void afterBegin() throws EJBException, RemoteException
    {
        System.out.println("**** StatefulSessionBean.afterBegin");
    }

    /**
     * Notifies the stateful session bean that a transaction has been
     * completed and whether the transaction was committed or not.
     * This method executes outside any transaction context.
     *
     * @param inCommitted True if transaction committed, false otherwise.
     * @throws EJBException If error occurred.
     * @throws RemoteException Not used.
     */
    @Override
    public void afterCompletion(boolean inCommitted) throws EJBException,
        RemoteException
    {
        System.out.println("**** StatefulSessionBean.afterCompletion: " +
            inCommitted);
    }

    /**
     * Notifies the stateful session bean that a transaction is about
     * to be completed.
     * This method executes in a transaction context.
     *
     * @throws EJBException If error occurred.
     * @throws RemoteException Not used.
     */
    @Override
    public void beforeCompletion() throws EJBException, RemoteException
    {
        System.out.println("**** StatefulSessionBean.beforeCompletion");
    }
}

```

Note that:

- The session bean implementation class is annotated with the `@TransactionManagement(TransactionManagementType.CONTAINER)` annotation to configure the session bean to use container-managed transactions.
- A reference to the *SessionContext* is injected into the EJB.
- The *greeting* method is annotated with the `@TransactionAttribute(TransactionAttributeType.REQUIRED)` annotation. This tells the container that the method require a transaction context. For details, see the section on transaction attributes below.
- The transaction can be marked for rollback by invoking the *setRollbackOnly* method on the *SessionContext* reference injected into the EJB.
- The EJB can inquiry about whether the transaction has been marked for rollback by calling the *getRollbackOnly* method on the *SessionContext* reference.
- There are no exception-handling code related to transaction management in the *greeting* method.

10.4.3. Transaction Attributes

References: EJB 3.1 Specification, sections 13.3.7 and 13.6.2.

A transaction attribute is connected to a method and specifies how the container is to manage transactions for the method in question when it is being invoked. Transaction attributes is what controls whether a transaction context will propagate when one EJB invokes another EJB, as described in the section [Transaction Propagation](#) above.

The following methods types have a transaction attribute:

- Methods in the business interface.
- Methods in the no-interface view.
- Message listener methods in message driven beans.
- Timeout callback methods.
- Web service endpoint methods of session beans.
- Methods in an EJB's home or component interface.
For EJB version <= 2.1.
- Singleton session bean's PostConstruct and PreDestroy lifecycle callback interceptor methods.

The following table lists the available transaction attributes and their meaning:

Transaction Attribute	Deployment Descriptor Value	Description
MANDATORY	Mandatory	The method will be executed in the transaction context of the caller. If no such context exists, a <i>javax.ejb.EJBTransactionRequired</i> exception is thrown. If EJB 2.1 client view is used, a <i>javax.transaction.TransactionRequiredException</i> (remote client) or <i>javax.ejb.TransactionRequiredLocalException</i> (local client) will be thrown.
REQUIRED	Required	The method executes in the client's transaction context. If no client transaction context, container starts a new transaction. Default transaction attribute.
REQUIRES_NEW	RequiresNew	Container starts a new transaction immediately prior to invoking the method. Any existing client transaction context is suspended.
SUPPORTS	Supports	If the caller executes in a transaction context, the method will also execute in that context. Otherwise, the method will run without a transaction context.
NOT_SUPPORTED	NotSupported	The method always executes without a transaction context. If the caller executes in a transaction context, the context will be suspended until the method finished executing.
NEVER	Never	If the caller executes in a transaction context, an <i>java.rmi.RemoteException</i> (remote client) or <i>javax.ejb.EJBException</i> (local client) will be thrown. Otherwise, the method will run without a transaction context.

The container will attempt to commit any transaction context created immediately prior to invoking a method on an EJB when the method execution has completed.

A transaction attribute of a method in an EJB can be specified either using the `@TransactionAttribute` annotation or the `ejb-jar.xml` deployment descriptor. As usual, any configuration in the deployment descriptor overrides configuration by annotations.

Certain methods in EJBs only allow certain transaction attributes:

- Message listener methods in message driven beans.
Only REQUIRED and NOT_SUPPORTED are allowed.
- Timeout callback methods of EJBs.
Only REQUIRED, REQUIRES_NEW and NOT_SUPPORTED are allowed.
- Asynchronous business methods in session beans.
Only REQUIRED, REQUIRES_NEW and NOT_SUPPORTED are allowed.
- PostConstruct and PreDestroy lifecycle callback methods in singleton session beans.
Only REQUIRED, REQUIRES_NEW and NOT_SUPPORTED are allowed.
- Methods in an EJB implementing the *SessionSynchronization* interface or using one or more of the corresponding annotations.
Only REQUIRED, REQUIRES_NEW or MANDATORY are allowed.

The following table summarizes how a transaction context is propagated from a client invoking a business method of an EJB to the business method and, finally, to resource managers used by the business method in the case of different transaction attributes. TC is the client's transaction context, TB is the transaction context of a transaction started immediately prior to the invocation of the business method.

Transaction Attribute	Client's Transaction	Business Method Transaction	Resource Managers Transaction
Not Supported	None	None	None
	TC	None	None
Required	None	TB	TB
	TC	TC	TC
Supports	None	None	None
	TC	TC	TC
RequiresNew	None	TB	TB
	TC	TB	TB
Mandatory	None	Exception	-
	TC	TC	TC
Never	None	None	None
	TC	Exception	-

Transaction context container passes to business methods and resource managers from a business method.

Transaction Attribute and Annotations

The following shows an example of how to specify a transaction attribute on a method in an EJB using annotations:

```
...
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public String greeting(final String inName)
{
...
}
```

Transaction Attributes in the Deployment Descriptor

The deployment descriptor can be used to specify transaction attributes for methods of EJBs or to override transaction attributes specified using annotations.

Below is an example of different ways of specifying the method(s) for which a transaction attribute is to be assigned.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">
...
<assembly-descriptor>
...
<!--
    Set all the methods in the EJB with the name StatelessSessionBean
    to have the transaction attribute Required.
    Note the use of the wildcard when specifying the method name(s).
    There must be only one <container-transaction> element per EJB
    that uses the wildcard when specifying method name.
-->
<container-transaction>
    <method>
        <ejb-name>StatelessSessionBean</ejb-name>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
<!--
    Set the method(s) named controlCitizen in the EJB with the name
    StateControlSessionBean to have the transaction attribute Mandatory.
    Applies to all methods with the specified name in the EJB.
-->
<container-transaction>
    <method>
        <ejb-name>StateControlSessionBean</ejb-name>
        <method-name>controlCitizen</method-name>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
</container-transaction>
<!--
    Set the method named diggForGold with the parameter-types
    String, int and MyClass in the EJB with the name
    LawlessSessionBean to have the transaction attribute RequiresNew.
    Specifies a single method using name and parameter signature.
-->
<container-transaction>
    <method>
        <ejb-name>LawlessSessionBean</ejb-name>
        <method-name>diggForGold</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>int</method-param>
            <method-param>com.ivan.scbcd6.MyClass</method-param>
        </method-params>
    </method>
    <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
```

```
</assembly-descriptor>  
</ejb-jar>
```

Note that:

- In the first <container-transaction> element, the <method-name> element contains the * character.
This is a wildcard character which matches all the methods in the EJB with the specified name.
- In the second <container-transaction> element, the name of the method(s) is explicitly specified.
In the case where there are multiple methods with the same name in the EJB in question, the transaction attribute will apply to all the methods.
- In the third <contianer-transaction> element, not only the method name of the method which is to be assigned the RequiresNew transaction attribute is specified but also the parameter signature of the method.
The ability to be able to specify parameter types enables assigning different transaction attributes to methods with the same name.

Transaction Attributes and Inheritance

Transaction attributes of an EJB class with a superclass is determined using the following rules:

- If no transaction attribute is specified for a class, then this is equal to the transaction attribute REQUIRED having been specified on the class.
- The transaction attribute defined on a class, or the default transaction attribute, applies to all methods defined in the class.
- A method defined in a superclass has the transaction attribute according to the previous point.
- A method defined in a superclass but overridden in a child class (to the superclass) has the transaction attribute as defined in the child class, or the default transaction attribute of the child class.

10.5. JMS Messaging and Transactions

References: EJB 3.1 Specification, section 13.3.5.

JMS messages sent from within a transaction are not delivered until the transaction commits. Thus an EJB should not attempt to send a JMS message and wait for a reply to the message within one and the same transaction, since there will never be a reply.

11.Add Security Checks to Business Logic

References: EJB 3.1 Specification, chapter 17 and section 15.8.

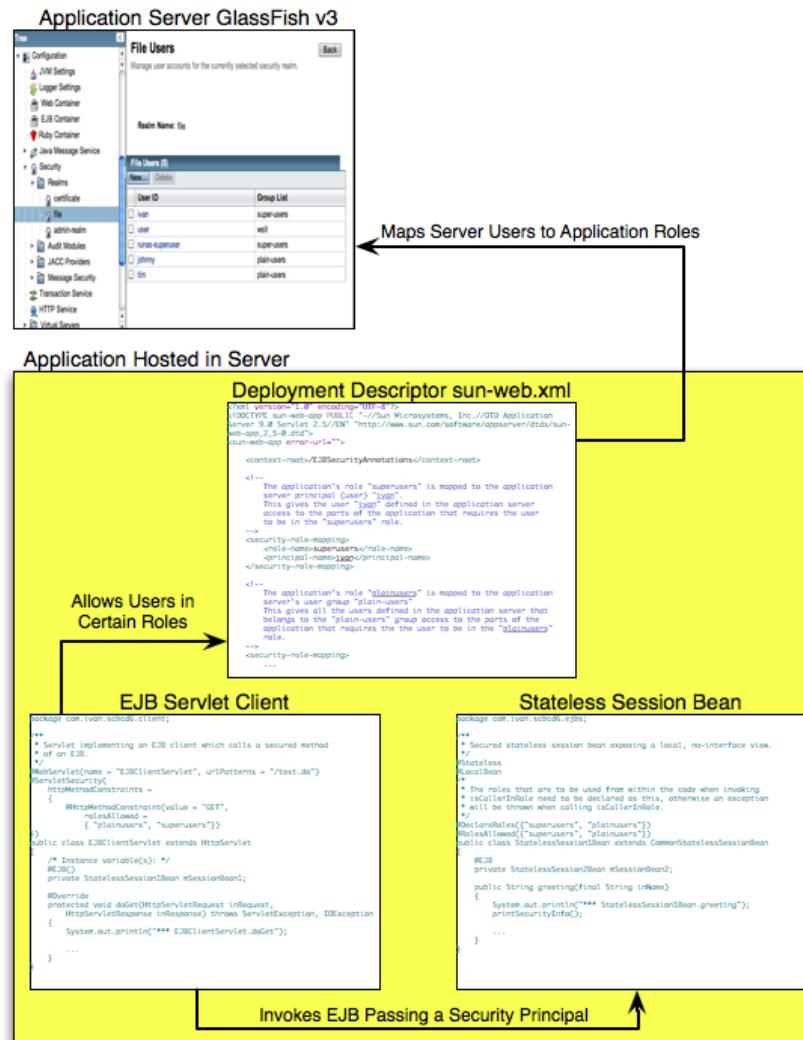
This chapter will deal with how security of EJBs works, which options are available and how to use declarative as well as programmatic security. For details on basic security configuration in the GlassFish v3 application server as well as explanations of some basic security concepts, please refer to [appendix G](#).

EJB security concerns itself with authorization and not authentication:

- Authorization
Ensure that the user only can access the resources which he has permission to access.
- Authentication
Confirm the identity of the user. Commonly through the process of supplying a login and password.

11.1. EJB Security Examples Overview

As customary, we'll start this chapter with an example. The example in this chapter involves not only developing an example application, but also security configuration of the GlassFish v3 application server. The following figure shows an overview of how the main parts related to security are related to each other; the application server, the product-specific deployment descriptor, the EJB client (a servlet) and finally the EJB:



Outline of the different security artifacts and their relations in this chapter's example program.

11.2. Create Users in GlassFish

In order to prepare for the security example programs, we set up the users in the GlassFish application server. According to the instructions in [appendix G](#), create the users listed in this table belonging to the named groups and with the listed passwords:

User Name	Group	Password
ivan	super-users	ivan
runas-superuser	super-users	super
johnny	plain-users	johnny

11.3. Annotations EJB Security Example

As usual, we'll develop the example program in a Dynamic Web project in Eclipse. Since a second EJB will be introduced later, some common functionality has been extracted to a common superclass.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project "EJBSecurityAnnotations".

11.3.1. Common Session Bean Superclass

In a later part of the example a second session bean will be introduced. The common session bean superclass extracts common properties of the session bean implementations to minimize code duplication.

- In the package *com.ivan.scbcd6.ejb*s, create the common superclass for the stateless session beans *CommonStatelessSessionBean*.

```
package com.ivan.scbcd6.ejb;

import java.security.Principal;
import java.util.Date;
import javax.annotation.Resource;
import javax.ejb.SessionContext;

/**
 * Class gathering common properties of the stateless session beans
 * in this example.
 */
public class CommonStatelessSessionBean
{
    /* Class variable(s): */
    protected static int sCurrentInstanceNumber = 1;

    /* Instance variable(s): */
    @Resource
    protected SessionContext mSessionContext;
    protected int mInstanceNumber;

    /**
     * Initializes session bean instance number.
     */
    public CommonStatelessSessionBean()
    {
        mInstanceNumber = sCurrentInstanceNumber++;
    }

    /**
     * Prints security information from the session context, if available.
     */
}
```

```

protected void printSecurityInfo()
{
    if (mSessionContext != null)
    {
        /*
         * Container may never return a null principal, so we don't
         * need to make sure it is not null.
         */
        Principal theCallerPrincipal = mSessionContext.getCallerPrincipal();
        System.out.println("    Principal name: " + theCallerPrincipal.getName());
        System.out.println("    Principal object: " + theCallerPrincipal);
        System.out.println("    Principal type: " + theCallerPrincipal.getClass());

        testCallerRole("superusers");
        testCallerRole("plainusers");
        testCallerRole("ivan");
    } else
    {
        System.out.println("    No session context available.");
    }
}

/**
 * Tests whether the caller is in the supplied role.
 */
private void testCallerRole(String inRoleName)
{
    try
    {
        System.out.println("    Caller in '" + inRoleName + "' role? " +
                           mSessionContext.isCallerInRole(inRoleName));
    } catch (Throwable theException)
    {
        System.out.println("    Cannot determine caller role: '" + inRoleName +
                           "'");
    }
}

/**
 * Creates a greeting for supplied name and bean with supplied name.
 *
 * @param inName Name of person to greet.
 * @param inBeanName Bean name doing the greeting.
 * @return Greeting.
 */
protected String assembleGreeting(String inName, String inBeanName)
{
    Date theCurrentTime = new Date();
    return "Hello " + inName + ", " + inBeanName + ", instance: " +
           mInstanceNumber + ". The time is now: " + theCurrentTime;
}
}

```

Note that:

- The class contains the *mSessionContext* instance variable, which is annotated with the *@Resource* annotation.
Despite the instance variable being declared in the superclass of a class implementing an EJB, dependency injection will take place.
- In the method *printSecurityInfo*, the method *getCallerPrincipal* is invoked on the session context object.
This retrieves the an object that identifies the caller of the EJB method. We'll look at more details in the section on [programmatic security](#).
- In the method *testCallerRole*, the method *isCallerInRole* is invoked on the session context.
This method tests wether the caller is in the security role with the supplied name. Again, further details in the section on [programmatic security](#).

11.3.2. First Session Bean

In the first version of the security example program, there will be only one single session bean.

- In the package `com.ivan.scbcd6.ejbs`, create the class `StatelessSession1Bean`:

```
package com.ivan.scbcd6.ejbs;

import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

/**
 * Secured stateless session bean exposing a local, no-interface view.
 */
@Stateless
@LocalBean
/*
 * The roles that are to be used from within the code when invoking
 * isCallerInRole need to be declared using the @DeclareRoles annotation,
 * otherwise an exception will be thrown when calling isCallerInRole.
 */
@DeclareRoles({"superusers", "plainusers"})
/*
 * When applied at class-level, the @RolesAllowed annotation specifies
 * which security-roles are allowed to access all the methods in the
 * EJB. @RolesAllowed may also be used at method-level.
 */
@RolesAllowed({"superusers", "plainusers"})
public class StatelessSession1Bean extends CommonStatelessSessionBean
{
    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        System.out.println("**** StatelessSession1Bean.greeting");
        printSecurityInfo();

        String theMessage = assembleGreeting(inName, "StatelessSession1Bean");
        return theMessage;
    }
}
```

Note that:

- The session bean class is annotated with the `@DeclareRoles` annotation in which a list of security roles is supplied.
This annotation declares the security roles that are used for programmatic security from within the code of the EJB. If a security role is not declared, it will be impossible to determine if the current user is in the security role in question using the `isCallerInRole` method in the `SessionContext`.
The code performing programmatic security is located in the superclass of the session bean.
- The session bean class is annotated with the `@RolesAllowed` annotation in which a list of security roles is supplied.
This annotation, which can be declared both at class- and method-level, specifies which security roles are allowed to invoke method(s) on the EJB.
If annotating the class, then the roles applies to all methods in the EJB. If annotating a single method, then the roles only applies to that single method.

11.3.3. EJB Client Servlet

As usual for most of the examples in this book, the client of the EJB is a servlet.

In this example, having a servlet as client gives us the opportunity of allowing the user to log in when running the example program.

- In the package *com.ivan.scbcd6.client*, create the class *EJBClientServlet*:

```
package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.HttpMethodConstraint;
import javax.servlet.annotation.ServletSecurity;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.ivan.scbcd6.ejbs.StatelessSession1Bean;

/**
 * Servlet implementing an EJB client which calls a secured method
 * of an EJB.
 */
@WebServlet(name = "EJBClientServlet", urlPatterns = "/test.do")
@ServletSecurity(
    httpMethodConstraints =
    {
        @HttpMethodConstraint(value="GET", rolesAllowed = { "plainusers", "superusers" })
    })
public class EJBClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB()
    private StatelessSession1Bean mSessionBean1;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        System.out.println("**** EJBClientServlet.doGet");
        PrintWriter theResponseWriter = inResponse.getWriter();
        String theRequestNameParam = inRequest.getParameter("name");
        if (theRequestNameParam == null)
        {
            theRequestNameParam = "Anonymous";
        }
        String theResponse = mSessionBean1.greeting(theRequestNameParam);
        theResponseWriter.println("Response from the EJB:\n" + theResponse);
    }
}
```

Despite this book not focusing on servlet technology, we should note that:

- The class is annotated with the `@ServletSecurity` annotation.
This annotation tells the server that only users in the roles “plainusers” and “superusers” may issue GET requests to the servlet.
- The name of the roles are not the same as the name of the user groups used when creating users in the GlassFish application server.

11.3.4. Mapping Security Roles in GlassFish

The next step consists of mapping security roles in the application to users and/or user groups in the application server. This involves creating the product-specific web deployment descriptor for GlassFish v3.

- In the WebContent/WEB-INF directory create a file named sun-web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/EJBAnnotations</context-root>

    <!--
        The application's role "superusers" is mapped to the application
        server principal (user) "ivan".
        This gives the user "ivan" defined in the application server
        access to the parts of the application that requires the user
        to be in the "superusers" role.
    -->
    <security-role-mapping>
        <role-name>superusers</role-name>
        <principal-name>ivan</principal-name>
    </security-role-mapping>

    <!--
        The application's role "plainusers" is mapped to the application
        server's user group "plain-users"
        This gives all the users defined in the application server that
        belongs to the "plain-users" group access to the parts of the
        application that requires the user to be in the "plainusers" role.
    -->
    <security-role-mapping>
        <role-name>plainusers</role-name>
        <group-name>plain-users</group-name>
    </security-role-mapping>

    <security-role-mapping>
        <role-name>runasadmin</role-name>
        <principal-name>runas-superuser</principal-name>
    </security-role-mapping>

    <class-loader delegate="true" />
</sun-web-app>
```

Note that:

- This deployment descriptor is specific to GlassFish v3.
Other application containers use different ways of mapping roles used in the code to principals and user groups in the application server.
- For GlassFish v3, the way of mapping security roles to principals and/or user groups is the same for the sun-web.xml (web application), sun-ejb-jar.xml (EJB) and sun-application.xml (enterprise application) deployment descriptors.
- In the first <security-role-mapping> element, the user (principal) with the name “ivan” that we created earlier is mapped to the security role “superusers”.
This gives the user “ivan” access to parts of the application that requires the user to be in the “superusers” role.
- In the second <security-role-mapping> element, the user group “plain-users”, that we assigned to some users, is mapped to the security role “plainusers”.
This gives the users in the group “plain-users”, for instance the user “johnny”, access to parts of the application that requires the user to be in the “plainusers” role.

11.3.5. Running the Example Program – First Time

After having deployed the application to GlassFish, we are now ready to give it a first try:

- In a browser, navigate to the [http://localhost:8080/EJBSecurityAnnotations/test.do?
name=Ivan](http://localhost:8080/EJBSecurityAnnotations/test.do?name=Ivan) URL.
- When asked to log in, log in using the name “ivan” with the password “ivan”.

In the browser, a message similar to the following should appear:

```
Response from the EJB:  
Hello Ivan, StatelessSession1Bean, instance: 3. The time is now: Wed Nov 24 05:50:41 CET  
2010
```

In the console, you should see the following:

```
INFO: *** EJBClientServlet doGet  
INFO: *** StatelessSession1Bean greeting  
INFO: Principal name: ivan  
INFO: Principal object: ivan  
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl  
INFO: Caller in 'superusers' role? true  
INFO: Caller in 'plainusers' role? false  
INFO: Cannot determine caller role: 'ivan'
```

Note that:

- You were asked to log in before being able to view the webpage.
As you recall, the servlet only allows users in the roles “plainusers” or “superusers” to invoke it.
- From the console output above, we can see that the user “ivan” indeed is in the “superusers” role.
- The session bean invoked by the servlet also only allows users in the “plainusers” or “superusers” roles to invoke it.
Servlets with security configuration will display a dialog asking the user to provide security credentials, something an EJB with security configuration will not do.
As an exercise, remove the `@ServletSecurity` annotation from the servlet and re-run the program. This will cause an exception to be thrown when the servlet tries to invoke the session bean.
Thus we can conclude that the security principal was propagated from the servlet to the session bean.
- When the session bean is invoked, the caller is in the “superusers” role but not in the “plainusers” role.
This of course depends on the user credentials used when logging in.
As an exercise, clear the cookies and history information in the browser and log in using the user “johnny” and the password “johnny” and re-run the application. This user has a different security role.
- When the session bean is invoked, it cannot determine whether the caller is in a role named “ivan”. Invoking the `isCallerInRole` method with a role name that has not been declared using the `@DeclareRoles` annotation will cause an exception to be thrown.

11.3.6. Adding a Second Session Bean

To look further at principal propagation, we will introduce a second session bean which will be called from the first session bean.

- In the package `com.ivan.scbcd6.ejbs`, create the class `StatelessSession2Bean`:

```
package com.ivan.scbcd6.ejbs;

import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

/**
 * Stateless session bean with some declarative security specified using
 * annotations.
 */
@Stateless
@LocalBean
/*
 * The roles that are to be used from within the code when invoking
 * isCallerInRole need to be declared using the @DeclareRoles annotation,
 * otherwise an exception will be thrown when calling isCallerInRole.
 */
@DeclareRoles({"superusers", "plainusers"})
/*
 * When applied at class-level, the @RolesAllowed annotation specifies
 * which security-roles are allowed to access all the methods in the
 * EJB. @RolesAllowed may also be used at method-level.
 */
@RolesAllowed({"superusers", "plainusers", "runasadmin"})
public class StatelessSession2Bean extends CommonStatelessSessionBean
{
    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        System.out.println("**** StatelessSession2Bean.greeting");
        printSecurityInfo();

        try
        {
            Thread.sleep(2000L);
        } catch (InterruptedException theException)
        {
            /* Ignore exceptions. */
        }

        return assembleGreeting(inName, "StatelessSession2Bean");
    }

    /**
     * Returns a string.
     *
     * @return String.
     */
    @RolesAllowed("superusers")
    public String superusersOnly()
    {
        System.out.println("**** StatelessSession2Bean.superusersOnly");
        printSecurityInfo();
        return "Bingo!";
    }
}
```

Note that:

- As with the [first session bean](#), this class is also annotated with the `@DeclareRoles` annotation in which a list of security roles used with programmatic security is specified.
The code performing programmatic security is located in the superclass of the session bean.
- This session bean class is also annotated with the `@RolesAllowed` annotation in which a list of security roles is supplied.
This annotation specifies the default security restrictions for all the methods of the session bean.
- The method `superusersOnly` is also annotated with the `@RolesAllowed` annotation, specifying only the “superusers” role.
This annotation overrides the security restrictions that the `@RolesAllowed` annotation specified for all the methods of the class and only allows users belonging to the “superusers” security group to access the method.

We also need to modify the first session bean, so that invokes the second session bean:

- Modify the implementation of the `StatelessSession1Bean` class to look like this:

```
package com.ivan.scbcd6.ejb;

import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
import javax.ejb.EJB;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

/**
 * Secured stateless session bean exposing a local, no-interface view.
 */
@Stateless
@LocalBean
/*
 * The roles that are to be used from within the code when invoking
 * isCallerInRole need to be declared using the @DeclareRoles annotation,
 * otherwise an exception will be thrown when calling isCallerInRole.
 */
@DeclareRoles(
{ "superusers", "plainusers" })
/*
 * When applied at class-level, the @RolesAllowed annotation specifies
 * which security-roles are allowed to access all the methods in the
 * EJB. @RolesAllowed may also be used at method-level.
 */
@RolesAllowed({ "superusers", "plainusers" })
public class StatelessSession1Bean extends CommonStatelessSessionBean
{
    @EJB
    private StatelessSession2Bean mSessionBean2;

    /**
     * Creates a greeting to the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    public String greeting(final String inName)
    {
        System.out.println("**** StatelessSession1Bean.greeting");
        printSecurityInfo();

        try
        {
            System.out.println("    Message for the superuser: " +
                mSessionBean2.superusersOnly());
        } catch (Throwable theException)
        {
            System.out.println("    No message for the superuser.");
        }
    }

    String theBean2Greeting = mSessionBean2.greeting(inName);
}
```

```

        String theMessage = assembleGreeting(inName, "StatelessSession1Bean")
        + "\n" + theBean2Greeting;
    return theMessage;
}

```

Note that:

- The class now contains the instance field `mSessionBean2`, annotated with the `@EJB` annotation.
This field holds a reference to the second session bean.
- The `greeting` method tries to invoke the `superusersOnly` method on the second session bean. A certain message will be printed if the invocation succeeds, another will be printed if the invocation fails.
- The `greeting` method also invokes the `greeting` method on the second session bean and appends the result to the message it produces.

11.3.7. Running the Example Program – Second Time

We are now ready to try the new version of the EJB security example program:

- Deploy the new version of the application to the GlassFish v3 server.
- Clear cookies and cached data in the browser used to access the example program.
- In the browser, navigate to the <http://localhost:8080/EJBSecurityAnnotations/test.do?name=Ivan> URL.
- When asked to log in, log in using the name “johnny” with the password “johnny”.

In the browser, a message similar to the following should appear:

```

Response from the EJB:
Hello Ivan, StatelessSession1Bean, instance: 3. The time is now: Wed Nov 24 17:12:50 CET
2010
Hello Ivan, StatelessSession2Bean, instance: 4. The time is now: Wed Nov 24 17:12:50 CET
2010

```

In the console, you should see the following (exception stacktrace edited to conserve space):

```

INFO: *** EJBClientServlet.doGet
INFO: *** StatelessSession1Bean.greeting
INFO: Principal name: johnny
INFO: Principal object: johnny
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? true
INFO: Cannot determine caller role: 'ivan'
INFO: JACC Policy Provider: Failed Permission Check,
context(EJBAnnotations/EJBAnnotations_internal)-
permission((javax.security.jacc.EJBMethodPermission StatelessSession2Bean
superusersOnly,Local, ))
WARNING: A system exception occurred during an invocation on EJB StatelessSession2Bean
method public java.lang.String
com.ivan.scbcd6.ejbs.StatelessSession2Bean.superusersOnly()
javax.ejb.AccessLocalException: Client not authorized for this invocation.
    at com.sun.ejb.containers.BaseContainer.preInvoke(BaseContainer.java:1850)
    at
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.invoke(EJBLocalObjectInvocationHa
ndler.java:188)
    at
com.sun.ejb.containers.EJBLocalObjectInvocationHandlerDelegate.invoke(EJBLocalObjectInvo
cationHandlerDelegate.java:84)
    at $Proxy220.superusersOnly(Unknown Source)
    at
com.ivan.scbcd6.ejbs._EJB31_Generated__StatelessSession2Bean__Intf____Bean__.superusers
Only(Unknown Source)

```

```

at
com.ivan.scbcd6.ejb.StatelessSession1Bean.greeting(StatelessSession1Bean.java:47)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at
org.glassfish.ejb.security.application.EJBSecurityManager.runMethod(EJBSecurityManager.java:1056)
at
org.glassfish.ejb.security.application.EJBSecurityManager.invoke(EJBSecurityManager.java:1128)
at com.sun.ejb.containers.BaseContainer.invokeBeanMethod(BaseContainer.java:5292)
at com.sun.ejb.EjbInvocation.invokeBeanMethod(EjbInvocation.java:615)
at
com.sun.ejb.containers.interceptors.AroundInvokeChainImpl.invokeNext(InterceptorManager.java:797)
at com.sun.ejb.EjbInvocation.proceed(EjbInvocation.java:567)
at
com.sun.ejb.containers.interceptors.SystemInterceptorProxy.doAround(SystemInterceptorProxy.java:157)
at
com.sun.ejb.containers.interceptors.SystemInterceptorProxy.aroundInvoke(SystemInterceptorProxy.java:139)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at
com.sun.ejb.containers.interceptors.AroundInvokeInterceptor.intercept(InterceptorManager.java:858)
at
com.sun.ejb.containers.interceptors.AroundInvokeChainImpl.invokeNext(InterceptorManager.java:797)
at
com.sun.ejb.containers.interceptors.InterceptorManager.intercept(InterceptorManager.java:367)
at com.sun.ejb.containers.BaseContainer._intercept(BaseContainer.java:5264)
at com.sun.ejb.containers.BaseContainer.intercept(BaseContainer.java:5252)
at
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.invoke(EJBLocalObjectInvocationHandler.java:190)
at
com.sun.ejb.containers.EJBLocalObjectInvocationHandlerDelegate.invoke(EJBLocalObjectInvocationHandlerDelegate.java:84)
at $Proxy221.greeting(Unknown Source)
at
com.ivan.scbcd6.ejb._EJB31_Generated__StatelessSession1Bean__Intf____Bean__.greeting(Unknown Source)
at com.ivan.scbcd6.client.EJBClientServlet doGet(EJBClientServlet.java:54)
...
INFO: No message for the superuser.
INFO: *** StatelessSession2Bean.greeting
INFO: Principal name: johnny
INFO: Principal object: johnny
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? true
INFO: Cannot determine caller role: 'ivan'

```

In addition to what we saw when running the first version of the example program, note that:

- We can see that the user “johnny” is in the “plainusers” role and not in the “superusers” role.
- The first session bean, *StatelessSession1Bean*, did not succeed in invoking the *superusersOnly* method on the second session bean, *StatelessSession2Bean*. The *superusersOnly* method may only be invoked by users in the “superusers” role, but “johnny” is in the “plainusers” role and the invocation thus fails. The resulting exception was caught and a stacktrace was printed to the console.
- Curiously, we can see that the container uses interceptors to apply security restrictions to EJBs (indications highlighted with green in the stacktrace).
- The first session bean succeeds in invoking the *greeting* method on the second session bean. This since users in the “plainusers” role are allowed to invoke this method. The messages from both the session beans are output in the browser.

11.3.8. Changing the Invocation Security Role

In this section we will modify the example program so that the second session bean is invoked using a fixed security role, which may or may not be the same as the original security role. More details about how to change the security role an EJB uses when invoking other EJBs can be found in the section on Declarative Security later in this chapter.

- Add the `@RunAs` annotation to the *StatelessSession1Bean* class, as shown in the listing below.

```
...
@RolesAllowed({ "superusers", "plainusers" })
@RunAs("runasadmin")
public class StatelessSession1Bean extends CommonStatelessSessionBean
{
...
}
```

- Deploy the new version of the application to the GlassFish v3 server.
- Clear cookies and cached data in the browser used to access the example program.
- In the browser, navigate to the <http://localhost:8080/EJBSecurityAnnotations/test.do?name=Ivan> URL.
- When asked to log in, log in using the name “johnny” with the password “johnny”.
- The message in the browser should be similar to what we have seen earlier.
- In the console, you should see the following (exception stacktrace edited to conserve space):

```
WARNING: Web login failed: Login failed: javax.security.auth.login.LoginException:
Failed file login for johnny.

INFO: *** EJBClientServlet.doGet
INFO: *** StatelessSession1Bean.greeting
INFO: Principal name: johnny
INFO: Principal object: johnny
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? true
INFO: Cannot determine caller role: 'ivan'
INFO: JACC Policy Provider: Failed Permission Check,
context(EJBAnnotations/EJBAnnotations_internal)-
permission((javax.security.jacc.EJBMethodPermission StatelessSession2Bean
superusersOnly,Local,))

WARNING: A system exception occurred during an invocation on EJB StatelessSession2Bean
method public java.lang.String
```

```

com.ivan.scbcd6.ejbs.StatelessSession2Bean.superusersOnly()
javax.ejb.AccessLocalException: Client not authorized for this invocation.
    at com.sun.ejb.containers.BaseContainer.preInvoke(BaseContainer.java:1850)
    ...
INFO:      No message for the superuser.
INFO: *** StatelessSession2Bean.greeting
INFO: Principal name: runas-superuser
INFO: Principal object: runas-superuser
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? false
INFO: Cannot determine caller role: 'ivan'

```

Note that:

- When executing in the *StatelessSession1Bean*, the name of the principal is “johnny” and the caller is in the security role “plainusers”.
- The first session bean, *StatelessSession1Bean*, did not succeed in invoking the *superusersOnly* method on the second session bean, *StatelessSession2Bean*. This is not entirely surprising, as the caller is in the role “plainusers” when executing in the first session bean.
- When executing in the *StatelessSession2Bean*, the name of the principal has changed to “runas-superuser” and the caller is neither in the security role “superusers” nor in the role “plainusers”.
- Remember that we configured a user named “runas-superuser” in the GlassFish server which belongs to the “super-users” group. So despite the “runas-superuser” belonging to the same group as the user “ivan”, running with the former principal still does not allow us to invoke the *superusersOnly* method on the *StatelessSession2Bean*. This is because the “runas-superuser” is mapped to another security role, the “runasadmin” role.

If we want to allow the “runas-superuser” principal to be able to execute the *superusersOnly* method, we can either map the “runas-superuser” principal to the “superusers” role in the sun-web.xml deployment descriptor or modify the annotation on the *superusersOnly* method to include the “runasadmin” role too.

- Apply one of the above suggested ways to enable the “runas-superuser” principal to execute the *superuserOnly* method.
- Deploy the new version of the example program to the GlassFish server.
- Clear cookies and cached data in the browser used to access the example program.
- In the browser, navigate to the <http://localhost:8080/EJBSecurityAnnotations/test.do?name=Ivan> URL.
- When asked to log in, log in using the name “johnny” with the password “johnny”.
- The message in the browser should be similar to what we have seen earlier.

(continued on next page)

- In the console, you should see the following:

```

INFO: *** EJBClientServlet.doGet
INFO: *** StatelessSession1Bean.greeting
INFO: Principal name: johnny
INFO: Principal object: johnny
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? true
INFO: Cannot determine caller role: 'ivan'
INFO: *** StatelessSession2Bean.superusersOnly
INFO: Principal name: runas-superuser
INFO: Principal object: runas-superuser
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? false
INFO: Cannot determine caller role: 'ivan'
INFO: Message for the superuser: Bingo!
INFO: *** StatelessSession2Bean.greeting
INFO: Principal name: runas-superuser
INFO: Principal object: runas-superuser
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? false
INFO: Cannot determine caller role: 'ivan'

```

Note that:

- Again, the principal name when executing in the *StatelessSession1Bean* is “johnny” and the user is in the security role “plainusers”.
- The *StatelessSession1Bean* now succeeds in invoking the *superuserOnly* method on the *StatelessSession2Bean*.
- When executing in the *superuserOnly* method, the principal name is “runas-superuser” and the user is still not in any of the roles “superusers” or “plainusers”.
This will give you a hint about the solution I used to allow execution of the *superusersOnly* method, which was by adding the role “runasadmin” to the *@RolesAllowed* annotation on the method.

This concludes the EJB security example that uses annotations. In the next section we will look at how to modify the example program and replace all annotations with configuration in the ejb-jar.xml deployment descriptor.

11.4. Deployment Descriptor EJB Security Example

When creating the example program that only uses the ejb-jar.xml deployment descriptor for configuration of the EJBs, you can either choose to modify the example program above or create a new one. I choose to create a new example program. If you modify the existing project, then skip the two first steps below.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “EJBSecurityDeploymentDescriptor”.
- Copy all the classes and the sun-web.xml deployment descriptor from the previous example program to the new project.
- Remove all the annotations from the three EJB-related classes *CommonStatelessSessionBean*, *StatelessSession1Bean*, *StatelessSession2Bean*.
Note: Do not remove the annotations in the servlet class!
- In the WebContent/WEB-INF directory, create the file ejb-jar.xml with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatelessSession2Bean</ejb-name>
            <local-bean/>
            <ejb-class>com.ivan.scbcd6.ejbs.StatelessSession2Bean</ejb-class>
            <session-type>Stateless</session-type>

            <!--
                Inject the session context.
                Note that the injection target class is the class in which
                the instance variable is declared, which is the superclass
                of the EJB implementation class.
            -->
            <resource-env-ref>
                <resource-env-ref-name>java:comp/EJBContext</resource-env-ref-name>
                <resource-env-ref-type>javax.ejb.EJBContext</resource-env-ref-type>
                <injection-target>
                    <injection-target-class>
                        com.ivan.scbcd6.ejbs.CommonStatelessSessionBean
                    </injection-target-class>
                    <injection-target-name>mSessionContext</injection-target-name>
                </injection-target>
            </resource-env-ref>

            <!--
                Defines a security role used in the code of the EJB, for
                instance when invoking isCallerInRole, and maps the
                role name to a security role defined in the
                <assembly-descriptor>.
                The security role mapping is local to the component it
                is defined in, in this case the StatelessSession2Bean EJB.
            -->
            <security-role-ref>
                <!--
                    The <role-name> element specifies the rolename
                    used for programmatic security, that is, calling
                    isCallerInRole.
                -->
                <role-name>superusers</role-name>
                <!--
                    Links the above role name to a security role defined
                    in a <security-role> element in the <assembly-descriptor>
                    below.
                -->
            </security-role-ref>
        </session>
    </enterprise-beans>

```

```

        <role-link>superusers</role-link>
    </security-role-ref>
    <security-role-ref>
        <role-name>plainusers</role-name>
        <role-link>plainusers</role-link>
    </security-role-ref>
</session>

<session>
    <ejb-name>StatelessSession1Bean</ejb-name>
    <local-bean/>
    <ejb-class>com.ivan.scbcd6.ejbs.StatelessSession1Bean</ejb-class>
    <session-type>Stateless</session-type>

    <!--
        Inject a reference to the second session bean.
    -->
    <ejb-local-ref>
        <ejb-ref-name>StatelessSession2Bean</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <injection-target>
            <injection-target-class>
                com.ivan.scbcd6.ejbs.StatelessSession1Bean
            </injection-target-class>
            <injection-target-name>mSessionBean2</injection-target-name>
        </injection-target>
    </ejb-local-ref>

    <!--
        Inject the session context.
    -->
    <resource-env-ref>
        <resource-env-ref-name>java:comp/EJBContext</resource-env-ref-name>
        <resource-env-ref-type>javax.ejb.EJBContext</resource-env-ref-type>
        <injection-target>
            <injection-target-class>
                com.ivan.scbcd6.ejbs.CommonStatelessSessionBean
            </injection-target-class>
            <injection-target-name>mSessionContext</injection-target-name>
        </injection-target>
    </resource-env-ref>

    <security-role-ref>
        <role-name>superusers</role-name>
        <role-link>superusers</role-link>
    </security-role-ref>
    <security-role-ref>
        <role-name>plainusers</role-name>
        <role-link>plainusers</role-link>
    </security-role-ref>
    <security-role-ref>
        <role-name>runasadmin</role-name>
        <role-link>runasadmin</role-link>
    </security-role-ref>

    <!--
        Specifies the security identity the EJB will use when
        calling other EJBs and Java EE components.
        Corresponds to the @RunAs annotation.
    -->
    <security-identity>
        <run-as>
            <role-name>runasadmin</role-name>
        </run-as>
    </security-identity>
</session>
</enterprise-beans>

<assembly-descriptor>
    <!--
        Defines the security roles used in the EJB module.
        These roles are mapped to application server principals or
        user groups in the application server specific deployment
        descriptor - for GlassFish this is sun-ejb-jar.xml.
    -->
    <security-role>
        <role-name>superusers</role-name>

```

```

</security-role>
<security-role>
    <role-name>plainusers</role-name>
</security-role>
<security-role>
    <role-name>runasadmin</role-name>
</security-role>

<!--
    Allow access to all of the methods in the StatelessSession1Bean
    for users in the superusers and plainusers security roles.
-->
<method-permission>
    <role-name>superusers</role-name>
    <role-name>plainusers</role-name>
    <method>
        <ejb-name>StatelessSession1Bean</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>

<!--
    Allow access to the greeting method in the StatelessSession2Bean
    for users in the superusers, plainusers and runasadmin
    security roles.
-->
<method-permission>
    <role-name>superusers</role-name>
    <role-name>plainusers</role-name>
    <role-name>runasadmin</role-name>
    <method>
        <ejb-name>StatelessSession2Bean</ejb-name>
        <method-name>greeting</method-name>
    </method>
</method-permission>

<!--
    Allow access to the superusersOnly method in the
    StatelessSession2Bean for users in the superusers and
    runasadmin security roles.
-->
<method-permission>
    <role-name>superusers</role-name>
    <role-name>runasadmin</role-name>
    <method>
        <ejb-name>StatelessSession2Bean</ejb-name>
        <method-name>superusersOnly</method-name>
    </method>
</method-permission>
</assembly-descriptor>
</ejb-jar>

```

Note that:

- There are two EJBs defined in the deployment descriptor; StatelessSession1Bean and StatelessSession2Bean.
As the names hints, both these EJBs are stateless session beans. Both EJBs have a local, no-interface, view.
- An EJB context is injected in both the session beans using a [<resource-env-ref> element](#).
Also note that the injection target class is the common session bean superclass, since the instance field is declared in this class.
- A reference to the StatelessSession2Bean is injected into the StatelessSession1Bean using a [<ejb-local-ref> element](#).
- In each of the session beans, there are a number of [<security-role-ref>](#) elements.
Such an element declares a mapping between a security role name used in the code of the EJB in which the element is declared (in the [<role-name>](#) element) and a security role name in the application (in the [<role-link>](#) element). The former security role name can only be used within the EJB in which it is declared. The latter security role name is global to the

application.

These mappings are only required for programmatic security.

- In the StatelessSession1Bean, there is a <security-identity> element. This element contains a <run-as> element, which in turn contains a <role-name> element. This construction specifies the security role that will be used by all methods of the EJB in which it appears when calling other EJBs and Java EE components. This corresponds to the @RunAs annotation.
- In the <assembly-descriptor> element, there are a number of <security-role> elements. Each of these elements declare a security role that can be used in the application. These security roles must be mapped to users (principals) and/or user groups in a server-specific deployment descriptor. In our case, these mappings are done in the sun-web.xml deployment descriptor, as described [earlier](#).
- In the <assembly-descriptor> element, there are three <method-permission> elements. Each of these elements declare the security role(s) that are allowed to execute certain method(s) on an EJB.
- The first <method-permission> element allows the security roles “superusers” and “plainusers” to execute all methods in the StatelessSession1Bean.
- The second <method-permission> element allows the security roles “superusers”, “plainusers” and “runasadmin” to execute the *greeting* method in the StatelessSession2Bean.
- Finally, the third <method-permission> element allows the security roles “superusers” and “plainusers” to execute the *superuserOnly* method in the StatelessSession2Bean.

With the ejb-jar.xml deployment descriptor in place, we are done and can try the example out.

- Deploy the of the example program to the GlassFish server.
- Clear cookies and cached data in the browser used to access the example program.
- In the browser, navigate to URL
<http://localhost:8080/EJBSecurityDeploymentDescriptor/test.do?name=Ivan>.
Note that the URL may differ depending on the name of the project in which the example was developed.
- When asked to log in, log in using the name “johnny” with the password “johnny”.

In the browser, a message similar to the following should appear:

```
Response from the EJB:  
Hello Ivan, StatelessSession1Bean, instance: 3. The time is now: Mon Nov 29 20:13:49 CET  
2010  
Hello Ivan, StatelessSession2Bean, instance: 4. The time is now: Mon Nov 29 20:13:49 CET  
2010
```

In the console, you should see the following:

```
INFO: *** EJBClientServlet.doGet
INFO: *** StatelessSession1Bean.greeting
INFO: Principal name: johnny
INFO: Principal object: johnny
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? true
INFO: Cannot determine caller role: 'ivan'
INFO: *** StatelessSession2Bean.superusersOnly
INFO: Principal name: runas-superuser
INFO: Principal object: runas-superuser
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? false
INFO: Cannot determine caller role: 'ivan'
INFO: Message for the superuser: Bingo!
INFO: *** StatelessSession2Bean.greeting
INFO: Principal name: runas-superuser
INFO: Principal object: runas-superuser
INFO: Principal type: class org.glassfish.security.common.PrincipalImpl
INFO: Caller in 'superusers' role? false
INFO: Caller in 'plainusers' role? false
INFO: Cannot determine caller role: 'ivan'
```

Not surprisingly, this is the same result as we saw when using annotations to configure EJB security. In the rest of this chapter we will look at the different options available in connection to EJB security.

11.5. Programmatic Security

Using programmatic security in EJBs is discouraged and has the following drawbacks:

- Mixes security infrastructure and business logic code in the EJB class.
- Requires code change to change security behaviour.
- Risk of reducing portability.

If programmatic security must be used, a better alternative is place programmatic security checks in an interceptor and thus separating it from the business logic code.

There are two security-related methods in the *EJBContext* interface that are not deprecated:

Method Name	Description
getCallerPrincipal()	Retrieves a principal object that identifies the caller. From the principal object the name of the principal can be retrieved and the principal object can also be compared with other principal objects for equality. This method always return an object and never return null.
isCallerInRole(String)	Tests if the caller is in the supplied security role.

The following method from the security examples in the beginning of this chapter shows use of the above security-related methods in the *EJBContext* interface:

```
...
protected void printSecurityInfo()
{
    if (mSessionContext != null)
    {
        /*
         * Container may never return a null principal, so we don't
         * need to make sure it is not null.
         */
        Principal theCallerPrincipal = mSessionContext.getCallerPrincipal();
        System.out.println("    Principal name: " +
                           theCallerPrincipal.getName());
        System.out.println("    Principal object: " + theCallerPrincipal);
        System.out.println("    Principal type: " + theCallerPrincipal.getClass());

        testCallerRole("superusers");
        testCallerRole("plainusers");
        testCallerRole("ivan");
    } else
    {
        System.out.println("    No session context available.");
    }
}

private void testCallerRole(String inRoleName)
{
    try
    {
        System.out.println("    Caller in '" + inRoleName + "' role? " +
                           mSessionContext.isCallerInRole(inRoleName));
    } catch (Throwable theException)
    {
        /* Catch exceptions thrown, for instance if role has not been declared. */
        System.out.println("    Cannot determine caller role: '" +
                           inRoleName + "'");
    }
}
```

```
}
```

```
...
```

Note that an *EJBContext* is not available at all occasions in the life-cycle of an EJB.

Programmatic security in EJBs is not purely programmatic, as we have seen. Names of roles used when calling *isCallerInRole* must be declared either using the *@DeclareRoles* annotation or the *<security-role-ref>* element in the ejb-jar.xml deployment descriptor.

The following code snippet shows how security role references are declared using the *@DeclareRoles* annotation:

```
...
@Stateless
@LocalBean
/*
 * The roles that are to be used from within the code when invoking
 * isCallerInRole need to be declared using the @DeclareRoles annotation,
 * otherwise an exception will be thrown when calling isCallerInRole.
 */
@DeclareRoles({ "superusers", "plainusers" })
/*
 * When applied at class-level, the @RolesAllowed annotation specifies
 * which security-roles are allowed to access all the methods in the
 * EJB. @RolesAllowed may also be used at method-level.
 */
@RolesAllowed({ "superusers", "plainusers" })
@RunAs("runasadmin")
public class StatelessSession1Bean extends CommonStatelessSessionBean
...
```

The following excerpt shows how the same roles declared using the annotation in the example above may be declared in the ejb-jar.xml deployment descriptor using two *<security-role-ref>* elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatelessSession2Bean</ejb-name>
            <local-bean/>
            <ejb-class>com.ivan.scbcd6.ejbs.StatelessSession2Bean</ejb-class>
            <session-type>Stateless</session-type>

            <resource-env-ref>
                <resource-env-ref-name>java:comp/EJBContext</resource-env-ref-name>
                <resource-env-ref-type>javax.ejb.EJBContext</resource-env-ref-type>
                <injection-target>
                    <injection-target-class>
                        com.ivan.scbcd6.ejbs.CommonStatelessSessionBean
                    </injection-target-class>
                    <injection-target-name>mSessionContext</injection-target-name>
                </injection-target>
            </resource-env-ref>

            <!--
                Defines a security role used in the code of the EJB, for
                instance when invoking isCallerInRole, and maps the
                role name to a security role defined in the
                <assembly-descriptor>.
                The security role mapping is local to the component it
                is defined in, in this case the StatelessSession2Bean EJB.
            -->
            <security-role-ref>
                <!--
                    The <role-name> element specifies the rolename
                    used for programmatic security, that is, calling
                -->
            </security-role-ref>
        </session>
    </enterprise-beans>

```

```

        isCallerInRole.
-->
<role-name>superusers</role-name>
<!--
    Links the above role name to a security role defined
    in a <security-role> element in the <assembly-descriptor>
    below.
-->
<role-link>superusers</role-link>
</security-role-ref>
<security-role-ref>
    <role-name>plainusers</role-name>
    <role-link>plainusers</role-link>
</security-role-ref>
</session>
...

```

Note that:

- In addition to the role name, it is also possible to declare a role link in a `<security-role-ref>` element.
- The `<role-name>` element in the `<security-role-ref>` element specifies the role-name that may be used in the code of the EJB, when calling `isCallerInRole`.
This security role name is component-local and not available outside of the EJB in which the `<security-role-ref>` element is declared.
- The `<role-link>` element links the component-local security role name specified in the `<role-name>` element to an application global security role name declared in a `<security-role>` element in the `<assembly-descriptor>` element.

Please refer to the example programs in the beginning of this chapter for the complete code and deployment descriptor!

11.6. Declarative Security

Declarative security is the preferred way of configuring security for EJBs. Security configuration can be accomplished using either annotations or the ejb-jar.xml deployment descriptor. We have already seen examples of declarative security using annotations and the deployment descriptor. In this section we will look more closely into the options available.

11.6.1. Declarative Security Using Annotations

The security-related annotations used with EJBs are all common Java EE security annotations. The comments below applies foremost to EJBs.

Annotation	Comments
DeclareRoles	Declared on class-level to specify the security roles available for programmatic security.
DenyAll	Declared on class- or method-level to specify that no security roles may invoke the methods of the class or the single annotated method.
PermitAll	Declared on class- or method-level to specify that all security roles may invoke the methods of the class or the single annotated method. @PermitAll on method-level overrides a class-level @RolesAllowed.
RolesAllowed	Declared on class- or method-level to specify the security roles that may invoke the methods of the class or the single annotated method. @RolesAllowed on method-level overrides a class-level @RolesAllowed.
RunAs	Declared on class-level to specify the security role the EJB is to use when invoking other Java EE components.

The following rules apply regarding inheritance of security roles declared using the @RolesAllowed annotation:

- A class-level @RolesAllowed annotation in the superclass applies to the (inherited) methods defined in the superclass.
- A method-level @RolesAllowed annotation in the superclass applies to the (inherited) method which it annotates.
- A method defined in a subclass does not inherit security roles from any @RolesAllowed annotation defined on the superclass or a method in the superclass.
- A method-level @RolesAllowed annotation overrides a class-level @RolesAllowed annotation.

Example: Given the following EJB and its superclass:

```
@RolesAllowed({ "superusers" })
public class CommonStatelessSessionBean
{
    public String doGreeting(String inName) { ... }
    public String doFarewell(String inName) { ... }
}

@Stateless
public class StatelessSession1Bean extends CommonStatelessSessionBean
{
    public String doIntroduction(String inName) { ... }

    @RolesAllowed({ "plainusers" })
    public String doPresentation(String inName, String inFriendName) { ... }
}
```

The methods in the *StatelessSession1Bean* can be invoked by users in the following roles:

- *doGreeting* can be invoked by users in the superusers role.
- *doFarewell* can be invoked by users in the superusers role.
- *doIntroduction* can be invoked by users in any role.

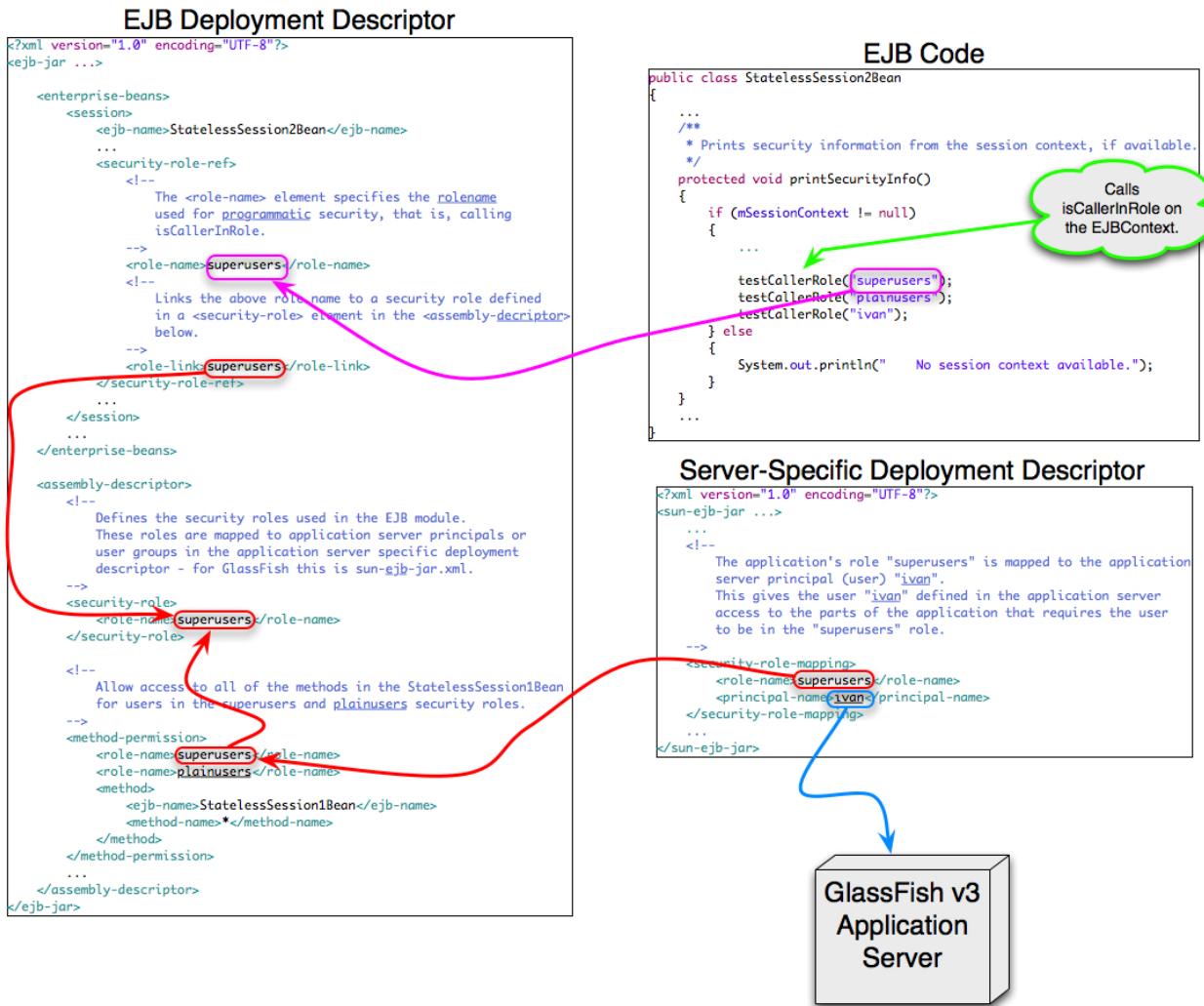
There is no security role specified for the method in question, so the container is to treat the method as an unchecked method. (EJB 3.1 Specification, section 17.3.2.3 and 17.3.2.2).

- *doPresentation* can be invoked by users in the plainusers role.

11.6.2. Declarative Security Using the Deployment Descriptor

As with earlier versions of EJB and as we have seen in one of the examples opening this chapter, EJB security can be configured in deployment descriptors, without use of annotations.

The following figure shows the relationships between an EJB implementation class using programmatic and declarative security, the ejb-jar.xml deployment descriptor and the application server-specific deployment descriptor:

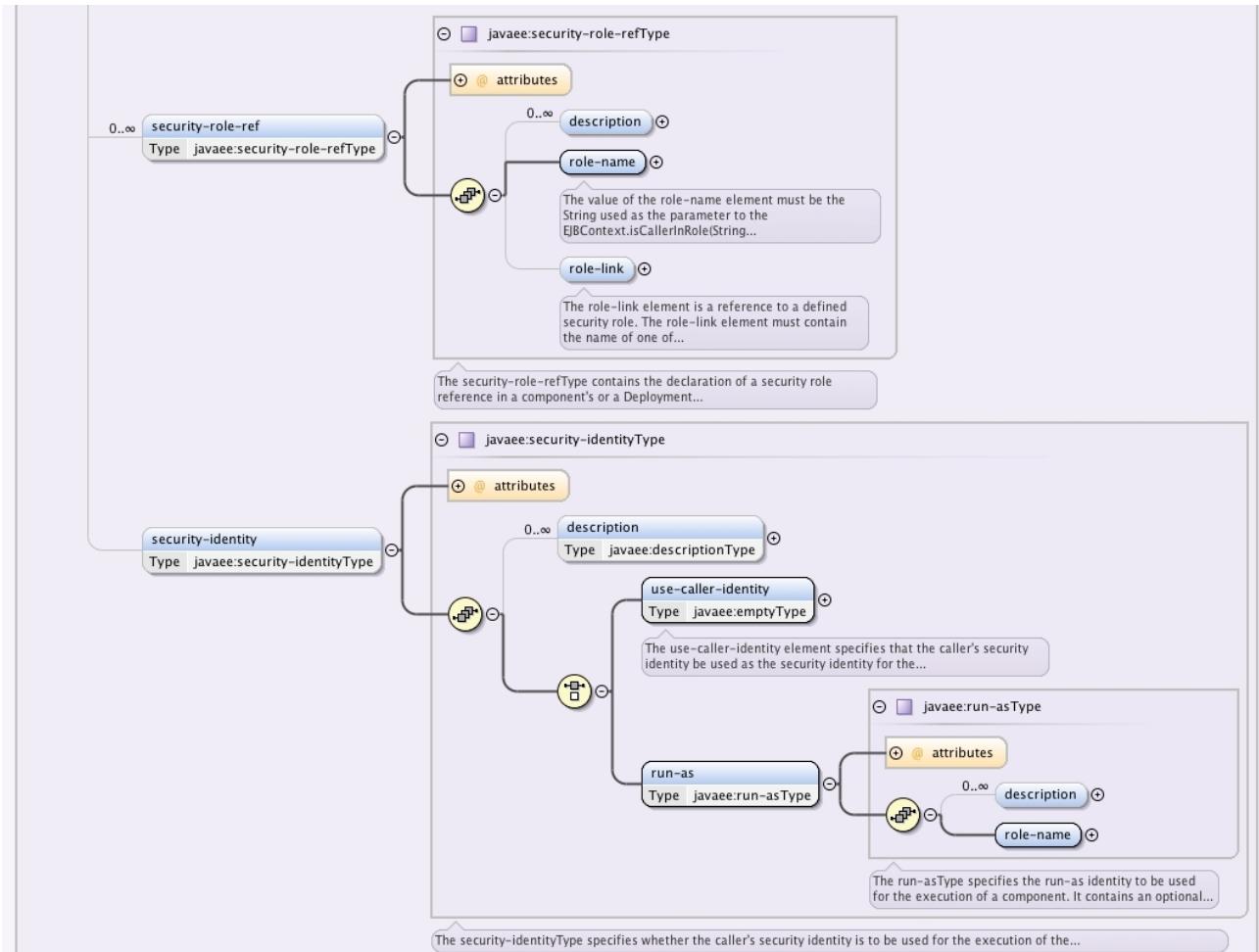


An EJB using programmatic and declarative security and its security-related artifacts.

There are two sections in the ejb-jar.xml deployment descriptor that are related to security; one is component-local and the other is EJB-module global.

Component-Local Security in the Deployment Descriptor

The component-local security section declares component-local security role reference declarations and security-identity declarations. The following XML schema fragment shows the parts of the ejb-jar.xml deployment descriptor containing component-local security declarations:



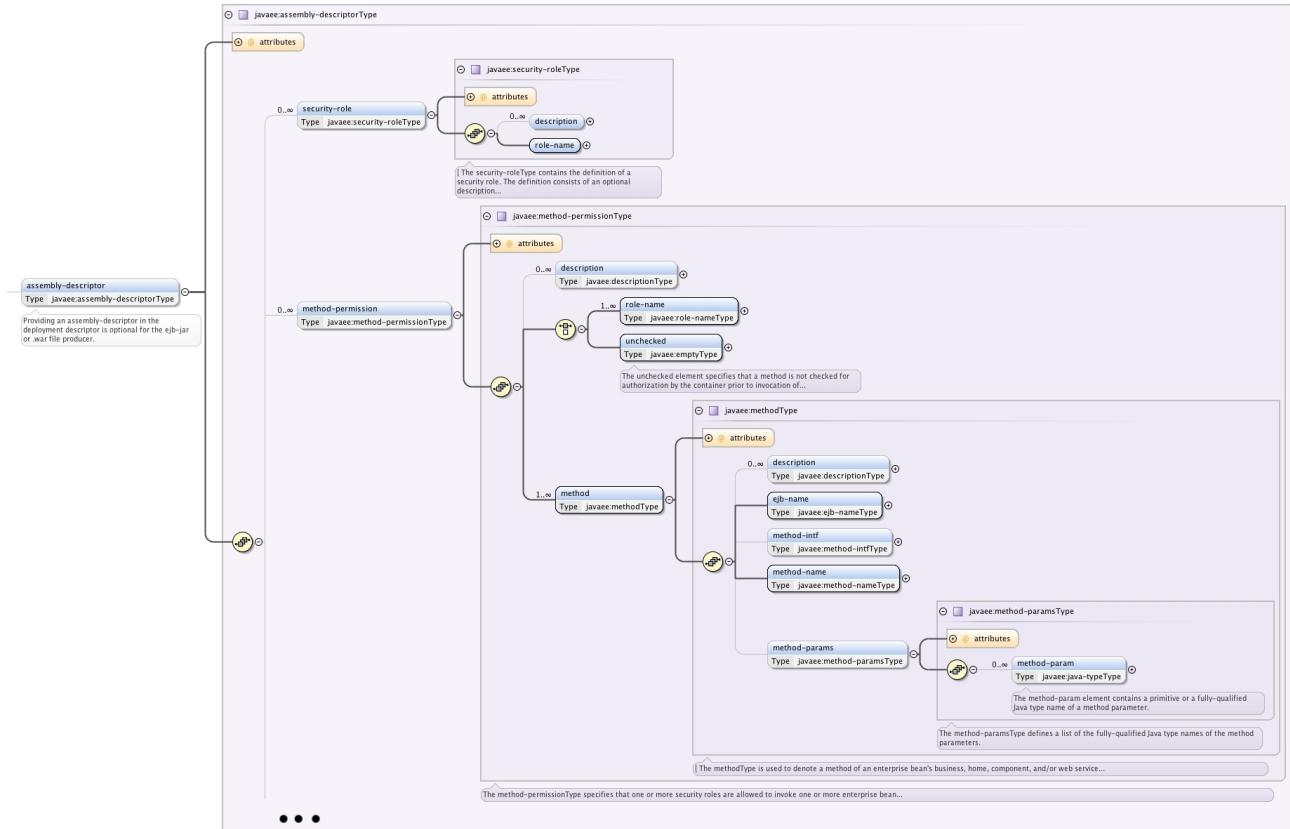
Both `<security-role-ref>` elements and `<security-identity>` elements are immediate child elements of elements containing EJB declarations, for instance `<session>` for session beans.

A `<security-role-ref>` element contains declaration of a security role name used with programmatic security when invoking `EJBContext.isCallerInRole`. The `<role-name>` element specifies the role name used as parameter to `isCallerInRole` method. The `<role-link>` element links the component-local role name specified in the `<role-name>` element to an EJB-module global role name defined in the `<assembly-descriptor>`. If the name of the security role in the `<role-name>` element is the same as the EJB-module global security role, then the `<role-link>` element need not be defined.

A `<security-identity>` element contains a specification of which security role is to be used when the EJB invokes other EJBs and Java EE components. If the `<use-caller-identity>` element is used, then the caller's security role will be passed on. If the `<run-as>` element is used, the security role with the name specified in the `<role-name>` element in the `<run-as>` will be used.

EJB-Module Security in the Deployment Descriptor

The EJB-module security section defines security roles used in the EJB module. The following XML schema fragment shows the parts of the ejb-jar.xml deployment descriptor containing EJB module security declarations:



Elements of the ejb-jar.xml deployment descriptor containing EJB-module global security declarations.

One or more `<security-role>` elements in the `<assembly-descriptor>` element defines the (logical) security roles used in the EJB module. These security roles are later mapped to principals and/or user groups in the application-server specific deployment descriptor, as we saw an example of in the [first security example program above](#).

(continued on next side)

One or more <method-permission> elements assigns one or more security roles to one or more methods of an EJB in the EJB module. A user in any of the security roles assigned to a method is allowed to invoke the method. The following are examples of how to specify method permissions in the ejb-jar.xml deployment descriptor:

- Using a wildcard method name.

This causes the method permission declaration to apply to all the methods in the specified EJB. Example:

```
...
<!--
   Allow access to all of the methods in the StatelessSession1Bean
   for users in the superusers and plainusers security roles.
-->
<method-permission>
  <role-name>superusers</role-name>
  <role-name>plainusers</role-name>
  <method>
    <ejb-name>StatelessSession1Bean</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
...
```

- Specify a method name.

The method permission declaration applies to all the methods with the specified name in the specified EJB. Example:

```
...
<!--
   Allow access to the greeting method in the StatelessSession2Bean
   for users in the superusers, plainusers and runasadmin
   security roles.
-->
<method-permission>
  <role-name>superusers</role-name>
  <role-name>plainusers</role-name>
  <role-name>runasadmin</role-name>
  <method>
    <ejb-name>StatelessSession2Bean</ejb-name>
    <method-name>greeting</method-name>
  </method>
</method-permission>
...
```

- Specify a method name and the parameters.

The method permission declaration applies to all methods with the specified name taking the specified parameters. Example:

```
...
<!--
    Allow access to the greeting method that takes a String parameter
    in the StatelessSession2Bean for users in the superusers, plainusers
    and runasadmin security roles.
-->
<method-permission>
    <role-name>superusers</role-name>
    <role-name>plainusers</role-name>
    <role-name>runasadmin</role-name>
    <method>
        <ejb-name>StatelessSession2Bean</ejb-name>
        <method-name>greeting</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </method>
</method-permission>
...

```

- Specify the EJB interface in which the method with the specified name appears.

The following interface types may be used:

Home, Remote, LocalHome, Local, ServiceEndpoint, Timer, MessageEndpoint

In addition, method parameter types may also be specified.

Example:

```
...
<!--
    Allow access to the greeting method that takes a String parameter
    in the local view of the StatelessSession2Bean
    for users in the superusers, plainusers and runasadmin
    security roles.
-->
<method-permission>
    <role-name>superusers</role-name>
    <role-name>plainusers</role-name>
    <role-name>runasadmin</role-name>
    <method>
        <ejb-name>StatelessSession2Bean</ejb-name>
        <method-intf>Local</method-intf>
        <method-name>greeting</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </method>
</method-permission>
...

```

12. Schedule Execution of Business Logic

References: EJB 3.1 Specification, chapter 18.

This chapter will look at how we can schedule execution of business logic implemented in version 3.1 EJBs. The business logic can be implemented in stateless session beans, singleton session beans and message driven beans but the EJB 3.1 specification does not allow scheduled execution of business logic implemented in stateful session beans.

Throughout this chapter, the expression “a timer expires” is used. It means that the time has arrived when the timer is supposed to trigger execution of some business logic.

12.1. EJB Business Logic Scheduling Example

As usual, we'll start with an example showing some of the features available in EJB 3.1 for scheduling execution of business logic.

12.1.1. Create the Project

First we'll create the Eclipse project in which we'll develop the two EJBs in the example:

- Create an EJB project in Eclipse, as described in [appendix B](#).
I call my project “TimerExample”.
- Create the package *com.ivan.scbcd6*.

12.1.2. Create the Scheduled Stateless Session Bean

The first EJB, a stateless session bean, that we'll develop shows how to use the `@Schedule` annotation to automatically create and schedule a timer that invokes the annotated timeout callback method.

- In the package *com.ivan.scbcd6*, implement the *ScheduledStatelessSessionBean* class as follows:

```
package com.ivan.scbcd6;

import java.util.Date;
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.EJBException;
import javax.ejb.NoSuchObjectLocalException;
import javax.ejb.Schedule;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.Timer;

/**
 * Stateless session bean that contains two timer callback methods that
 * are scheduled using the @Schedule annotation.
 * An EJB may contain any number of methods annotated with @Schedule
 * that all are invoked according to their own schedule.
 */
@Stateless
public class ScheduledStatelessSessionBean
{
    /* Constant(s): */

    /* Class variable(s): */
    private static int sCurrentInstanceNo = 1;

    /* Instance variable(s): */
    private int mInvocationCounter;
    private int mInstanceNo = sCurrentInstanceNo++;
}
```

```

@Resource
private SessionContext mBeanContext;

@PostConstruct
public void initialize()
{
    /* Log the session bean instance creation. */
    System.out.println("ScheduledStatelessSessionBean created: "
        + mInstanceNo + " at: " + new Date());
}

/**
 * Scheduled method invoked every 20th and 45th second every minute
 * between 6 o'clock in the morning and 22 o'clock in the evening.
 *
 * @param inTimer Timer that caused the timeout callback invocation.
 */
@SuppressWarnings("unused")
@Schedule(second = "20, 45", minute = "*", hour = "6-22",
    dayOfWeek = "Mon-Fri", dayOfMonth = "*", month = "*", year = "*",
    info = "MyTimer")
private void scheduledMethod1(final Timer inTimer)
{
    System.out.println("ScheduledStatelessSessionBean.scheduledMethod1: "
        + mInstanceNo + " entering at: " + new Date());
    System.out.println("    Rollback only: " + mBeanContext.getRollbackOnly());
    System.out.println("    Timer info: " + inTimer.getInfo());

    /*
     * Wait some time to show what happens with multiple timer
     * callback methods being invoked on a session bean that
     * has container managed concurrency.
     */
    waitSeconds(15);

    cancelOverdue(inTimer);

    System.out.println("ScheduledStatelessSessionBean.scheduledMethod1: "
        + mInstanceNo + " exiting at: " + new Date());
}

/**
 * Scheduled method invoked every 10th second within the minute
 * starting at 15th second.
 * Timeout callback methods need not take a Timer object, as is the
 * case with this method.
 */
@SuppressWarnings("unused")
@Schedule(second = "15/10", minute = "*", hour = "*")
private void scheduledMethod2()
{
    System.out.println("ScheduledStatelessSessionBean.scheduledMethod2: "
        + mInstanceNo + " entering at: "
        + new Date());

    /*
     * Wait some time to show what happens with multiple timer
     * callback methods being invoked on a session bean that
     * has container managed concurrency.
     */
    waitSeconds(2);

    System.out.println("ScheduledStatelessSessionBean.scheduledMethod2: "
        + mInstanceNo + " exiting at: " + new Date());
}

private void cancelOverdue(final Timer inTimer)
throws IllegalStateException, NoSuchObjectLocalException, EJBException
{
    /* Cancel timer after certain number of invocations. */
    if (mInvocationCounter++ > 5)
    {
        System.out.println("Cancelling " + mInstanceNo + "...");
        inTimer.cancel();
    }
}

```

```

private void waitSeconds(final long inSeconds)
{
    try
    {
        Thread.sleep(inSeconds * 1000L);
    } catch (InterruptedException theException)
    {
        // Ignore exceptions.
    }
}

```

Note that:

- The EJB is a stateless session bean.
Recall that stateless session beans have container managed concurrency, which does not allow two threads to execute on one and the same instance simultaneously.
- The session bean context is injected into the *mBeanContext* instance variable.
- There is a method annotated with `@PostConstruct`.
This lifecycle callback method logs the creation of instances of the EJB.
- There is a method named *scheduledMethod1* that takes a parameter of the type *Timer*.
This method is a timeout callback method.
- The *scheduledMethod1* is annotated with the `@Schedule` annotation.
This annotation specifies the schedule for invocation of the method. In this case, the method is to be invoked on the 20th and 45th second every minute, Mondays to Fridays from 06 in the morning to 22 in the evening. The method is to be invoked regardless of which day of the month it is, every month, every year.
More about declarative scheduling [later](#).
- The *scheduledMethod1* has private visibility.
Timeout callback methods, like *scheduledMethod1*, may have private visibility.
- There is a delay for 15 seconds in the *scheduledMethod1*.
This delay was introduced in order to see how the container handles timer expiration during the execution of a timeout callback method.
- There is another timeout callback method named *scheduledMethod2*.
This method also has private visibility but does not take any parameters.
- The *scheduledMethod2* is also annotated with the `@Schedule` annotation.
The method is scheduled to be invoked every 10th second in every minute every hour, starting at the 15th second in a minute. The invocation schedule is thus on seconds 15, 25, 35, 45 and 55 in every minute.
More about declarative scheduling [later](#).
- There is a delay for two seconds in the *scheduledMethod2* method.
This delay was introduced in order to see how the container handles timer expiration during the execution of a timeout callback method.
- The *cancelOverdue* method cancels a timer if a counter has reached a certain number.

12.1.3. Running the Example Program – Declarative Scheduling

Having implemented the stateless session bean in the previous section we are now ready to try the example program out for the first time. There is no need for a client to the EJB since, when it is deployed, the timeout callback methods will be scheduled for invocation and later invoked by the container.

- Deploy the example program to GlassFish.
- After some time, you should see output in the GlassFish log similar to this:

```
...
INFO: Portable JNDI names for EJB ScheduledStatelessSessionBean :
[ java:global/TimerExample/ScheduledStatelessSessionBean!
com.ivan.scbcd6.ScheduledStatelessSessionBean,
java:global/TimerExample/ScheduledStatelessSessionBean]
INFO: TimerExample was successfully deployed in 205 milliseconds.

INFO: ScheduledStatelessSessionBean created: 2 at: Wed Dec 08 07:12:15 CET 2010
INFO: ScheduledStatelessSessionBean.scheduledMethod2: 2 entering at: Wed Dec 08 07:12:15
CET 2010

INFO: ScheduledStatelessSessionBean.scheduledMethod2: 2 exiting at: Wed Dec 08 07:12:17
CET 2010

INFO: ScheduledStatelessSessionBean.scheduledMethod1: 2 entering at: Wed Dec 08 07:12:20
CET 2010
INFO:     Rollback only: false
INFO:     Timer info: MyTimer

INFO: ScheduledStatelessSessionBean created: 3 at: Wed Dec 08 07:12:25 CET 2010
INFO: ScheduledStatelessSessionBean.scheduledMethod2: 3 entering at: Wed Dec 08 07:12:25
CET 2010

INFO: ScheduledStatelessSessionBean.scheduledMethod2: 3 exiting at: Wed Dec 08 07:12:27
CET 2010
...
```

Note that:

- An instance of the EJB with number 2 is created.
- The method *scheduledMethod2* is invoked and finishes executing on the EJB instance with number 2.
- The method *scheduledMethod1* is invoked and starts executing on the EJB instance with number 2.
- The method *scheduledMethod2* is scheduled for execution prior to *scheduledMethod1* has finished executing.
- A new instance of the EJB, with number 3, is created.
- The method *scheduledMethod2* is invoked, and finishes executing, on the EJB instance with number 3.
- The timer info output to the console is the string specified in the @Schedule annotation.
- We can thus conclude that scheduling collisions are resolved by GlassFish by creating a new instance of the stateless session bean and invoking the timeout callback method to be executed next on the new instance.

Note that this is the behaviour displayed by the version of GlassFish I am using and not behaviour stipulated by the EJB 3.1 specification.

12.1.4. Programmatic Timer Creation Example

Adding to our scheduling example project, we will create a singleton session bean that, when the application starts up, programmatically schedules a recurring timer. We will also look at the use of transactions in timeout callback methods.

- Optionally, comment out the `@Schedule` annotations in the `ScheduledStatelessSessionBean` class developed in the previous section.
This is done in order to avoid output not relevant to the programmatic timer creation EJB in the console.
- In the package `com.ivan.scbcd6`, implement the `ProgrammaticTimerStartedBean` class as follows:

```
package com.ivan.scbcd6;

import java.util.Date;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.SessionContext;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.Timer;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

/**
 * Singleton session bean that programmatically starts a timer
 * at the startup of the application in which the EJB is deployed.
 * Note that the class implements the TimedObject interface, which
 * could have been replaced by using the @Timeout annotation.
 *
 * The TimerService object used for programmatic scheduling can be
 * obtained either from the EJB context or it can be injected directly,
 * as seen with the second instance variable below.
 *
 * Timer(s) created programmatically in an EJB will invoke one single
 * callback method, either the ejbTimeout method from the TimedObject
 * interface or a method annotated with @Timeout.
 */
@Singleton
@LocalBean
@Startup
public class ProgrammaticTimerStartedBean implements TimedObject
{
    /* Constant(s): */

    /* Instance variable(s): */
    @Resource
    private SessionContext mBeanContext;
    @Resource
    private TimerService mTimerService;

    /**
     * Initializes the EJB and starts the single-action timer that
     * is to be invoked on this EJB.
     * Programmatic creation of timers can be done in a method with
     * any transaction attribute (container managed transactions).
     */
    @PostConstruct
    @TransactionAttribute(TransactionAttributeType.NEVER)
    public void initialize()
    {
        System.out.println("ProgrammaticTimerStartedBean.initialize: "
            + new Date());
        System.out.println("    Context: " + mBeanContext);
    }
}
```

```

        System.out.println("    Rollback only: "
                           + mBeanContext.getRollbackOnly());

        /*
         * A TimerConfig object may be used to supply additional
         * information that will be enclosed in the Timer object
         * when the timeout callback method is invoked on the bean.
         * The "Timer info" string in the example below is the arbitrary
         * serializable object that we later can retrieve when the timeout
         * callback method is invoked.
         */
        TimerConfig theTimerConfig = new TimerConfig("Timer info", false);

        /*
         * Retrieve the timer service object from the bean context and
         * create a timer.
         * We could have used the timer service object in the instance
         * variable mTimerService with the same effect.
         */
        mTimerService.createIntervalTimer(5000, 5000, theTimerConfig);
    }

    /**
     * Callback method in the TimedObject interface that is invoked when
     * a timer associated with the instance expires.
     * Such a method in an EJB with container managed transactions may
     * only have the REQUIRED or REQUIRES_NEW transaction attributes.
     * Successful execution of a timer callback method means that the
     * transaction in which the method executes is committed.
     *
     * @param inTimer Timer that expired and caused invocation of the
     * callback method.
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void ejbTimeout(final Timer inTimer)
    {
        /*
         * Output the time at which the method was invoked and the
         * arbitrary serializable data enclosed when creating the
         * TimerConfig and the timer.
         */
        System.out.println("ProgrammaticTimerStartedBean.ejbTimeout: " +
                           new Date() + ", info: " + inTimer.getInfo());

        /* Timer method always execute with an unauthorized principal. */
        System.out.println("    Security principal: "
                           + mBeanContext.getCallerPrincipal());

        /*
         * Retrieves the number of milliseconds remaining until the
         * next invocation of the timeout callback method.
         */
        System.out.println("    Time remaining: " + inTimer.getTimeRemaining());

        /*
         * Sometimes the transaction in which the timer callback is
         * executed rolls back. This causes the container to retry the
         * timer and, with GlassFish, eventually cancel the timer if
         * a certain number of consecutive rollbacks occur.
         */
        if (Math.random() > 0.8)
        {
            System.out.println("    Rolling back!");
            mBeanContext.setRollbackOnly();
        }

        System.out.println("    Rollback only: "
                           + mBeanContext.getRollbackOnly());
    }
}

```

Note that:

- The class is annotated with the `@Singleton`, the `@LocalBean` and the `@Startup` annotations. The EJB is thus a singleton session bean with a local view that will be eagerly loaded upon application startup.
- The class has two instance variables, `mBeanContext` and `mTimerService`, both annotated with the `@Resource` annotation.
We have already seen the EJB context of an EJB being injected into EJB instances. This example also shows that an instance of `TimerService` can be injected into an EJB instance.
- The `initialize` method is annotated with the `@PostConstruct` annotation.
Since the singleton session bean is to be eagerly loaded on application startup, this method will be executed when the application starts up. This gives us an opportunity to schedule any events that are to occur when the application is running in the application server.
- The `initialize` method is annotated with the `@TransactionAttribute(TransactionAttributeType.NEVER)` annotation.
We'll [later](#) look at how, despite this annotation, we can mark the current transaction for rollback and undo the scheduling of any timers done in the method.
- In the `initialize` method, the rollback-state of the transaction is examined by calling `getRollbackOnly`.
- When creating the timer, a `TimerConfig` object is first created and the timer service is then asked to create an interval timer.
The `TimerConfig` object enables us to set persistency of the timer as well as enclose a serializable object holding additional data for our timeout callback method.
- The class implements the `TimedObject` interface and contains a method named `ejbTimeout`.
The `ejbTimeout` method is the timeout callback method that will be invoked by the timer(s) created programmatically in the EJB.
An alternative to the `TimedObject` interface is to annotate a method with the `@Timeout` annotation.
- The `ejbTimeout` method takes a parameter of the type `Timer`.
This parameter holds a reference to the timer that caused the timeout callback method to be invoked.
- The time remaining until the next invocation and other data related to the scheduling of the timeout callback method can be retrieved from the `Timer` object. In this example we retrieve the time in milliseconds that remain until the next invocation of the `ejbTimeout` method.
- The `ejbTimeout` method is annotated with the `@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)` annotation.
A timeout callback method is required to execute in a transaction context and the only two allowed transaction attributes are `REQUIRES_NEW` and `REQUIRED`.
- The transaction in which the `ejbTimeout` method is executing is randomly set to roll back.
We will see how this affects the timer when we run the example program.

12.1.5. Running the Example Program – Programmatic Scheduling

Having implemented the singleton session bean in the previous section we are now ready to try the example program out. There is no need for a client to the EJB since, when it is deployed, the singleton session bean will be eagerly initialized and the timer scheduled programmatically.

- Deploy the example program to GlassFish.

After some time, you should see output in the GlassFish log similar to this (part of the output omitted to conserve space):

```
INFO: Portable JNDI names for EJB ScheduledStatelessSessionBean :  
[ java:global/TimerExample/ScheduledStatelessSessionBean!  
com.ivan.scbcd6.ScheduledStatelessSessionBean,  
java:global/TimerExample/ScheduledStatelessSessionBean]  
INFO: Portable JNDI names for EJB ProgrammaticTimerStartedBean :  
[ java:global/TimerExample/ProgrammaticTimerStartedBean,  
java:global/TimerExample/ProgrammaticTimerStartedBean!  
com.ivan.scbcd6.ProgrammaticTimerStartedBean]  
INFO: ProgrammaticTimerStartedBean.initialize: Wed Dec 08 19:38:53 CET 2010  
INFO: Context: ProgrammaticTimerStartedBean; id: [B@32f5c51c  
INFO: Rollback only: false  
INFO: TimerExample was successfully deployed in 209 milliseconds.  
  
INFO: ProgrammaticTimerStartedBean.ejbTimeout: Wed Dec 08 19:38:58 CET 2010, info: Timer  
info  
INFO: Security principal: ANONYMOUS  
INFO: Time remaining: 4997  
INFO: Rollback only: false  
  
...  
  
INFO: ProgrammaticTimerStartedBean.ejbTimeout: Wed Dec 08 19:39:23 CET 2010, info: Timer  
info  
INFO: Security principal: ANONYMOUS  
INFO: Time remaining: 4999  
INFO: Rolling back!  
INFO: Rollback only: true  
  
INFO: ProgrammaticTimerStartedBean.ejbTimeout: Wed Dec 08 19:39:28 CET 2010, info: Timer  
info  
INFO: Security principal: ANONYMOUS  
INFO: Time remaining: 4996  
INFO: Rolling back!  
INFO: Rollback only: true  
INFO: EJB5119:Expunging timer [ '12@@1291786701392@server@@domain1' 'TimedObject =  
ProgrammaticTimerStartedBean' 'Application = TimerExample' 'BEING_DELIVERED' 'PERIODIC'  
'Container ID = 84658510992769024' 'Wed Dec 08 19:38:58 CET 2010' '5000' ] after [2]  
failed deliveries
```

Note that:

- The string “Timer info” is retrieved from the *Timer* object passed as parameter to the *ejbTimeout* method.
This is the additional data enclosed to the *TimerConfig* object created when scheduling the timer.
- The security principal obtained in the *ejbTimeout* method is the principal representing an unidentified user – when using GlassFish v3 the name is ANONYMOUS.
- After having set the transaction to roll back two consecutive times in a row, the timer is cancelled by the container. No further invocations of the timeout callback method are made. This feature is product specific and may differ between different application servers.

12.1.6. Rolling Back Programmatically Scheduled Timers

Programmatic scheduling of timers can be part of a transaction and may thus be rolled back, as we will see in this little addition to our example program.

- Modify the *initialize* method of the *ProgrammaticTimerStartedBean* to look like this:

```
...
    /**
     * Initializes the EJB and starts the single-action timer that
     * is to be invoked on this EJB.
     * Programmatic creation of timers can be done in a method with
     * any transaction attribute (container managed transactions).
     */
    @PostConstruct
    @TransactionAttribute(TransactionAttributeType.NEVER)
    public void initialize()
    {
        System.out.println("ProgrammaticTimerStartedBean.initialize: "
            + new Date());
        System.out.println("    Context: " + mBeanContext);
        System.out.println("    Rollback only: "
            + mBeanContext.getRollbackOnly());

        /*
         * A TimerConfig object may be used to supply additional
         * information that will be enclosed in the Timer object
         * when the timeout callback method is invoked on the bean.
         * The "Timer info" string in the example below is the arbitrary
         * serializable object that we later can retrieve when the timeout
         * callback method is invoked.
         */
        TimerConfig theTimerConfig = new TimerConfig("Timer info", false);

        /*
         * Retrieve the timer service object from the bean context and
         * create a timer.
         * We could have used the timer service object in the instance
         * variable mTimerService with the same effect.
         */
        Timer theTimer = mTimerService.createIntervalTimer(5000, 5000,
            theTimerConfig);

        /*
         * Note that we can roll back timer creation despite the method
         * having the transaction attribute NEVER.
         */
        mBeanContext.setRollbackOnly();

        System.out.println("    Creation rollback only: "
            + mBeanContext.getRollbackOnly());
        System.out.println("    Timer next timeout: " + theTimer.getNextTimeout());
    }
...
}
```

- Deploy the example program to GlassFish.

After some time, you should see output in the GlassFish log similar to this (part of the output omitted to conserve space):

```
INFO: Portable JNDI names for EJB ScheduledStatelessSessionBean : ....
INFO: Portable JNDI names for EJB ProgrammaticTimerStartedBean : ...
INFO: ProgrammaticTimerStartedBean.initialize: Fri Dec 17 07:12:10 CET 2010
INFO:   Context: ProgrammaticTimerStartedBean; id: [B@57a8bdbe
INFO:   Rollback only: false
INFO:   Creation rollback only: true
INFO:   Timer next timeout: Fri Dec 17 07:12:15 CET 2010

INFO: TimerExample was successfully deployed in 254 milliseconds.
```

Note that:

- A *Timer* object is obtained as a result of scheduling a timer.
Using this object we can get information about the timer.
- The rollback-only flag is set to true when execution reaches the end of the *initialize* method.
According to the EJB 3.1 specification, the *setRollbackOnly* and *getRollbackOnly* methods are not available in PostConstruct methods of stateless session beans since these methods are executed in an unspecified transaction context. These methods should not be used as in this example.
- The timeout callback method is never called.
- We are able to obtain a date telling us when the next timeout of the timer is to happen.
If we look at the API documentation for the *getNextTimeout* method in the *Timer* interface, we can see that it will throw an exception if called when the timer is cancelled.
The timer has thus not been cancelled at this stage, but is cancelled sometime after the *initialize* method has finished executing.

This concludes the scheduling EJB business logic example program.

12.2. Timeout Callback Methods

References: EJB 3.1 Specification, section 18.2.5 and 18.2.8.

Common for timeout callback methods used in connection to declarative and programmatic scheduling is that, as the result of a timer expiring, one timeout callback method will be invoked. One timeout callback method may be associated with multiple timers.

Timeout callback methods must be defined according to the following specifications:

- Can have public, protected, private or package visibility.
- Must not be final or static.
- Must have one of the following two signatures:

*void someMethodName()
void someMethodName(Timer inTimer)*

Where “someMethodName” may be substituted by any method name.

- Must not throw application exceptions.
- Must have either REQUIRED or REQUIRES_NEW transaction attributes when the EJB uses container managed transactions.
- Can be defined in the EJB implementation class or in a superclass of the EJB implementation class, if any.

In addition:

- Timeout callback methods are executed without a client security context.
- If a timeout callback method rolls back the transaction it is executing in, the container will retry the timer.

As we saw in the example program, GlassFish destroyed the timer associated with a timeout callback method that two consecutive times rolled back the transaction it executed in. This behaviour is not stipulated by the EJB specification.

The following list contains links to tables that specifies, among other things, what operations are allowed for timeout callback methods in the different kinds of EJBs. In addition the tables tells us when a timer service object is available, with which we can do programmatic scheduling.

- [Stateless session beans](#)
- [Singleton session beans](#)
- [Message driven beans](#)

Information specific to timeout callback methods of declarative and programmatically scheduled timers will be discussed in the sections on [Declarative Scheduling](#) and [Programmatic Scheduling](#) below.

12.3. The Timer Interface

An object implementing the *Timer* interface is obtained when scheduling a new timer. It may also be enclosed as a parameter when the container invokes timeout callback methods.

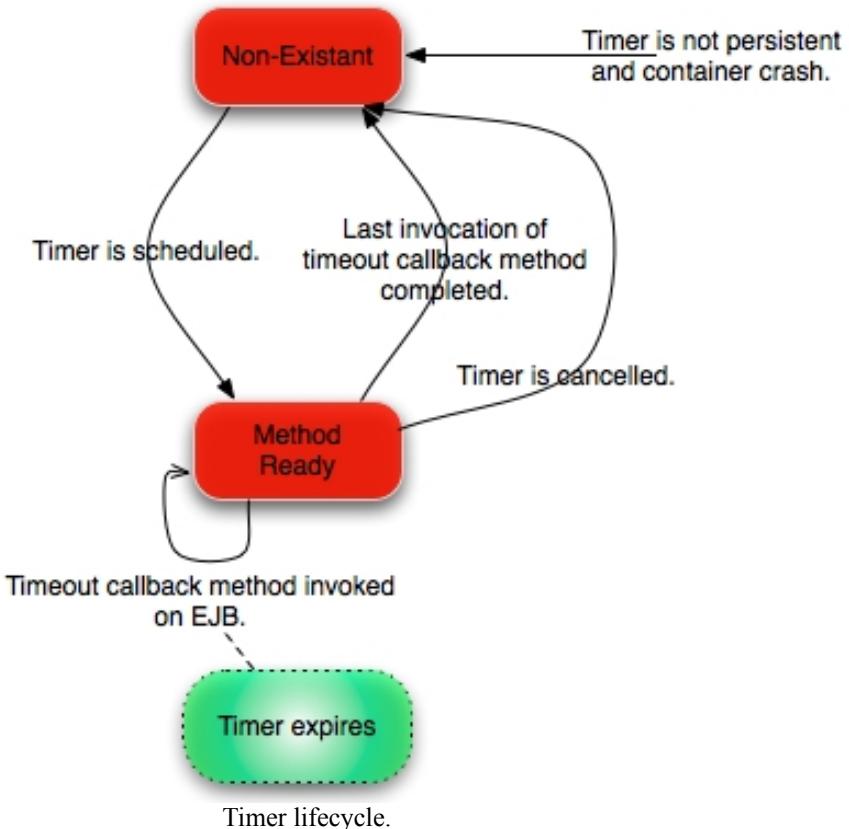
The *Timer* interface contains the following methods:

Method Signature	Comments
void cancel()	Cancels the timer. The timeout callback method associated with the timer will not be invoked further as a result of the cancelled timer.
TimerHandle getHandle()	Obtains a serializable representation of the timer that later can be used to re-create the timer.
Serializable getInfo()	Retrieves the timer info object set when the timer was created.
Date getNextTimeout()	Retrieves a <i>Date</i> object specifying the point in time at which next timeout expiration will occur.
ScheduleExpression getSchedule()	Retrieves the timer schedule for calendar-based timers. If the timer is not calendar-based, an exception will be thrown.
long getTimeRemaining()	Retrieves the number of milliseconds remaining until the next timeout expiration of the timer.
boolean isCalendarTimer()	Returns true if the timer is a calendar-based timer, false otherwise.
boolean isPersistent()	Returns true if the timer is persistent and thus will survive container shutdown etc, false otherwise.

In addition to the above methods, objects implementing the *Timer* interface must also implement the *equals* and *hashCode* methods, facilitating comparison etc of *Timer* objects.

12.3.1. Timer Life-Cycle

A timer's life spans from the moment it is scheduled until after the last invocation of the timeout callback method of the timer has finished executing or until the timer is cancelled. If an attempt is made to invoke a method on a timer that no longer exists, a *NoSuchObjectLocalException* will be thrown.



12.3.2. Timers and Persistence

Objects implementing the *Timer* interface must, according to the EJB 3.1 specification, not be serializable. To serialize a timer, retrieve an object implementing the *TimerHandle* interface using the *getHandle* method in the *Timer* interface.

A timer handle can be used during the lifetime of the timer from which it was obtained.

Persistent timers that are restored after, for instance, a container crash and that have expired during the time when the container was down will have their timeout callback methods invoked as follows:

- Single-event timers.
Timeout callback method will be invoked once.
- Interval timer.
Timeout callback method will be invoked at least once, even if multiple expirations occurred.
- Scheduled timer.
Timeout callback method will be invoked at least once, even if multiple expirations occurred.

12.4. Declarative Scheduling

Declarative scheduling of business methods in EJBs can be accomplished using either annotations or the deployment descriptor. Common to both kinds of declarative scheduling is that there may be any number of timeout callback methods in an EJB that are invoked as a result of declarative scheduling. It is also possible to have multiple timers scheduled declaratively to invoke one and the same timeout callback method.

All timers created with declarative scheduling, be it using annotations or the deployment descriptor, are calendar-based timers.

12.4.1. Declarative Scheduling with Annotations

To schedule a method in an EJB for invocation when timeout expiration event(s) occur, the method just needs to be annotated with either the `@Schedule` annotation or, if multiple schedules are to be used to activate one and the same method, the `@Schedules` annotation. Any number of methods in an EJB can be scheduled this way, each with its own schedule if so desired.

The following annotations connected to declarative scheduling of business methods in EJBs are available:

Annotation	Description
Schedule	Annotates a method that will be the timeout callback method of an automatically created timer scheduled as specified in the annotation elements.
Schedules	Allows for the scheduling of multiple timers to invoke one and the same timeout callback method (the annotated method).
Timeout	Designates the annotated method to be the timeout callback method for programmatically created timers. Alternatively, the EJB may implement the <code>TimedObject</code> interface and implement the <code>ejbTimeout</code> method.

The `@Schedule` annotation has the following optional elements:

Optional Element	Description
year	Year during which the timer expires. Default: * Allowed values: Four digit year. Example: 2010
month	Month(s) during which the timer expires. Default: * Allowed values: 1 to 12, Jan, Feb, Mar, ..., Dec Examples: 1, Jan, 1-4
dayOfMonth	Day(s) within month on which the timer expires. Default: * Allowed values: 1 to 31, -1 to -7 (number of days before the last day of the month), 1st to 5th, Last. Examples: 27, -5, 2nd, Last
dayOfWeek	Day(s) within a week on which the timer expires. Default: * Allowed values: 0 to 7 (0 and 7 both refer to Sunday), Mon to Sun. Examples: 3, Sat.

hour	Hour(s) during a day during which the timer expires. Default: 0 Allowed values: 0 to 23 Examples: 13, 18
minute	Minute(s) in an hour during which the timer expires. Default: 0 Allowed values: 0 to 59 Examples: 48, 13
second	Second(s) in a minute at which the timer expires. Default: 0 Allowed values: 0 to 59 Examples: 32, 48
timezone	Timezone of which the timer time is kept. Default: “” (empty string – the current timezone) Allowed values: Time zone names (see EJB 3.1 specification).
persistent	Will the timer be persisted to survive application shutdown etc? Default: true
info	String that will be enclosed to the timeout callback method as timer info when it is invoked.

An example of how to use the @Schedule annotation:

```
...
    @Schedule(second = "20, 45", minute = "*", hour = "6-22",
              dayOfWeek = "Mon-Fri", dayOfMonth = "*", month = "*", year = "*",
              info = "MyTimer")
    private void scheduledMethod1(final Timer inTimer)
    {
...
}
```

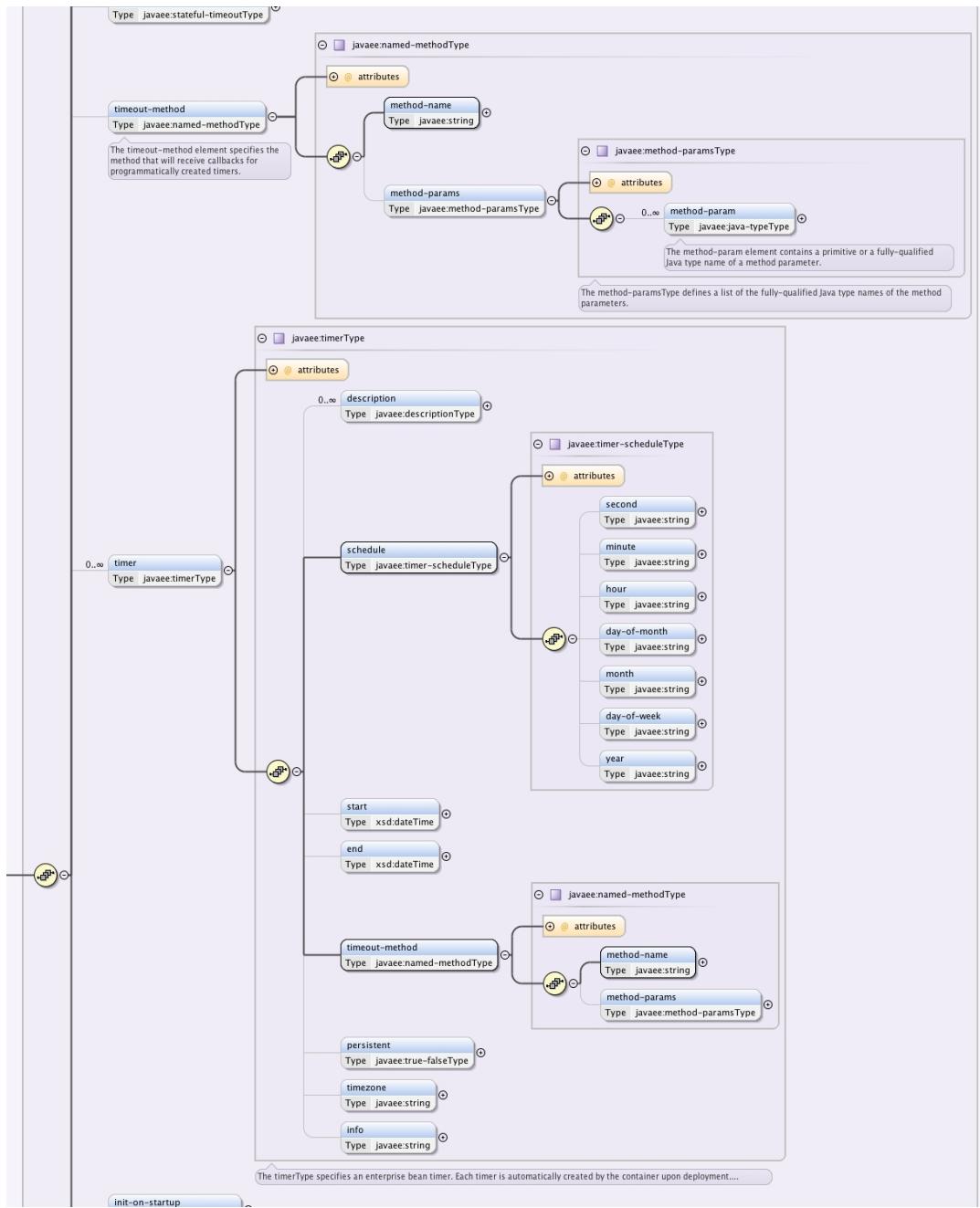
An example of how to use the @Schedules annotation:

```
...
    @Schedules(
    {
        @Schedule(second = "15/10", minute = "*", hour = "*"),
        @Schedule(second = "18/10", minute = "*", hour = "*")
    })
    private void scheduledMethod2()
    {
...
}
```

Detailed information on scheduling with calendar-based expressions can be found in section 18.2.1 in the EJB 3.1 Specification.

12.4.2. Declarative Scheduling in the Deployment Descriptor

The features of the `@Schedule` and `@Schedules` annotations are also available using the `ejb-jar.xml` deployment descriptor. The following figure shows the excerpt of the deployment descriptor XML schema relevant to scheduling execution of business logic:



Excerpt of the `ejb-jar.xml` deployment descriptor XML schema
relevant to scheduling of business method execution.

The following shows an example of declarative scheduling of business logic using the ejb-jar.xml deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

<enterprise-beans>
    <session>
        <ejb-name>ScheduledStatelessSessionBean</ejb-name>
        <local-bean/>
        <ejb-class>com.ivan.scbcd6.ScheduledStatelessSessionBean</ejb-class>
        <session-type>Stateless</session-type>

        <!--
            Zero or more <timer> elements, each defining a timer
            that schedules invocation of some business logic in the EJB.
        -->
        <timer>
            <!--
                Specify the timer schedule.
            -->
            <schedule>
                <second>15/10</second>
                <minute>*</minute>
                <hour>*</hour>
            </schedule>
            <!--
                An optional start and/or end time of the timer may be
                specified. These times take precedence over the
                schedule. Thus scheduled invocations of the timeout
                callback method will occur according to the schedule
                specified in the <schedule> element, but only inside
                the time period specified by the <start> and <end>
                times.
            -->
            <start>2010-12-01T00:00:00Z</start>
            <end>2012-12-24T00:00:00Z</end>
            <!--
                Specify the name and, optionally, the parameters
                of the timeout callback method that is to be invoked
                when the timer expires.
            -->
            <timeout-method>
                <method-name>scheduledMethod1</method-name>
            </timeout-method>
        </timer>
    </session>
</enterprise-beans>
</ejb-jar>
```

(continued on next page)

Note that:

- There is a <timer> element in the <session> element defining the ScheduledStatelessSessionBean EJB.
This element defines a timer and specifies the timeout callback method that is to be invoked when the timer expires.
- Zero or more timers can be defined in the ejb-jar.xml deployment descriptor for an EJB.
- The <timer> element contains a <schedule> element.
This defines the schedule that determines when the timeout callback method will be invoked.
For detailed information on how to define a schedule, please refer to section 18.2.1 in the EJB 3.1 Specification.
- An optional <start> element may be specified.
This element specifies the starting point in time when the timer will start to expire.
Thus, regardless of the schedule specified in the <schedule> element, the timeout callback method will never be invoked prior to the time in the <start> element.
- An optional <end> element may be specified.
This element specifies the ending point in time after which the timer will cease to expire.
Thus, regardless of the schedule specified in the <schedule> element, the timeout callback method will never be invoked after the time in the <end> element.
- The <timeout-method> element and child elements specifies the timeout callback method to be invoked when the timer expires.
The <method-name> element contains the name of the timeout callback method. Optionally, a <method-params> element containing one or more <method-param> elements may be used to specify the method parameter types in order to distinguish methods with the same names in the EJB.

12.5. Programmatic Scheduling

Programmatic scheduling of timers is, as we have seen, accomplished using an object that implements the *TimerService* interface. Such an object can be obtained in any of the following ways:

- Dependency injection.
- Calling the *getTimerService* method of *EJBContext*.
- JNDI lookup.

The *TimerService* allows for creation of the following kinds of timers:

- Calendar timers
Depending on their configuration, calendar timers can execute business logic once or at a recurring interval.
- Interval timers.
Interval timers will execute business logic at a recurring interval.
- Single-action timers.
Single-action timers will execute the scheduled business logic once.

12.5.1. Timeout Callback Methods for Programmatically Scheduled Timers

Timeout callback methods of programmatically scheduled timers must follow the restrictions that applies to all timeout callback methods as discussed [earlier](#).

All timers created programmatically by an EJB will invoke one and the same timeout callback method on the EJB. This timeout callback method can be specified in three ways:

- Annotate one method with the *@Timeout* annotation.
- Specify the timeout method using the *ejb-jar.xml* deployment descriptor.
- Let the EJB implementation class implement the *TimedObject* interface and, in the EJB, implement the *ejbTimeout(Timer)* method.

Timeout Callback Methods with Annotations

The first example shows an example of a timeout callback method specified using the *@Timeout* annotation:

```
...
 /**
 * Processes timeout expiration events of programmatically scheduled timers
 * in this EJB.
 * Such a method in an EJB with container managed transactions may
 * only have the REQUIRED or REQUIRES_NEW transaction attributes.
 * Successful execution of a timer callback method means that the
 * transaction in which the method executes is committed.
 *
 * @param inTimer Timer that expired and caused invocation of the
 * callback method.
 */
@TransactionalAttribute(TransactionAttributeType.REQUIRES_NEW)
@Timeout
public void myTimeout(final Timer inTimer)
{
...
}
```

Timeout Callback Methods in the Deployment Descriptor

The following example shows how the above method *myTimeout* is specified to be the timeout callback method of the EJB ProgrammaticTimerStartedBean:

```
...
<session>
    <ejb-name>ProgrammaticTimerStartedBean</ejb-name>
    <!--
        Specifies the single timeout callback method to be invoked
        when programmatically scheduled timers expire.
    -->
    <timeout-method>
        <method-name>myTimeout</method-name>
        <method-params>
            <method-param>javax.ejb.Timer</method-param>
        </method-params>
    </timeout-method>
</session>
...
```

Note that, despite the EJB 3.1 specification saying that a `<timeout-method>` element without a `<method-params>` element specified will match both a parameterless timeout callback method as well as a timeout callback method taking a *Timer* object parameter, this is not the case with the version of GlassFish v3 used when writing the examples of this book.

Timeout Callback Methods and the TimedObject Interface

The final example shows how the timeout callback method of an EJB is specified by implementing the *TimedObject* interface:

```
public class ProgrammaticTimerStartedBean implements TimedObject
{
    ...

    /**
     * Callback method in the TimedObject interface that is invoked when
     * a timer associated with the instance expires.
     * Such a method in an EJB with container managed transactions may
     * only have the REQUIRED or REQUIRES_NEW transaction attributes.
     * Successful execution of a timer callback method means that the
     * transaction in which the method executes is committed.
     *
     * @param inTimer Timer that expired and caused invocation of the
     *                callback method.
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void ejbTimeout(final Timer inTimer)
    {
        ...
    }
    ...
}
```

12.5.2. Creating Calendar Timers

The programmatic version of the @Schedule annotation is calendar timers configured with a *ScheduleExpression* instance. The configuration of the *ScheduleExpression* instance is similar to the configuration of the @Schedule annotation described [earlier](#) with the same default values.

Depending on the schedule expression, the timer can be executed once or at a recurring interval.

The following example schedules a timer to execute every 15th second, each minute in every hour:

```
...
    TimerConfig theTimerConfig = new TimerConfig("Timer info", false);

    /*
     * Programmatically create a schedule expression that determines
     * when the timer will expire.
     * In this case, expiration will take place every 15th second
     * in every minute in every hour.
     */
    ScheduleExpression theTimerSchedule = new ScheduleExpression().
        hour("*").minute("*").second(15);
    Timer theTimer = mTimerService.createCalendarTimer(theTimerSchedule,
        theTimerConfig);
...
}
```

In the above example, an optional *TimerConfig* object was also supplied. For details on timer configuration, see [below](#).

Specifying a Calendar with *ScheduleExpression*

The *ScheduleExpression* class enables specifying timer schedules that occurs at one single given point in time or at a certain recurring interval. The *ScheduleExpression* class has the following properties:

Property Name	Default Value	Allowed Values	Comments
timezone	JVM default timezone.	Time zone names. See EJB 3.1 specification.	
startDate	none		Date from which the schedule is activated.
endDate	none		Date after which the schedule is no longer active.
year	*	Four-digit calendar year.	Year at which the scheduled event is to occur.
month	*	1 to 12, Jan to Dec	One or more months within a year at which the scheduled event is to occur.
dayOfMonth	*	1 to 31, -7 to -1, Last, 1st to 5th, Sun to Sat	One or more days within a month at which the scheduled event is to occur. Negative numbers mean number of days before the last day of the month. 1st etc means the first occurrence of the first day of the week in the month.
dayOfWeek	*	0 to 7, Sun to Sat	Day of week at which the scheduled event is to occur. 0 and 7 both refer to Sunday.
hour	0	0 to 23	One or more hours within a day at which the scheduled event is to occur.
minute	0	0 to 59	One or more minutes within an hour at which the scheduled event is to occur.
second	0	0 to 59	One or more seconds within a minute at which the scheduled event is to occur.

The “*” character is a wildcard representing all possible values of a property.

Note that all the setter-methods, which have the same name as the property without any “set” prefix, of the *ScheduleExpression* class returns the *ScheduleExpression* instance on which the method was invoked. This enables stringing multiple method invocations together like in this example:

```
...
ScheduleExpression theSchedule = new ScheduleExpression();
theSchedule.hour("*").minute(15).second(0);
...
```

Also note that each setter-method comes in two versions; one that take a string as a parameter and another that take an integer as a parameter.

Please refer to section 18.2.1 of the EJB 3.1 specification for additional details on schedule expressions.

12.5.3. Creating Interval Timers

Interval timers are timers which first expiration occurs after an initial delay and then expire at a regular interval.

There are four methods in the *TimerService* interface that facilitates scheduling of interval timers:

- `createIntervalTimer(Date, long, TimerConfig)`
- `createTimer(Date, long, Serializable)`
- `createIntervalTimer(long, long, TimerConfig)`
- `createTimer(long, long, Serializable)`

The first two methods creates a timer that first expire at a point in time specified by the supplied *Date* object and with subsequent expirations at an interval specified by the long integer parameter. The latter two methods creates a timer that expires after the number of milliseconds specified by the first long integer parameter and with subsequent expirations at an interval specified by the second long integer parameter.

Timers are either configured according to the supplied *TimerConfig* object or with the supplied serializable timer info object.

The following example schedules a timer that waits 42 milliseconds before first expiration and then expire every 15th second.

```
...
/*
 * Creates a timer that waits 42 milliseconds until first
 * expiration and thereafter expires every 15th second.
 * A null TimerConfig object is supplied, indicating that the
 * default timer configuration is to be used.
 */
Timer theTimer = mTimerService.createIntervalTimer(42, 15000, null);
...
```

In the above example, a null *TimerConfig* object was supplied, indicating that the default timer configuration is to be used.

12.5.4. Creating Single-Action Timers

Single-action timers are timers that expires only once; either at a given point in time or after a certain delay. There are four methods in the *TimerService* interface that facilitates scheduling of single-action timers:

- `createSingleActionTimer(Date, TimerConfig)`
- `createTimer(Date, Serializable)`
- `createSingleActionTimer(long, TimerConfig)`
- `createTimer(long, Serializable)`

The first two methods creates a timer that expire at a point in time specified by the supplied *Date* object. The latter two methods creates a timer that expires after the number of milliseconds specified by the long integer parameter.

Timers are either configured according to the supplied *TimerConfig* object or with the supplied serializable timer info object.

For details on timer configuration, see [below](#).

The following example schedules a single execution of business logic 20 seconds into the future.

```
...
    TimerConfig theTimerConfig = new TimerConfig("Timer info", false);

    /*
     * Creates a timer that will execute business logic once
     * at a time specified by the supplied date.
     */
    Date theTimerDate = new Date(System.currentTimeMillis() + 20000);
    Timer theTimer = mTimerService.createSingleActionTimer(theTimerDate,
        theTimerConfig);
...

```

12.5.5. Timer Configuration

Timers are, as default, persistent. This means that they will survive the application server in which the application runs being shut down. Creating non-persistent timers is possible by using one of the methods that take a *TimerConfig* object as parameter. Supplying a *TimerConfig* also enables us to supply a serializable object, a timer info object, that is later supplied every time the timeout callback method is invoked.

```
...
    /*
     * Create a timer configuration object that sets the timer info
     * to the string "Timer info" and makes the timer(s) created
     * with the configuration non-persistent.
     */
    TimerConfig theTimerConfig = new TimerConfig("Timer info", false);

    /* Create a timer using the timer configuration. */
    Timer theTimer = mTimerService.createIntervalTimer(5000, 5000,
        theTimerConfig);
...

```

If a null *TimerConfig* object is supplied when creating a timer, the timer will be persistent and will have a null timer info object.

12.6. *Timers and Transactions*

References: EJB 3.1 Specification, sections 18.2.8, 18.4.3, 18.4.4.

Timers can be created and cancelled in transactions. Invocation of timeout callback methods also occur inside transactions.

- Timers can be created in transactions.
If the transaction rolls back, the timer creation or cancellation is also rolled back.
- Timers can be cancelled in transactions.
If the transaction is rolled back, the timer will be restored as if it had not been cancelled.
- Timeout callback methods are invoked inside transactions.
If the transaction is rolled back, the container will retry the invocation of the timeout callback method. The EJB 3.1 specification stipulates that the container must retry invoking the timeout callback method at least once. This behaviour applies to both persistent and non-persistent timers.

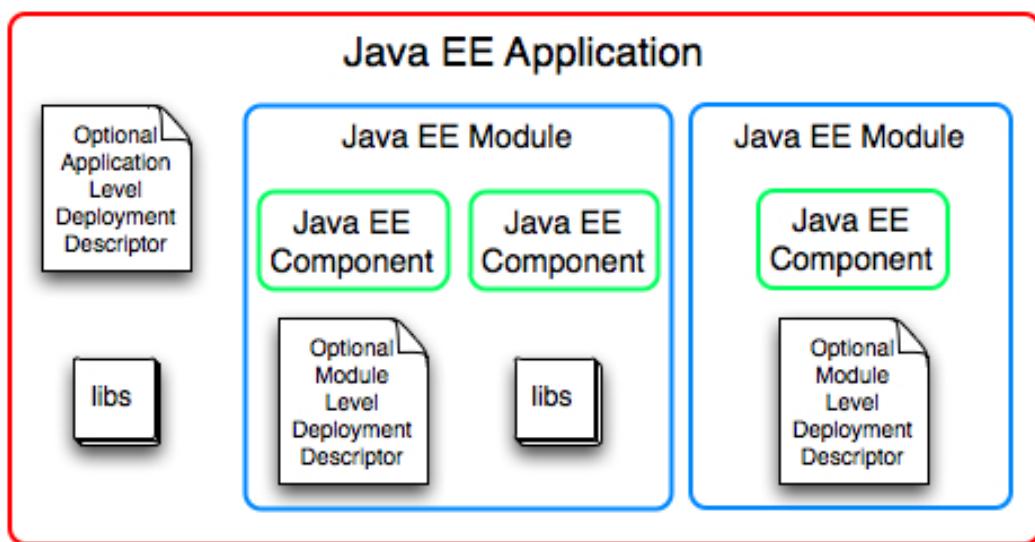
13. Package and Deploy EJB Applications

References: EJB 3.1 Specification, chapters 19 and 20, [JAR File Specification](#), JavaEE 6 Specification, chapter EE.8.

In this chapter we'll look at the options available when packaging and deploying modules that contains EJBs - be it a complete application or a standalone module that only contains EJBs.

13.1. Java EE Application Assembly

Java EE modules are the basic building-blocks with which Java EE applications are assembled. Java EE modules, in turn, are composed of Java EE components.



Example of a Java EE application assembly.

Java EE applications are contained in EAR files, where EAR stands for Enterprise Application aRchive. Java EE applications may contain a number of libraries on which one or more modules in the application may depend.

Java EE modules are:

- Web application modules.
Contained in WAR (Web Application aRchive) files.
- EJB modules.
Contained in EJB-JAR files holding enterprise Java business components.
- Application client modules.
- Resource adapter modules.
- Libraries.
Contained in JAR files.

Java EE modules may contain a number of libraries on which one or more components in the module may depend.

Java EE components required by the Java EE 6 specification are:

- EJBs
Enterprise Java Beans. Commonly contains business logic.
- Servlets
Includes JSP pages, JSF pages, filters and listeners.
Typically produces HTML that is rendered in a browser to create a user interface for the application.
- Application clients.
Client programs running on desktop computers. Typically so-called fat clients.
- Applets.
User interface components executing in web browsers.

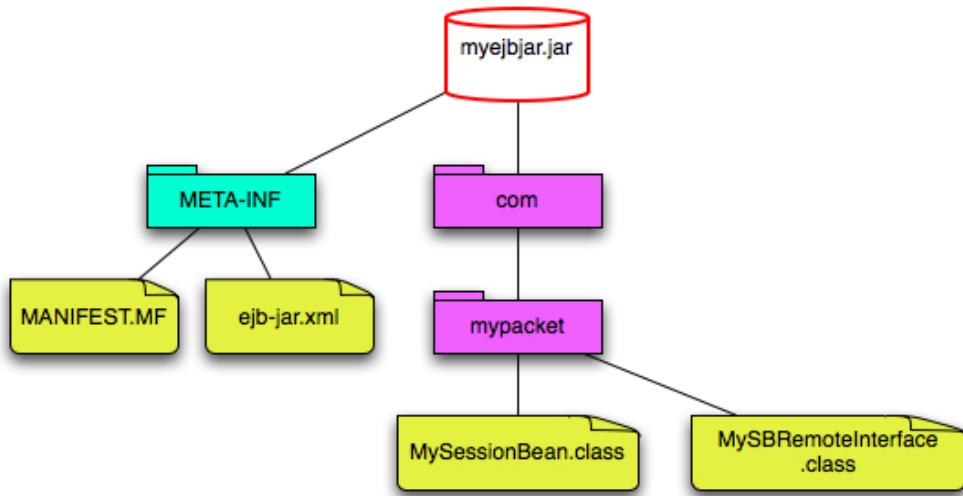
13.2. *Packaging Options*

Starting with EJB 3.1, the following options how to package EJBs are available:

- In EJB-JAR files.
This is the standard way of packaging EJBs and has also been used in earlier versions of the EJB specification.
EJB-JAR files may in turn be packaged in EAR files.
- In WAR files.
Starting with EJB 3.1, EJBs can now be packaged in web application archives.
WAR files may in turn be packaged in EAR files.
- In standard JAR files.
Using the embeddable container available in EJB 3.1, standalone Java SE applications can now take advantage of EJBs.

13.2.1. EJB-JAR Files

EJB-JAR files are used to package standalone EJB modules. Such modules can be deployed separately or be included in an EAR file. The following figure shows what the structure of an EJB-JAR file may look like:



Example of structure of an EJB-JAR containing a standalone EJB module.

Everything in the EJB-JAR file is optional:

- The META-INF directory is only needed if the EJB-JAR is to contain a MANIFEST.MF file, an ejb-jar.xml deployment descriptor or some other file that is to reside in this directory.
- The MANIFEST.MF file is only required if the EJB needs to include classes in external JAR files onto the classpath.
- The ejb-jar.xml deployment descriptor is not required if all configuration of the EJBs in the module has been done using annotations.
- EJB implementation classes, interfaces and other classes can be included by including other JAR files on the classpath.

Classes can be included either by being contained in the EJB-JAR file or by being contained in another file that is referenced from the MANIFEST.MF file (included on the classpath).

An EJB-JAR file is a module with its own namespace, as described in the section on [Portable JNDI Names for Session Beans](#) above. This means that the EJB-JAR file can be included in an EAR (enterprise application archive) file together with other modules that contain components with the same names as the components found in the EJB-JAR file without name collisions.

Additional considerations when deploying EJBs to an EJB-JAR file:

- If the EJB-JAR file contains an ejb-jar.xml deployment descriptor for EJB 3.x and the deployment descriptor is marked as metadata-complete, then annotations in classes in the EJB-JAR file will not be processed.

If the EJB-JAR is included in, for instance, a WAR, then the metadata-completeness of its deployment descriptor does not affect whether annotations of EJB classes in the WAR is processed or not.

The following is an example of an EJB 3.1 deployment descriptor which has been marked as being metadata-complete:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  metadata-complete="true">

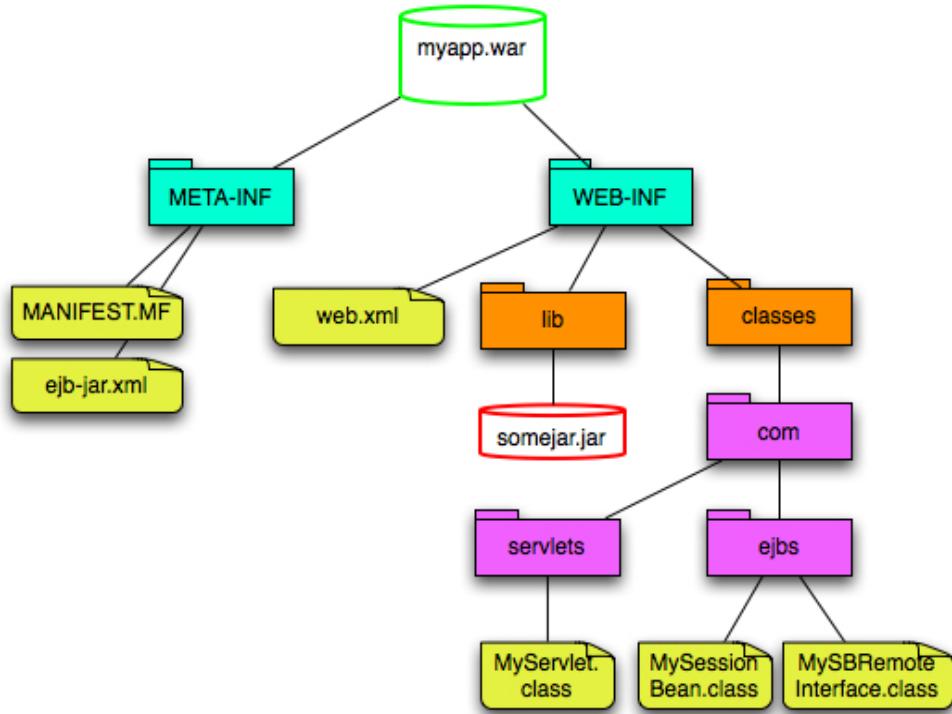
  ...

</ejb-jar>
```

13.2.2. WAR Files

References: EJB 3.1 Specification, section 20.4, JavaEE 6 Tutorial.

WAR files are used to package web application modules. Starting with EJB 3.1, version 3.x EJBs may be deployed in WAR files. The following figure shows the structure of an example WAR file that contains an EJB:



Example of structure of a WAR file containing an EJB.

Note that the above figure does not describe the complete structure of a WAR file.

As with EJB-JAR files, everything in a WAR file is optional:

- The META-INF directory is only needed if the WAR is to contain a MANIFEST.MF file, an ejb-jar.xml deployment descriptor or some other file that is to reside in this directory.
- The MANIFEST.MF file is only required if the application needs to include classes in external JAR files onto the classpath.
- The ejb-jar.xml deployment descriptor is not required if all configuration of the EJBs in the module has been done using annotations.
- The web.xml deployment descriptor is optional.
The Servlet 3.0 specification states that the web.xml deployment descriptor is optional and can be left out if all relevant configuration can be accomplished using annotations.
- EJB implementation classes, interfaces and other classes can be included by including other JAR files on the classpath or in the WEB-INF/lib directory.
Classes can be included either by being contained in the WAR file or by being contained in another file that is referenced from the MANIFEST.MF file (included on the classpath).

Additional factors that should be considered when deploying EJBs to WAR files:

- If included, the ejb-jar.xml deployment descriptor applies to all the EJBs in the WAR file.
- All components in a WAR file reside within one and the same module and share one and the same (JNDI) naming environment.
Care should be taken as to avoid naming conflicts. For details on name context, please refer to the section on [Portable JNDI Names for Session Beans](#) above.
- Annotated EJB implementation classes may be located either in the WEB-INF/classes directory or in JAR files located in the WEB-INF/lib directory.
Any ejb-jar.xml deployment descriptors in such archives will not be recognized by the container. Annotated EJB classes and interfaces in archives in the WEB-INF/lib directory will be found and recognized by the container.
- EJB-JAR files located in the WEB-INF/lib directory will only be recognized as regular JAR files and not as Java EE modules.
The no interface and local business interface view of an EJB deployed in a WAR file are only visible to components within the WAR.
If the WAR file is part of an application and other modules of the application wishes to access the no interface or local business interface view of an EJB, then the EJB must be deployed in a separate EJB-JAR file in the application.
- Annotations on EJB classes will not be processed if there is no ejb-jar.xml deployment descriptor and the web.xml deployment descriptor is marked as metadata-complete or if its version is earlier than 2.5.
The deployment descriptor, of which a fragment is displayed below, is metadata-complete and of version 3.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Deployment descriptor version is specified using the version attribute
    in the <web-app> element.
    Whether the deployment descriptor is metadata-complete is specified using
    the metadata-complete attribute in the <web-app> element.
-->
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0" metadata-complete="true">
...
</web-app>
```

- If the WAR file contains an ejb-jar.xml deployment descriptor for EJB 3.x and the deployment descriptor is marked as metadata-complete, then annotations in EJB classes in the WAR file will not be processed.
- Packaging of pre-EJB 3.x EJBs in WAR files is not supported.
- Packaging of JAX-RPC endpoint implementation classes in WAR files is not supported.

13.2.3. JAR Files

References: EJB 3.1 Specification, chapter 22.

Using the embeddable EJB container, which is new for EJB 3.1, it is also possible to run EJBs in, for instance, standalone Java SE applications or unit tests.

[Appendix H](#) shows an example of a Java SE application that uses the embeddable EJB 3.1 container to run an EJB.

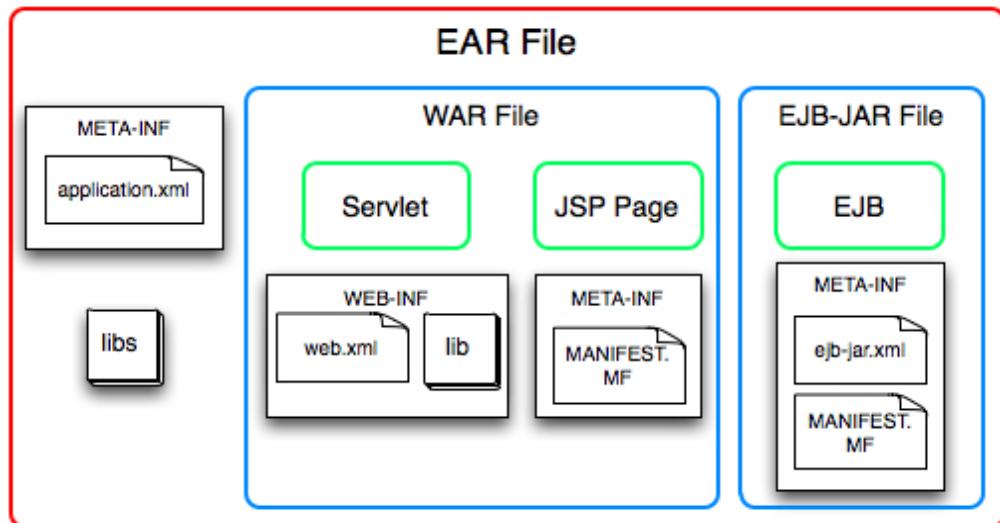
If all EJB configuration is done using annotations, then there is no need to include any EJB-specific metadata, such as a deployment descriptor.

If an ejb-jar.xml deployment descriptor is used, it is to be located in the META-INF directory in the JAR file in order for it to be automatically found by the embeddable EJB container.

A MANIFEST.MF file must be included if we want to specify the main class and the class-path of the application. It does not contain any EJB-related information.

13.2.4. EAR Files

EAR, Enterprise Application aRchive, files are the top-level containers of Java EE applications.



Structure of an example EAR file containing a WAR and an EJB-JAR file.

When deployed in an EAR file, the global JNDI change to have the application name inserted. This is described in the section on portable [JNDI names](#) earlier.

Starting from Java EE 5, the EAR application.xml deployment descriptor is optional and may be left out.

If the application.xml deployment descriptor is used, the <library-directory> element may be used to specify a directory of libraries. All JAR files in this directory are made available on the class-path of all components of all components in the EAR. See the Java EE specification for details.

13.2.5. EJB-Client JAR Files

Optionally, the classes and interfaces that make up the client view of an EJB can be packaged in a separate JAR file - an EJB-Client JAR file. Such a JAR file contains all that clients of the EJB needs to interact with the EJB. The client view of an EJB consists of the following:

- Business interface(s) of the EJB.
- Superinterface(s) of the EJB's business interfaces.
- Classes and interfaces used as method parameters in the EJB's business interface(s). This includes classes used as members of collections that are method parameters.
- Classes and interfaces used as method return values in the EJB's business interface(s). This includes classes used as members of collections that are method return values.
- Classes and interfaces used as method exceptions in the EJB's business interface(s).

In the module containing the EJB implementation, the location of the EJB-Client JAR file is specified in the ejb-jar.xml deployment descriptor, relative to the archive containing the deployment descriptor.

The deployment descriptor fragment below specifies the location of the EJB-Client JAR file having the name “employee_service_client.jar” as being located at the same location in the containing EAR file as the archive (EJB-JAR or WAR) that the deployment descriptor is contained in:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    ...
    <!--
        The optional <ejb-client-jar> element specifies the location,
        relative to the EJB-JAR or WAR file in which this
        deployment descriptor is contained, in the EAR of the EJB-client
        JAR file.
        The EJB-client JAR contains all the classes and interfaces a
        client needs to access the EJB(s) located in the archive in which
        this deployment descriptor is located.
    -->
    <ejb-client-jar>employee_service-client.jar</ejb-client-jar>
</ejb-jar>
```

The contents of the EJB-Client JAR can be included in one of two ways:

- By reference.
The client view classes and interfaces are contained in the EJB-Client JAR, which is referenced from the EJB-JAR or WAR file containing the EJB implementation class(es) by including the EJB-Client JAR on the classpath.
- By copy.
The client view classes and interfaces are duplicated both in the EJB-Client JAR and in the EJB-JAR or WAR in which the EJB implementation class(es) are located.

13.3. Packaging Requirements

The following things are required to be included in, or referenced from, all the different packaging archives in which EJBs can be distributed:

- EJB implementation class.
- EJB business interface(s).
If EJB(s) have business interfaces.
- Web service endpoint interface(s).
If one or more EJBs are exposed as web services.
- Interceptor classes.
If interceptors are configured.
- Class files for all classes and interfaces the above classes depend on, except JavaSE and JavaEE classes.
This includes, but is not limited to, EJB superclasses, method parameter classes, method result parameter classes, exception classes.

13.4. Module Visibility

Classes and interfaces in JAR files deployed to the lib directory of an EAR file or pointed out using the <library-directory> element in the application.xml deployment descriptor will be made available to all the modules in the EAR.

EJB components in EJB modules in an EAR file are accessible by other components in the EAR without any addition class-path configuration.

Empirical results tell me that components in a web module in an EAR file are not accessible by components outside of the web module.

13.5. The ejb-jar.xml Deployment Descriptor

References: EJB 3.1 Specification, section 19.2 and 19.3.

Descriptions of the different sections of the ejb-jar.xml deployment descriptor are scattered throughout this book and won't be repeated. Instead we'll take a look at responsibilities related to the deployment descriptor and what may and may not be changed immediately prior to deployment. Configuration can be accomplished using either annotations or the ejb-jar.xml deployment descriptor, as we have seen earlier.

The two roles that are likely to manipulate EJB deployment information are:

- Bean provider.
Develops EJBs.
- Application assembler.
Assembles the applications in which the EJBs are used.

The following EJB-related information may be modified when applications containing EJBs are assembled:

- Environment entry values.
- Description fields.
- Relationship names for EJB 2.x beans.
- Message selectors of message driven beans.
- Timer schedule attributes.

The following EJB-related information must not be modified when applications containing EJBs are assembled:

- An EJB's abstract schema name.
Related to entity beans. May be modified if EJB-QL query strings containing schema names are modified accordingly.
- Relationship role source element.
Again, related to entity beans. May also be modified if EJB-QL query strings containing schema names are modified accordingly.

The following EJB-related information may optionally be specified when an application containing EJBs is assembled:

- EJB reference bindings.
Using the <ejb-link> element.
- Message destination links.
See [appendix C](#) for an example on how message destination links are used.
- Security roles.
- Method permissions.
- Singleton session bean initialization ordering.
See section on [singleton session bean initialization and destruction dependencies](#).
- Timeout of stateful session beans.
- Singleton session bean eager initialization.
See section on [singleton session bean eager initialization](#).
- Concurrency access timeouts.
See section on [concurrency management](#).
- Linking security role references.
Security related. See [example on how to configure EJB security using the deployment descriptor](#).
- Invocation security role.
See [example on how to change the invocation security role](#).
- Transaction attributes of business methods.
Applies only to EJBs with container-managed transactions.
- Interceptors.
Interceptor configurations may be changed or overridden, interceptors may be added or reordered.

Part Two

Extras and Refactored Information

14. Session Beans

References: EJB 3.1 Specification, chapter 4.

This chapter does not correspond to any item in the exam topics, but rather it is a common place for information about session beans that otherwise would be scattered throughout the book or duplicated at several places.

Session beans are objects that implement some business logic that runs in an EJB container. Session beans have an identity but are not persistent and does not survive a crash or container restart.

The EJB container provides the following services to session beans, all of which are transparent to clients of the session bean:

- Security.
- Concurrency management.
- Transactions.
- Swapping to secondary storage.
For stateful session beans.

14.1. Session Beans in the Deployment Descriptor

Despite fashionable options such as annotations, it is still possible to define an EJB using only the ejb-jar.xml deployment descriptor and a Java class completely free from annotations, as shown in this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <!--
            Minimal configuration to define a stateless session bean
            with a no-interface view using only the deployment
            descriptor and no annotations.
        -->
        <session>
            <ejb-name>DDConfigedStatelessBean</ejb-name>
            <local-bean/>
            <ejb-class>com.ivan.scbcd6.DDConfigedStatelessBean</ejb-class>
            <session-type>Stateless</session-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

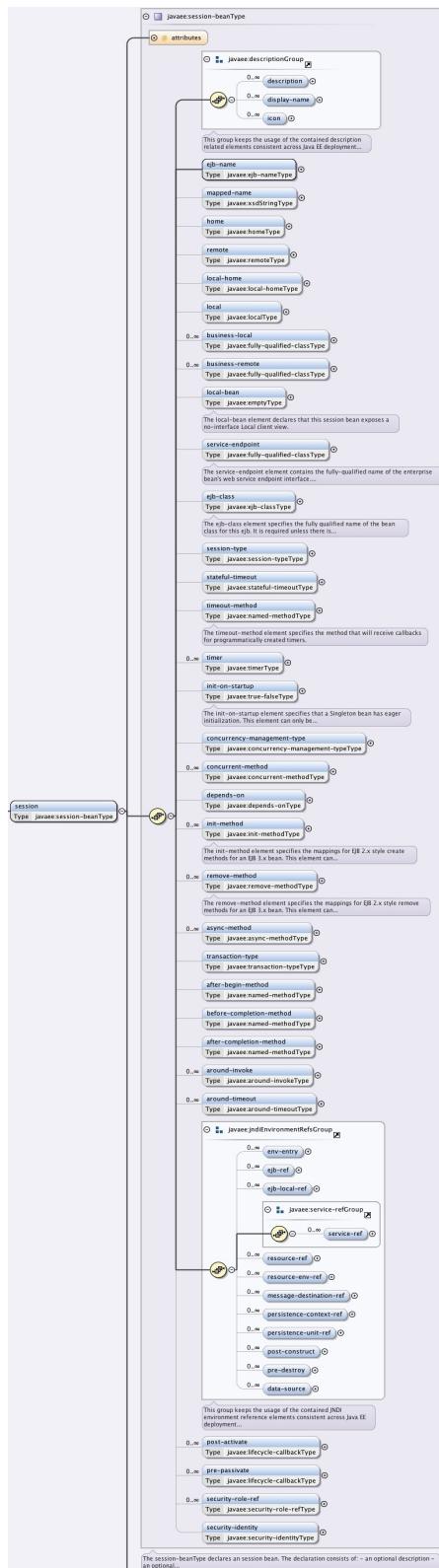
For the curious, the corresponding class that implements the DDConfigedStatelessBean looks like this:

```
package com.ivan.scbcd6;

public class DDConfigedStatelessBean
{
    public String sayHello()
    {
        return "Hello!";
    }
}
```

EJBs defined in the deployment descriptor are equal to annotated EJBs. For instance, they are also eligible for dependency injection using annotated fields or methods in the EJB implementation class.

The following XML schema fragment describes configuration options available when defining a session bean in the ejb-jar.xml deployment descriptor:



Different sections of the deployment descriptor will be discussed further in relevant sections.

14.2. Session Bean Types

There are three types of session beans:

- Stateless
The bean does not contain state related to any particular client. Any instance serves any client.
- Stateful
The bean may contain state related to a particular client. A one-to-one relationship exists between client and the bean that serves the requests of the client.
- Singleton
A single instance shared by all clients. Allows for bean or container managed concurrent access.

14.3. Session Bean Metadata

Session bean metadata is data that specifies the type of a session bean, the view(s) it makes available, the transaction policy, the life-cycle methods of the bean etc. Such data may be specified using either annotations or the ejb-jar.xml deployment descriptor.

14.3.1. Session Bean Type Metadata

To let the container know that a class we have written is to become a session bean, we can either use the `@Stateful`, `@Stateless` or the `@Singleton` annotation or a deployment descriptor entry.

The `@Stateful`, `@Stateless` or the `@Singleton` are to annotate the class implementing the session bean. They share the following optional elements:

Optional Element	Description
description	Description of the session bean. For human consumption.
mappedName	Can be used to specify vendor-specific deployment information. Use of this element is non-portable.
name	EJB-name of the bean. Defaults to unqualified name of the class the annotation is applied to.

An example of a deployment-descriptor fragment defining a stateful session bean with a local business interface:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
    version="3.1"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatefulSession1</ejb-name>
            <local>com.ivan.scbcd6.StatefulSession1Local</local>
            <ejb-class>com.ivan.scbcd6.StatefulSession1Bean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    ...
</ejb-jar>
```

To choose between the different types of session bean in the deployment descriptor, use the `<session-type>` element with one of the following possible values:

- Singleton
- Stateful
- Stateless

14.3.2. Session Bean View Metadata

The @LocalBean, @Local and @Remote annotations are used to specify the view(s) made available by a session bean.

- **@LocalBean**
Annotates the session bean implementation class that is to present a no-interface local view.
- **@Local**
Annotates either the session bean implementation class or the local business interface(s) of a session bean.
When annotating the session bean implementation class, the annotation declares the local business interface(s) of the session bean. The value of the annotation lists the local business interface(s).
When annotating one or more interfaces, it designates the interface(s) as a local business interface(s). No value need to be provided.
- **@Remote**
Annotates either the session bean implementation class or the remote business interface(s) of a session bean.
When annotating the session bean implementation class, the annotation declares the remote business interface(s) of the session bean. The value of the annotation lists the remote business interface(s).
When annotating one or more interfaces, it designates the interface(s) as a remote business interface(s). No value need to be provided.

In the deployment descriptor, the <local-bean>, <local> and <remote> elements can be used to specify the different kinds of views. See example in the previous section.

14.3.3. Session Bean Life-Cycle Annotations

References: EJB 3.1 Specification, sections 4.3.4 and 4.6.3..

Annotations can be used to define methods that are to be invoked in connection to different events in the life-cycle of a session bean:

Annotation	Comments
@PostConstruct	Invoked after a bean instance has been created and before it is taken into use. Allows for initialization of the bean instance.
@PreDestroy	Invoked when the EJB instance is to be taken out of service by the container. Allows for cleanup and release of resources used by the instance. Not invoked if the EJB instance throws a system exception, if the container crashes or if timeout occurs when a stateful session bean is passivated.
@PrePassivate	Invoked immediately prior to a stateful session bean being passivated. Allows for release of resources used by the active bean instance and handling of non-serializable references of the bean prior to serialization (as part of the bean being passivated).
@PostActivate	Invoked immediately after a stateful session bean being activated. Allows for obtaining resources used by the active bean instance and setting of non-serializable references prior to the bean being used after activation.

The above annotations available in EJB 3.0 and later eliminates the need for implementing the *javax.ejb.SessionBean* interface with its lifecycle callback methods.

14.4. Session Bean Concurrency

In this section, an example program showing concurrency behaviour of stateful and stateless session beans will be examined. The concurrency behaviour of singleton session beans is discussed extensively in the [Singleton Session Bean Concurrency](#) section in chapter four.

14.4.1. Stateful Session Bean Concurrency

First the example program showing the concurrent behaviour of stateful session beans is implemented. We'll later modify this program slightly to look at the concurrent behaviour of stateless session beans.

- Create a dynamic web project in Eclipse, as described in [appendix A](#). I call my project “SessionBeanConcurrentAccess”.
- In the package *com.ivan.scbcd6*, create the session bean implementation class according to the following listing:

```
package com.ivan.scbcd6;

import javax.ejb.Stateful;

/**
 * This class implements a session bean on which concurrent
 * access will be tested.
 */
@Stateful
public class SessionBeanA
{
    public void slowMethod()
    {
        System.out.println("SessionBeanA - Entering slowMethod " +
                           this);

        waitSomeTime(10);

        System.out.println("SessionBeanA - Exiting slowMethod " +
                           this);
    }

    public void fastMethod()
    {
        System.out.println("SessionBeanA - Entering fastMethod " +
                           this);

        waitSomeTime(1);

        System.out.println("SessionBeanA - Exiting fastMethod " +
                           this);
    }

    private void waitSomeTime(final long inSecondsDelay)
    {
        try
        {
            Thread.sleep(1000L * inSecondsDelay);
        } catch (InterruptedException e)
        {
            // Ignore exceptions.
        }
    }
}
```

- In the package `com.ivan.scbcd6.client`, create the servlet that will act as a client to the session bean:

```

package com.ivan.scbcd6.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ivan.scbcd6.SessionBeanA;

/**
 * Servlet acting as a client of the session bean.
 */
@WebServlet(name = "SessionBeanClientServlet", urlPatterns = "/test.do")
public class SessionBeanClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @EJB
    private SessionBeanA mSessionBean;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    @Override
    protected void doGet(HttpServletRequest inRequest,
                         HttpServletResponse inResponse) throws ServletException, IOException
    {
        System.out.println("**** Entering SessionBeanClientServlet");

        testConcurrentAccess();

        System.out.println("**** Exiting SessionBeanClientServlet");

        /* Display a message on the web page. */
        PrintWriter theResponseWriter = inResponse.getWriter();
        theResponseWriter.println("Finished invoking session bean " +
                               "concurrency test.");
    }

    private void testConcurrentAccess()
    {
        System.out.println("**** Entering testConcurrentAccess");

        /*
         * Prepare the two threads from which the different methods
         * of the stateful session bean will be called.
         */
        Thread theSlowMethodThread = new Thread()
        {
            @Override
            public void run()
            {
                mSessionBean.slowMethod();
            }
        };

        Thread theFastMethodThread = new Thread()
        {
            @Override
            public void run()
            {
                mSessionBean.fastMethod();
            }
        };

        System.out.println("      Calling slowMethod... ");
    }
}

```

```

        theSlowMethodThread.start();
        System.out.println("    Calling fastMethod... ");
        theFastMethodThread.start();

        System.out.println("**** Exiting testConcurrentAccess");
    }
}

```

- Deploy the project to the GlassFish application server.
- In a browser issue a request to the following URL:
<http://localhost:8080/SessionBeanConcurrentAccess/test.do>
 You should see the following console output:

```

INFO: *** Entering SessionBeanClientServlet
INFO: *** Entering testConcurrentAccess
INFO:     Calling slowMethod...
INFO:     Calling fastMethod...
INFO: SessionBeanA - Entering slowMethod
com.ivan.scbcd6._SessionBeanA_Serializable@f684bc0
INFO: *** Exiting testConcurrentAccess
INFO: *** Exiting SessionBeanClientServlet

INFO: SessionBeanA - Exiting slowMethod
com.ivan.scbcd6._SessionBeanA_Serializable@f684bc0
INFO: SessionBeanA - Entering fastMethod
com.ivan.scbcd6._SessionBeanA_Serializable@f684bc0

INFO: SessionBeanA - Exiting fastMethod
com.ivan.scbcd6._SessionBeanA_Serializable@f684bc0

```

Note that:

- The session bean is stateful.
- The execution of the method *slowMethod* does not overlap with the execution of the *fastMethod*.
- Both the *fastMethod* and the *slowMethod* are invoked on the same session bean instance.
- If we try to annotate the session bean class with the `@ConcurrencyManagement(ConcurrencyManagementType.BEAN)` annotation, we can see that the behaviour of the bean does not change.

The conclusion is that bean-managed concurrency does not apply to stateful session beans.

For details on the `ConcurrencyManagement` annotation, please see the section on [Concurrency Management](#) in the chapter on singleton session beans.

14.4.2. Stateless Session Bean Concurrency

To look at the concurrent behaviour of stateless session beans, we simply change the annotation on the bean class in the example program from the previous section:

- Change the annotation on the *SessionBeanA* class to `@Stateless`:

```
...  
@Stateless  
public class SessionBeanA  
{  
...  
}
```

- Deploy the project to the GlassFish application server.
- In a browser issue a request to the following URL:
<http://localhost:8080/SessionBeanConcurrentAccess/test.do>

You should see the following console output:

```
INFO: *** Entering SessionBeanClientServlet  
INFO: *** Entering testConcurrentAccess  
INFO: Calling slowMethod...  
INFO: Calling fastMethod...  
INFO: *** Exiting testConcurrentAccess  
INFO: *** Exiting SessionBeanClientServlet  
INFO: SessionBeanA - Entering fastMethod com.ivan.scbcd6.SessionBeanA@75225d4a  
INFO: SessionBeanA - Entering slowMethod com.ivan.scbcd6.SessionBeanA@30a7a93a  
INFO: SessionBeanA - Exiting fastMethod com.ivan.scbcd6.SessionBeanA@75225d4a  
INFO: SessionBeanA - Exiting slowMethod com.ivan.scbcd6.SessionBeanA@30a7a93a
```

Note that:

- The session bean is stateless.
- The execution of the method *slowMethod* overlaps with the execution of the *fastMethod*.
- The *fastMethod* and the *slowMethod* are invoked on different session bean instances.
- We can conclude that the container does not allow concurrent access of a stateless session bean instance. If a stateless session bean instance is busy serving a request (*fastMethod*), another instance is fetched from the pool and the second request (*slowMethod*) is invoked on the second instance.
- If we try to annotate the session bean class with the `@ConcurrencyManagement(ConcurrencyManagementType.BEAN)` annotation, we can see that the behaviour of the bean does not change.

The conclusion is that bean-managed concurrency does not apply to stateless session beans. For details on the `ConcurrencyManagement` annotation, please see the section on [Concurrency Management](#) in the chapter on singleton session beans.

14.5. Session Bean Initialization

References: EJB 3.1 Specification, section 3.4.4.

The EJB 3.1 specification recommends any initialization code of a session bean instance to be placed in a method annotated with the `@PostConstruct` annotation and not in a constructor of the bean implementation class.

15. EJB Dependency Injection and Resource Lookup

References: EJB 3.1 Specification, chapter 16.

This chapter describes dependency injection and resource lookup facilities available to all kinds of EJBs. Again, this is not an exam topic in itself, but rather a place to collect information that applies to session beans as well as message driven beans.

An EJB instance can have various kinds of data or references injected into annotated fields using dependency injection.

Type of Dependency Injection	Examples	Comments
Simple Environment Entries	<pre>@Resource int myConfigParam = 5; @Resource public void setMyConfigParam(int inMyConfigParam) ... </pre>	
EJB References	<pre>@EJB private MySuperBean mMyBean; @EJB public void setMyBean(MySuperBean inMyBean) ... </pre>	
Web Service References	<pre>@WebServiceRef MyService serviceRef;</pre>	
Resource Manager Connection Factory Resources	<pre>@Resource javax.sql.DataSource myAppDB;</pre>	
Resource Environment References	<pre>@Resource SomeRsrcEnvRef myRsrcEnvRef;</pre>	
Message Destination References	<pre>@Resource javax.jms.Queue workQueue;</pre>	
Persistence Unit References	<pre>@PersistenceUnit(unitName="WarehouseManagement") EntityManagerFactory warehouseEMF;</pre>	
Persistence Context References	<pre>@PersistenceContext(type=EXTENDED) EntityManager myEntityMgr;</pre>	
UserTransaction Interface	<pre>@Resource UserTransaction transaction;</pre>	Only available to session- and message-driven beans with bean-managed transactions.
ORB References	<pre>@Resource ORB corbaOrg;</pre>	
TimerService References	<pre>@Resource TimerService mTimerService;</pre>	
EJBContext References	<pre>@Resource SessionContext mSessionBeanContext;</pre>	

15.1. The @Resource Annotation

Commonly used for injection or declaration of various things is the @Resource annotation. This annotation may be applied to:

- A class.
Declaring a resource that the application will look up at runtime.
- An instance field.
Declaring a field, with any visibility modifier, that is to be injected with the specified resource when an instance of the class is initialized.
- A setter method.
Declares a method that will set specified resource when an instance of the class is initialized.

The @Resource annotation has the following optional elements:

Element Name	Comments
authenticationType	Who is to be responsible for authentication for this resource; the container or application. Possible values: Resource.AuthenticationType.CONTAINER, Resource.AuthenticationType.APPLICATION Default value: CONTAINER
description	Description of the resource. For human consumption.
mappedName	Product specific, non-portable, name the resource is to be mapped to. Default: “”
name	JNDI name of the resource. Default: “”
shareable	Boolean indicating whether this resource can be used simultaneously by more than one component (EJB). Default: true
type	Java type of the resource. Default: java.lang.Object

For some resource types, the *shareable* and *authenticationType* elements must not be used, since resources of that type does not require authentication and are not shareable.

15.2. The @EJB Annotation

The @EJB annotation can be applied at the following targets with the corresponding result:

Target of @EJB Annotation	Effect
Class	Declares an entry in the annotated beans environment. This use corresponds to EJB references in the deployment descriptor.
Method	Indicates a dependency on an EJB which will be injected by invoking the annotated method.
Instance field	Indicates a dependency on an EJB which will be injected by setting the instance field.

Entries declared in the annotated beans environment using an @EJB annotation is to be looked up by the bean using either the *EJBContext.lookup* method or the JNDI API.

Optional elements of the @EJB annotation are:

Element	Type	Description
beanInterface	Class	Holds one of the following types of the target EJB (with DI: the EJB to be injected): Local business interface, remote business interface, bean class (no-interface view), local home interface (EJB 2.1), remote home interface (EJB 2.1).
beanName	String	The EJB name of the EJB to which this reference is mapped. With dependency injection, the name of the EJB which reference will be injected.
description	String	
lookup	String	The portable JNDI name of the target EJB (with DI: the EJB to be injected).
mappedName	String	Product-specific, non-portable, name of the EJB to which this reference is mapped.
name	String	Logical name of the EJB reference within the java:comp/env environment.

When using the @EJB annotation to perform dependency injection, either the *beanName* or the *lookup* element, but not both elements, can be used to resolve the dependency.

15.2.1. Declaring JNDI Names with the @EJB Annotation

In the first example an @EJB annotation is used to declare the “java:comp/env/ejb/BookingRecord” JNDI environment entry referring to the home interface of another EJB. This JNDI name can later be used to look up a reference to the home interface from within the annotated EJB implementation class.

```
@EJB(name="ejb/BookingRecord", beanInterface=BookingRecordHome.class)
@Stateless
public class BookingServiceBean implements BookingService
{
    ...
}
```

15.2.2. Setter Method Injection with the @EJB Annotation

In the second example, a setter method setting a reference to another EJB is annotated. The method will be invoked after an instance of the EJB implementation class is created and before it is taken into use. Dependency injection takes place prior to any @PostConstruct method is being invoked. As an alternative to the *beanName* element in the @EJB annotation, one of the *lookup* or *name* elements can be used specifying either the full JNDI name or the JNDI name relative to “java:comp/env” used to look up the bean reference to be injected. In the case of no ambiguity, all elements can be omitted.

```
...
@Stateful
@LocalBean
public class StatefulSession1Bean
{
    ...

    @EJB(beanName="DummyStatelessBean")
    public void setDummyReference(DummyStatelessBean inBeanReference)
    {
        // Store the reference to the injected bean.
    }
    ...
}
```

15.2.3. Instance Field Injection with the @EJB Annotation

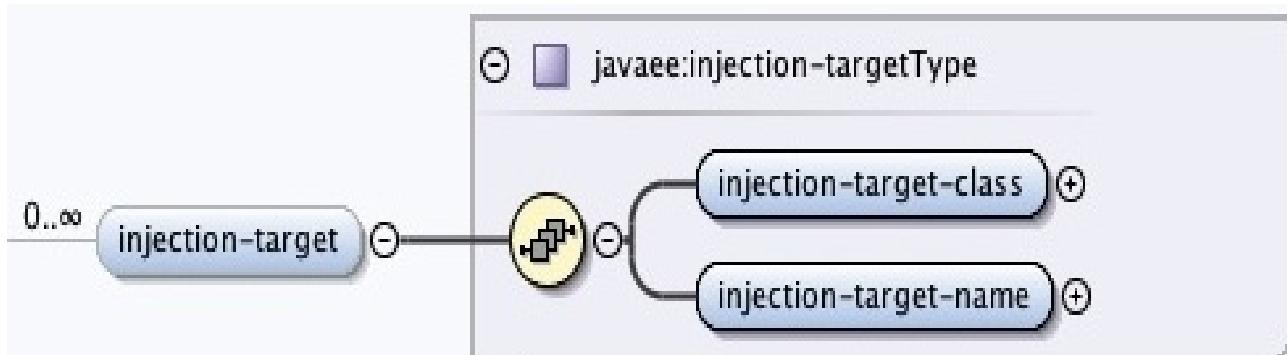
In the final example, an instance field holding a reference to another EJB is annotated. Dependency injection into instance fields also takes place prior to the invocation of any @PostConstruct method. As in the case of annotating a setter-method, the *beanName*, *name* or *lookup* elements of the @EJB annotation may be used to specify which reference to inject. Again, in the case of no ambiguity, all elements can be omitted.

```
@Stateful
@LocalBean
public class StatefulSession1Bean
{
    @EJB
    private DummyStatelessBean mDummyStatelessRef;
    ...
}
```

15.3. Injection Using the Deployment Descriptor

Injection of resource references can be accomplished without annotations, using the ejb-jar.xml deployment descriptor. Since this is common to all the different resource reference types, it is described in this, separate, section.

The <injection-target> element appears as a child element of an element defining a resource reference and specifies the class and instance field name in which the resource reference is to be injected. The fragment of the XML schema defining the <injection-target> element looks like this:



The fragment from the ejb-jar deployment descriptor XML schema defining the <injection-target> element and child elements, which are used to perform dependency injection from the deployment descriptor.

Example:

Define the field to be injected in the EJB. Note that the field is not annotated.

```
...
/* Simple Environment Entry Injection without Annotations. */
private int turboCounter = 99;
...
```

Specify the value to inject and where in the EJB the value is to be injected in the ejb-jar.xml deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatefulSession2Bean</ejb-name>

            ...
            <env-entry>
                <env-entry-name>turboCounter</env-entry-name>
                <env-entry-type>java.lang.Integer</env-entry-type>
                <env-entry-value>15</env-entry-value>

                <!--
                    Specifies where to inject the value/resource reference.
                -->
                <injection-target>
                    <injection-target-class>
                        com.ivan.scbcd6.StatefulSession2Bean
                    </injection-target-class>
                    <injection-target-name>
                        turboCounter
                    </injection-target-name>
                </injection-target>
            </env-entry>
        </session>
    </enterprise-beans>

```

```
</session>
</enterprise-beans>
</ejb-jar>
```

Since a value was provided as an initialization value of the instance field, this is the value the field will have if it is not injected with a value in the ejb-jar.xml deployment descriptor. Be aware that if no value is provided in the deployment descriptor, then trying to look up the value using the JNDI API will fail with an error.

In the above example, the message destination reference is injected into the instance field *turboCounter* in the class *com.ivan.scbcd6.StatefulSession2Bean*.

With resource references, the type of the instance field must match the resource reference type.

15.4. Programmatic Lookup of References from EJBs

Another thing common to different kinds of resources is programmatic lookup. In order to be able to perform programmatic lookup, a reference has to be specified in the ejb-jar.xml deployment descriptor or a JNDI name has to be assigned to the resource in an annotation.

When looking up a resource using its JNDI name, there are two options for performing a lookup from within an EJB:

- Creating and using an instance of *InitialContext*.
- Using an *EJBContext* being injected into the EJB.

Below are examples of how to use each method to lookup a resource from within an EJB.

15.4.1. Lookup Using InitialContext

Looking up a JNDI name using an instance of *InitialContext* is quite simple. The example below is from an EJB running in the GlassFish v3 application server:

```
Context theInitialContext = new InitialContext();
Context theJavaCompEnv = (Context)theInitialContext.lookup("java:comp/env");
ReferencedLocal theLocalRef =
    (ReferencedLocal)theJavaCompEnv.lookup("ejb/AnotherBean");
```

In the above example, an *InitialContext* is created and, using that context, the “java:comp/env” sub-context is looked up. The sub-context is then used to look up a reference to an EJB implementing the *ReferencedLocal* interface.

References may, of course, be looked up using the *InitialContext* instance directly, eliminating the step in which the sub-context is retrieved.

15.4.2. Lookup Using EJBContext

When using an `EJBContext` to look up resource references, we must first see to that an instance of the context is injected into the EJB in which we want to perform the lookup. This is accomplished by creating an instance field and annotating it with the `@Resource` annotation:

```
@Stateful  
public class StatefulSession2Bean  
{  
    @Resource  
    private SessionContext mSessionBeanContext;  
    ...
```

There are four different types of EJB contexts that can be injected.

EJB Context Type	Can Be Injected Into	Comments
<code>EJBContext</code>	Session beans, message driven beans, entity beans	Basic EJB context type, usable with any type of EJB.
<code>EntityContext</code>	Entity beans.	Contains additional methods for, for example, retrieval of an entity's primary key.
<code>MessageDrivenContext</code>	Message driven beans.	No additional functionality compared to <code>EJBContext</code> .
<code>SessionContext</code>	Session beans.	Contains additional methods for, for example, retrieval of invoked business interface, cancellation of asynchronously invoked methods on the EJB.

For complete details on the different context-types, please refer to the EJB 3.1 API documentation.

Available in the `EJBContext`, and thus in all types of contexts, is the `lookup` method, which we can use to look up a resource references. Three things should be taken into consideration when using the `EJBContext.lookup` method:

- Narrowing references to remote homes and remote interfaces using the `PortableRemoteObject.narrow` method is not needed.
- If looking up names referring to entries within the private component namespace, for instance the reference to the local view of an EJB, then the “`java:comp/env/`” part of the JNDI name can be omitted.
- As a consequence of the previous point; when looking up entries not within the private component namespace, then entire JNDI names must be used.

Finally, we can look at some examples of lookups made using an `EJBContext`:

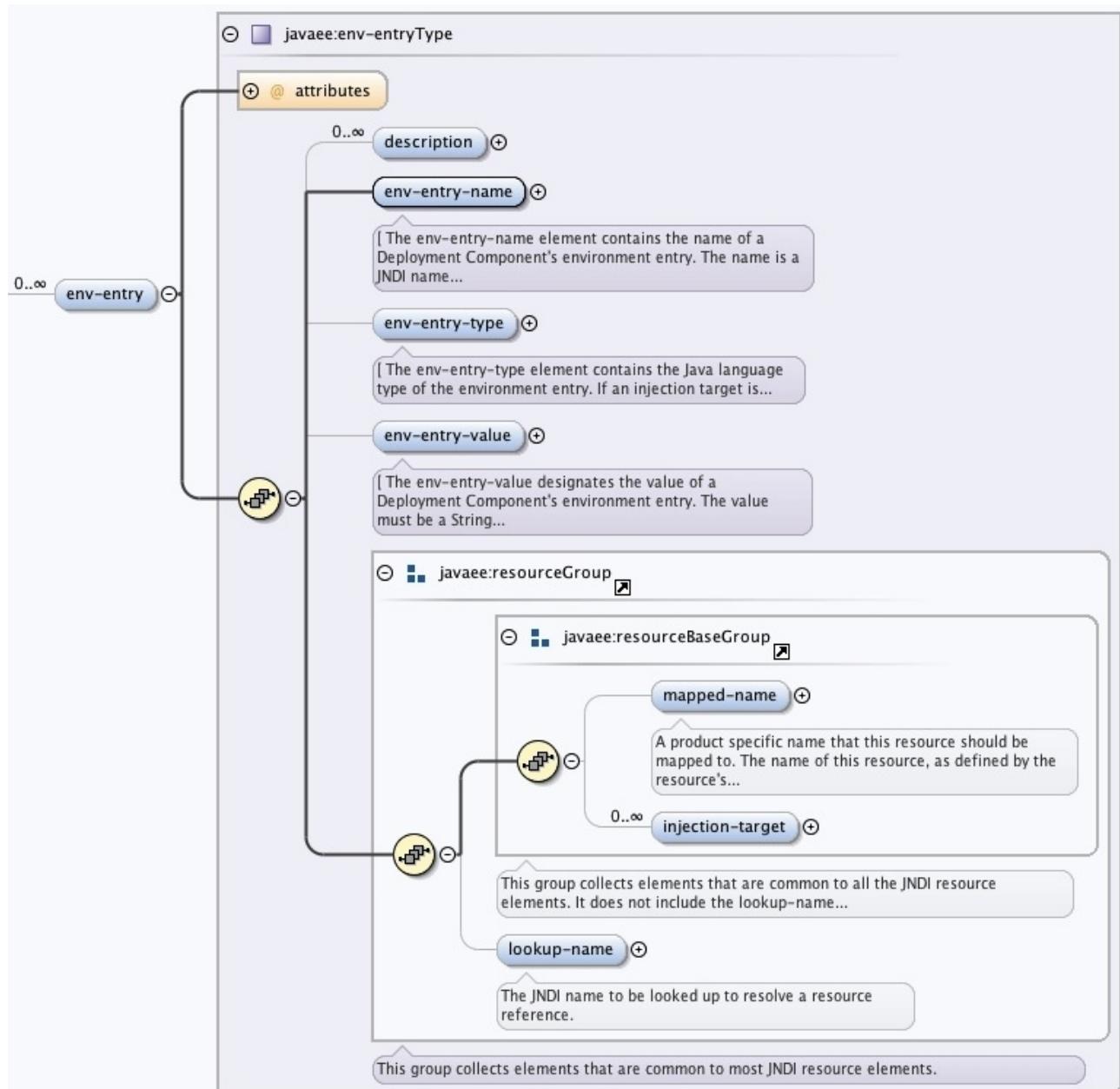
```
Integer theTurboCounter = (Integer)mSessionBeanContext.lookup( "turboCounter" );  
ReferencedLocal theLocalRef =  
    (ReferencedLocal)mSessionBeanContext.lookup( "AnotherReferencedBeanRef" );
```

Both the above references were declared in the ejb-jar.xml deployment descriptor; the first using an `<env-entry>` element and the second using an `<ejb-local-ref>` element.

15.5. Simple Environment Entries

References: EJB 3.1 Specification, section 16.4.

Simple environment entries are values of one of the following types: *String*, *Character*, *Integer*, *Boolean*, *Double*, *Byte*, *Short*, *Long*, *Float*. Values to be injected are defined in the deployment descriptor using `<env-entry>` elements. The XML schema fragment defining the `<env-entry>` element looks like this:



The fragment from the ejb-jar deployment descriptor XML schema defining the `<env-entry>` element, which is used to define simple environment entries.

15.5.1. Simple Environment Entry Injection Using Annotations

When using the `@Resource` annotation to annotate an instance field or setter-method to facilitate injection of simple environment entries, the `authenticationType` and `shareable` elements of the annotation must not be used, since simple environment entries neither require authentication, nor are they shareable.

Example:

Annotate the field to be injected in the EJB:

```
... /* Simple Environment Entry Injection. */
@Resource(name="turboFlag")
private boolean mTurboFlag;
...
```

A value must also be specified in the deployment descriptor, as showed in the next section.

15.5.2. Defining Simple Environment Entries in the Deployment Descriptor

Specify the value to inject in the ejb-jar.xml deployment descriptor using an `<env-entry>` element in the bean that the value is to be injected into:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatefulSession2Bean</ejb-name>

            <!-- Simple environment entry. -->
            <env-entry>
                <env-entry-name>turboFlag</env-entry-name>
                <env-entry-type>java.lang.Boolean</env-entry-type>
                <env-entry-value>true</env-entry-value>
            </env-entry>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Specifying the instance field in which the environment entry is to be injected into can also be specified in the deployment descriptor using the `<injection-target>` element, as [previously](#) described.

15.5.3. Programmatic Lookup of Simple Environment Entries

Finally, simple environment entries can be looked up as described in the section on [Programmatic Lookup of References from EJBs](#) above.

15.6. EJB References

References: EJB 3.1 Specification, section 16.5.

EJB references are references to business interfaces, no-interface views or home interfaces of other EJBs.

15.6.1. Reference Injection Using Annotations

Injection of references to other EJBs can be injected into an EJB into fields annotated using the `@EJB` annotation. Detailed information on the `@EJB` annotation can be found in section [5.3.1](#). When injecting an EJB reference, a field or setter method is annotated. The type of the field or the type of the parameter of the setter method can be a no-interface view, a local or remote business interface.

Example:

If we want to provide a custom name of the EJB which reference is to be injected, we use the optional `name` element in the `@Stateful` or `@Stateless` annotation:

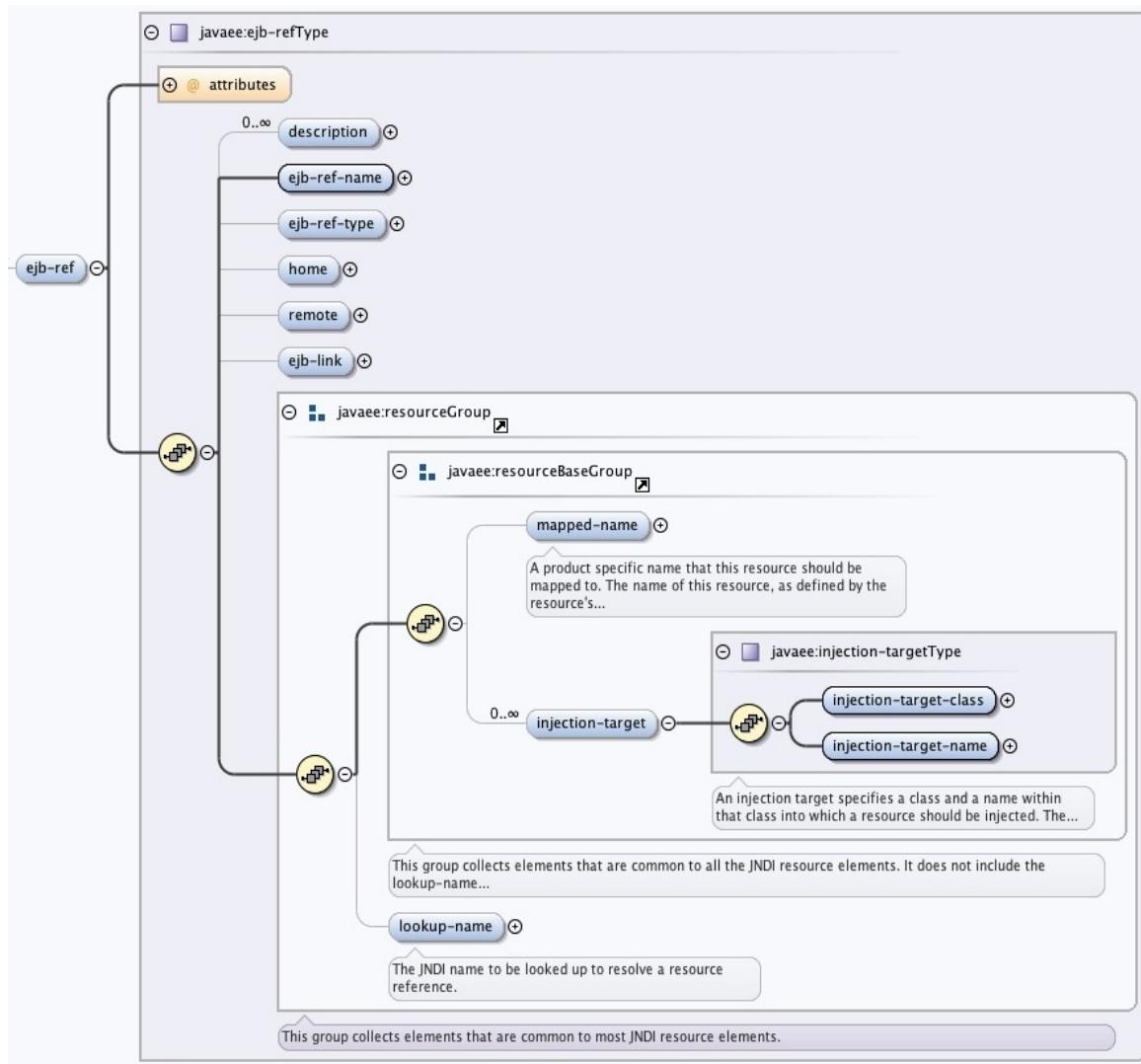
```
...
@Stateless(name="ReferencedBean")
public class ReferencedBean implements ReferencedLocal {
...
```

The field or setter method to be injected is annotated with the `@EJB` annotation, specifying the name of the EJB which reference to be injected using the `beanName` optional element:

```
...
/* EJB Reference Injection. */
@EJB(beanName="ReferencedBean")
private ReferencedLocal mReferencedBean;
...
```

15.6.2. Defining EJB References in the Deployment Descriptor

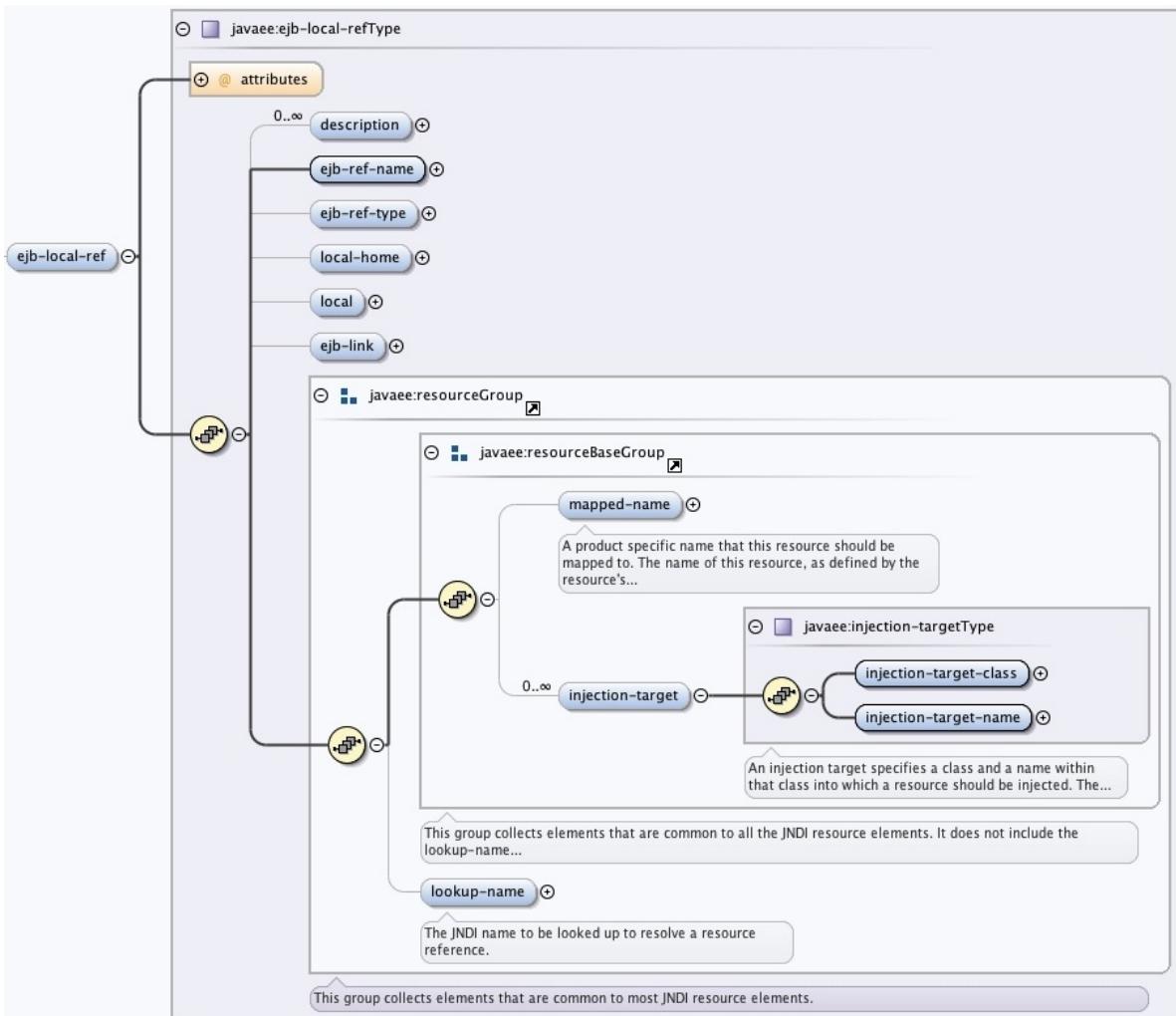
EJB references can be defined in the deployment descriptor using the `<ejb-ref>` or `<ejb-local-ref>` elements. The XML schema fragment defining the `<ejb-ref>` element looks like this:



The XML schema fragment defining the `<ejb-ref>` element,
which is used to define local and remote EJB references.

Annotating a class with the [@EJB annotation](#) can also be used to declare references to other EJBs.

The XML schema defining the <ejb-local-ref> looks this this:



The fragment from the ejb-jar deployment descriptor XML schema defining the `<ejb-local-ref>` element, which is used to define local EJB references.

Example:

In the ejb-jar.xml deployment descriptor, the following configuration is made for a reference to an EJB with a local view:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

  <enterprise-beans>
    <session>
      <!--
          Supplying the name of the EJB which we want override the
          annotation configuration of is enough.
      -->
      <ejb-name>StatefulSession2Bean</ejb-name>

      ...
      <ejb-local-ref>
        <!--
            Desired JNDI name. "java:comp/env/" is automatically
            prefixed. Must be specified.
        -->
        <ejb-ref-name>AnotherReferencedBeanRef</ejb-ref-name>
        <!-- Specifies name of EJB to inject reference to. -->
        <ejb-link>AnotherReferencedBean</ejb-link>
        <!--
            Instead of using the <ejb-link> element above, the following
            element can be used to determine which EJB's reference to
            inject (the two elements are mutually exclusive):
            <lookup-name>
                java:global/EJBDependencyInjection/AnotherReferencedBean
            </lookup-name>
        -->
        </ejb-local-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

The EJB reference defined above can then, for instance, be programmatically looked-up, as described in the next section.

As with other references, it is also possible to specify the instance field in which the EJB reference is to be injected into using the `<injection-target>` element in the ejb-jar.xml deployment descriptor, as [previously](#) described.

15.6.3. Programmatic Lookup of EJB References

EJB References can be looked up as described in the section on [Programmatic Lookup of References from EJBs](#) above.

15.6.4. Overriding an @EJB Annotation with the Deployment Descriptor

The following rules exist regarding overriding an @EJB annotation with a deployment descriptor:

- The *name* element in the @Stateless, @Stateful, @Singleton or @MessageDriven annotation is used to match an entry in the deployment descriptor with the same name in the <ejb-name> element.
This is only used by the container to find the appropriate deployment descriptor fragment and does not override anything.
- The type specified in the deployment descriptor's <remote>, <local>, <remote-home> or <local-home> elements and any bean referenced by the <ejb-link> element must be assignable to the type specified by:
 - The @EJB annotation's *beanInterface* element.
 - The type of the instance field annotated by the @EJB annotation.
 - The parameter type of the setter-method annotated by the @EJB annotation.Put more clearly: You can only substitute an EJB being injected into a target by another EJB that can be assigned to the same field or is compatible with the parameter type of the setter method in question.
- Any description in the deployment descriptor overrides the *description* element in the @EJB annotation.
- If an <injection-target> is specified in the deployment descriptor, then it must name exactly the instance field or setter-method annotated with the @EJB annotation.

15.7. Web Service References

References: EJB 3.1 Specification, section 16.6.

Using the `@WebServiceRef` or `@WebServiceRefs` annotation, one or more references to web services may be injected into EJBs.

15.8. Resource Manager Connection Factory References

References: EJB 3.1 Specification, section 16.7.

A resource manager connection factory, for instance an object implementing `javax.sql.DataSource`, is a factory that creates connections, for instance `java.sql.Connection` objects, to a resource manager.

References to such connection factories are special entries in the EJB environment.

15.8.1. Reference Injection Using Annotations

Using the `@Resource` annotation on an instance field or a setter-method allows for references to resource manager connection factories to be injected into EJBs.

The `authenticationType` element can be used to specify whether the container or the application is to be responsible for authentication. The `shareable` element indicates whether connections obtained from the factory can be shared or not.

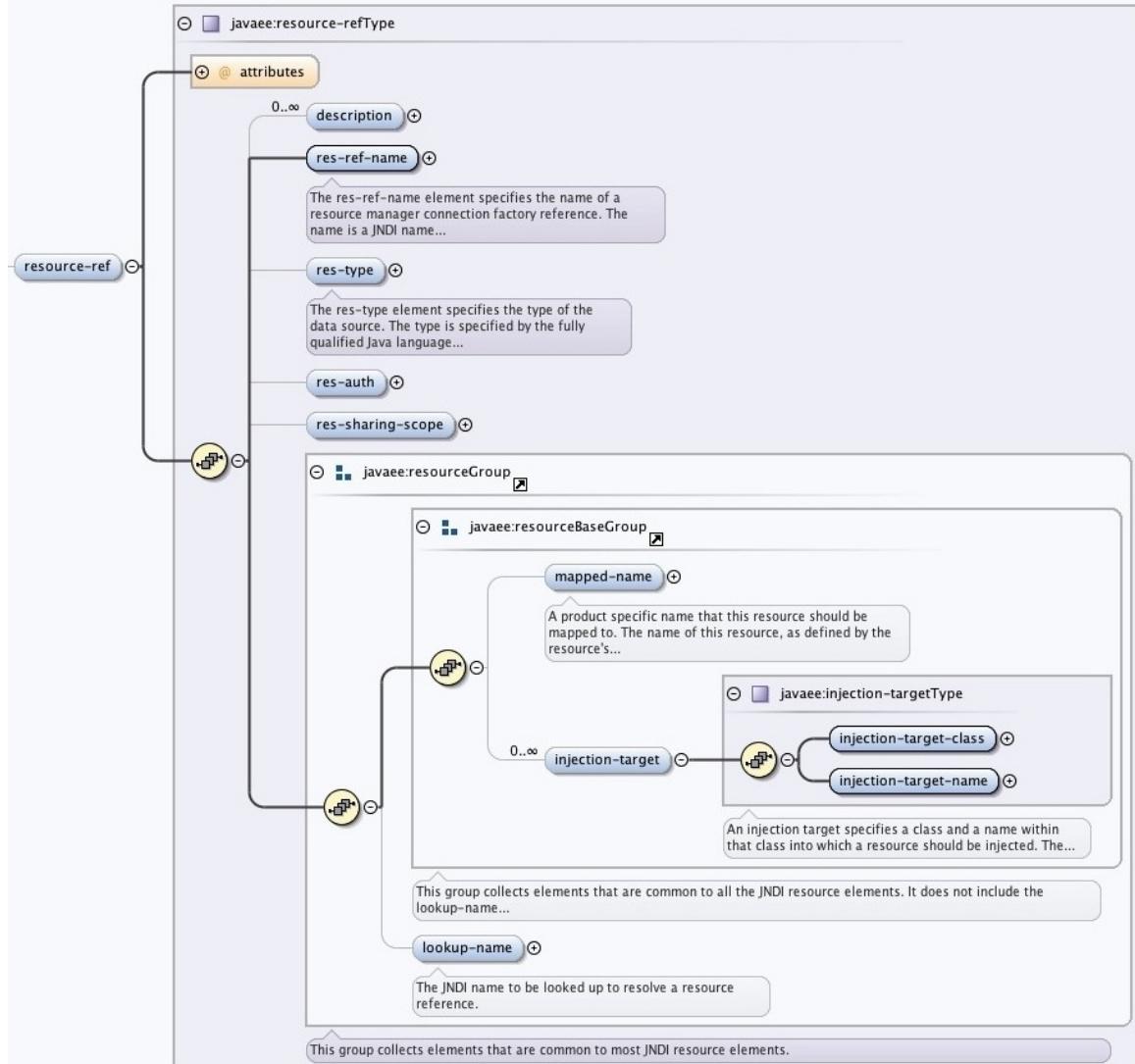
Example:

```
@Stateless
public class WarehouseInventoryBean
{
    ...
    /* Warehouse DB connection factory. */
    @Resource(name="jdbc/WarehouseAppDB")
    javax.sql.DataSource mWarehouseAppDB;
    ...

    ...
    public int countItems(itemId inItemId)
    {
        ...
        java.sql.Connection theConnection = mWarehouseAppDB.getConnection();
        /* Now we can query the database. */
        ...
    }
}
```

15.8.2. Defining Resource Manager Connection Factory References in the Deployment Descriptor

Resource manager connection factory references are defined in the ejb-jar.xml deployment descriptor using <resource-ref> elements. The XML schema fragment specifying the structure of the <resource-ref> element looks like this:



The fragment from the ejb-jar deployment descriptor XML schema defining the <resource-ref> element, which is used to define resource manager connection factory references.

The following listing shows an example of an ejb-jar.xml deployment descriptor with a resource manager connection factory reference declared in the StatefulSession2Bean session bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatefulSession2Bean</ejb-name>
            <!--
                Resource manager connection factory reference to
                a database (JDBC connection pool).
            -->
            <resource-ref>
                <res-ref-name>jdbc/WarehouseAppDB</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <!--
                    Configures how resource manager authentication is configured:
                    - "Container" = the deployer configures the resource manager factory
                    with authentication information. EJBs using the resource manager
                    factory does not need to provide authentication information.
                    - "Application" = the EJB programmatically authenticates with
                    the resource manager factory.
                -->
                <res-auth>Container</res-auth>
                <!--
                    A resource connection can be Shareable or Unshareable,
                    which indicates whether a connection obtained from the
                    connection factory can be shared by other EJBs in the same
                    application using the same resource in the same transaction
                    context.
                    Default is Shareable.
                -->
                <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
            ...
        </session>
    </enterprise-beans>
</ejb-jar>
```

As with other references, it is also possible to specify the instance field in which the resource manager connection factory reference is to be injected into using the <injection-target> element in the ejb-jar.xml deployment descriptor, as [previously](#) described.

15.8.3. Programmatic Access of Resource Manager Connection Factory References

In order to be able to retrieve and use a resource manager connection factory programmatically from within the code of an EJB, the bean provider must follow this procedure:

- Declare a resource manager connection factory reference in the ejb-jar.xml deployment descriptor, as in the example in the above section.
- It is recommended, but not mandated, that resource manager connection factory references are organized into different subcontexts, depending on the resource type. For instance:
 - JDBC DataSource declared in the “java:comp/env/jdbc” subcontext.
 - JMS connection factories declared in the “java:comp/env/jms” subcontext.
 - JavaMail connection factories declared in the “java:comp/env/mail” subcontext.
 - URL connection factories declared in the “java:comp/env/url” subcontext.
- In the code, look up the resource manager connection factory object using either the JNDI API or the *lookup* method in the *EJBContext*.
- Invoke the resource manager connection factory to obtain one or more connections to the resource. Example:

```
@Stateless
public class WarehouseInventoryBean
{
    ...
    /* EJB context object used to lookup resource references. */
    @Resource
    private SessionContext mSessionContext;
    ...

    ...
    public int countItems(ItemId inItemId)
    {
        ...
        /*
         * Lookup the resource manager connection factory using its
         * JNDI name. Note that the name is automatically prefixed with
         * "java:comp/env/".
         */
        javax.sql.DataSource theWarehouseAppDB =
            mSessionContext.lookup("jdbc/EmployeeAppDB"));

        /* Retrieve a resource (database) connection. */
        java.sql.Connection theConnection = theWarehouseAppDB.getConnection();

        /* Now we can query the database. */
        ...
    }
    ...
}
```

15.9. Resource Environment References

References: EJB 3.1 Specification, section 16.8.

To inject resource environment references, we use the old faithful `@Resource` annotation on an instance field or setter-method. Since resource environment references are not shareable and does not require authentication, the `authenticationType` and `shareable` elements of the `@Resource` annotation must not be used.

The only thing that sets injection of resource environment references apart from injection of simple environment entries is that the type of the data to be injected is not one of the data types allowed with [injection of simple environment entries](#).

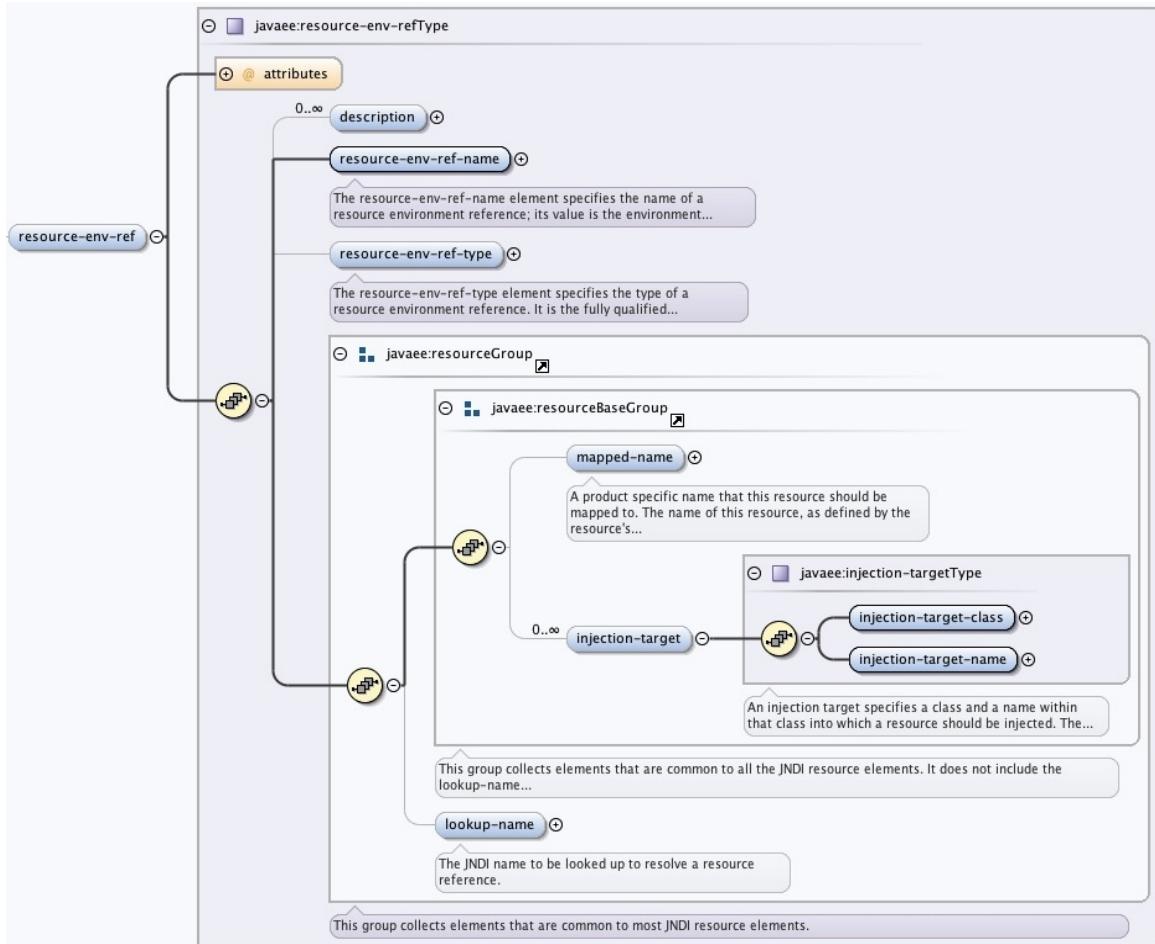
15.9.1. Injecting a Resource Environment Reference Using Annotations

To inject the resource environment reference created above using annotations, all that is needed is to annotate an instance field or a setter-method with the `@Resource` annotation, specifying the JNDI name of the resource reference to inject:

```
...
@Stateful
@LocalBean
public class StatefulSession2Bean
{
    ...
    /* Resource environment reference with annotation. */
    @Resource(name="resref/MyRsrcEnvRef1")
    private Properties mResourceEnvRef1;
    ...
}
```

15.9.2. Defining Resource Environment References in the Deployment Descriptor

Resource environment references are defined using <resource-env-ref> element in the ejb-jar.xml deployment descriptor. The <resource-env-ref> element has the following structure:



The fragment from the ejb-jar deployment descriptor XML schema defining the <resource-ref> element, which is used to define resource manager connection factory references.

Below is an example showing a resource environment reference declared in a ejb-jar.xml deployment descriptor. The resource references is an instance of the *java.util.Properties* class, configured in the application server. Please see the inline comments for details.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <!--
                Supplying the name of the EJB which we want override the
                annotation configuration of is enough
            -->
            <ejb-name>StatefulSession2Bean</ejb-name>

            <resource-env-ref>
                <!-- JNDI name relative to java:comp/env of the reference. Required. -->
                <resource-env-ref-name>resref/MyRsrcEnvRef</resource-env-ref-name>
                <!-- Type of the reference. Optional. -->
                <resource-env-ref-type>java.util.Properties</resource-env-ref-type>
                <!-- Specifies instance field to inject the reference into. -->
                <injection-target>
                    <injection-target-
class>com.ivan.scbcd6.StatefulSession2Bean</injection-target-class>
                    <injection-target-name>mResourceEnvRef2</injection-target-name>
                </injection-target>
            </resource-env-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

The resource environment reference must also be defined in the application server in which the EJB is to be executed. For a description on how to define such references in GlassFish v3, see [appendix E](#).

15.9.3. Programmatically Retrieving Resource Environment References

Resource environment references can be looked up as described in the section on [Programmatic Lookup of References from EJBs](#) above. The procedure for defining and accessing a resource environment reference is:

- Define the resource in the container in which the application is to run.
- In the ejb-jar.xml deployment descriptor, declare a <resource-env-ref> in the EJB that is to lookup the resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatefulSession2Bean</ejb-name>

            <resource-env-ref>
                <!-- JNDI name relative to java:comp/env of the reference. Required. -->
                <resource-env-ref-name>resref/MyRsrcEnvRef2</resource-env-ref-name>
                <!--
                    Type of the reference. Optional if an injection target is
                    specified, required if not.
                -->
                <resource-env-ref-type>java.util.Properties</resource-env-ref-type>
            </resource-env-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

- From within the EJB, look up the resource reference using either the JNDI API or an *EJBContext* object.

15.10. Message Destination References

References: EJB 3.1 Specification, section 16.9.

Message destination references are references to message destinations, commonly JMS topics or queues, that an EJB can use to produce or consume messages.

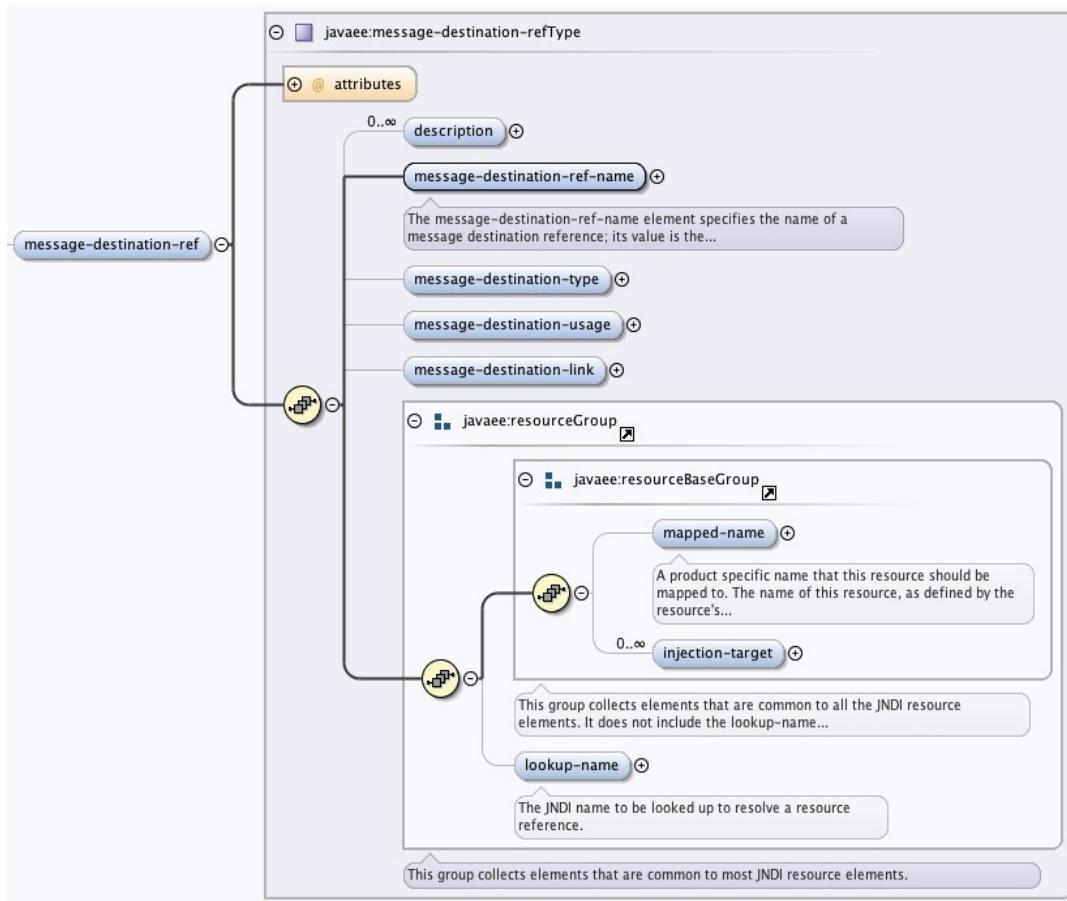
A message destination reference adds a name to the environment of an EJB, by which the EJB can look up a reference to a queue or topic from which it consumes messages or to which it publishes messages. In addition, the queue or topic must be created in the application server, except for in the special case with message destination links.

15.10.1. Configuring Message Destination References in the Deployment Descriptor

Message destination reference configuration using the ejb-jar.xml deployment descriptor offers the additional functionality, compared to annotation-based configuration:

- Linking of message destinations.
Enables creation of application-local transmitting of messages between EJBs; one EJB produces a message to queue or topic which is looked-up using a JNDI name that does not exist outside of the application. The message is consumed by an EJB in the same application, which uses the same, application-local, JNDI name.
- Specifying whether a message destination reference is used to produce, consume or both produce and consume messages.

The XML schema fragment defining message destination references in the deployment descriptor looks like this:



The fragment from the ejb-jar deployment descriptor XML schema defining the <message-destination-ref> element, which is used to define message destination references.

The following listing shows an example of a message destination reference declaration in an ejb-jar.xml deployment descriptor:

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:ejb="http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1">

  <enterprise-beans>
    <message-driven>
      <ejb-name>ListenerFacadeMDB</ejb-name>
      <message-destination-link>ListenerMsgDest</message-destination-link>

      <message-destination-ref>
        <!--
          JNDI name used in the EJB to look up the destination
          reference. Note that this name does not have to correspond
          to a destination in the application server if a message
          destination link is used.
          When a message destination link is used, this reference
          has to be looked up programmatically or being injected
          from the deployment descriptor and cannot be
          injected using annotations.
        -->
        <message-destination-ref-name>
          jms/PublishingTopic
        </message-destination-ref-name>
      </message-destination-ref>
    </message-driven>
  </enterprise-beans>

```

```

<!--
    Here, the type of the destination which is referenced
    can be specified using a <message-destination-type>
    element. We can also specify how the destination reference
    is used; whether the EJB produces or consumes messages
    using the reference.
    Both these configurations are optional.
-->
<message-destination-type>javax.jms.Topic</message-destination-type>
<message-destination-usage>Produces</message-destination-usage>
<!--
    Message destination link enabling linking message
    producers with message consumers within one and the
    same application.
    The <message-destination-link> element is defined in
    the message producer.
-->
<message-destination-link>ListenerMsgDest</message-destination-link>
</message-destination-ref>
</message-driven>
...
</enterprise-beans>

<assembly-descriptor>
<!--
    This element is used to specify a link between a logical message
    destination name, used in <message-destination-link> elements
    above, and a physical JNDI name.
-->
<message-destination>
    <!-- Logical name. -->
    <message-destination-name>ListenerMsgDest</message-destination-name>
    <!--
        JNDI name of a physical queue defined in the application
        server
    -->
    <lookup-name>jms/StockInfo</lookup-name>
</message-destination>
</assembly-descriptor>
</ejb-jar>

```

Please refer to [appendix C](#) for a complete example on using message destination links.

15.10.2. Injecting a Message Destination Reference Using Annotations

The following example shows how a JMS connection factory and a JMS message destination is injected into a servlet using annotations. Again, we use the [@Resource annotation](#). Note how the *mappedName* elements of the @Resource annotations contains JNDI names.

```
...
@WebServlet(name = "MyServlet", urlPatterns = {"/MyServlet"})
public class PostMessage extends HttpServlet{

    @Resource(mappedName = "jms/MyConnectionFactory")
    private ConnectionFactory mMyConnectionFactory;
    @Resource(mappedName = "jms/MyQueue")
    private Queue mMyQueue;
...
}
```

When using message destination links, the JNDI name specified in a <message-destination-ref-name> element must be looked-up programmatically or injected using the <injection-target> child element of the <message-destination-ref> element in the deployment descriptor, as described [earlier](#).

15.10.3. Programmatically Retrieving Message Destination References

Message Destination References can be looked up as described in the section on [Programmatic Lookup of References from EJBs](#) above.

15.11. Persistence Unit References

References: EJB 3.1 Specification, section 16.10.

Persistence unit references are references in the environment of an EJB that refer to JPA entity manager factories. Such entity manager factories are configured in persistence.xml files.

A JPA entity manager factory is used to create JPA entity managers, also called persistence contexts, which are used to create, read, update, delete etc. entities.

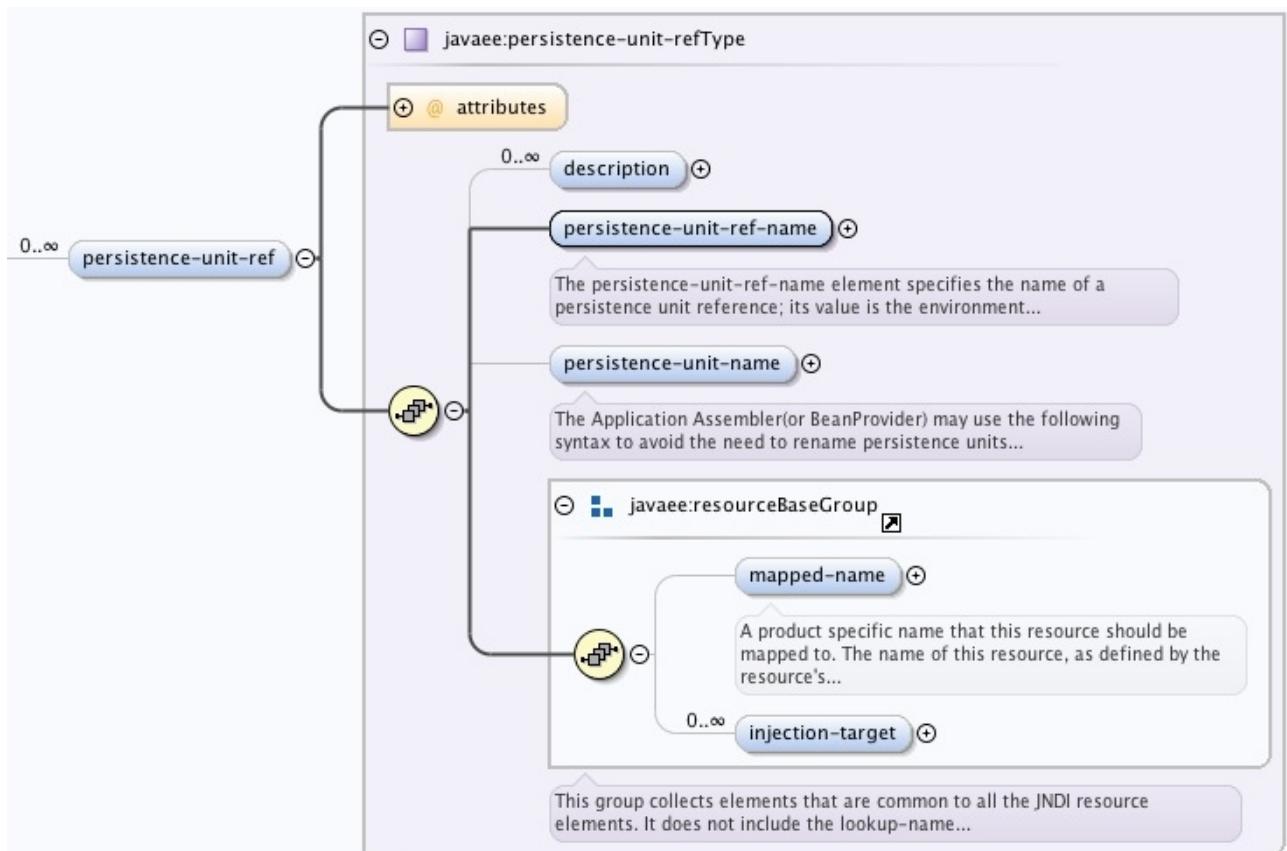
Persistence unit references are less commonly used, compared to persistence context references which will be discussed in a subsequent [section](#).

For details on how to set up JDBC Resources in GlassFish v3, see [appendix F](#).

15.11.1. Persistence Unit References in the Deployment Descriptor

If, for some reason, we do not want to use annotations to inject a reference to a persistence unit, then we need to declare it in the ejb-jar.xml deployment descriptor, in order to be able to look it up programmatically in an EJB.

The XML schema fragment defining persistence unit references in the deployment descriptor looks like this:



The fragment from the ejb-jar deployment descriptor XML schema defining the `<persistence-unit-ref>` element, which is used to define persistence unit references.

The `<injection-target>` element in the `<persistence-unit-ref>` element allows for dependency injection of persistence unit references, as described [earlier](#).

The following is an example of a persistence unit reference declaration in a deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:ejb="http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1">

  <enterprise-beans>
    <message-driven>
      <ejb-name>NewMessageMDB</ejb-name>

      <persistence-unit-ref>
        <!--
          Appended to "java:comp/env", becomes the JNDI name
          by which the persistence unit reference can be looked
          up programmatically.
        -->
        <persistence-unit-ref-name>
          persistence/MyNewDB
        </persistence-unit-ref-name>
        <!-- Persistence unit name, as defined in persistence.xml -->
        <persistence-unit-name>MsgDrvEx2PU</persistence-unit-name>
      </persistence-unit-ref>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

When overriding a `@PersistenceUnit` annotation using the `ejb-jar.xml` deployment descriptor, the following rules exists:

- The container uses the JNDI name, be it the default name or a provided name, of the `@PersistenceUnit` annotation to determine whether a `<persistence-unit-ref>` entry in the deployment descriptor is to override the annotation.
- The value of the `unitName` element of the `@PersistenceUnit` annotation is overridden by the value in the `<persistence-unit-name>` element.
- The injection target, if used, must specify the exact name of the annotated field or property method.

15.11.2. Persistence Unit Reference Injection Using Annotations

Injection of persistence unit references into instance fields or setter-methods is accomplished by annotating the field or method with the `@PersistenceUnit` annotation, specifying the name of the persistence unit to be used using the `unitName` element.

Example:

```
...
  @PersistenceUnit(unitName="MsgDrvEx2PU", name="persistence/MyEntityMgrFactory")
  private EntityManagerFactory mEx2PUEntityMgrFactory;
...
```

Optionally, the `name` element in the annotation may be used to define a JNDI name by which the reference can be looked up programmatically. In the above example, a reference to the entity manager factory can be looked up using the JNDI name “`java:comp/env/persistence/MyEntityMgrFactory`”.

15.11.3. Programmatic Retrieval of Persistence Unit References

Persistence Unit References can be looked up as described in the section on [Programmatic Lookup of References from EJBs](#) above.

15.12. Persistence Context References

References: EJB 3.1 Specification, section 16.11.

Persistence context references are references in the environment of an EJB that refer to JPA entity managers. Such entity managers are configured in persistence.xml files.

A JPA entity manager is used to create, read, update, delete etc. entities.

Important note:

Instances of *EntityManager* are not thread-safe!

This does not pose a problem in EJBs, except perhaps in singleton session beans, depending on the concurrency configuration.

To avoid threading issues with *EntityManager* instances, use JNDI lookup and store the object in a local variable instead of using dependency injection.

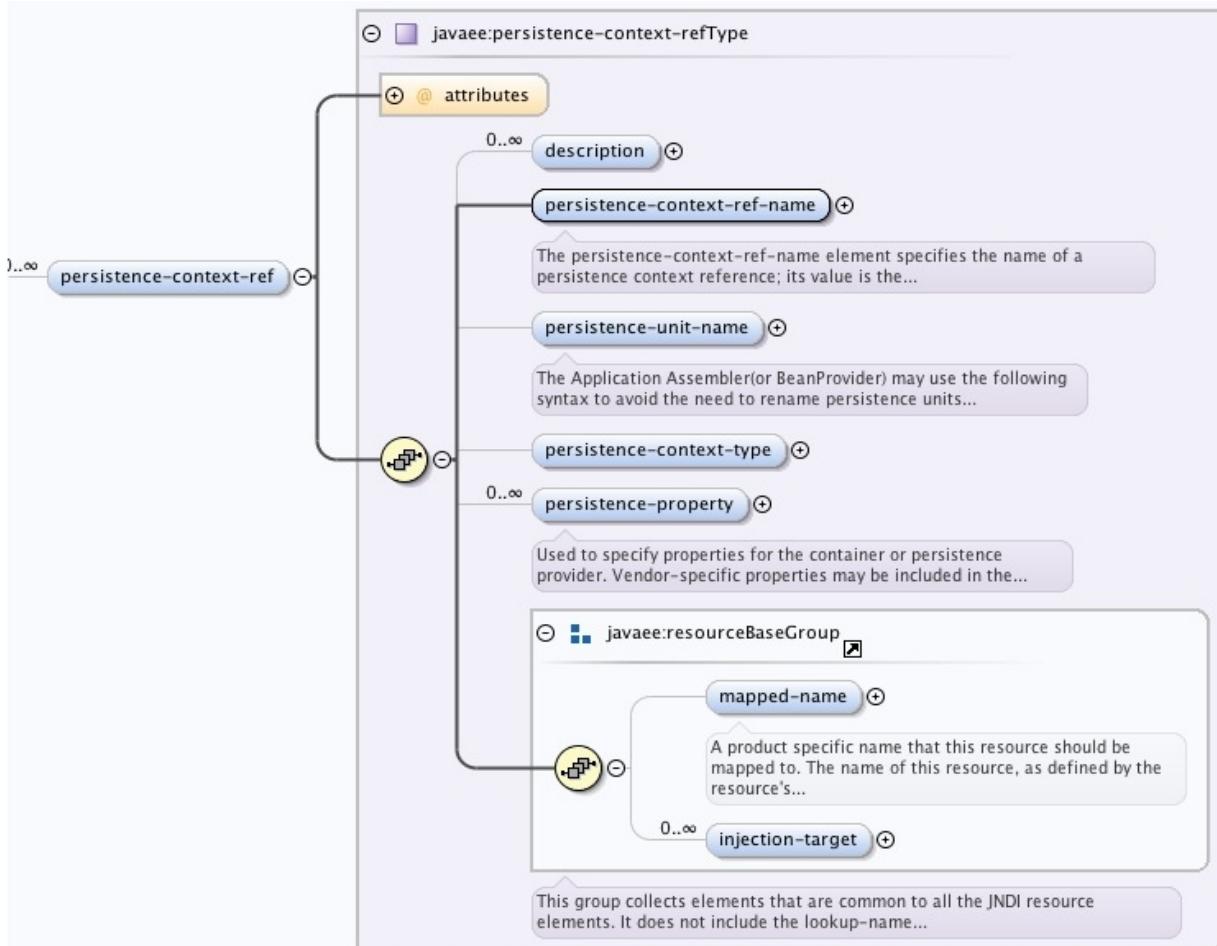
Instances of *EntityManagerFactory* are thread-safe.

For details on how to set up JDBC Resources in GlassFish v3, see [appendix F](#).

15.12.1. Persistence Context References in the Deployment Descriptor

If, for some reason, we do not want to use annotations to inject a reference to a persistence context, then we need to declare it in the ejb-jar.xml deployment descriptor, in order to be able to look it up programmatically in an EJB.

The XML schema fragment defining persistence context references in the deployment descriptor looks like this:



The fragment from the ejb-jar deployment descriptor XML schema defining the `<persistence-context-ref>` element, which is used to define persistence context references.

The `<injection-target>` element in the `<persistence-context-ref>` element allows for dependency injection of persistence context references, as described [earlier](#).

The following is an example of a persistence unit reference declaration in a deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <ejb-name>StatefulSession2Bean</ejb-name>

            <persistence-context-ref>
                <!-- JNDI name relative to java:comp/env of the reference. -->
                <persistence-context-ref-name>
                    persistence/MyPersistenceContext
                </persistence-context-ref-name>
                <!--
                    Name of the persistence unit, as specified in
                    persistence.xml.
                -->
                <persistence-unit-name>MsgDrvEx2PU</persistence-unit-name>
                <!--
                    Persistence context type (default: Transaction:
                    Transaction - Transaction-scoped persistence context.
                    Extended   - Extended scope persistence context.
                                Can only be used with stateful session beans.
                -->
                <persistence-context-type>Extended</persistence-context-type>
            </persistence-context-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

When overriding a `@PersistenceContext` annotation using the `ejb-jar.xml` deployment descriptor, the following rules exists:

- The container uses the JNDI name, be it the default name or a provided name, of the `@PersistenceContext` annotation to determine whether a `<persistence-context-ref>` entry in the deployment descriptor is to override the annotation.
- The value of the `unitName` element of the `@PersistenceContext` annotation is overridden by the value in the `<persistence-unit-name>` element.
- If the `<persistence-context-type>` element is specified, then it overrides the `type` element of the annotation.
- Any `<persistence-property>` element introduced in the deployment descriptor are added to those specified in the annotation, unless one or more properties have the same name as those in the annotation in which case the properties in the annotation are overridden.
- The injection target, if used, must specify the exact name of the annotated field or property method.

15.12.2. Persistence Context Reference Injection Using Annotations

Injection of persistence context references into instance fields or setter-methods is accomplished by annotating the field or method with the `@PersistenceContext` annotation, specifying the name of the persistence unit to be used using the `unitName` element.

Example:

```
@PersistenceContext(unitName = "MsgDrvEx2PU",
    type = PersistenceContextType.EXTENDED, name="persistence/MyEntityMgr")
private EntityManager mSomeEntityMgr;
```

Optionally, the `name` element in the annotation may be used to define a JNDI name by which the reference can be looked up programmatically. In the above example, a reference to the entity manager factory can be looked up using the JNDI name “`java:comp/env/persistence/MyEntityMgr`”.

The `type` element of the `@PersistenceContext` annotation is used to specify the scope of the `EntityManager` injected. The default value is `PersistenceContextType.TRANSACTION`. Possible values are:

Persistence Context Type	Notes
PersistenceContextType.TRANSACTION	Scope in which entity state is managed is one transaction. Usable with any type of EJB. Default value.
PersistenceContextType.EXTENDED	Gives the ability to manage entity state in a scope that spans multiple method calls on an EJB, even if the methods run in different transactions. May only be used with stateful session beans.

The `properties` element enables specifying name-value pairs for properties that are passed to the container or persistence provider. Properties with names that are not recognized are ignored.

15.13. UserTransaction Interface

References: EJB 3.1 Specification, section 16.12.

EJBs with bean managed transactions can either request an object implementing the *UserTransaction* interface to be injected or look up such an object using its JNDI name. Such an object allows for the EJB to manage transaction boundaries programmatically.

In order to be able to have a *UserTransaction* object injected or to look such an object up, the bean must be configured to use bean-managed transactions, using either the

```
@TransactionManagement(TransactionManagementType.BEAN)
```

annotation on the bean class or the

```
<transaction-type>Bean</transaction-type>
```

element in the ejb-jar.xml deployment descriptor.

Failing to do this will result in an exception being thrown at runtime when attempting to perform dependency injection or JNDI lookup.

15.13.1. UserTransaction Object References in the Deployment Descriptor

UserTransaction object references are declared in the ejb-jar.xml deployment descriptor in the same way as [resource environment references](#).

The example configuration below injects a *UserTransaction* object into the instance field *mAnotherTransaction* in the EJB with the name StatefulSession2Bean:

```
...
<session>
  <!--
    Supplying the name of the EJB which we want override the
    annotation configuration of is enough
  -->
  <ejb-name>StatefulSession2Bean</ejb-name>

  <resource-env-ref>
    <!-- JNDI name of the reference. Required. -->
    <resource-env-ref-name>java:comp/UserTransaction</resource-env-ref-name>
    <!-- Type of the reference. Optional. -->
    <resource-env-ref-type>
      javax.transaction.UserTransaction
    </resource-env-ref-type>
    <!-- Specifies instance field to inject the reference into. -->
    <injection-target>
      <injection-target-class>
        com.ivan.scbcd6.StatefulSession2Bean
      </injection-target-class>
      <injection-target-name>mAnotherTransaction</injection-target-name>
    </injection-target>
  </resource-env-ref>
...

```

15.13.2. UserTransaction Reference Injection Using Annotations

Injection of *UserTransaction* references into instance fields or setter-methods is accomplished by annotating the field or method with the `@Resource` annotation. The `authenticationType` and `shareable` element of the [@Resource annotation](#) must not be used.

Example:

```
@Stateful(name = "StatefulSession2Bean")
@TransactionManagement(TransactionManagementType.BEAN)
public class StatefulSession2Bean
{
    @Resource
    private UserTransaction mUserTransaction;

    ...
}
```

15.13.3. Programmatic Retrieval of UserTransaction Objects

UserTransaction objects can be retrieved programmatically, either by [performing a JNDI lookup](#) using the special name “java:comp/UserTransaction” or by calling the `getUserTransaction()` method on an object implementing the `EJBContext`.

15.14. ORB References

References: EJB 3.1 Specification, section 16.13.

EJBs that want to use a CORBA ORB can either request an object of the type `org.omg.CORBA.ORB` to be injected or look up such an object using the special JNDI name “java:comp/ORB”.

The reference to the ORB may only be used within the EJB in which the reference was injected or that performed the lookup.

15.14.1. ORB References in the Deployment Descriptor

ORB references are declared in the ejb-jar.xml deployment descriptor in the same way as [resource manager connection factories](#).

The example configuration below injects an ORB reference into the instance field `mDDInjectedCorbaOrb` in the EJB with the name `StatefulSession2Bean`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <!--
                Supplying the name of the EJB which we want override the
                annotation configuration of is enough
            -->
            <ejb-name>StatefulSession2Bean</ejb-name>

            <!-- Inject an ORB reference into the EJB. -->
            <resource-ref>
                <res-ref-name>java:comp/ORB</res-ref-name>
                <res-type>org.omg.CORBA.ORB</res-type>
            <!--
                Request a reference to an ORB instance that is not
                shared.
                Default is for ORB instances to be shared.
            -->
        
```

```

<res-sharing-scope>Unshareable</res-sharing-scope>
<!-- Specifies where to inject the value. -->
<injection-target>
    <injection-target-class>
        com.ivan.scbcd6.StatefulSession2Bean
    </injection-target-class>
    <injection-target-name>mDDInjectedCorbaOrb</injection-target-name>
</injection-target>
</resource-ref>
...
</session>
</enterprise-beans>
</ejb-jar>

```

In the above example, a reference to an ORB instance that is not shared is request by using the `<res-sharing-scope>` element with the value “Unshareable”.

15.14.2. ORB Reference Injection Using Annotations

Injection of ORB references into instance fields or setter-methods is accomplished by annotating the field or method with the `@Resource` annotation.

The `authenticationType` element of the [@Resource annotation](#) need not be used.

The `shareable` element of the [@Resource annotation](#) may be used to request a reference to a non-shared ORB instance by setting its value to false.

Example:

```

@Stateful(name = "StatefulSession2Bean")
public class StatefulSession2Bean
{
    @Resource(shareable=false)
    private ORB mInjectedCorbaOrb;

    ...
}

```

15.14.3. Programmatic Retrieval of ORB References

A reference to an ORB object can be retrieved programmatically by [performing a JNDI lookup](#) using the special name “java:comp/ORB”. References to ORB instances obtained from a JNDI lookup may always be shared.

15.15. TimerService References

References: EJB 3.1 Specification, section 16.14.

EJBs that want to schedule execution of business logic can either request an object implementing the *javax.ejb.TimerService* interface to be injected or look up such an object using the special JNDI name “java:comp/TimerService”. For examples on how to use *TimerService* references, see [chapter 12](#).

15.15.1. TimerService References in the Deployment Descriptor

TimerService references are declared in the ejb-jar.xml deployment descriptor in the same way as [resource environment references](#).

The example configuration below injects a *TimerService* reference into the instance field *mDDInjectedTimerService* in the EJB with the name StatefulSession2Bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

    <enterprise-beans>
        <session>
            <!-- Name of EJB in which to inject the reference. -->
            <ejb-name>StatefulSession2Bean</ejb-name>

            <!-- Inject reference to TimerService. -->
            <resource-env-ref>
                <!-- JNDI name of the reference. Required. -->
                <resource-env-ref-name>java:comp/TimerService</resource-env-ref-name>
                <!-- Type of the reference. Optional. -->
                <resource-env-ref-type>javax.ejb.TimerService</resource-env-ref-type>
                <!-- Specifies instance field to inject the reference into. -->
                <injection-target>
                    <injection-target-class>
                        com.ivan.scbcd6.StatefulSession2Bean
                    </injection-target-class>
                    <injection-target-name>
                        mDDInjectedTimerService
                    </injection-target-name>
                </injection-target>
            </resource-env-ref>
            ...
        </session>
    </enterprise-beans>
</ejb-jar>
```

15.15.2. TimerService Reference Injection Using Annotations

Injection of *TimerService* references into instance fields or setter-methods is accomplished by annotating the field or method with the @Resource annotation.

The *authenticationType* and *shareable* elements of the [@Resource annotation](#) must not be used.

Example:

```
@Stateful(name = "StatefulSession2Bean")
public class StatefulSession2Bean
{
    @Resource
    private TimerService mTimerService;

    ...
}
```

15.15.3. Programmatic Retrieval of TimerService References

A reference to a *TimerService* object can be retrieved programmatically by [performing a JNDI lookup](#) using the special name “java:comp/TimerService”.

15.16. EJBContext References

References: EJB 3.1 Specification, section 16.15.

EJBs that want to access the runtime context provided by the EJB container can either request an object implementing the *javax.ejb.EJBContext* interface to be injected or look up such an object using the special JNDI name “java:comp/EJBContext”.

Depending on the type of EJB, the subtypes *EntityContext*, *MessageDrivenContext* and *SessionContext* types may be used.

15.16.1. EJBContext References in the Deployment Descriptor

EJBContext references are declared in the ejb-jar.xml deployment descriptor in the same way as [resource environment references](#).

The example configuration below injects a *EJBContext* reference into the instance field *mDDInjectedEjbContext* in the EJB with the name StatefulSession2Bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd">

  <enterprise-beans>
    <session>
      <!-- Name of EJB in which to inject the reference. -->
      <ejb-name>StatefulSession2Bean</ejb-name>

      <!-- Inject reference to EJBContext. -->
      <resource-env-ref>
        <!-- JNDI name of the reference. Required. -->
        <resource-env-ref-name>java:comp/EJBContext</resource-env-ref-name>
        <!-- Type of the reference. Optional. -->
        <resource-env-ref-type>javax.ejb.EJBContext</resource-env-ref-type>
        <!-- Specifies instance field to inject the reference into. -->
        <injection-target>
          <injection-target-class>
            com.ivan.scbcd6.StatefulSession2Bean
          </injection-target-class>
          <injection-target-name>mDDInjectedEjbContext</injection-target-name>
        </injection-target>
      </resource-env-ref>
      ...
    </session>
  </enterprise-beans>
</ejb-jar>
```

15.16.2. EJBContext Reference Injection Using Annotations

Injection of *EJBContext* references into instance fields or setter-methods is accomplished by annotating the field or method with the `@Resource` annotation.

The *authenticationType* and *shareable* elements of the [@Resource annotation](#) must not be used.

Example:

```
@Stateful(name = "StatefulSession2Bean")
public class StatefulSession2Bean
{
    @Resource
    private EJBContext mInjectedEjbContext;

    ...
}
```

15.16.3. Programmatic Retrieval of EJBContext References

A reference to an *EJBContext* object can be retrieved programmatically by [performing a JNDI lookup](#) using the special name “java:comp/EJBContext”.

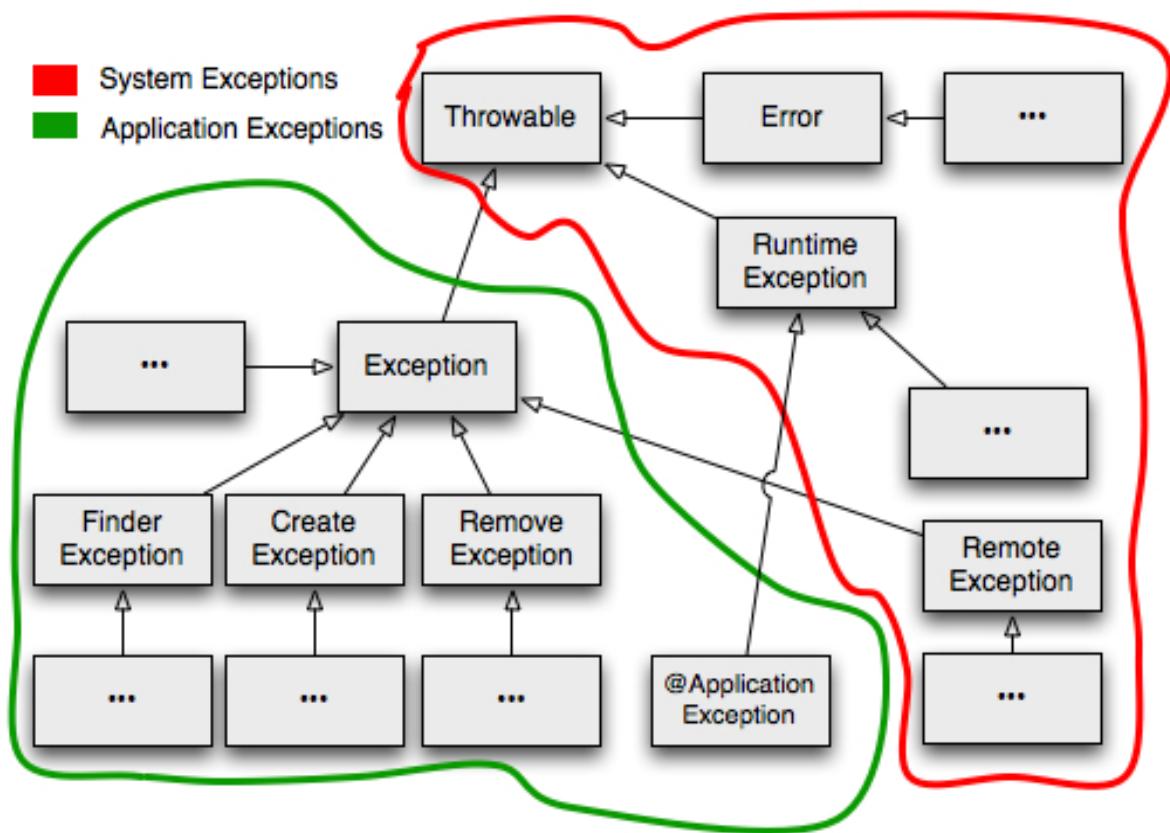
16. EJBs and Exceptions

References: EJB 3.1 Specification, chapter 14 and section 4.7.3

In this chapter we will look at the semantics of exceptions in connection to EJBs; categorization of exceptions, how exceptions affect transactions, how exceptions affect an EJB and how clients are supposed to act in response to an exception being thrown.

The chapter is not an explicit part of the certification objectives, but a place to collect common information that otherwise would have had to be duplicated at several places.

Exceptions from EJBs are categorized into two groups; application exceptions and system exceptions. This categorization applies to all types of EJBs, not only session beans.



Exceptions from EJBs categorized into system exceptions and application exceptions.

Further details about application and system exceptions in the upcoming sections.

16.1. Application Exceptions

Application exceptions are exceptions defined by the bean provider used to report problems connected to the business logic of an application. An application exception does not necessarily cause a business process to be aborted or rolled back. Clients of EJBs are expected to be able to catch, and recover from, application exceptions.

Application exceptions are:

- The class *java.lang.Exception* and subclasses.
Excluded are *java.rmi.RemoteException* and subclasses, which are system exceptions.
- Unchecked exceptions (*java.lang.RuntimeException* and subclasses) annotated with the *@ApplicationException* or marked with the *<application-exception>* element in the ejb-jar.xml deployment descriptor and listed in the throws-clause of one or more business methods.
- The class *javax.ejb.CreateException* and subclasses.
- The class *javax.ejb.RemoveException* and subclasses.
- The class *javax.ejb.FinderException* and subclasses.

Application exceptions are typically included in the throws clause of methods in the different kinds of views that can be presented by an EJB.

The *@ApplicationException* annotation is used to mark an exception as being an application exception and that it should be reported to the client without being wrapped. The annotation has the following optional elements:

Element	Description
inherited	Indicates whether subclasses also should become application exceptions. Default value: true
rollback	Indicates whether transaction rollback should be caused when exception is thrown. Default value: false

The *<application-exception>* element in the ejb-jar.xml deployment descriptor has the corresponding configuration possibilities.

16.1.1. Application Exceptions and Transactions

Application exceptions does not automatically result in the transaction in which the exception was thrown as being marked for rollback unless the *rollback* element of the *@ApplicationException* annotation or corresponding configuration in the deployment descriptor has the value true.

Prior to throwing an application exception, the bean provider must do one of the following:

- Ensure that the state of the EJB instance is consistent.
This in order to allow for continued use of the EJB instance.
- Mark the transaction for rollback.
Not necessary if the exception thrown will cause transaction rollback.
This in order to prevent that the transaction is committed and thereby causing loss of data integrity.

16.2. System Exceptions

System exceptions usually indicate that there was an unknown problem that the EJB or the EJB container throwing the exception were not able to recover from or that the problem was an unexpected one.

System exceptions are:

- The *java.rmi.RemoteException* and subclasses.
- Unchecked exceptions (*java.lang.RuntimeException* and subclasses) that are not annotated with the *@ApplicationException* and not marked with the *<application-exception>* element in the *ejb-jar.xml* deployment descriptor.
Such exceptions should not be listed in the throws-clauses of business methods.

The EJB specification gives the following recommendations regarding system exceptions:

- System exceptions should be propagated to the container.
That is, there is no need to catch and try to handle system exceptions in an EJB.
- If the EJB is unable to recover from a checked exception, it should be wrapped in an *EJBException* and the *EJBException* thrown.
- Other unexpected errors should result in *EJBException-s* being thrown.

Note that *EJBException* is an unchecked exception.

An EJB instance throwing a system exception will be destroyed, unless it is a singleton session bean. Thus only singleton session beans need to make sure the state of the EJB is consistent before throwing a system exception, since the bean will continue to serve any future requests.

An attempt to invoke a stateful session bean having thrown a system exception will result in a *NoSuchEJBException* being thrown.

16.2.1. System Exceptions and Transactions

The container will automatically roll back any transaction the EJB were executing in if a system exception is thrown from within a method in the EJB.

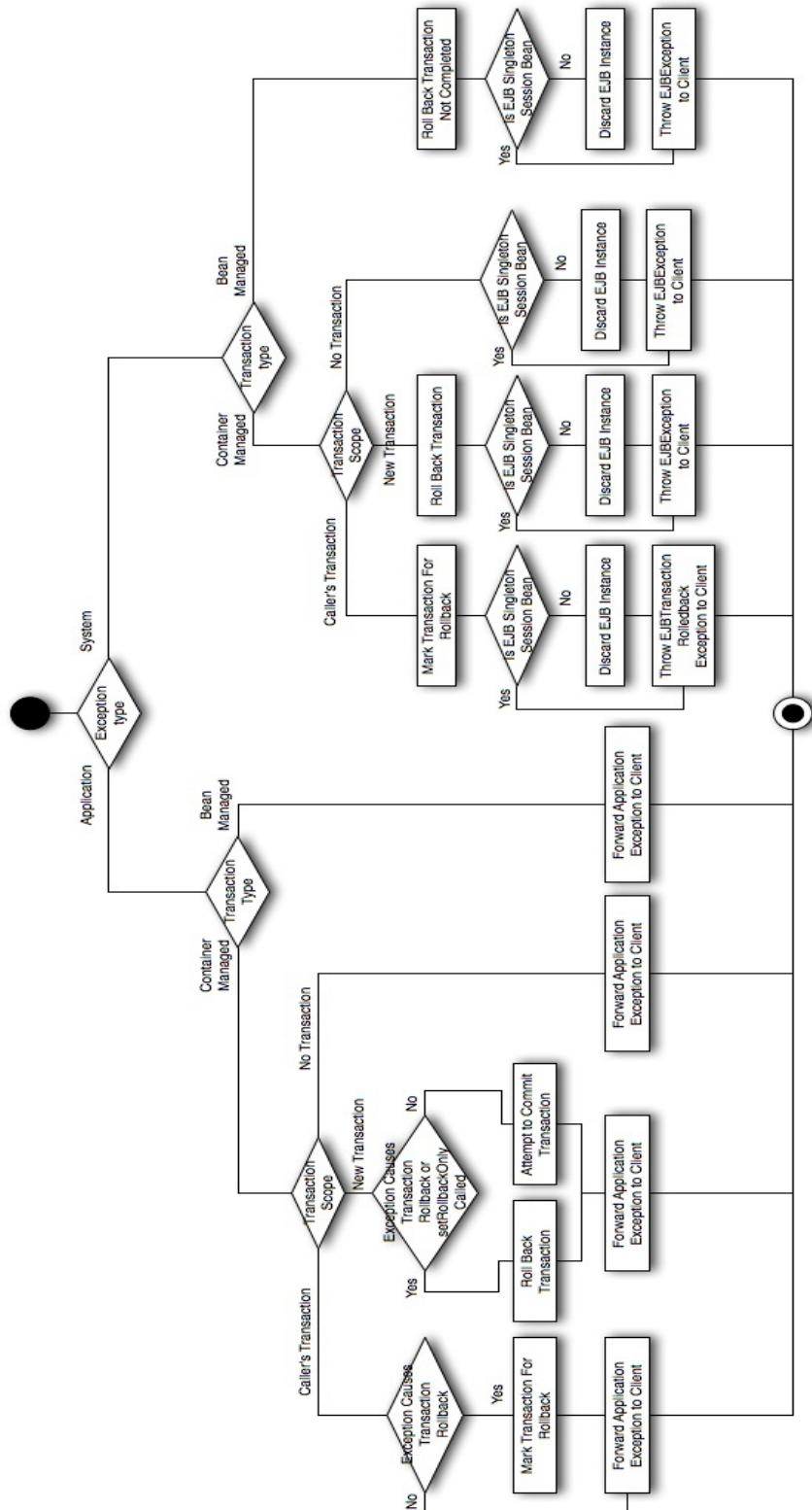
16.3. Session Bean Exception Handling

The figures in this section show how exceptions from session bean methods in different views are handled, how transactions are affected and what exceptions clients receives.

Omitted from the diagrams are logging to the container's log, which always takes place in the case of system exceptions.

16.3.1. Business Interface and No-Interface View Method Exception Handling

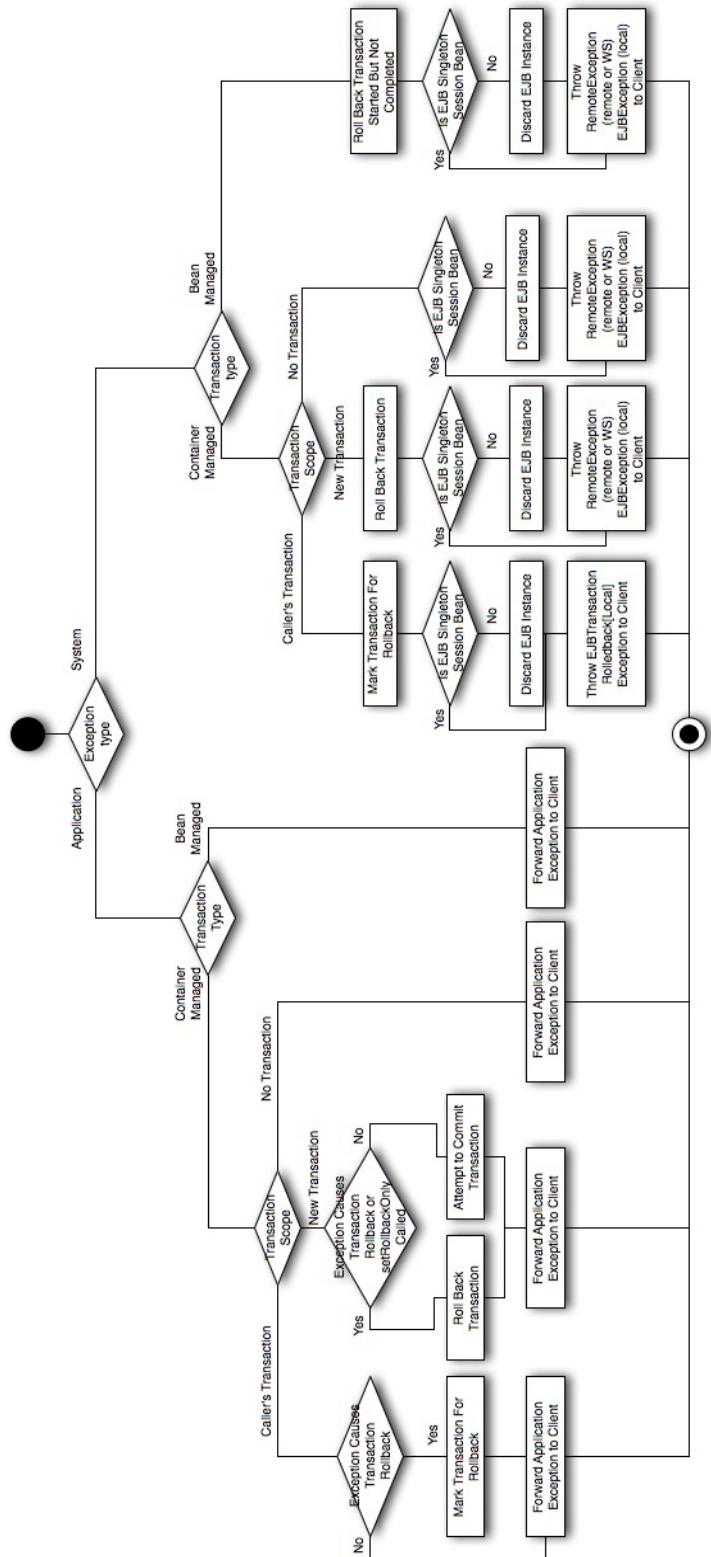
The following figure shows the exception handling for methods in the business interface or no-interface view of session beans:



Exception handling for business methods of session beans.

16.3.2. Web Service Client View and EJB 2.1 Client View Method Exception Handling

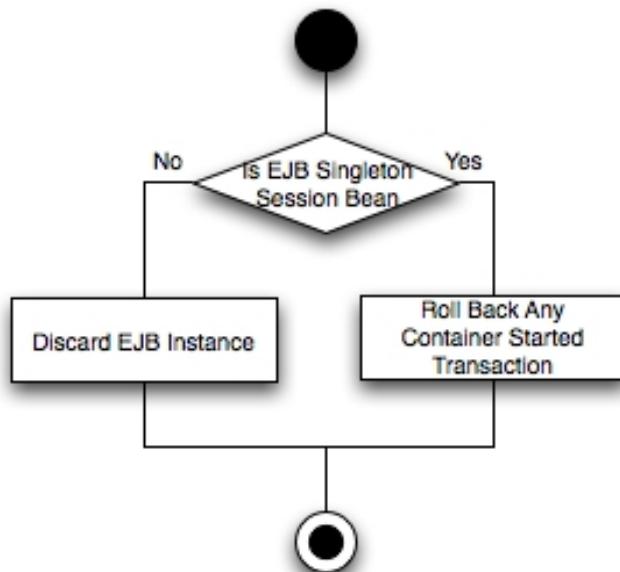
The following figure shows the exception handling for methods in the web service client view and methods in the EJB 2.1 client view:



Exception handling for web service methods and methods in the EJB 2.1 view of session beans.

16.3.3. PostConstruct and PreDestroy Method Exception Handling

The following figure shows how system exceptions thrown by a PostConstruct or PreDestroy method of a session bean are handled. These methods are not allowed to throw application exceptions.



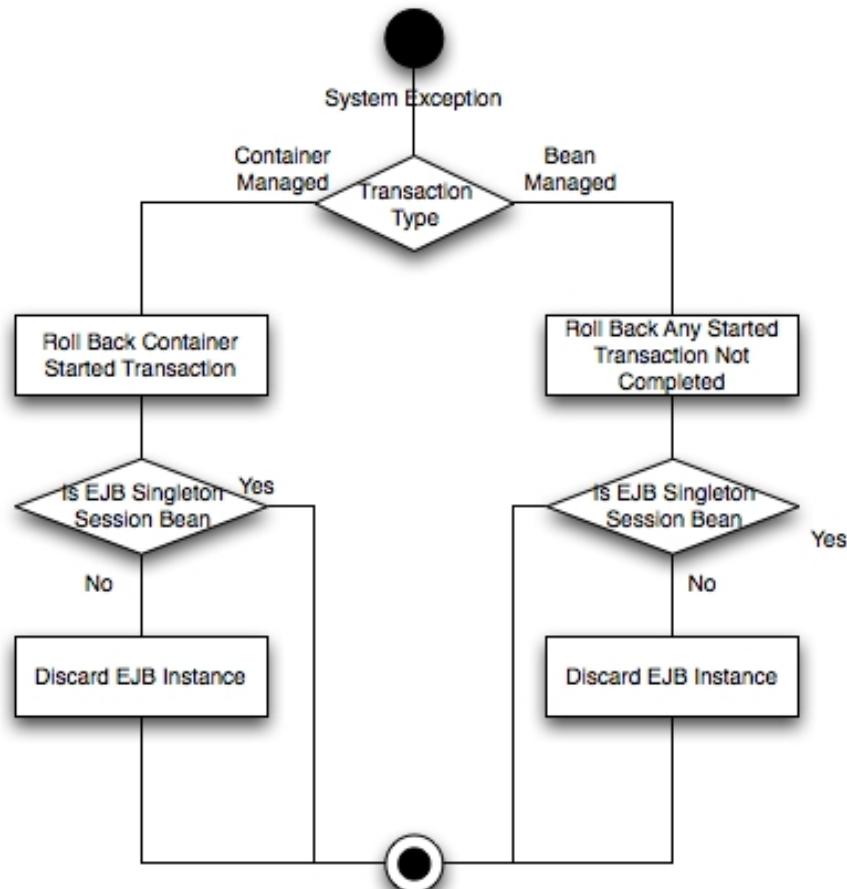
PostConstruct and PreDestroy session bean method exception handling.

16.4. Timeout Callback Method Exception Handling

References: EJB 3.1 Specification, section 14.3.6.

The exception handling described in this section applies to all kinds of EJBs, not only session beans.

Note that this figure describes the container's exception handling semantics for timeout callback methods in all kinds of EJBs.



Timeout callback method exception handling.

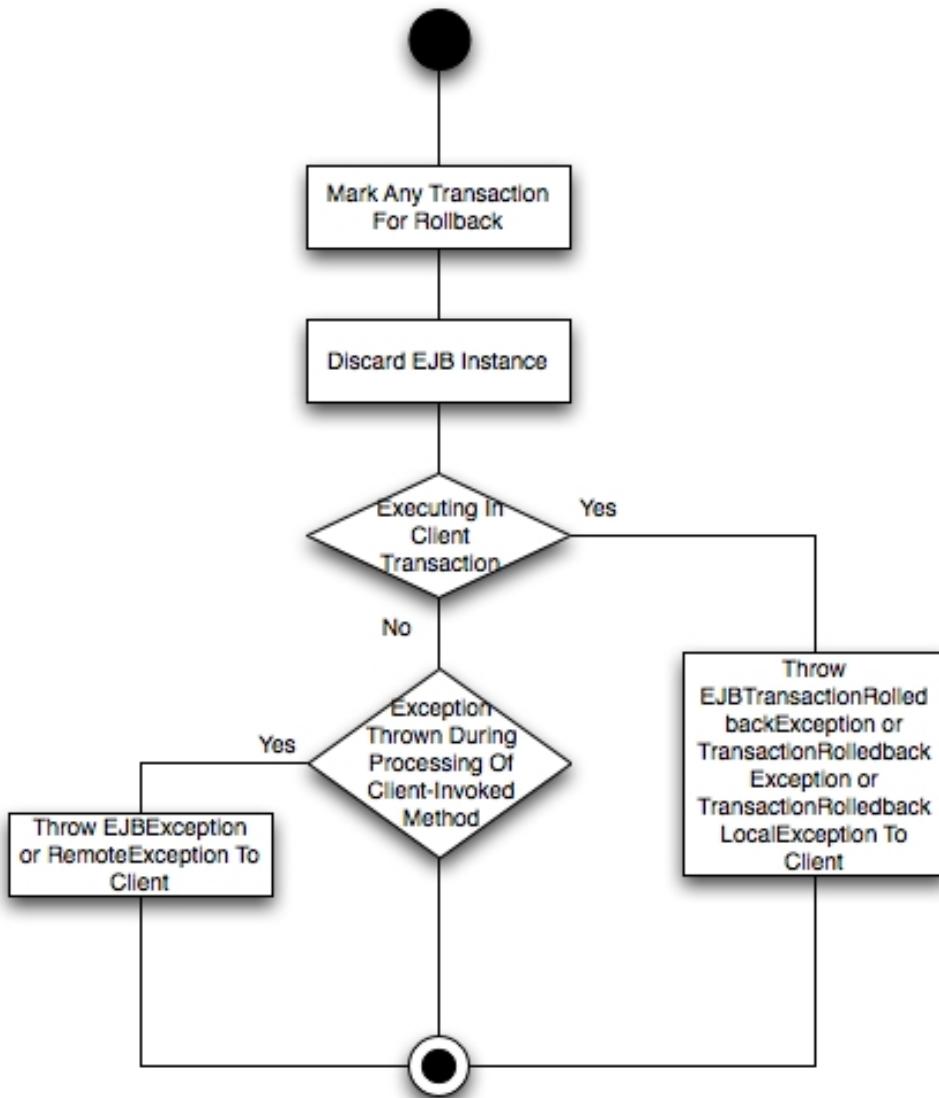
16.5. Exception Handling in Other Callback Methods

References: EJB 3.1 Specification, section 14.3.7.

The exception handling described in this section applies to all kinds of EJBs, not only session beans.

The figure below shows the handling of exceptions thrown from other callback methods invoked by the container. All exception are handled in the same way. Such methods are:

- Dependency injection methods.
- The following methods in the *EntityBean* interface:
 - *ejbActivate*
 - *ejbPassivate*
 - *ejbLoad*
 - *ejbStore*
 - *setEntityContext*
 - *unsetEntityContext*
- PostActivate methods.
- PrePassivate methods.
- The following methods in the *SessionBean* interface:
 - *ejbActivate*
 - *ejbPassivate*
 - *setSessionContext*
- The method *setMessageDrivenContext* in the *MessageDrivenBean* interface.
- The following session synchronization methods:
 - *afterBegin*
 - *beforeCompletion*
 - *afterCompletion*



Exception handling in other callback methods.

Whether *EJBException* or *RemoteException* is thrown depends on:

- If the EJB 2.1 remote client view or the JAX-RPC web service client view is used, a *RemoteException* is thrown.
- If the EJB 2.1 local client view is used, throw an *EJBException*.
- If any EJB 3.1 client view is used, throw an *EJBException*.

Whether *EJBTransactionRolledbackException*, *TransactionRolledbackException* or *TransactionRolledbackLocalException* is thrown depends on:

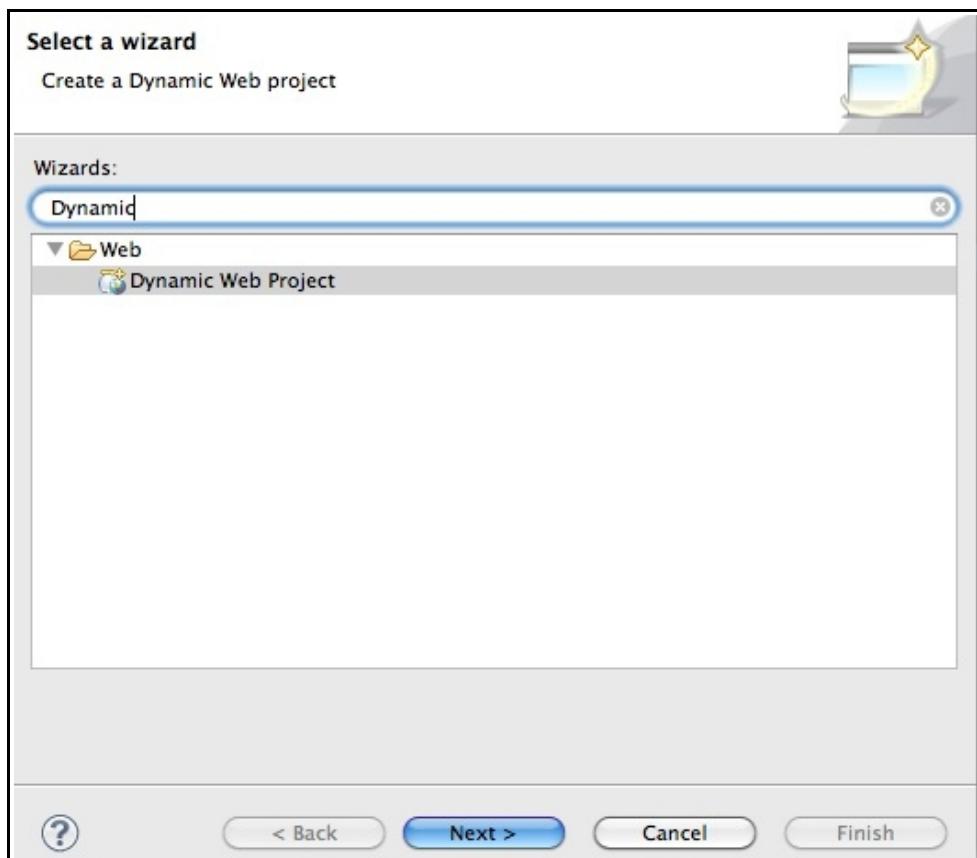
- If the EJB executed in the client's transaction, throw an *EJBTransactionRolledbackException*.
- If the EJB 2.1 remote client view or the web service client view is used, a *TransactionRolledbackException* is thrown.
- If the EJB 2.1 local client view is used, a *TransactionRolledbackLocalException* is thrown.

Appendix A – Setting Up a Dynamic Web Project in Eclipse

This appendix describe the procedure of setting up a dynamic web project in Eclipse Helios for deployment of a JavaEE 6 application to be deployed on GlassFish v3.

- Select New -> Project... in the File menu.
- Select the Dynamic Web Project wizard.

I have typed “dynamic” to apply filtering of the project wizards shown in the figure below.

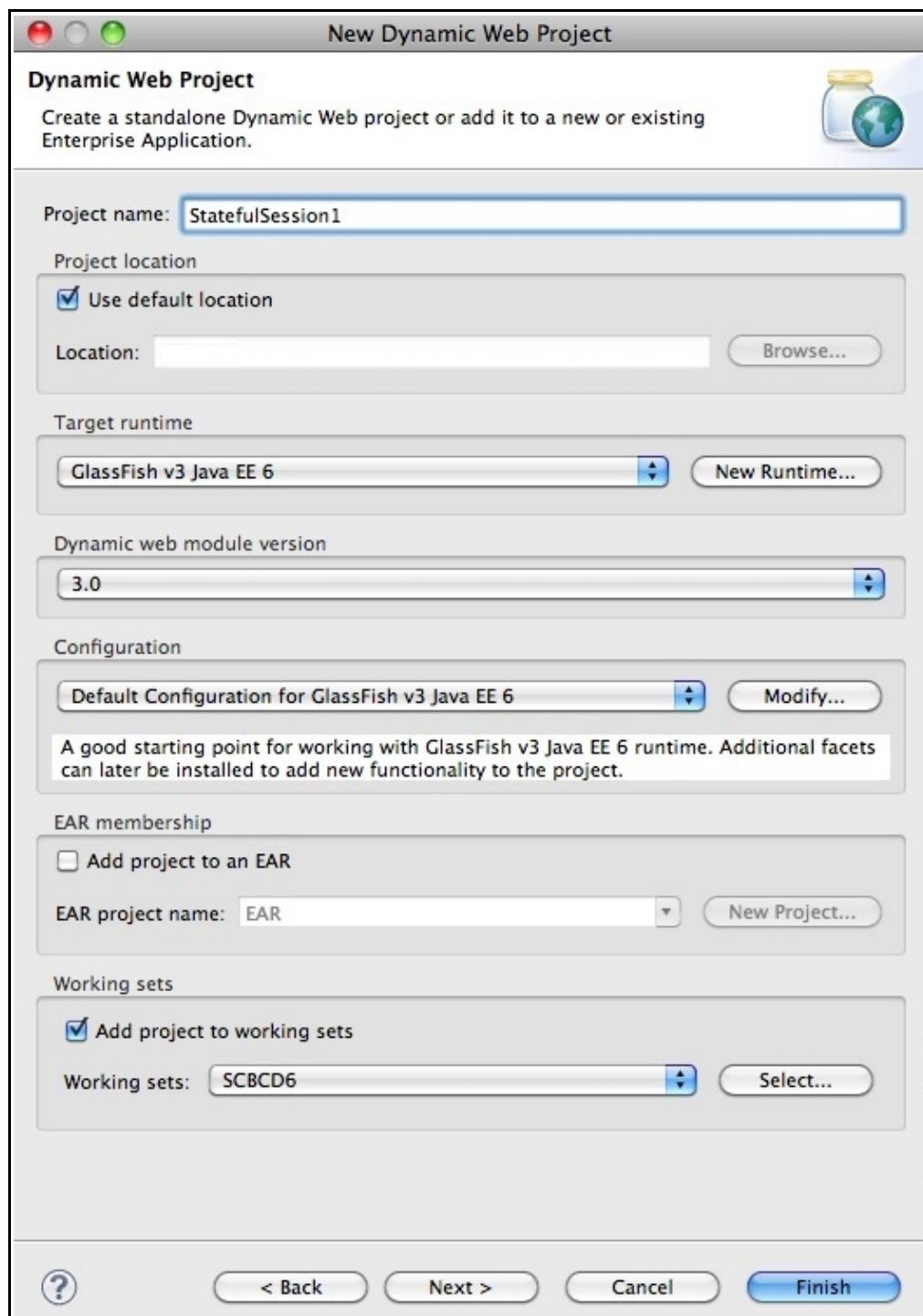


- Click the Next button.

- Configure the dynamic web project.

Selecting GlassFish v3 Java EE 6 target runtime is mandatory, as is a version 3.0, or later, of the Dynamic Web Module Version. It is also strongly suggested to use the default configuration for the application server, as seen in the figure below.

Project name and whether the project is to belong to a working set or not can be chosen as desired.



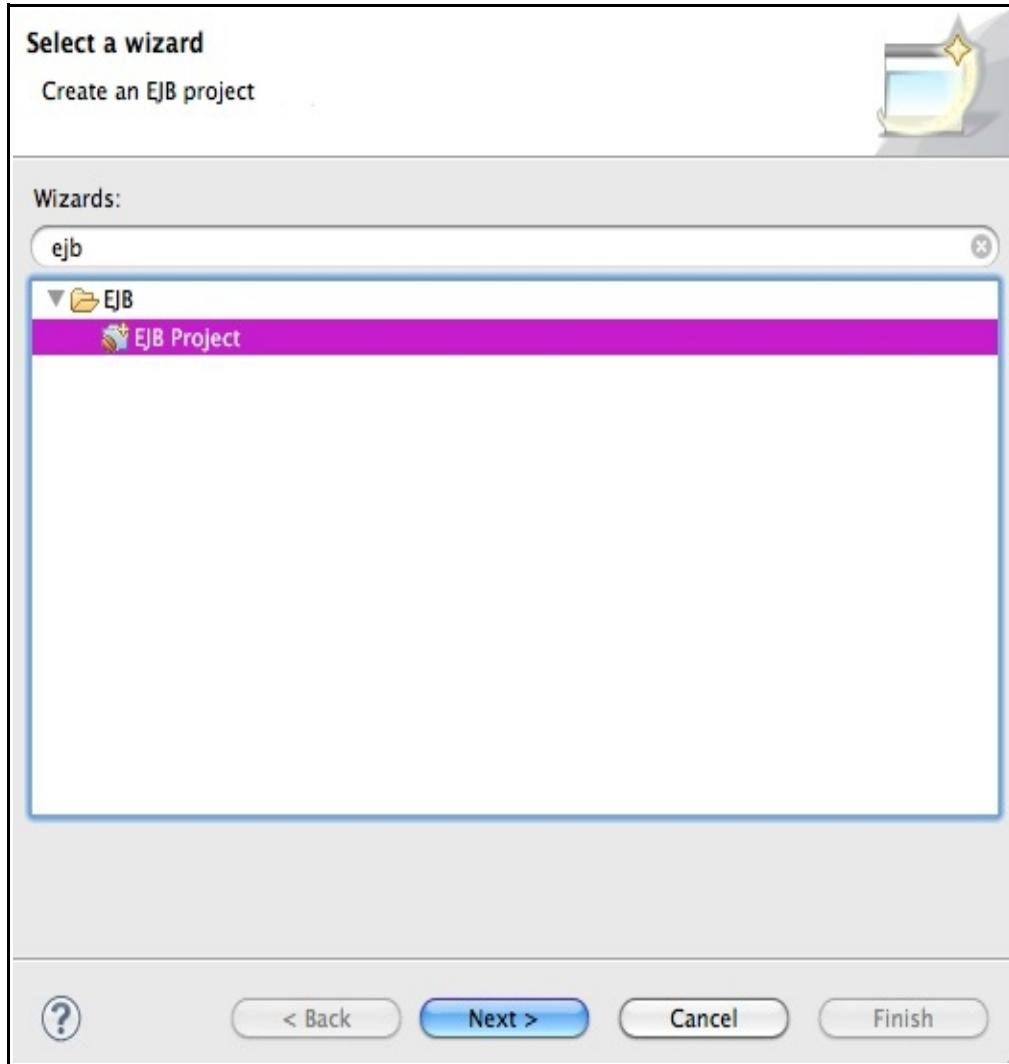
- Click the Finish button to finish creation of the project.

Appendix B – Setting Up an EJB Project in Eclipse

This appendix describe the procedure of setting up an EJB project in Eclipse Helios for deployment of one or more JavaEE 6 EJBs to be deployed on GlassFish v3. The project will not be part of an enterprise application.

- Select New -> Project... in the File menu.
- Select the EJB Project wizard.

I have typed “EJB” to apply filtering of the project wizards shown in the figure below.



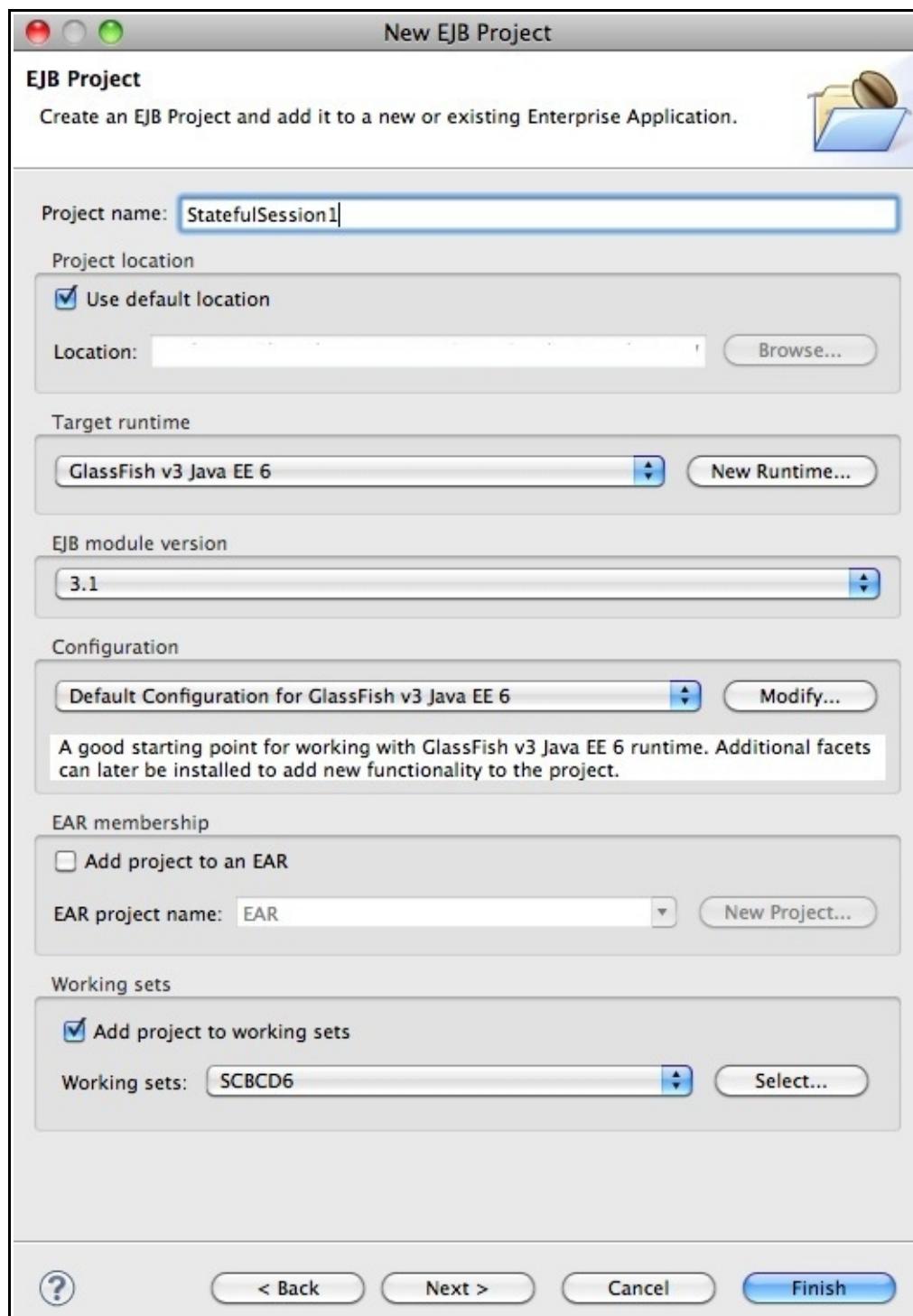
- Click the Next button.

- Configure the EJB project.

Configure the dynamic web project.

Selecting GlassFish v3 Java EE 6 target runtime is mandatory, as is a version 3.1, or later, of the EJB Module Version. It is also strongly suggested to use the default configuration for the application server, as seen in the figure below.

Project name and whether the project is to belong to a working set or not can be chosen as desired.

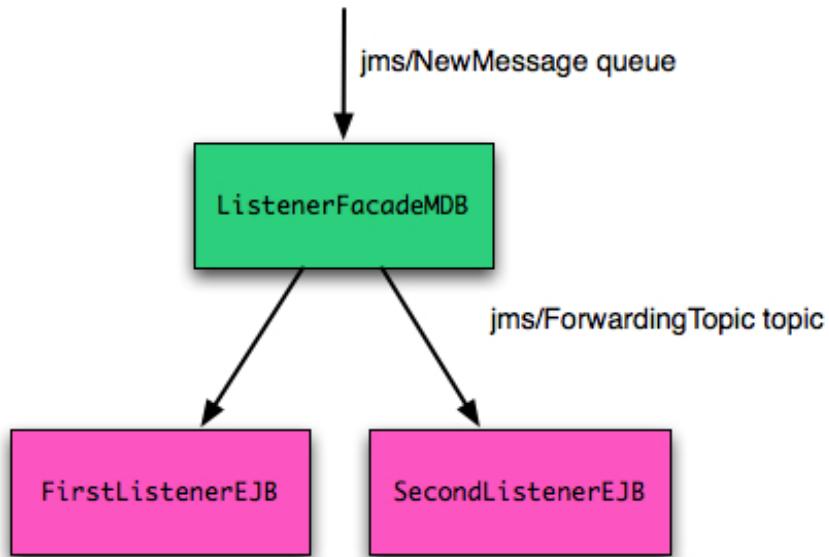


- Click the Finish button to finish creation of the project.

Appendix C – Message Destination Link Example

This appendix contains a complete example showing how to use message destination links to send messages between EJBs in one and the same application.

In the example, there are three message-driven beans. The first acts as a facade, listening for messages on a queue and, having received such a message, forwards it to the other EJBs. The two other EJBs are also message-driven beans, but listening on one and the same topic.



The three message-driven beans in the message destination link example program.

The example program is to be developed in an EJB Project in Eclipse, which is set up as described in [appendix B](#). All in all there are three classes in the project, which implement the three message-driven beans. They are all located in the `com.ivan.listeners` package.

Listener Facade EJB

As before, the listener facade bean listens for messages on a JMS queue and passes them on to a JMS topic.

```
package comivan.listeners;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.Topic;
import javax.naming.InitialContext;

/**
 * Implements a listener facade bean that receives messages from a queue
 * and passes them on to a topic.
 *
 * @author Ivan A Krizsan
 */
@MessageDriven(name = "ListenerFacadeMDB", mappedName = "jms/NewMessage",
activationConfig =
{
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue")
})
public class ListenerFacadeEJB implements MessageListener
{
    /* Constant(s): */
    private final static String PUBLISH_TOPIC_JNDI_NAME =
        "java:comp/env/jms/PublishingTopic";

    /* Instance variable(s): */
    @Resource(mappedName="jms/NewMessageFactory")
    private ConnectionFactory mJMSConnectionFactory;
    private Topic mForwardingTopic;

    /**
     * Default constructor.
     */
    public ListenerFacadeEJB()
    {
        System.out.println("***** ListenerFacadeEJB created");
    }

    @PostConstruct
    public void initialize() throws Exception
    {
        /*
         * This dependency, being a message destination reference declared
         * in the deployment descriptor cannot be injected, but must be
         * looked up this way.
         */
        InitialContext theContext = new InitialContext();
        mForwardingTopic = (Topic)theContext.lookup(PUBLISH_TOPIC_JNDI_NAME);
    }

    /**
     * @see MessageListener#onMessage(Message)
     */
    @Override
    public void onMessage(Message inReceivedMessage)
    {
        MessageProducer theForwardingMsgProducer = null;
        Connection theForwardingConnection = null;
```

```

try
{
    /*
     * To send a message to a queue or topic:
     * - Create a connection using a JMS connection factory.
     * In this case, the connection factory can create connections
     * for both queues and topics.
     * - Create a JMS session for outgoing messages.
     * - Create a message producer, which is the object that is
     * to send the message.
     */
    theForwardingConnection = mJMSSConnectionFactory.createConnection();
    Session theForwardingSession =
        theForwardingConnection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
    theForwardingMsgProducer =
        theForwardingSession.createProducer(mForwardingTopic);

    /*
     * Don't need to create a new message, we just set a flag in the
     * received message indicating that it is re-delivered.
     */
    inReceivedMessage.setJMSRedelivered(true);

    /* Send the message to the topic. */
    theForwardingMsgProducer.send(inReceivedMessage);
} catch (final Exception theException)
{
    theException.printStackTrace();
} finally
{
    closeJMSResources(theForwardingMsgProducer, theForwardingConnection);
}

System.out.println("***** ListenerFacaceEJB - Forwarded message");
}

private void closeJMSResources(MessageProducer inForwardingMsgProducer,
    Connection inForwardingConnection)
{
    if (inForwardingMsgProducer != null)
    {
        try
        {
            inForwardingMsgProducer.close();
        } catch (JMSEException theException)
        {
            // Ignore exceptions.
        }
    }
    if (inForwardingConnection != null)
    {
        try
        {
            inForwardingConnection.close();
        } catch (JMSEException theException)
        {
            // Ignore exceptions.
        }
    }
}
}

```

Note that:

- The reference to the topic to pass the messages on to must be looked up using its JNDI name, as is done in the *initialize* method. It cannot be dependency-injected, because no destination resource with the JNDI name exists in the application server.

First Topic Listener EJB

The first and second topic listener EJBs are identical, except for the name of the EJBs and some text in the messages. Upon receiving a message, they just print output to the console.

```
package comivan.listeners;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

/**
 * This class implements a message driven EJB that listens for messages
 * on a topic.
 * Note that the @MessageDriven annotation contains an element named
 * mappedName, which must be present and must have a JNDI name that
 * exists in the application server (GlassFish v3).
 * This is, however, not the JNDI name that the EJB will listen on.
 * This JNDI name will be overridden by the <message-destination-link>
 * in the deployment descriptor.
 *
 * @author Ivan A Krizsan
 */
@MessageDriven(name="ListenerMDB1", mappedName = "jms/StockInfo", activationConfig =
{
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Topic")
})
public class FirstListenerEJB implements MessageListener
{

    /**
     * Default constructor.
     */
    public FirstListenerEJB()
    {
        System.out.println("***** FirstListenerEJB created");
    }

    /**
     * Delivers a message to the message driven bean.
     *
     * @param inMessage Message to be delivered.
     */
    @Override
    public void onMessage(final Message inMessage)
    {
        System.out.println("***** FirstListenerEJB onMessage start");

        if (inMessage instanceof ObjectMessage)
        {
            ObjectMessage theObjMsg = (ObjectMessage)inMessage;
            System.out.println("      Object message: " + theObjMsg);
        } else
        {
            System.out.println("      Not object message.");
        }

        System.out.println("***** FirstListenerEJB onMessage end");
    }
}
```

Note that:

- The optional element mappedName in the @MessageDriven annotation must have a value that is the JNDI name of an existing destination resource in the application server (GlassFish v3, in my case – perhaps this is a peculiarity of GlassFish), otherwise we won't be able to deploy the EJB to the server. This JNDI name can be any valid JNDI name, though, since this is not the JNDI name of the queue the EJB will listen for messages on – the <message-destination-link> in the ejb-jar.xml deployment descriptor overrides this configuration.

Second Topic Listener EJB

As before, the second topic listener is almost identical to the first topic listener discussed above – please refer to the comments above!

```
package com.ivan.listeners;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

/**
 * This class implements a message driven EJB that listens for messages
 * on a topic.
 * Note that the @MessageDriven annotation contains an element named
 * mappedName, which must be present and must have a JNDI name that
 * exists in the application server (GlassFish v3).
 * This is, however, not the JNDI name that the EJB will listen on.
 * This JNDI name will be overridden by the <message-destination-link>
 * in the deployment descriptor.
 */
@MessageDriven(name="ListenerMDB2", mappedName = "jms/StockInfo", activationConfig =
{
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Topic")
})
public class SecondListenerEJB implements MessageListener
{
    /**
     * Default constructor.
     */
    public SecondListenerEJB()
    {
        System.out.println("***** SecondListenerEJB created");
    }

    /**
     * Delivers a message to the message driven bean.
     *
     * @param inMessage Message to be delivered.
     */
    @Override
    public void onMessage(final Message inMessage)
    {
        System.out.println("***** SecondListenerEJB onMessage start");

        if (inMessage instanceof ObjectMessage)
        {
            ObjectMessage theObjMsg = (ObjectMessage)inMessage;
            System.out.println("      Object message: " + theObjMsg);
        } else
        {
            System.out.println("      Not object message.");
        }

        System.out.println("***** SecondListenerEJB onMessage end");
    }
}
```

EJB Deployment Descriptor

The ejb-jar.xml deployment descriptor is where messages produced by the listener facade EJB are routed to the two topic listener EJBs.

In the producer, a <message-destination-ref> element, with a number of child elements, are used to specify the following things:

- A JNDI name referencing a message destination to which messages will be produced to. This JNDI name is the JNDI name that is used in the EJB when programmatically looking up the destination resource to which to publish the messages.
This JNDI name does not have to correspond to a destination resource in the application server.
- Optionally, we can also specify the destination type, that is, whether the producer publishes messages to a queue or a topic.
- We can also, optionally, specify whether the EJB produces to, consumes from or both produces and consumes from the destination resource.
- The message destination link.
The message destination link name, which is required, acts as an alias for a JNDI name referring to the destination resource in the application server to which messages will be produced and from which messages will be consumed.

In the deployment descriptor configuration of the consumer(s), the message driven bean(s), only two discrete elements are used in connection to message linking:

- The optional <message-destination-type> element specifies whether the EJB is listening on a queue or a topic.
- The <message-destination-link> element, which is required, again acts as an alias for a JNDI name and specifies the JNDI name used to look up the reference to the destination resource the message-driven bean is listening on.
Specifying a <message-destination-link> in a MDB overrides any annotation-based configuration.

Finally, lets take a look at the finished ejb-jar.xml deployment descriptor, which is to be located in the META-INF directory in the root of the source-path of the project:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:ejb="http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1">

  <enterprise-beans>
    <message-driven>
      <ejb-name>ListenerFacadeMDB</ejb-name>
      <message-destination-link>ListenerMsgDest</message-destination-link>

      <message-destination-ref>
        <!--
          JNDI name used in the EJB to look up the destination
          reference. Note that this name does not have to correspond
          to a destination in the application server if a message
          destination link is used.
          When a message destination link is used, this reference
          has to be looked up programmatically or being injected
          from the deployment descriptor and cannot be
          injected using annotations.
        -->
    
```

```

-->
<message-destination-ref-name>jms/PublishingTopic</message-destination-
ref-name>
<!--
    Here, the type of the destination which is referenced
    can be specified using a <message-destination-type>
    element. We can also specify how the destination reference
    is used; whether the EJB produces or consumes messages
    using the reference.
    Both these configurations are optional.
-->
<message-destination-type>javax.jms.Topic</message-destination-type>
<message-destination-usage>Produces</message-destination-usage>
<!--
    Message destination link enabling linking message
    producers with message consumers within one and the
    same application.
    The <message-destination-link> element is defined in
    the message producer.
-->
<message-destination-link>ListenerMsgDest</message-destination-link>
</message-destination-ref>
</message-driven>

<message-driven>
    <ejb-name>ListenerMDB1</ejb-name>
    <ejb-class>com.ivan.listeners.FirstListenerEJB</ejb-class>
    <messaging-type>javax.jms.MessageListener</messaging-type>
    <transaction-type>Container</transaction-type>

    <message-destination-type>javax.jms.Topic</message-destination-type>
    <message-destination-link>ListenerMsgDest</message-destination-link>
</message-driven>

</enterprise-beans>

<assembly-descriptor>
    <!--
        This element is used to specify a link between a logical message
        destination name, used in <message-destination-link> elements
        above, and a physical JNDI name.
    -->
    <message-destination>
        <!-- Logical name. -->
        <message-destination-name>ListenerMsgDest</message-destination-name>
        <!--
            JNDI name of a physical queue defined in the application
            server
        -->
        <lookup-name>jms/ForwardingTopic</lookup-name>
    </message-destination>
</assembly-descriptor>
</ejb-jar>

```

Note the last section of the deployment descriptor, the <assembly-descriptor> element with its <message-destination> child element. It is there that the mapping between the logical message destination link name and the physical JNDI name is done.

The advantage of using message destination links is that the JNDI name is used only at one single place and exchanging it for another JNDI name becomes very easy.

This concludes the message destination link example program.

Appendix D – Configuring Messaging in GlassFish v3

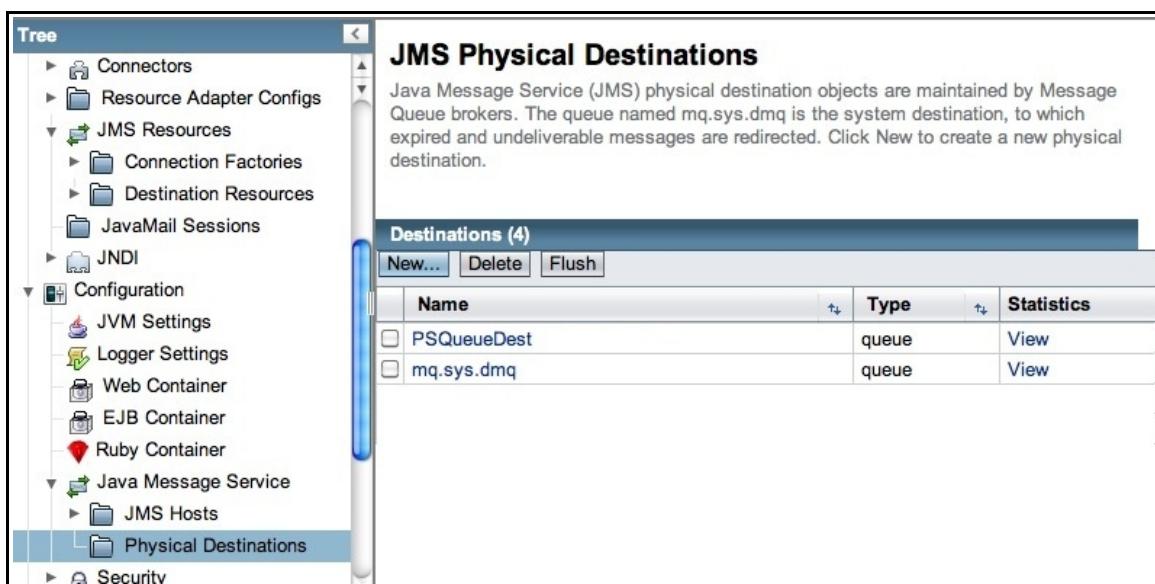
In order to be able to, for instance, receive messages in a message-driven EJB that is running in an application in GlassFish v3 we need to configure messaging in the server. The following procedure creates a new physical destination, which is the actual queue or topic in the message broker, and then creates the associated connection factory and destination resource.

- Log in to the GlassFish administration console.

Create a JMS Physical Destination

The first step is to create a JMS Physical Destination in GlassFish. If you already have a physical destination that you want to use, this step can be skipped.

- In the left pane, go to the Configuration -> Java Message Service -> Physical Destinations node.



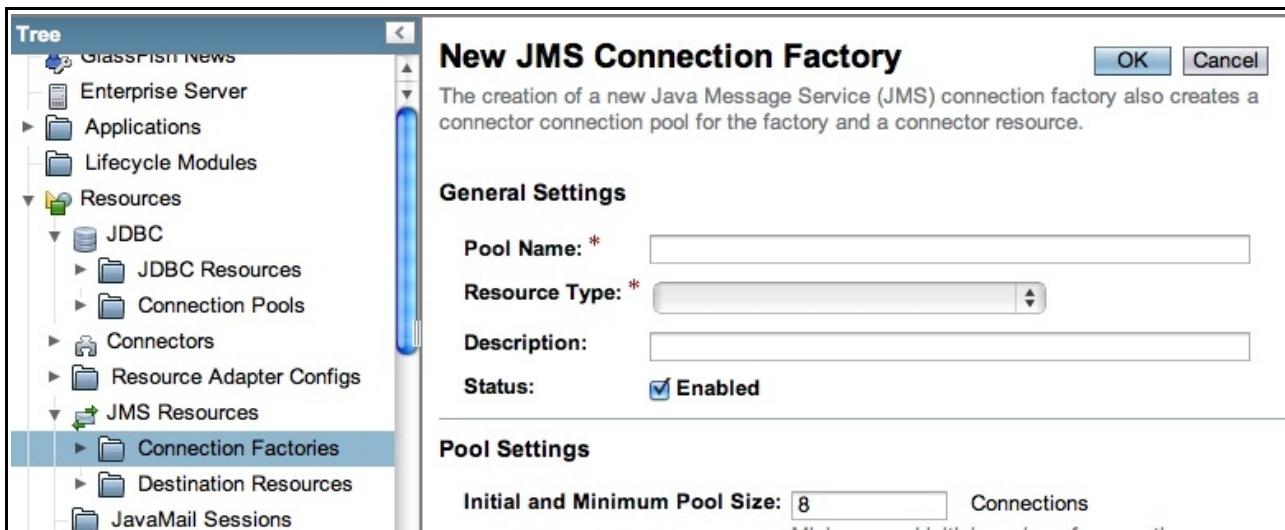
JMS Physical Destinations configuration node in GlassFish v3.

- Create a new JMS Physical Destination by clicking the New button.
The following steps will describe configuration of the new physical destination.
- Enter the name of the new JMS Physical Destination.
Note down this name, since we will need it when configuring the new JMS Destination Resource we'll create later.
- Select the type of the JMS Physical Destination.
In the default configuration of GlassFish v3, this is either *javax.jms.Topic* or *javax.jms.Queue*.
- Configure the additional settings of the new JMS Physical Destination as desired.
If you do not have any special requirements, then use the default settings. It is possible to change the settings of a physical destination later.
- Click the Save button to finish creation of the new JMS Physical Destination.

Create a JMS Connection Factory

This step creates and configures a new JMS Connection Factory in GlassFish v3. Again, if you already have a connection factory you want to use, skip this step.

- In the left pane, go to Resources -> JMS Resources -> Connection Factories.
- Create a new JMS Connection Factory by clicking the New button.



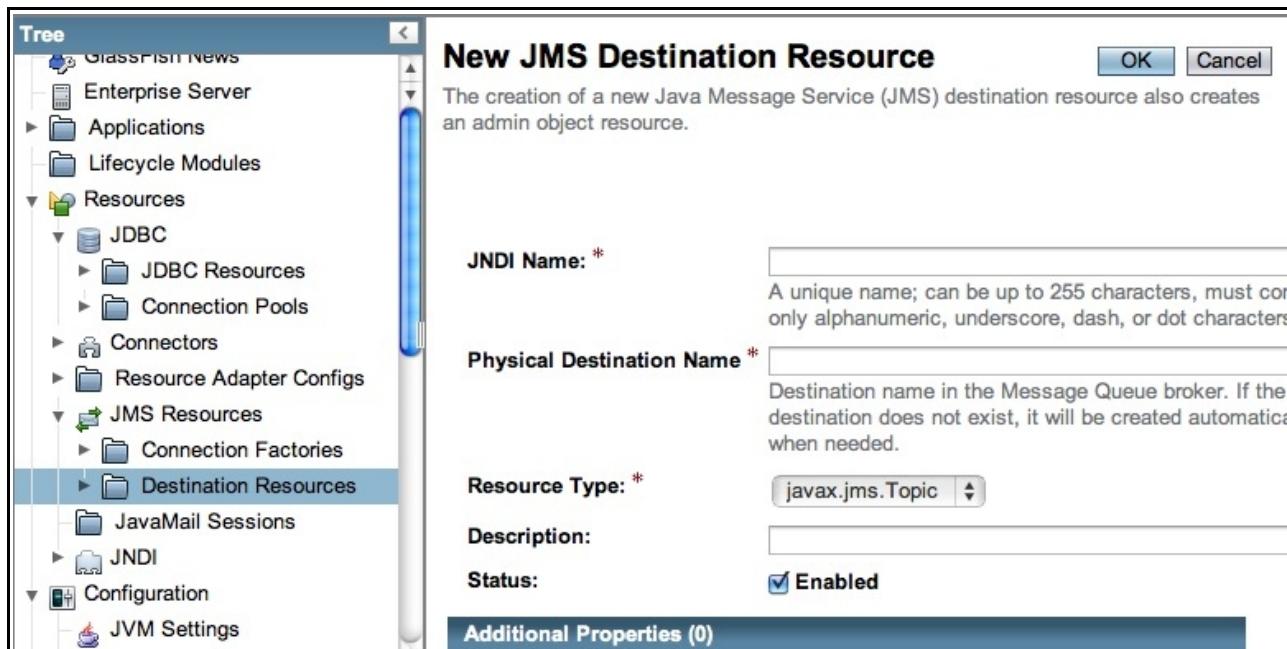
Creating a new JMS Connection Factory in GlassFish v3.

- Enter a Pool Name.
This name will also become the JNDI name used when injecting or looking up a reference to the connection factory later.
- Choose the Resource Type.
There are three different types to choose from:
 - *javax.jms.TopicConnectionFactory* – Factory creating connections to be used when sending/receiving messages from a JMS topic.
 - *javax.jms.QueueConnectionFactory* – Factory creating connections to be used when sending/receiving messages from a JMS queue.
 - *javax.jms.ConnectionFactory* – Factory creating connections that can be used with both JMS topics and JMS queues.
- Configure the additional settings of the JMS Connection Factory.
If you do not have any special requirements, then use the default settings. It is possible to change the settings of a connection factory later.
- Click the Save button to finish creation of the new JMS Connection Factory.

Create a JMS Destination Resource

The final step is creating a JMS Destination Resource, which links a JNDI name with a physical destination.

- In the left pane, go to Resources -> JMS Resources -> Destination Resources.
- Create a new JMS Destination Resource by clicking the New button.



Creating a new JMS Destination Resource in GlassFish v3.

- Enter a JNDI Name.
This is the name later used when injecting or looking up the JMS Destination Resource. The name must be unique.
- Enter a Physical Destination Name.
This is the name of the JMS Physical Destination we created earlier.
- Specify the Resource Type.
This can be either *javax.jms.Topic* or *javax.jms.Queue* and must match the type of the JMS Physical Destination used.
- Click the OK button to finish creation of the JMS Destination Resource.

Appendix E – Defining Resource Environment References in GlassFish v3

This section describes how to create a resource environment reference to a *Properties* object holding a number of key-value pairs in the GlassFish v3 application server.

- Log in to the GlassFish administration console.
- Click on the JNDI node in the list to the left.
- Click on Custom Resources in the pane on the right.
- Click the New button in the Custom Resources pane.
- Fill in data about the new resource.

In the picture below, there is an example of how I set up a reference to a *Properties* object holding a single key-value pair.

New Custom Resource

Ensure that the resource references and EJB references are linked to the configured server-wide resources,

JNDI Name: * A unique name; can be up to 255 characters, must contain only alphanumeric, underscore, and dash characters.

Resource Type: java.util.Properties

Enter a fully qualified type following the format xxx.xxx (for example, javax.jms.Topic)

Factory Class: * Factory class for resource; implements javax.naming.spi.ObjectFactory

Description:

Status: Enabled

Additional Properties (1)

Name	Value
<input type="checkbox"/> property	value1

Creating a custom resource reference in GlassFish v3.

- When finished entering data, click the OK button.

See the section on [resource environment reference injection](#) below for an example on how to consume the resource.

Appendix F – Setting Up JDBC Resources in GlassFish v3

In this appendix I will show how to configure a Connection Pool and a JDBC Resource using the embedded Derby database in GlassFish v3. These preparations is necessary if you want to inject a [persistence unit reference](#) or a [persistence context reference](#) into an EJB.

Finally, I will show how to configure a JPA persistence.xml file, in order to be able to access the persistence resources from a web application.

As with other GlassFish configuration, the first step is:

- Log in to the GlassFish administration console.

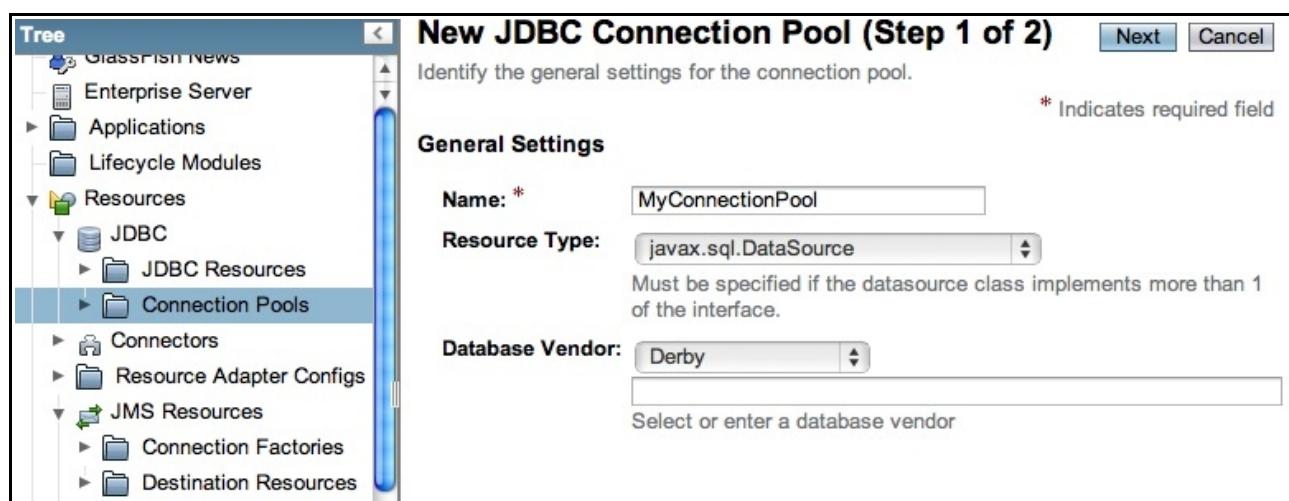
Create a JDBC Connection Pool

In this step we'll create a connection pool – a necessary prerequisite before we can create a JDBC resource.

- In the left pane, go to the Resources -> JDBC -> Connection Pools node.

- Create a new Connection Pool by clicking the New button.

The following steps will describe configuration of the new connection pool.



Creating a new JDBC Connection Pool in GlassFish v3 – step 1 of 2.

- Enter the Name of the new connection pool.

Note down this name, as we will need it when creating the JDBC resource later.

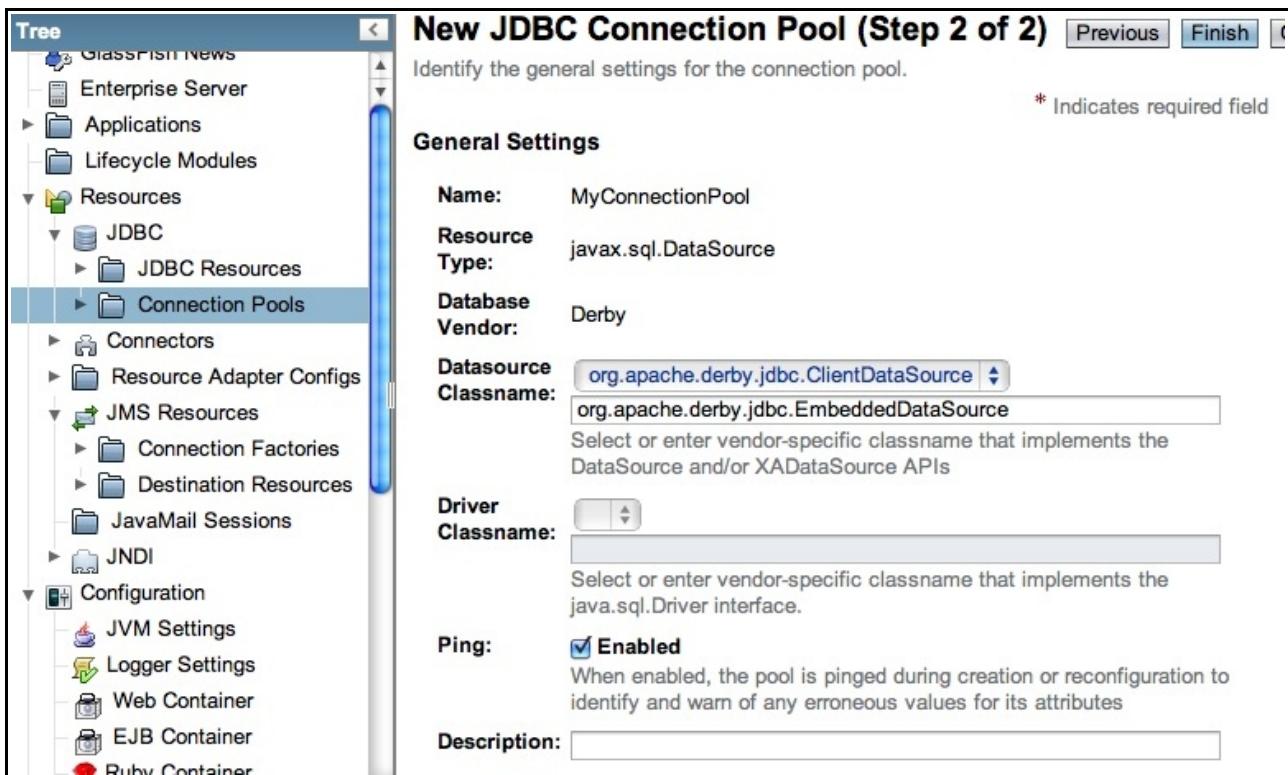
- Select the Resource Type of the new connection pool.

Choose the *javax.sql.DataSource* type.

- Select the Database Vendor.

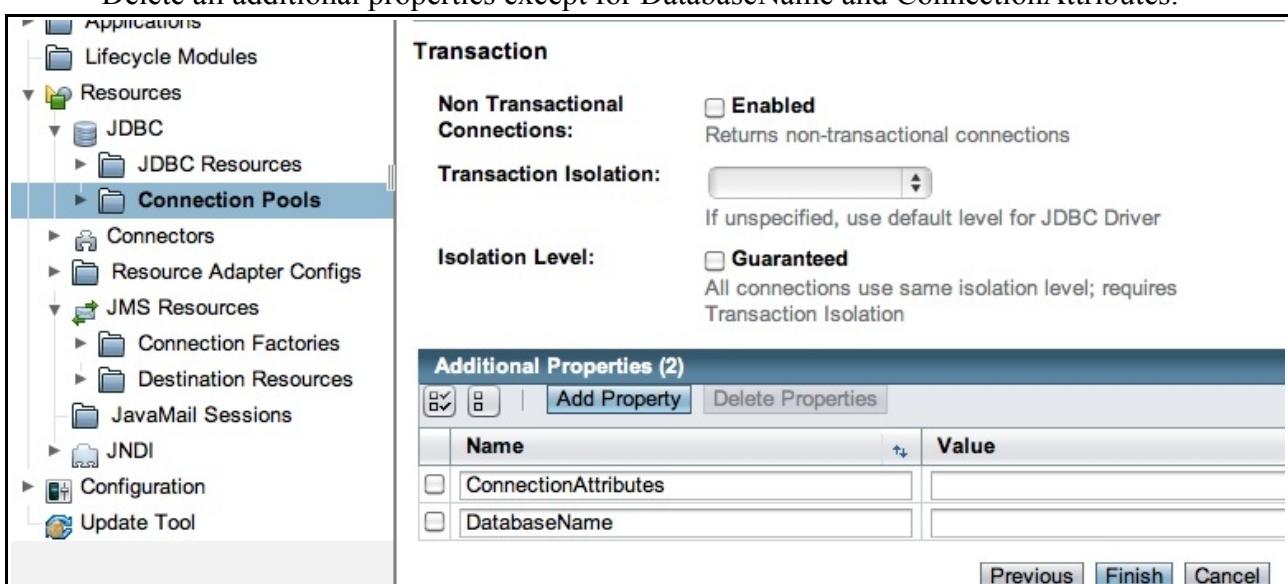
We will use the embedded Derby database, so select Derby.

- Having finished the first step of the JDBC Connection Pool creation, click the Next button.



Creating a new JDBC Connection Pool in GlassFish v3 – step 2 of 2.

- In the text-field below the Datasource Classname popup-menu, enter the classname *org.apache.derby.jdbc.EmbeddedDataSource*.
This datasource class allows us to use the embedded Derby database.
- Optionally, check the Ping checkbox.
- Scroll down to the Additional Properties section.
- Delete all additional properties except for DatabaseName and ConnectionAttributes.



Setting the JDBC Connection Pool additional properties in step 2 of 2.

- Set the DatabaseName property value to the desired name of your database. I set it to “MyNewDB”, without the quotes. The data of the database will be stored in a directory with the same name as the database in the config directory in the GlassFish domain directory to which the application is deployed.
- Set the ConnectionAttributes property value to “;create=true”, again without the quotes.
- Create a new property named URL and set its value to “jdbc:derby:” and append the name of the database. In my case, the URL is “jdbc:derby:/MyNewDB”.
- The remaining parameters are left at their default values.
- Having finished configuring the connection pool, click the Finish button.

Create a JDBC Resource

Having created a JDBC Connection Pool, we are now ready to create a JDBC Resource. The JDBC Resource links a connection pool to a JNDI name and enables us to access the underlying database from within our (web) application.

- In the left pane, go to the Resources -> JDBC -> JDBC Resources node.
- Create a new JDBC Resource by clicking the New button.

The following steps will describe configuration of the new resource.



Creating a new JDBC Resource in GlassFish v3.

- Enter the JNDI Name of the new JDBC Resource. JNDI names of JDBC resources are commonly prefixed with jdbc. In my case, the JNDI name of the new resource is “jdbc/MyJDBCResource” without the quotes.
- Select a Pool Name. Using the popup-menu, select the name of the JDBC connection pool created earlier.
- Having finished configuring the JDBC resource, click the OK button.

Configuring Persistence Settings in Web Applications

Finally, after having set up both a JDBC Connection Pool and a JDBC Resource, we are ready to configure persistence for a web application. JPA persistence configuration is entered in a file named persistence.xml.

This file must, when the web application is deployed, be located in the WEB-INF/classes/META-INF directory. This can be accomplished by creating a META-INF directory in the root source director and, in this directory, create the persistence.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

  <!--
    Note that this file must be placed so that it ends up in
    WEB-INF/classes/META-INF, otherwise the persistence unit
    will not be created!
  -->
  <persistence-unit name="MsgDrvEx2PU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/MyJDBCResource</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation" value="create-tables" />
    </properties>
  </persistence-unit>
</persistence>
```

We configure the following things in the persistence.xml file:

- A persistence provider class.
Configured in the <provider> element. In the above example, we use the EclipseLink persistence provider. EclipseLink is embedded in GlassFish v3.
- The JNDI name of the JDBC resource the persistence unit is to use.
Configured in the <jta-data-source> element.
- Additional properties for the provider.
The property in the above example instructs EclipseLink to create tables in the database.

Appendix G – GlassFish Security Configuration

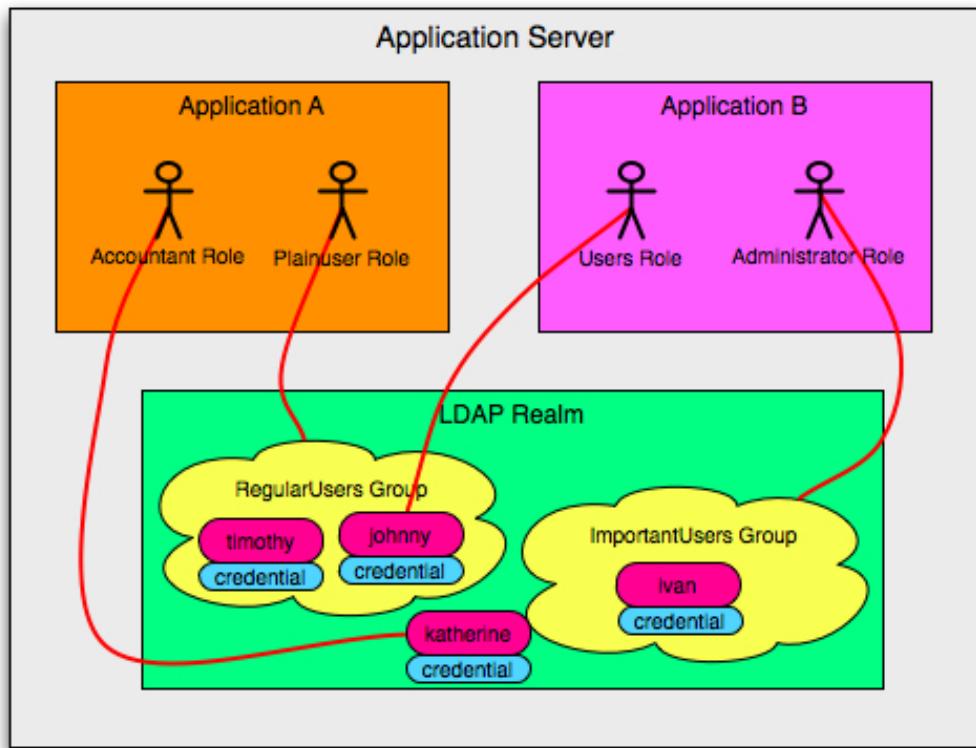
In this appendix there will be a very brief description of how to configure basic web application and EJB Java EE security when using GlassFish v3. It is meant as a reminder or as a way to quickly get started with the security-related examples in this book. Please consult other books and reference material for an in-depth discussion on Java EE and GlassFish security.

Basic Concepts

When dealing with EJB and web application security, there are a few concepts that will appear frequently:

- **Credential**
Information used to authenticate a principal.
A common type of credential is a password.
- **Group**
A number of users with some common characteristics, such as access privileges.
For instance, users that are allowed to use a system but not allowed to administer it may belong to a group called “plainusers”.
- **Principal**
Connects an identity with a credential which can be used for authentication. For instance, associates a user with a password.
- **Role**
A concept used to assign access privileges in an application. Roles are used in the application to specify different areas of the application that requires different privileges to access. For instance, an EJB implementing account management may require only regular user privileges to list the accounts, but may require privileges of an accountant to create and remove accounts. The application may thus declare a “regular-user” role and an “accountant” role. The roles are later mapped to application server users and/or user groups using a server-specific deployment descriptor. In GlassFish v3 this is the sun-web.xml or the sun-ejb-jar.xml deployment descriptors.
- **Security Realm**
A security realm specifies an identity storage technique, like JDBC, LDAP or flat file, and defines how to interact with the identity storage in order to store and retrieve user's authentication and grouping information.
- **User**
One individual user of the system that, having presented some kind of credentials, is allowed to use the system in whole or part.

The figure on the next page shows the relation of some of the concepts listed above.



The relationships between roles, users, groups, security realms, principal and credentials.

In words, the above figure can be expressed:

- There is only one single Security Realm in the application server – the LDAP Realm. Looking at its name, a qualified guess would be that it stores and retrieves user information and credentials using LDAP.
- The LDAP security realm contains two Groups; the “RegularUsers Group” and the “ImportantUsers Group”.
- The “RegularUsers Group” contains two Users; “timothy” and “johnny”.
- The “ImportantUsers Group” contains one User; “ivan”.
- The User “katherine” does not belong to any Group.
- Each User has some kind of credential associated with it. This can be a password, a password hash etc.
- Application A defines the “Accountant” and “Plainuser” roles.
- In Application A, the “Accountant” Role is mapped to the Principal “katherine”.
- In Application A, the “Plainuser” Role is mapped to the Group “RegularUsers Group”.
- Application B defines the “Users” and “Administrator” roles.
- In Application B, the “Users” Role is mapped to the single User “johnny”.
- In Application B, the “Administrator” Role is mapped to the Group “ImportantUsers Group”.

Creating Users in GlassFish

In order to add security to a Java EE application, we need to create one or more users in a security realm in the GlassFish application server.

- Open the GlassFish v3 administration console and log in.
- In the tree on the left, navigate to the Configuration -> Security -> Realms -> file node.

The screenshot shows the GlassFish administration console interface. On the left, there is a tree view of configuration nodes. Under the 'Security' section, the 'Realms' node is expanded, and its child node 'file' is selected and highlighted with a blue selection bar. The main right panel is titled 'Edit Realm' and contains the following fields:

- Realm Name:** file
- Class Name:** com.sun.enterprise.security.auth.realm.file.FileRealm
- Properties specific to this Class**
 - JAAS Context:** fileRealm
 - Key File:** \${com.sun.aas.instanceRoot}/config/keyfile
 - Assign Groups:** (empty)
- Additional Properties (0)**

Name	Value	Description
No items found.		

Locating the file security realm in the GlassFish v3 administration console.

- Click the Manage Users button in the upper left corner of the Edit Realm frame. There will be a list of user in the file realm, similar to this:

File Users

Manage user accounts for the currently selected security realm.

Realm Name: file

File Users (5)		
New...	Delete	
	User ID	Group List
<input type="checkbox"/>	user	wsit

List of users in the file realm.

- Click the New button to create a new user.
- Enter the name of the user, the name of the groups the user belong to and the user's password.

New File Realm User

Create new user accounts for the currently selected security realm.

* Indicates required field

Realm Name:	file
User ID *	ivan
Name of a user to be granted access to this realm; name can be up to 255 characters, must contain only alphanumeric, underscore, dash, or dot characters	
Group List	super-users,plain-users
Separate multiple groups with commas	
New Password
Confirm New Password

Entering data of a new file realm user.

- Click the OK button in the upper right corner of the frame. Upon successful creation of the user, you will be taken back to the file realm user list with the new user appearing in the list.

Security Role Mapping

In order for a user or a group of users defined according to the previous section to have access privileges in an application, the user(s) or group(s) must be mapped to a role used in the application. Such role-to-principal or role-to-group mapping can be done in the following GlassFish specific deployment descriptors:

- sun-application.xml
- sun-web.xml
- sun-ejb-jar.xml

The mapping described below shows how to map a single user and a group of users to roles in the sun-web.xml deployment descriptor, but the principle is the same for all the different deployment descriptors.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">

    <context-root>/EJBAnnotations</context-root>

    <!--
        The application's role "superusers" is mapped to the application
        server principal (user) "ivan".
        This gives the user "ivan" defined in the application server
        access to the parts of the application that requires the user
        to be in the "superusers" role.
    -->
    <security-role-mapping>
        <role-name>superusers</role-name>
        <principal-name>ivan</principal-name>
    </security-role-mapping>

    <!--
        The application's role "plainusers" is mapped to the application
        server's user group "plain-users"
        This gives all the users defined in the application server that
        belongs to the "plain-users" group access to the parts of the
        application that requires the user to be in the "plainusers"
        role.
    -->
    <security-role-mapping>
        <role-name>plainusers</role-name>
        <group-name>plain-users</group-name>
    </security-role-mapping>

    <class-loader delegate="true" />
    <jsp-config>
        <property name="keepgenerated" value="true" />
    </jsp-config>
</sun-web-app>
```

Note that:

- Each <security-role-mapping> element allows the mapping of one or more principals (users) and/or one or more user groups to a single role.
This is not shown in the example above, where only a single user or group is mapped to a role.
- Each <security-role-mapping> element contains one <role-name> child element.
- A <role-name> element specifies the name of the role in the application to which the following principals (users) and groups are to be mapped to.
- Each <principal-name> element specifies the name of a user defined in a realm in the GlassFish application server.
- Each <group-name> element specifies the name of a user group defined in a realm in the GlassFish application server.
- The first mapping mapping the user “ivan” to the role “superusers” allows the user “ivan” defined in the application server access (after having presented the appropriate credentials (a password)) to the parts of the application that requires the user to be in the “superusers” role.
- The second mapping mapping the user group “plain-users” to the role “plainusers” allows all the users in the group “plain-users” defined in the application server access (again after having presented the appropriate credentials) to the parts of the application that requires the user to be in the “plainusers” role.

Appendix H – Embeddable EJB Container Example

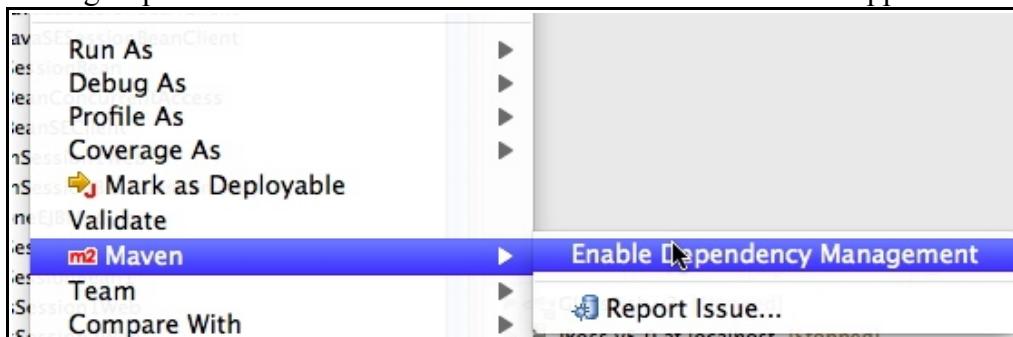
References: EJB 3.1 Specification, chapter 22.

This appendix contains an example Java SE application that uses the embeddable GlassFish EJB 3.1 container. The example was written gleaned on an example by Arun Gupta – thanks!

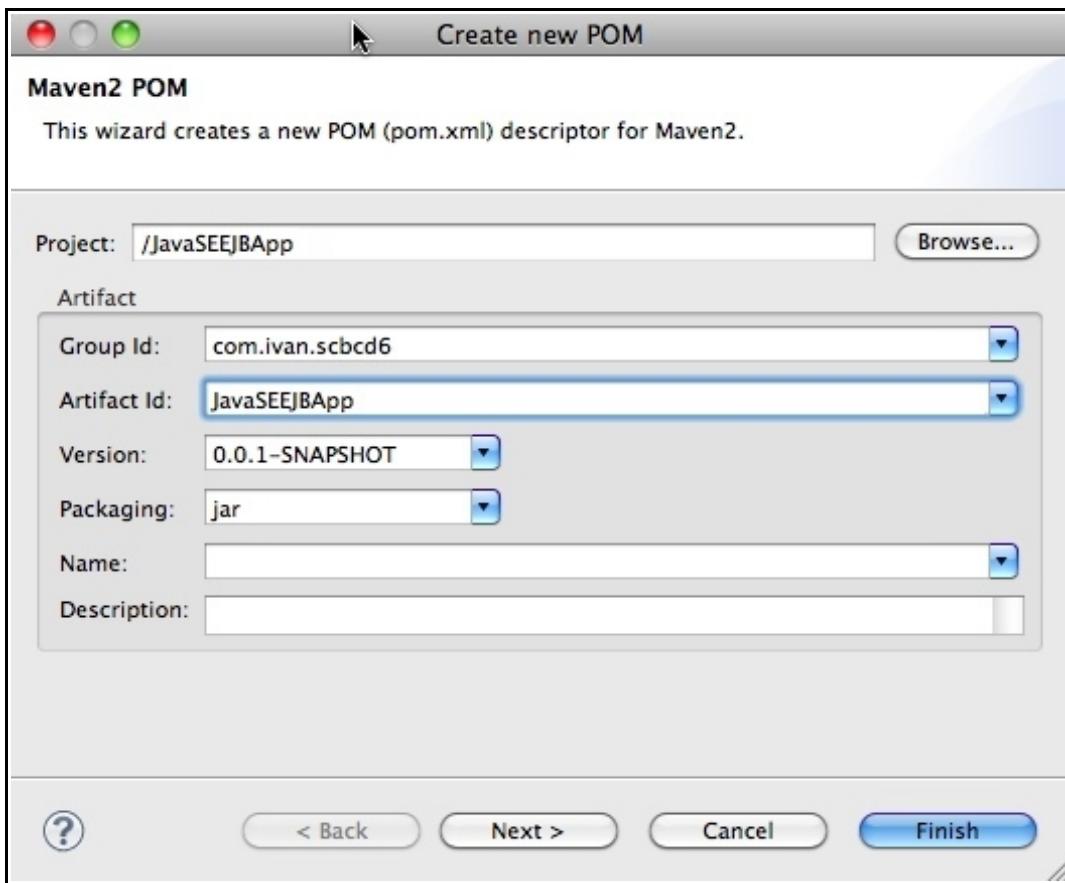
Creating the Example Project

The example project is a regular Java SE project with Maven dependency management.

- In Eclipse, create a Java Project.
I call my project “JavaSEEJBApp”.
- Enable Maven dependency management for the project.
In the Package or Project Explorer, right-click the project, select Maven -> Enable Dependency Management.
I use the group id “com.ivan.scbcd6” and the artifact id “JavaSEEJBApp”.



Enabling Maven dependency management on the project.

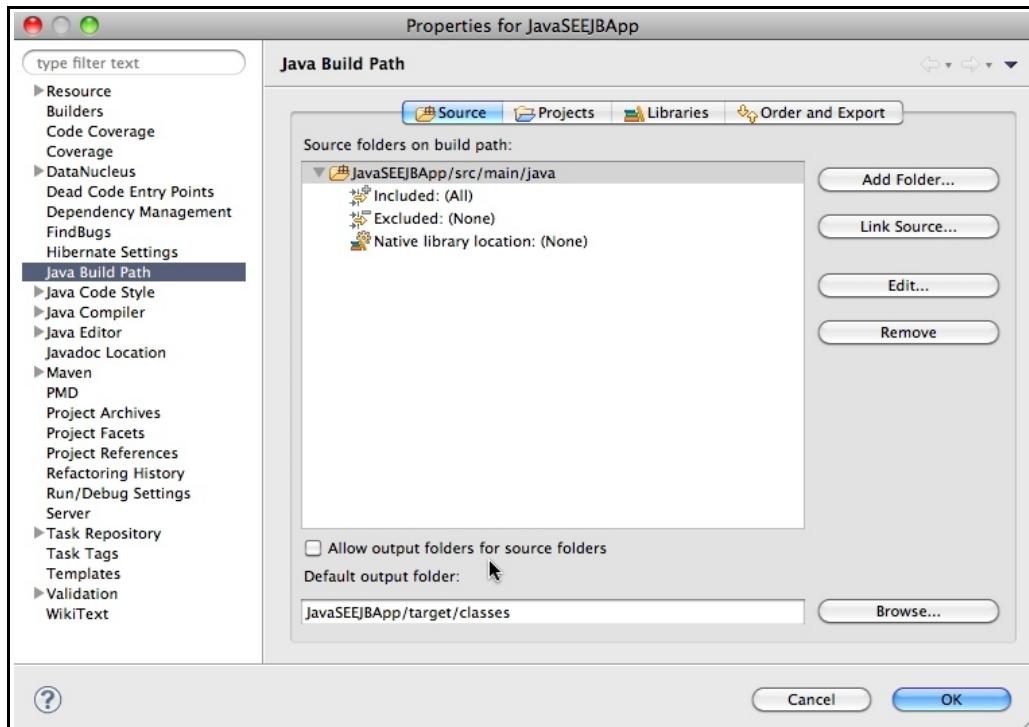


Entering Maven group and artifact id as part of enabling Maven dependency management.

- Create the Java source folders.

In the src folder, create a folder named “main” and in the “main” folder, create a folder named “java”.

- In the Eclipse project properties, set the “java” folder created in the previous step to be a source folder. The result should look like this:



The project settings with the /src/main/java folder assigned to be a source folder.

- Also in the Eclipse project properties for the project, in the Java Compiler tab, disable project specific settings.
It is assumed that the default Eclipse JRE is a Java 6 JRE.
- Make sure that Java SE 6 libraries are included on the project classpath.

Updating the Maven pom.xml File

The Maven pom.xml file need to be updated with the GlassFish embeddable EJB container dependency and information on the repository from which the JAR can be downloaded.

- Open the project's pom.xml file and modify the contents so the final result is like in the listing below.

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.ivan.scbcd6</groupId>
  <artifactId>JavaSEEJBApp</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <repositories>
    <repository>
      <id>download.java.net</id>
      <name>Java.net Maven Repository</name>
      <url>http://download.java.net/maven/glassfish</url>
    </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>org.glassfish.extras</groupId>
      <artifactId>glassfish-embedded-all</artifactId>
      <version>3.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

Note that:

- The group and artifact ids may vary, depending on your project setup.
- In order for Maven to find the JAR containing the GlassFish embeddable container we need to add a the Java.net repository.

Creating the EJB Class

The EJB class used in this example is a stateless session bean that has a local view.

- In the source folder, create the *com.ivan.scbcd6.ejb*s package.
- In the newly created package, implement the EJB:

```
package com.ivan.scbcd6.ejb;

import java.util.Date;
import javax.ejb.Local;
import javax.ejb.Stateless;

@Local
@Stateless(name="MyStatelessSessionBean")
public class StatelessSessionBean
{
    public String sayHello(String inName)
    {
        return "Hello " + inName + ", it is now " + new Date();
    }
}
```

Creating the Client Class

The client class is responsible for starting up the embeddable EJB container, after which it can retrieve a reference to the EJB developed in the previous section.

- In the source folder, create the *com.ivan.scbcd6.client* package.
- Implement the JavaSEEJBClientMain class:

```
package com.ivan.scbcd6.client;

import java.util.HashMap;
import java.util.Map;
import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import javax.naming.NamingException;
import com.ivan.scbcd6.ejb.StatelessSessionBean;

/**
 * Java SE EJB client using the embeddable EJB 3.1 container.
 * Based on an example by Arun Gupta.
 */
public class JavaSEEJBClientMain
{
    /*
     * EJB JNDI name.
     * If we do not set the application name when creating the embeddable
     * EJB container, the EJB will be given the following JNDI names,
     * depending on where the application is run:
     * In Eclipse: java:global/classes/MyStatelessSessionBean
     * Standalone: java:global/[JAR file name]/MyStatelessSessionBean
     */
    public final static String EJB_JNDI_NAME =
        "java:global/JavaSEEJBApp/MyStatelessSessionBean";
    public final static String APPLICATION_NAME = "JavaSEEJBApp";

    public static void main(String[] args)
    {
        /*
         * Creates an embeddable EJB container instance.
         * Setting the application name gives us consistent JNDI names
         * of EJBs, regardless of where the application is run.
         */
        Map<String, Object> theEjbContainerProperties =
            new HashMap<String, Object>();
```

```

theEjbContainerProperties.put(EJBContainer.APP_NAME, APPLICATION_NAME);
EJBContainer theEjbContainer = EJBContainer.createEJBContainer(
    theEjbContainerProperties);

try
{
    /*
     * Retrieve the JNDI naming context from the embeddable
     * EJB container.
     */
    Context theJndiContext = theEjbContainer.getContext();

    /*
     * Retrieve a reference to the EJB from the JNDI context.
     */
    StatelessSessionBean theEjb = (StatelessSessionBean)
        theJndiContext.lookup(EJB_JNDI_NAME);

    /* Call the EJB and print the result. */
    String theMessage = theEjb.sayHello("Ivan");
    System.out.println("Message from EJB: " + theMessage);
} catch (NamingException theException)
{
    theException.printStackTrace();
} finally
{
    /*
     * Make sure the embeddable EJB container is properly
     * shut down.
     */
    theEjbContainer.close();
}
}
}
}

```

Note that:

- When creating the embeddable EJB container, a map containing properties is enclosed.
In this map, we enclose a property that sets the application name.
See the API documentation of the *EJBContainer* class for further details!
- The application name is set in order to have a consistent JNDI name of the EJB, regardless of whether we run the embeddable EJB container in a unit test or in a standalone Java SE application.
If the application name were not set, then the EJB's JNDI name would become:
In Eclipse: java:global/classes/MyStatelessSessionBean
Standalone: java:global/[JAR file name]/MyStatelessSessionBean
- The embeddable EJB container needs to be explicitly shut down using the *close()* method.
If this is not done, the JVM in which the application or unit test is running will keep executing even after our main class has finished executing.

Running the Example Program in Eclipse

We are now ready to try out the example program in Eclipse.

- Right-click in the class file editor or on the class in the package explorer and select Run As
-> Java Application.
- Console output similar to the following should appear (some output has been omitted to conserve space):

```
Jan 11, 2011 5:56:48 PM org.glassfish.ejb.embedded.EJBContainerProviderImpl getValidFile
INFO: GlassFish v3 (74.2) startup time : Embedded(432ms) startup services(367ms)
total(799ms)
Jan 11, 2011 5:56:49 PM
com.sun.enterprise.transaction.JavaEETransactionManagerSimplified initDelegates
...
INFO: Security service(s) started successfully....
Jan 11, 2011 5:56:55 PM com.sun.ejb.containers.BaseContainer initializeHome
INFO: Portable JNDI names for EJB MyStatelessSessionBean :
[ java:global/JavaSEEJBApp/MyStatelessSessionBean!
com.ivan.scbcd6.ejbs.StatelessSessionBean,
java:global/JavaSEEJBApp/MyStatelessSessionBean]
Message from EJB: Hello Ivan, it is now Tue Jan 11 17:56:55 CET 2011
Jan 11, 2011 5:56:55 PM org.glassfish.admin.mbeanserver.JMXStartupService shutdown
INFO: JMXStartupService and JMXConnectors have been shut down.
Jan 11, 2011 5:56:55 PM com.sun.enterprise.v3.server.AppServerStartup stop
INFO: Shutdown procedure finished
Jan 11, 2011 5:56:55 PM AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] exiting
```

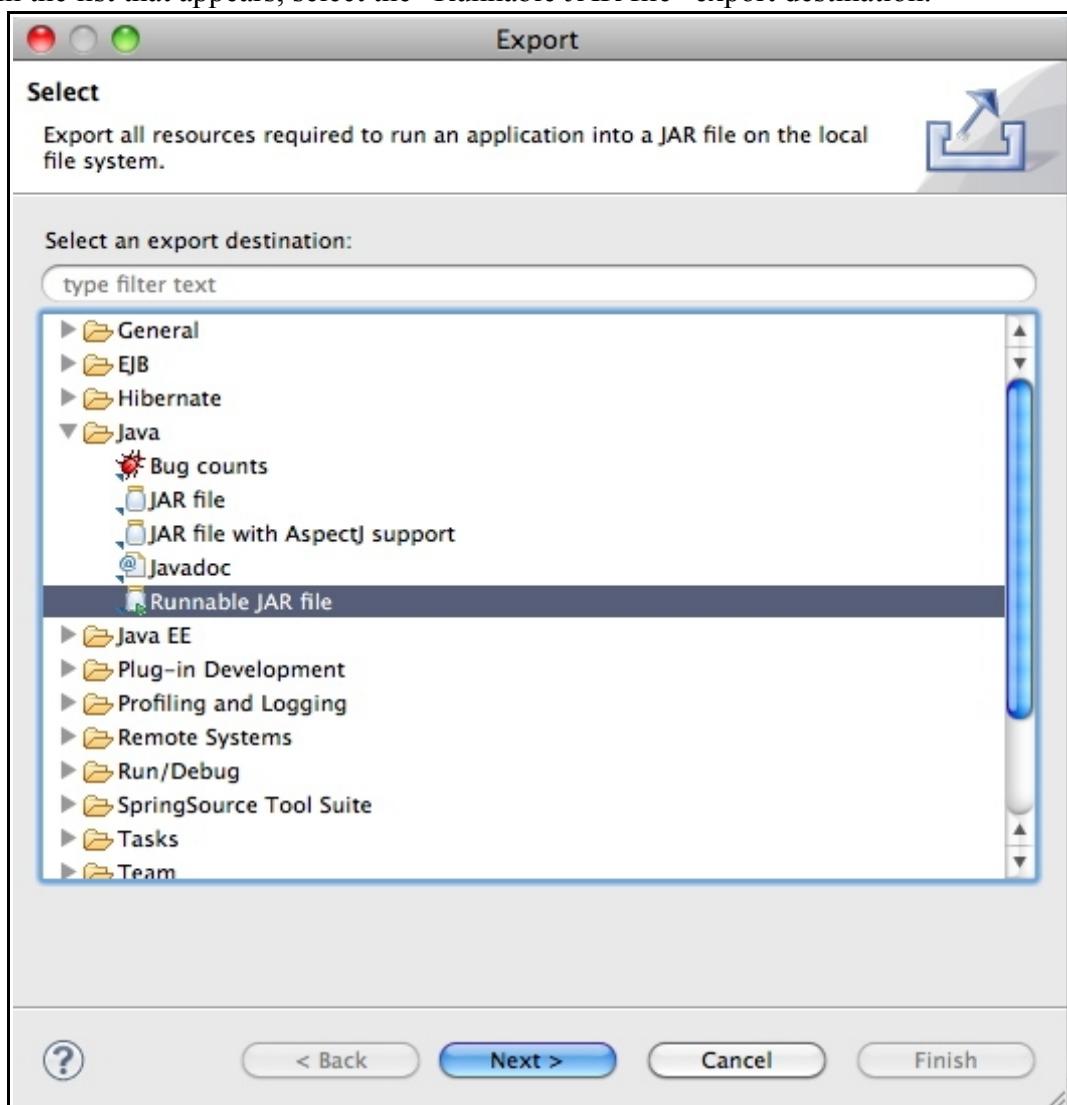
Note that:

- The portable JNDI names for the EJB MyStatelessSessionBean are:
java:global/JavaSEEJBApp/MyStatelessSessionBean!
com.ivan.scbcd6.ejbs.StatelessSessionBean
and
java:global/JavaSEEJBApp/MyStatelessSessionBean
- A message from the EJB is printed.
- The embeddable EJB container is properly shut down and the JVM terminates.

Creating the Standalone Application

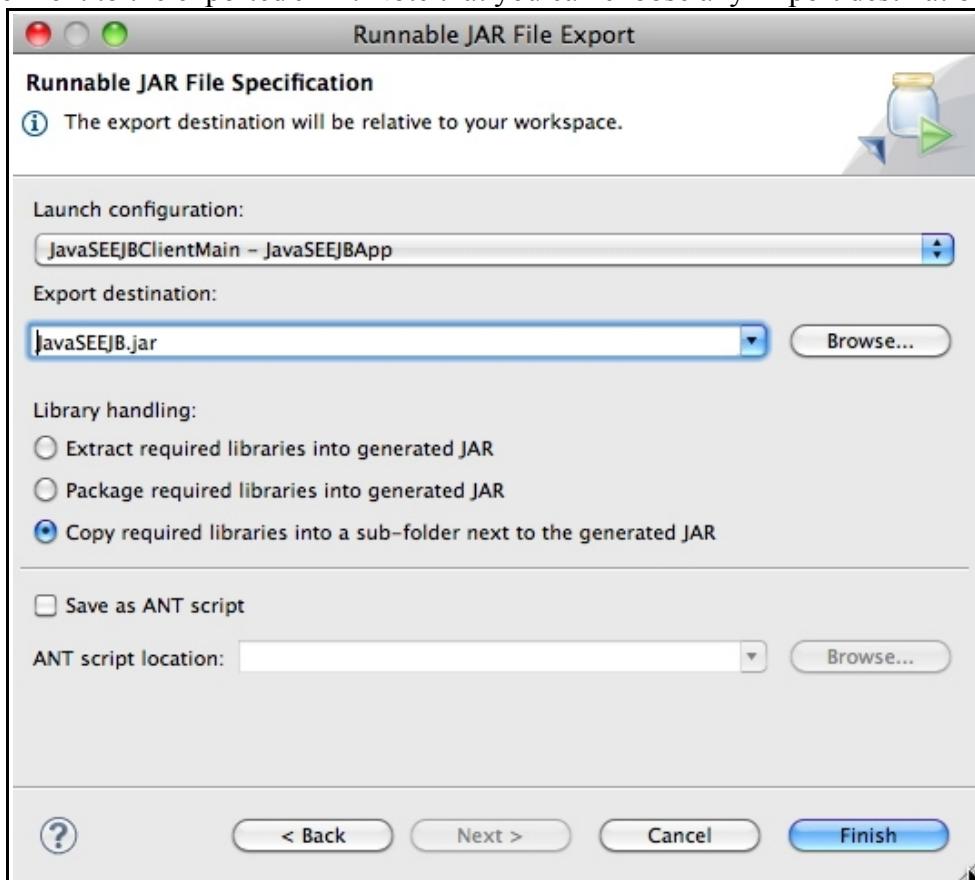
To show that the example program also works as expected when run in a standalone Java SE application, we need to create a JAR file containing the application. In this example, the JAR library file used is written to a directory next to the generated JAR file containing the application.

- In the Package Explorer, right-click on the project and select Export.
- In the list that appears, select the “Runnable JAR file” export destination.



Exporting the example application into a runnable JAR, first step.

- Configure the Runnable JAR file export as in the picture below with libraries copied to a folder next to the exported JAR. Note that you can choose any Export destination.



Configuring the JAR export creating a standalone application that uses the embeddable EJB container.

- Click the Finish button.

The result should be a JAR file with the name of your choice. In my case, it is JavaSEEJB.jar. Next to the JAR file there should be a folder in which you should find the glassfish-embedded-all-3.0 JAR library.

Running the Standalone Application

To verify that the standalone application we built in the previous section also is able to use the EJB, we take it for a test-drive:

- Open a terminal window.
- Navigate to the directory containing the runnable JAR file we created in the previous section.
- Issue the command:
`java -jar JavaSEEJB.jar`

The generated output should be similar to the output generated when we ran the application in Eclipse. Some output has been omitted to conserve space and highlighting has been added to mark interesting sections:

```
...
Jan 13, 2011 7:06:43 AM com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm certificate of classtype
com.sun.enterprise.security.auth.realm.certificate.CertificateRealm successfully
created.
Jan 13, 2011 7:06:43 AM com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security service(s) started successfully....
Jan 13, 2011 7:06:45 AM com.sun.ejb.containers.BaseContainer initializeHome
INFO: Portable JNDI names for EJB MyStatelessSessionBean :
[java:global/JavaSEEJBApp/MyStatelessSessionBean!
com.ivan.scbcd6.ejbs.StatelessSessionBean,
java:global/JavaSEEJBApp/MyStatelessSessionBean]
Message from EJB: Hello Ivan, it is now Thu Jan 13 07:06:45 CET 2011
Jan 13, 2011 7:06:45 AM org.glassfish.admin.mbeanserver.JMXStartupService shutdown
INFO: JMXStartupService and JMXConnectors have been shut down.
Jan 13, 2011 7:06:45 AM com.sun.enterprise.v3.server.AppServerStartup stop
INFO: Shutdown procedure finished
Jan 13, 2011 7:06:45 AM AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] exiting
```

Note that:

- The portable JNDI names for the EJB MyStatelessSessionBean are the same as when the example program was run in Eclipse, that is:
`java:global/JavaSEEJBApp/MyStatelessSessionBean!`
`com.ivan.scbcd6.ejbs.StatelessSessionBean`
and
`java:global/JavaSEEJBApp/MyStatelessSessionBean`
- A message from the EJB is printed.
- The embeddable EJB container, and also the process the example program runs in, is properly shut down and the JVM terminates.

This concludes the Embeddable EJB Container Example.