

# **Enterprise Java Development with EJB**

---

## **EJB3 Session, Entity and Message-Driven Beans**

**COLLABORATORS**

	<i>TITLE :</i> Enterprise Java Development with EJB	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY	Dr. Fritz Solme	15 April 2010

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>I</b>	<b>Introduction and overview</b>	<b>1</b>
<b>1</b>	<b>Architecture Overview</b>	<b>2</b>
1.1	The Java Platform, Enterprise Edition (Java EE) architecture . . . . .	2
1.1.1	What is Java EE? . . . . .	2
1.1.2	Java EE frameworks . . . . .	2
1.1.2.1	What is a Java EE framework? . . . . .	2
1.1.2.2	Example frameworks . . . . .	3
1.1.3	Java EE as a MVC architecture for internet deployment . . . . .	3
1.1.4	overview of the Java EE architecture . . . . .	3
1.1.4.1	Application-server generated layers . . . . .	4
1.1.4.1.1	The enterprise services layer . . . . .	4
1.1.4.1.2	Database mapping layer . . . . .	5
1.1.5	The Java EE APIs . . . . .	5
1.1.6	Roles defined for within JavaEE . . . . .	6
1.1.6.1	Introduction . . . . .	6
1.1.6.2	JavaEE Container/Application Server Provider . . . . .	6
1.1.6.3	Tool Providers . . . . .	6
1.1.6.4	Enterprise Bean Developer . . . . .	7
1.1.6.5	Client Application Developer . . . . .	7
1.1.6.6	Web Component Developer . . . . .	7
1.1.6.7	Application Assembler . . . . .	7
1.1.6.8	Application Deployer . . . . .	7
1.1.6.9	System Administrator . . . . .	7
1.1.7	What does the EJB container provide? . . . . .	8
1.1.7.1	Introduction . . . . .	8
1.1.7.2	Concurrency support . . . . .	8
1.1.7.3	Component pooling . . . . .	8
1.1.7.4	Network enabling . . . . .	8

1.1.7.5	Persistence . . . . .	8
1.1.7.6	Component location transparency . . . . .	9
1.1.7.7	Transaction support . . . . .	10
1.1.7.8	Security support . . . . .	10
1.1.7.9	Session management . . . . .	10
1.1.7.10	Interception . . . . .	10
1.1.7.11	Resource connection pooling . . . . .	10
1.1.7.12	Implicit monitoring . . . . .	11
1.1.8	Enterprise beans as business-logic components . . . . .	11
1.1.8.1	Introduction . . . . .	11
1.1.8.2	Types of enterprise beans . . . . .	11
1.1.8.2.1	Introduction . . . . .	11
1.1.8.2.1.1	Beans as lightweight components . . . . .	11
1.1.8.2.2	Session Beans . . . . .	11
1.1.8.2.2.1	Stateless Session Beans . . . . .	12
1.1.8.2.2.2	Stateful Session Beans . . . . .	12
1.1.8.2.3	Message-Driven Beans . . . . .	12
1.1.8.2.3.1	Synchronous versus asynchronous bean requests . . . . .	12
1.1.8.2.3.2	The onMessage() service . . . . .	12
1.1.8.2.3.3	Providing a response . . . . .	13
1.1.8.3	Elements of an enterprise bean . . . . .	13
1.1.8.3.1	Introduction . . . . .	13
1.1.8.3.2	The Remote Interface . . . . .	13
1.1.8.3.3	Local Interfaces . . . . .	13
1.1.8.3.4	The enterprise bean implementation class . . . . .	13
1.1.8.3.5	The EJBObject . . . . .	13
1.1.8.3.6	Deployment Descriptors . . . . .	14
1.1.8.4	The deployment package . . . . .	14
1.1.8.5	How enterprise bean elements collaborate? . . . . .	15
1.1.8.6	Enterprise bean restrictions . . . . .	15
1.1.8.6.1	Beans can't give clients direct access to the bean instance . . . . .	15
1.1.8.6.2	Enterprise beans may not accept network server connections . . . . .	16
1.1.8.6.3	Enterprise beans should be single-threaded . . . . .	16
1.1.8.6.4	Enterprise beans should not create a user interface . . . . .	16
1.1.8.6.5	Stateful session beans can't have persistent class fields . . . . .	16
1.1.8.6.6	Enterprise beans may not use any native libraries . . . . .	16
1.1.9	Entity objects . . . . .	16
1.1.10	Enterprise bean as flyweights . . . . .	17
1.1.11	Clustering . . . . .	17

---

1.1.11.1	What needs to be clustered . . . . .	17
1.1.11.2	Availability versus reliability . . . . .	17
1.1.11.3	Using Storage Area Networks (SAN) . . . . .	18
1.1.11.4	Replication algorithms . . . . .	18
1.1.12	Client containers . . . . .	18
1.1.13	Java EE and quality attributes . . . . .	18
1.1.13.1	Flexibility/modifiability . . . . .	18
1.1.13.2	Reliability and availability . . . . .	19
1.1.13.3	Scalability . . . . .	19
1.1.13.4	Manageability . . . . .	19
1.1.13.5	Security . . . . .	19
1.1.13.6	Performance . . . . .	19
1.1.13.7	Integrability . . . . .	19
1.1.13.8	Time-to-market . . . . .	19
1.1.14	Java EE best practices . . . . .	20
1.1.15	Java EE and SOA . . . . .	20
1.1.16	Exercises . . . . .	21
<b>2</b>	<b>Maven</b> . . . . .	<b>22</b>
2.1	Introduction . . . . .	22
2.1.1	Guiding principles of Maven . . . . .	22
2.1.2	What does Maven provide? . . . . .	23
2.1.3	Maven versus Ant . . . . .	23
2.1.4	What is Maven really? . . . . .	23
2.2	Maven's Project Object Model (POM) . . . . .	24
2.2.1	Introduction . . . . .	24
2.2.2	POM structure . . . . .	24
2.2.3	Project Identifiers . . . . .	26
2.2.3.1	Group ID . . . . .	26
2.2.3.2	ArtifactId . . . . .	26
2.2.3.3	Version . . . . .	26
2.2.3.4	Packaging . . . . .	26
2.2.4	POM inheritance . . . . .	27
2.2.4.1	Declaring the parent POM . . . . .	27
2.2.4.2	The Super POM . . . . .	27
2.2.4.3	The effective POM . . . . .	28
2.2.5	Project dependencies . . . . .	28
2.2.5.1	Specifying project dependencies . . . . .	28
2.2.5.1.1	Specifying version ranges . . . . .	29

2.2.5.1.2	Specifying the scope of a dependency	29
2.2.5.2	Transitive dependencies	30
2.2.5.3	Dependency management	30
2.2.5.4	Inclusions	31
2.2.6	Project modules	32
2.2.7	Build customization	32
2.2.7.1	Customizing/configuring plugin behaviour	33
2.2.7.2	Adding a goal to a life cycle phase	33
2.2.8	Distribution Information	35
2.2.8.1	Specifying distribution Info	35
2.2.8.2	Specifying login credentials	35
2.3	Maven's life cycles	36
2.3.1	Introduction	36
2.3.2	Maven's default build life cycle	37
2.3.3	Maven's clean life cycle	38
2.3.4	Maven's site life cycle	38
2.3.4.1	Default goal bindings for the site life cycle	38
2.3.5	Package-based goal bindings	39
2.3.6	Project based life cycle goals	39
2.4	Executing Maven	40
2.4.1	Executing a plugin goal	40
2.4.2	Executing a life cycle phase	40
2.5	Maven repositories	40
2.5.1	Introduction	40
2.5.2	Repository structure	41
2.5.3	Repository locations	41
2.5.3.1	Remote repositories	41
2.5.3.1.1	Accessing remote repositories	41
2.5.3.1.2	Specifying the location of remote repositories	41
2.5.3.2	The local repository	42
2.5.4	How Maven uses Respositories	42
2.5.5	Repository tools	42
2.6	Hello World via Maven	43
2.6.1	Introduction	43
2.6.2	Creating a project via the Archetype plugin	43
2.6.3	Executing default life cycle phases	44
2.6.3.1	Deploying onto a server	44
2.6.3.1.1	Specifying distribution info	45
2.6.3.1.2	SSH key for login without password	45

---

2.6.3.1.3	Specifying authentication settings	46
2.6.4	Executing specific plugin goals	46
2.6.4.1	Executing a program from Maven	47
2.6.5	Generating documentation	47
2.6.6	Cleaning the project	48
2.7	Maven JAXB Sample	48
2.7.1	Introduction	48
2.7.2	The schema	48
2.7.3	The Test Application	49
2.7.4	Java-5 POM	51
2.7.5	Java-6 POM	52
2.7.6	Executing goals	54
2.8	Maven JAXWS Sample	54
2.8.1	Introduction	54
2.8.2	Java-6 POM	55
2.8.3	The Test Application	56
2.8.4	Executing the web service test	57
<b>3</b>	<b>A simple JavaEE-6 example: WeatherBuro</b>	<b>59</b>
3.1	Overview	59
3.1.1	Core elements of the example	59
3.2	Environment setup	59
3.2.1	Database server	60
3.2.1.1	Installation	60
3.2.1.1.1	Installing PostgreSql on Gentoo Linux	60
3.2.1.2	Creating the datab, users and permissions	60
3.2.1.2.1	Creating the system user	61
3.2.2	Application server	61
3.2.2.1	Installing the Glassfish JavaEE application server	61
3.2.2.2	Creating a domain for your system domain	61
3.2.2.3	Setting up the database connection pool and JDBC resource	61
3.2.2.4	Setting up a JDBC security realm for container based authentication	62
3.3	The build and testing environment	63
3.3.1	The parent project	63
3.3.2	The business logic module	64
3.3.3	The web module	66
3.4	The domain objects layer	67
3.4.1	The domain objects model	67
3.4.2	Implementation mapping of the domain objects layer	68

3.4.2.1	Geographic coordinates . . . . .	68
3.4.2.2	Locations . . . . .	70
3.4.2.3	Weather readings . . . . .	72
3.4.3	The persistence descriptor . . . . .	75
3.5	The services layer . . . . .	76
3.5.1	Location services . . . . .	76
3.5.2	LocationServicesBean . . . . .	77
3.5.3	WeatherServices . . . . .	79
3.5.4	WeatherServicesBean . . . . .	81
3.6	Unit testing . . . . .	83
3.6.1	Initializing the embedded container and registering the enterprise beans . . . . .	83
3.6.2	LocationServicesTest . . . . .	84
3.6.3	WeatherServicesTest . . . . .	87
3.7	JSF2/Facelets based web presentation layer . . . . .	89
3.7.1	Core elements of JSF2/Facelets based web presentation layers . . . . .	89
3.7.2	Configuration . . . . .	90
3.7.2.1	faces-config . . . . .	90
3.7.2.2	faces-config . . . . .	91
3.7.2.3	faces-config . . . . .	92
3.7.2.4	Directory structure . . . . .	92
3.7.3	The index.html file . . . . .	92
3.7.4	The facelet template . . . . .	92
3.7.5	The main menu . . . . .	94
3.7.6	Facelets for adding a new location . . . . .	95
3.7.6.1	The AddLocation facelet page . . . . .	95
3.7.6.2	The AddLocationPanel . . . . .	95
3.7.6.2.1	Navigation . . . . .	96
3.7.6.3	AddLocation binding object . . . . .	96
3.7.6.3.1	Navigation . . . . .	97
3.7.6.4	The LocationPanel . . . . .	97
3.7.6.5	Location binding object . . . . .	98
3.7.6.6	The GeographicCoordinatesPanel . . . . .	99
3.7.6.7	GeographicCoordinatesBinding object . . . . .	99
3.7.7	Facelets for viewing and the locations location . . . . .	100
3.7.7.1	The Locations facelet page . . . . .	100
3.7.7.2	The LocationsPanel . . . . .	101
3.7.7.3	LocationsBinding object . . . . .	101

<b>II Persistence via the Java Persistence API (JPA)</b>	<b>103</b>
<b>4 JPA overview</b>	<b>104</b>
4.1 What is JPA? . . . . .	104
4.2 What does JPA provide? . . . . .	104
4.3 JPA providers . . . . .	104
<b>5 Persistence contexts</b>	<b>105</b>
5.1 Overview persistence contexts . . . . .	105
5.1.1 What is a persistence context? . . . . .	105
5.1.2 What is an entity manager? . . . . .	105
5.1.3 Optimistic concurrency control . . . . .	105
5.1.4 The life span of an entity manager . . . . .	106
5.1.4.1 Transaction-scoped persistence contexts . . . . .	106
5.1.4.2 Extended persistence contexts . . . . .	106
5.1.5 The transaction management for a persistence context . . . . .	106
5.1.6 The scope of a persistence context . . . . .	106
5.1.7 How are entity managers obtained? . . . . .	107
5.1.7.1 Obtaining an entity manager in a container managed environment . . . . .	107
5.1.7.2 Manual creation of an entity manager in a JavaSE application . . . . .	107
5.1.8 Detaching objects from a persistence context . . . . .	108
5.2 Persistence context configuration . . . . .	108
5.2.1 Typical persistence context for non-managed Java applications . . . . .	108
5.2.2 Typical persistence context for managed applications . . . . .	108
<b>6 Entities</b>	<b>110</b>
6.1 Overview . . . . .	110
6.2 Simple entities . . . . .	110
6.2.1 Declaring an entity . . . . .	110
6.2.2 Requirements for entities . . . . .	110
6.2.3 What is persisted? . . . . .	111
6.2.3.1 Valid persistent field types . . . . .	112
6.2.3.2 Collection variables . . . . .	113
6.2.3.3 Transient fields . . . . .	113
6.2.3.4 Field validation . . . . .	113
6.2.4 Embedded classes . . . . .	113
6.2.4.1 Defining an embeddable class . . . . .	114
6.2.4.1.1 Specifying the access type . . . . .	114
6.2.4.2 Embedding and embeddable class within an entity . . . . .	114
6.2.5 Primary keys . . . . .	114

6.2.5.1	Simple primary keys . . . . .	114
6.2.5.1.1	Valid types for simple primary keys . . . . .	115
6.2.5.1.2	Specifying the primary key field . . . . .	115
6.2.5.1.3	Automatically generating primary keys . . . . .	115
6.2.5.2	Primary key classes (composite keys) . . . . .	116
6.2.5.2.1	Interface for the primary key class . . . . .	116
6.2.5.2.2	Implementation of a primary key class . . . . .	116
6.2.5.2.3	Entity bean using the primary key class . . . . .	117
6.2.6	Specifying column mappings . . . . .	117
6.2.7	Column constraints . . . . .	118
6.2.8	Primitive collections and maps . . . . .	118
6.3	Relationships . . . . .	118
6.3.1	Summary of UML relationships . . . . .	119
6.3.1.1	Dependency . . . . .	119
6.3.1.2	Association . . . . .	119
6.3.1.3	Aggregation . . . . .	120
6.3.1.4	Composition . . . . .	120
6.3.1.5	Realisation . . . . .	120
6.3.1.6	Specialisation . . . . .	120
6.3.1.7	Containment . . . . .	120
6.3.1.8	Shopping for relationships . . . . .	121
6.3.2	Composition between entity beans . . . . .	121
6.3.3	Relationship types . . . . .	121
6.3.3.1	Relationship owner . . . . .	122
6.3.3.2	Unidirectional single-valued relationships . . . . .	122
6.3.3.3	Bidirectional one-to-one relationships . . . . .	122
6.3.3.4	Bidirectional many-to-one relationships . . . . .	123
6.3.3.5	Cascading Operations . . . . .	124
6.3.4	Fetching strategies . . . . .	125
6.3.4.1	Eager fetching . . . . .	125
6.3.4.2	Defaults fetching strategies . . . . .	125
6.3.5	Specialization relationships . . . . .	125
6.3.5.1	Mapping onto relational databases . . . . .	126
6.3.5.2	Joined subclass . . . . .	126
6.3.5.3	Single table per class hierarchy . . . . .	126
6.3.5.4	Single table per class . . . . .	126

---

<b>7 The Java Persistence Query Language (JPQL)</b>	<b>128</b>
7.1 Introduction . . . . .	128
7.2 JPQL versus SQL . . . . .	128
7.2.1 Result Collections in JPQL . . . . .	128
7.2.2 Selecting entity Attributes . . . . .	129
7.3 JPQL statement types . . . . .	129
7.3.1 Elements of query statements . . . . .	129
7.3.2 Elements of Update and delete statements . . . . .	129
7.4 Polymorphism . . . . .	130
7.5 Navigating object graphs . . . . .	130
7.5.1 Simple paths . . . . .	130
7.5.2 Single-Valued versus Multi-Valued Paths . . . . .	131
7.6 Specifying the source of a query . . . . .	132
7.6.1 Selecng from multiple domains . . . . .	132
7.6.2 Joins . . . . .	132
7.6.2.1 Inner joins . . . . .	132
7.6.2.1.1 Implicit inner joins . . . . .	132
7.6.2.1.2 Explicit inner joins . . . . .	132
7.6.2.2 Left outer joins . . . . .	133
7.7 Collapsing Multi-Valued Paths into Single-Valued Paths via Collection Variables . . . . .	133
7.8 Constraining a result set via a WHERE clause . . . . .	133
7.8.1 Comparison operators which can be used in WHERE clauses . . . . .	133
7.8.2 Calculation and string operators . . . . .	134
7.8.3 Using collection variables in WHERE clauses . . . . .	134
7.9 Constructing result objects . . . . .	134
7.10 Nested queries . . . . .	134
7.11 Ordering . . . . .	135
7.12 Grouping . . . . .	135
7.13 The Java Persistence Query Language (JPQL) . . . . .	135
7.13.1 Positional parameters . . . . .	135
7.13.2 Named parameters . . . . .	135
<b>8 Constructing and executing queries</b>	<b>136</b>
8.1 Named queries . . . . .	136

<b>III Enterprise beans</b>	<b>137</b>
<b>9 Session Beans</b>	<b>138</b>
9.1 Introduction . . . . .	138
9.2 Business Interfaces . . . . .	138
9.2.1 What is a component? . . . . .	138
9.2.2 EJBs as business logic components . . . . .	138
9.2.3 Remote interfaces . . . . .	138
9.2.3.1 Defining a remote interface . . . . .	139
9.2.3.2 A bean realising the business contract . . . . .	139
9.2.3.3 Access path when using a remote interface . . . . .	140
9.2.4 Local interfaces . . . . .	140
9.2.4.1 Defining a local interface . . . . .	140
9.2.4.2 Access path when using a local interface . . . . .	141
9.2.5 Switching between local and remote interfaces . . . . .	141
9.2.5.1 Parameter handling in plain Java objects . . . . .	141
9.2.5.2 Parameter handling in remote Java objects (via RMI) . . . . .	141
9.2.5.3 Providing access locally and remotely . . . . .	142
9.2.6 Automatically generated business interfaces . . . . .	143
9.2.6.1 Automatic interface naming . . . . .	143
9.2.6.2 Which services are published in the automatically generated interface? . . . . .	143
9.2.6.3 Automatically generated local interfaces . . . . .	143
9.2.6.4 Automatically generated remote interfaces . . . . .	143
9.2.6.5 Automatically generated interfaces: A good idea? . . . . .	143
9.3 A generic Ant build script for EJB projects . . . . .	143
9.3.1 Source packaging . . . . .	144
9.3.2 Common targets . . . . .	144
9.3.3 The common ant build targets . . . . .	145
9.3.4 Location of projects, common ant script and common libraries . . . . .	150
9.4 Stateless session beans . . . . .	150
9.4.1 Stateless session beans as a services facade . . . . .	150
9.4.2 Life cycle of a stateless session bean . . . . .	150
9.4.3 Writing the business interfaces . . . . .	151
9.4.3.1 Average.java . . . . .	151
9.4.3.2 AverageRemote.java . . . . .	151
9.4.3.3 AverageLocal.java . . . . .	151
9.4.3.4 AverageBean.java . . . . .	152
9.4.3.5 Building and deploying the bean . . . . .	152
9.5 A simple session bean client . . . . .	152

9.5.1	Client.java . . . . .	152
9.5.2	Packaging the JNDI properties file with the client . . . . .	154
9.5.3	Running the client application . . . . .	154
9.5.4	Providing a fully self-contained client . . . . .	154
9.6	A simple web client . . . . .	154
9.6.1	Should servlets use local or remote interfaces? . . . . .	154
9.6.2	Client input form: average.jsp . . . . .	154
9.6.3	Client controller: AverageServlet.java . . . . .	155
9.6.4	Client result view: averageResult.jsp . . . . .	156
9.6.5	Web deployment descriptor . . . . .	157
9.6.6	Using a web client . . . . .	157
9.7	Enterprise application archives . . . . .	157
9.7.1	The application deployment descriptor . . . . .	158
9.8	Stateful session beans . . . . .	158
9.8.1	Stateful session beans as workflow managers . . . . .	158
9.8.2	Life cycle of a stateful session bean . . . . .	158
9.8.2.1	Stateful session beans are realised using flyweight and memento patterns . . . . .	159
9.8.3	Writing the business interfaces . . . . .	159
9.8.3.1	ShoppingCart.java . . . . .	159
9.8.3.2	Value objects: CartContent.java and Product.java . . . . .	160
9.8.3.3	ShoppingCartBean.java . . . . .	162
9.8.3.4	Building and deploying the bean . . . . .	163
9.9	Reacting to life cycle events . . . . .	163
9.9.1	The life cycle of standard Java objects . . . . .	163
9.9.2	The life cycle of session beans . . . . .	164
9.9.3	Callback listener methods . . . . .	164
9.10	Asynchronous session bean services . . . . .	164
9.10.1	Why should one not create threads within code executed within an application server? . . . . .	164
9.10.2	Asynchronous processing of requests . . . . .	165
9.10.3	Deferred synchronous processing of requests . . . . .	165
9.10.4	Asynchronous beans . . . . .	166
9.11	A web adapter keeping the control of business processes in the business logic layer . . . . .	166
9.11.1	Introduction . . . . .	166
9.11.2	A Call-Return mapping adapter . . . . .	167
9.11.2.1	Implementations of Call-Return mapping adapter . . . . .	167
9.11.2.1.1	JavaEE implementation of a Call-Return mapping adapter . . . . .	167
9.11.2.1.1.1	The service layer . . . . .	169
9.11.2.1.1.2	Stateful Session Bean adapter using asynchronous session bean call . . . . .	171
9.12	Sending an email . . . . .	173

9.12.1 Registering an email server . . . . .	173
9.12.2 Sending an email from an enterprise bean . . . . .	174
9.13 Web services access . . . . .	174
9.13.1 The generated web services contract . . . . .	177
9.13.2 Customising the web services mapping . . . . .	179
9.14 Exercises . . . . .	179
<b>10 EJB Interception</b>	<b>180</b>
10.1 Introduction . . . . .	180
10.2 Dependency injection . . . . .	180
10.2.1 Dependency injection as a form of inversion of control . . . . .	180
10.2.2 Which dependencies are injected ? . . . . .	180
10.2.3 Setter injection . . . . .	181
10.2.3.1 Method access level modifier . . . . .	182
10.2.4 Field injection . . . . .	182
10.2.4.1 Injecting context objects . . . . .	182
10.2.4.2 Field access level modifier . . . . .	183
10.2.5 How are injection resources resolved? . . . . .	183
10.2.6 Linking logical EJB names to physical JNDI names . . . . .	183
10.3 Interceptors . . . . .	184
10.3.1 When should one use interceptors . . . . .	185
10.3.2 Defining interceptors . . . . .	185
10.3.2.1 The invocation context . . . . .	185
10.3.2.2 Interceptor methods . . . . .	185
10.3.2.2.1 Aborting an interception . . . . .	186
10.3.2.3 Interceptor classes . . . . .	186
10.3.2.3.1 Interceptor life span and state . . . . .	186
10.3.2.3.1.1 Interceptor state . . . . .	186
10.3.2.3.1.2 Activation and de-activation . . . . .	186
10.3.2.3.1.3 Life cycle callback services . . . . .	186
10.3.2.3.2 Requirements for interceptor classes . . . . .	186
10.3.2.3.3 Interceptor exceptions . . . . .	187
10.3.2.3.4 Declaring an interceptor class in a deployment descriptor . . . . .	187
10.3.2.3.5 Interceptor specialization . . . . .	187
10.3.3 Applying Interceptors . . . . .	187
10.3.3.1 Application domain for interceptors . . . . .	187
10.3.3.2 Method level interceptors . . . . .	187
10.3.3.2.1 Applying method level interceptors via annotations . . . . .	188
10.3.3.2.2 Applying method level interceptors in the deployment descriptor of a bean . . . . .	188

---

10.3.3.3 Class level interceptors . . . . .	188
10.3.3.3.1 Applying class level interceptors via annotations . . . . .	189
10.3.3.3.2 Applying class level interceptors in the deployment descriptor of a bean . . . . .	189
10.3.3.4 Default interceptors . . . . .	190
10.3.3.5 Interception order . . . . .	190
10.3.3.6 Specifying exclusions for class and default interceptors . . . . .	191
10.3.3.6.1 Specifying exclusions via annotations . . . . .	191
10.3.3.6.2 Specifying exclusions in a deployment descriptor . . . . .	191
10.3.3.7 Injecting dependencies into interceptors . . . . .	191
10.3.4 A simple example: AstrologyFeeder . . . . .	192
10.3.4.1 The context contract . . . . .	192
10.3.4.2 The interceptor . . . . .	193
10.3.4.3 The bean implementation class . . . . .	194
10.3.4.4 A simple RMI client . . . . .	195
10.3.5 Exercises . . . . .	196
<b>11 Messaging</b> . . . . .	<b>197</b>
11.1 Introduction . . . . .	197
11.1.1 Support for JMS as a Managed Resource . . . . .	197
11.1.2 Message-Driven Beans . . . . .	197
11.1.3 How does a Message-Driven Bean Work . . . . .	197
11.1.4 Robustness features . . . . .	197
11.1.4.1 Guaranteed Message Delivery . . . . .	197
11.1.4.2 Certified Message Delivery . . . . .	198
11.2 Why Message Driven Beans? . . . . .	198
11.2.1 Typical Applications of Message-Driven Beans . . . . .	198
11.2.2 When Should You Consider Using MDBs? . . . . .	198
11.2.3 When Should You Consider Avoiding MDBs? . . . . .	199
11.3 The Java Messaging Service (JMS) . . . . .	199
11.3.1 Styles of Messages: Point-Point vs Publish-Subscribe Domains . . . . .	199
11.3.1.1 The Point-to-Point Messaging Domain . . . . .	200
11.3.1.1.1 Characteristics of point-to-point messaging . . . . .	200
11.3.1.2 The Publish-Subscribe Messaging Domain . . . . .	200
11.3.1.2.1 Characteristics of publish-subscribe messaging . . . . .	200
11.3.2 Message Types . . . . .	200
11.3.3 Using JMS queues and topics . . . . .	201
11.3.3.1 General Algorithm for Connecting to a Queue or Topic . . . . .	201
11.3.3.2 Using JMS queues . . . . .	201
11.3.3.2.1 Connecting to A Message Queue . . . . .	202

---

---

11.3.3.2.2	Developing Queue Senders . . . . .	202
11.3.3.2.3	Creating Queue Receivers . . . . .	202
11.3.3.3	An Example Application using Point-To-Point Messaging . . . . .	202
11.3.3.3.1	MessageSender.java . . . . .	203
11.3.3.3.2	MessageRecipient.java . . . . .	204
11.3.3.4	Using JMS Topics . . . . .	205
11.3.3.4.1	Connecting to a topic . . . . .	205
11.3.3.4.2	Developing Publishers for a Topic . . . . .	206
11.3.3.4.3	Developing Subscribers for a Topic . . . . .	206
11.3.3.5	An Example Application using Publish-Subscribe Messaging . . . . .	206
11.3.3.5.1	GoldPricePublisher.java . . . . .	207
11.3.3.5.2	GoldPriceSubscriber.java . . . . .	208
11.3.4	The Structure of a JMS Message . . . . .	209
11.3.4.1	JMS Message Headers . . . . .	210
11.3.5	Durable Subscribers . . . . .	210
11.3.6	Messages Participating in Transactions . . . . .	210
11.3.7	Selected message retrieval . . . . .	211
11.4	Message-driven beans . . . . .	211
11.4.1	Features of Message-Driven Beans . . . . .	211
11.4.2	Developing Message-Driven Beans . . . . .	211
11.4.2.1	Developing message senders . . . . .	212
11.4.2.2	Deploying Message-driven beans . . . . .	212
11.5	Exercises . . . . .	212
<b>12</b>	<b>Singleton beans</b> . . . . .	<b>214</b>
12.1	Introduction . . . . .	214
12.2	Declaring singleton beans . . . . .	214
12.3	Using singleton beans . . . . .	214
12.4	Starup and shutdown callbacks . . . . .	215
<b>13</b>	<b>Scheduled tasks with the Timer Service</b> . . . . .	<b>216</b>
13.1	Introduction . . . . .	216
13.2	What can the EJB Timer Service be used for? . . . . .	216
13.3	Architecture of the timer service . . . . .	217
13.3.1	Timer services are persistent . . . . .	217
13.3.2	Automatic versus programmatic timer registration . . . . .	217
13.4	Specifying time instants . . . . .	217
13.4.1	Parameters and their possible and default values . . . . .	217
13.4.2	Specifying time expressions . . . . .	217
13.5	Automatic timer creation . . . . .	218
13.6	Programmatic timer creation . . . . .	218

---

<b>14 Transactions</b>	<b>220</b>
14.1 Conceptual overview of transactions . . . . .	220
14.1.1 Introduction . . . . .	220
14.1.1.1 Why Transactions? . . . . .	220
14.1.1.1.1 Atomic Operations . . . . .	221
14.1.1.1.2 Safe Concurrent Access to Shared Information . . . . .	221
14.1.1.1.3 Graceful Failure Recovery . . . . .	221
14.1.2 ACID guarantees by transaction processors . . . . .	221
14.1.2.1 Atomicity . . . . .	221
14.1.2.2 Consistency . . . . .	221
14.1.2.3 Isolation . . . . .	222
14.1.2.4 Durability . . . . .	222
14.1.3 Rolling Transaction Back . . . . .	222
14.1.4 Compensating Transactions . . . . .	222
14.1.5 Transactions across distributed resources . . . . .	222
14.1.5.1 Two-Phase Commit . . . . .	222
14.2 Overview . . . . .	222
14.2.1 JTS and JTA in EJB . . . . .	223
14.2.2 Transaction managers . . . . .	223
14.2.3 JTA transactions . . . . .	223
14.2.3.1 The status of a transaction . . . . .	224
14.2.3.2 Marking a transaction as failed . . . . .	224
14.2.4 Support for multiple resources . . . . .	224
14.3 Declarative Transaction Demarcation . . . . .	224
14.3.1 Transaction Attributes . . . . .	225
14.3.1.1 BeanManaged . . . . .	225
14.3.1.2 NotSupported . . . . .	225
14.3.1.3 Required . . . . .	225
14.3.1.4 Supports . . . . .	225
14.3.1.5 Manadatory . . . . .	226
14.3.1.6 RequiresNew . . . . .	226
14.3.1.7 Never . . . . .	226
14.3.2 Selecting a Transaction Attribute . . . . .	226
14.3.3 Annotating transaction attributes . . . . .	226
14.3.4 Transaction boundaries on method boundaries . . . . .	226
14.4 Transaction Isolation Levels . . . . .	227
14.4.1 Serializable . . . . .	227
14.4.2 Repeatable Reads . . . . .	227
14.4.3 Read Committed . . . . .	227
14.4.4 ReadUncommitted . . . . .	227
14.5 Setting EJB Isolation Levels . . . . .	227

<b>15 Java EE Security</b>	<b>228</b>
15.1 Java EE Security Overview . . . . .	228
15.1.1 Authentication and authorization . . . . .	228
15.1.2 Mechanisms to specify security services . . . . .	229
15.1.3 Containers Shared Security . . . . .	229
15.2 Declarative security . . . . .	229
15.2.1 Authorization annotations . . . . .	229
15.2.2 Specifying authorization requirements . . . . .	229
15.2.2.1 Specifying the authorization requirements for a service . . . . .	229
15.2.2.2 Default authorization requirements . . . . .	230
15.2.2.2.1 Overriding default authorization requirements . . . . .	230
15.2.3 Run-as . . . . .	231
15.2.4 Declaring additional roles . . . . .	232
15.3 Programmatic security . . . . .	232
<b>16 The JEE Connector Architecture</b>	<b>233</b>
16.1 Introduction . . . . .	233
16.1.1 What is the Java Connector Architecture? . . . . .	233
16.1.2 Example Enterprise Information (EIS) systems . . . . .	233
16.1.3 Simplifying resource managers via the Java Connector Architecture . . . . .	234
16.1.4 Who uses connector resources? . . . . .	234
16.1.5 Who provides JCA adapters? . . . . .	234
16.1.6 Features of Java Connector Architecture . . . . .	234
16.1.7 Why use resource adapters? . . . . .	234
16.2 Managed environments . . . . .	235
16.2.1 What is a managed environment? . . . . .	235
16.2.1.1 Examples of managed environments . . . . .	235
16.3 Design of the Java EE Connector Architecture . . . . .	235
16.3.1 The high level contracts defined for the Java Connector Architecture . . . . .	236
16.3.1.1 The Common Client Interface (CCI) . . . . .	236
16.3.1.2 The Service Provider Interface (SPI) . . . . .	236
16.3.2 Responsibility allocations across core Java Connector components . . . . .	237
16.3.3 Resource Adapter Life Cycle Management . . . . .	237
16.3.3.1 The typical life cycle of a resource adapter . . . . .	237
16.3.3.2 The ResourceAdapter class . . . . .	238
16.3.3.2.1 The bootstrap context . . . . .	239
16.4 Example outbound connector: ElectionConnector . . . . .	239
16.4.1 The election server . . . . .	239
16.4.2 The election connector . . . . .	242

16.4.2.1	The Service Provider Interface (SPI) . . . . .	242
16.4.2.1.1	The resource adapter . . . . .	242
16.4.2.1.2	The managed connection . . . . .	245
16.4.2.1.3	The managed connection factory . . . . .	250
16.4.2.1.4	Connection request info . . . . .	254
16.4.2.1.5	Managed connection meta data . . . . .	255
16.4.2.2	The elements of the Common Client Interface (CCI) . . . . .	256
16.4.2.2.1	Connection . . . . .	256
16.4.2.2.2	The connection implementation . . . . .	257
16.4.2.2.3	The connection factory . . . . .	259
16.4.2.2.4	The connection factory implementation . . . . .	260
16.4.2.3	The deployment descriptor for the resource adapter . . . . .	261
16.4.2.4	The deployment descriptor for the data source defined for the resource adapter . . . . .	263
16.4.2.5	The Ant build script for the resource adapter . . . . .	263
16.4.2.6	Deploying the resource adapter . . . . .	265
16.4.3	A client application for the resource adapter . . . . .	266
16.4.3.1	The session bean . . . . .	266
16.4.3.2	A JSP-based user interface . . . . .	267
16.4.3.3	A servlet controller . . . . .	270
16.4.3.4	The web deployment descriptor . . . . .	270
16.4.3.5	The application deployment descriptor . . . . .	271
16.4.3.6	The Ant build script . . . . .	271
16.4.3.7	Deploying the application . . . . .	275
16.4.3.8	Using the web client . . . . .	276

# List of Figures

1.1	The Java EE architecture . . . . .	4
1.2	The JNDI Architecture . . . . .	9
1.3	Bean Service Request Processing . . . . .	14
1.4	The EJB-Jar package . . . . .	15
1.5	How do the bean elements collaborate . . . . .	15
2.1	The structure of Maven’s POM . . . . .	25
3.1	A UML class diagram for the weather buro domain model . . . . .	68
6.1	Summary of UML relationships . . . . .	119
6.2	A unidirectional one-to-one relationship . . . . .	122
6.3	A bidirectional one-to-one relationship . . . . .	123
6.4	A bidirectional many-to-one relationship . . . . .	124
7.1	UML class diagram for a bond . . . . .	130
7.2	UML class diagram for presentations of courses for a course schedule . . . . .	131
9.1	Directory structure for source artifacts of our EJB projects . . . . .	144
9.2	Life cycle of a stateless session bean . . . . .	150
9.3	Life cycle of a stateful session bean . . . . .	159
9.4	Workflow as per technology neutral design (URDAD) . . . . .	168
9.5	Workflow using a call-return adapter . . . . .	169
15.1	Java EE Container Security . . . . .	229
16.1	Contracts around the resource adapter . . . . .	236
16.2	Responsibility allocations across JCA components . . . . .	237
16.3	Life cycle of a resource adapter . . . . .	238

# List of Tables

13.1 Time instant parameter values . . . . .	217
15.1 Supported EJB authorization annotations . . . . .	230

# Preface: About Solms TCD

## Copyright

We at Solms Training, Consulting and Development (STCD) believe in the open and free sharing of knowledge for public benefit. To this end, we make all our knowledge and educational material freely available. The material may be used subject to the *Solms Public License* (SPL), a free license similar to the Creative Commons license.

### Solms Public License (SPL)

The *Solms Public License* (SPL) is modeled closely on the GNU, BSD, and attribution assurance public licenses commonly used for open-source software. It extends the principles of open and free software to knowledge components and documentation including educational material.

#### Terms

- ‘Material’ below, refers to any such knowledge components, documentation, including educational and training materials, or other work.
- ‘Work based on the Material’ means either the Material or any derivative work under copyright law: that is to say, a work containing the Material or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.)
- ‘Licensee’ means the person or organization which makes use of the material.

#### Application domain

This License applies to any Material (as defined above) which contains a notice placed by the copyright holder saying it may be distributed under the terms of this public license.

#### Conditions

- The licensee may freely copy, print and distribute this material or any part thereof provided that the licensee
  1. prominently displays (e.g. on a title page, below the header, or on the header or footer of a page or slide) the author’s attribution information, which includes the author’s name, affiliation and URL or e-mail address, and
  2. conspicuously and appropriately publishes on each copy the *Solms Public License*.
- Any material which makes use of material subject to the *Solms Public License* (SPL) must itself be published under the SPL.
- The knowledge is provided free of charge. There is no warranty for the knowledge unless otherwise stated in writing: the copyright holders provide the knowledge ‘as is’ without warranty of any kind.

- In no event, unless required by applicable law or agreed to in writing, will the copyright holder or any party which modifies or redistributes the material, as permitted above, be liable for damages including any general, special, incidental or consequential damages arising from using the information.
- Neither the name nor any trademark of the Author may be used to endorse or promote products derived from this material without specific prior written permission.

## Overview

The training pillar of Solms TCD focuses on vendor-neutral training which provides both, short- and long-term value to the attendees. We provide training for architects, designers, developers, business analysts and project managers.

### **Vendor-neutral, concepts-based training with short- and long-term value**

None of our courses are specific to vendor solutions, and a lot of emphasis is placed on entrenching a deeper understanding of the underlying concepts. In this context, we only promote and encourage public (open) standards.

Nevertheless, candidates obtain a lot of hands-on and practical experience in these open-standards based technologies. This enables them to immediately become productive and proficient in these technologies, and entrenches the conceptual understanding of these technologies.

When a ‘product’ is required for practicals, we tend towards selecting open-source solutions above vendor products. Typically, open source solutions adhere more closely to public standards, and the workflows are less often hidden behind convenient wizards, exposing the steps more clearly. This typically helps to gain a deeper understanding of the appropriate technologies.

## Training methods

We provide instructor based training at our own training centre as well as on-site training anywhere in the world. In addition to this we provide further guidance in the form of mentoring and consulting services.

### **Instructor-Led Training**

Our instructors have extensive theoretical knowledge and practical experience in various sciences and technologies.

After familiarising ourselves with the individual skill and needs of candidates, we adapt our training to be most effective in the candidate’s context - deviating from the set course notes if necessary.

Students spend half their time in lectures and half their time in hands-on, instructor-assisted practicals. Class sizes are typically limited to 12 to enable instructors to monitor the progress of each candidate effectively.

## About the Author(s)

### Fritz Solms



Fritz Solms is the MD and one of the founding members of the company.

Besides the management role, he is particularly focused on architecture, design, software development processes and requirements management.

Fritz Solms has a PhD and BSc degrees in Theoretical Physics from the University of Pretoria and a Masters degree in Physics from UNISA. After completing a short post-doc, he took up a senior lectureship in Applied Mathematics at the Rand Afrikaans University. There he founded, together with Prof W.-H. Steeb, the *International School for Scientific Computing* - developing a large number of courses focused on the immediate needs of industry. These include *C++, Java, Object-Oriented Analysis and Design, CORBA, Neural Networks, Fuzzy Logic and Information Theory and Maximum Entropy Inference*. The ISSC was the first institution in South Africa offering courses in *Java* and the *Unified Modeling Language*. During this period he was also responsible for presenting the OO Training for the Education Division of IBM South Africa.

In 1998 he joined the Quantitative Applications Division of the *Standard Corporate and Merchant Bank* (SCMB). Here he was the key person developing the architecture and infrastructure for the QAD library and applications. These were based on Java and CORBA technologies, with a robust object-oriented analysis and design backbone.

In 2000 he and Ellen Solms founded Solms TCD.

E-Mail:	fritz@solms.co.za
Tel:	011 646 6459

### Dawid Loubser



Dawid Loubser is a director of Solms TCD, and a specialist in Java, XML, and various graphics- and media related technologies across a range of application domains. He has a long history in the field of architecture and design, and a strong interest in Semantic Web technologies and user interface design.

After working at the JSE (Johannesburg Stock Exchange) as architect and lead developer for web-based systems, he joined Solms TCD to pursue a broad and varied journey of training, consulting and development. He presents courses on various SOA, Java

and XML-based topics, and is actively involved in research around the Java language, XML Schema, and is co-creator of the URDAD analysis and design methodology.

E-Mail:	dawidl@solms.co.za
Telephone:	+27 (11) 646-6459

## Solms TCD Guarantee

We hope that the course will comply with your expectations, and hopefully exceed it! Should you for some reason you feel that you are not satisfied with

- the course content,
- the teaching methods,
- the course presenter or
- any auxillary services supplied

Please feel free to discuss any complaints you may have with us. We will do our best to address your complaints. Should you feel that your complaints are not satisfactorily addressed within our organization, then you can raise your complaints with:

- Professor W.-H. Steeb from the *University of Johannesburg*, Tel: +27 (11) 486-4270, E-Mail: whs@rau.ac.za
- Dr A. Gerber from the Computer Science Department of the *University of South Africa*, E-Mail: gerberaj@unisa.ac.za

At the time of writing we are in the process of obtaining *ISETT SETA* accreditation. Complaints can be raised directly to that institution.

## **Part I**

# **Introduction and overview**

# Chapter 1

## Architecture Overview

### 1.1 The Java Platform, Enterprise Edition (Java EE) architecture

Java EE, the *Java Platform, Enterprise Edition* architecture is one of the most widely used reference architectures for large, interactive enterprise systems.

#### 1.1.1 What is Java EE?

The Java Platform, Enterprise Edition (Java EE) aims to provide an industry standard for developing portable, robust, scalable and secure server-side Java applications. Building on the Java Platform, Standard Edition (Java SE), Java EE provides a layered infrastructure supporting

- component models for both presentation layer and business logic components,
- resource management, including management of processing, memory, persistence and communication resources,
- application management,
- an integration infrastructure,
- a persistence infrastructure,
- infrastructure for security, and
- infrastructure for reliability and auditability.

#### 1.1.2 Java EE frameworks

The Java Enterprise Edition (JEE) provides a reference architecture within which multi-tier enterprise applications are developed.

##### 1.1.2.1 What is a Java EE framework?

The specification provides a range of standard APIs which must be implemented by Java EE *frameworks*. But what is a Java EE framework?

*A Java EE framework is a tool that can host systems built according to the Java EE reference architecture.*

### 1.1.2.2 Example frameworks

There are many frameworks available which implement this reference architecture. Some of these are vendor solutions like

- *IBM Websphere*
- *BEA WebLogic*
- *Oracle Application Server*

In addition to these vendor solutions there are also a number of open-source frameworks that realise the Java EE reference architecture, including

- JBoss,
- Glassfish,
- Apache Geronimo, and
- Jonas.

### 1.1.3 Java EE as a MVC architecture for internet deployment

The Java Enterprise Edition architecture defines a Model-View-Controller architecture for enterprise systems with

- the *view* being hosted in a *web container* with a rich web-based presentation layer infrastructure consisting of Servlets, Java Server Pages and Java Server Faces.
- the business logic *controller* being the session- and message-driven beans, and the presentation layer controllers being the servlets for web based presentation layers and the event listeners for application based clients
- the *model* which contains the business logic and data being hosted in standard Java classes with support for persistence via object-relational mappers through the Java Persistence API (JPA) and general persistence support via Java Data Objects (JDO). The application server still supports object pooling for entities (entity beans).

### 1.1.4 overview of the Java EE architecture

On the surface the Java EE architecture can be seen as a *5-layered architecture* with the layers being

- the presentation layer,
- a services layer,
- a domain objects layer,
- the infrastructure or middleware layer, and
- the back-end layer.

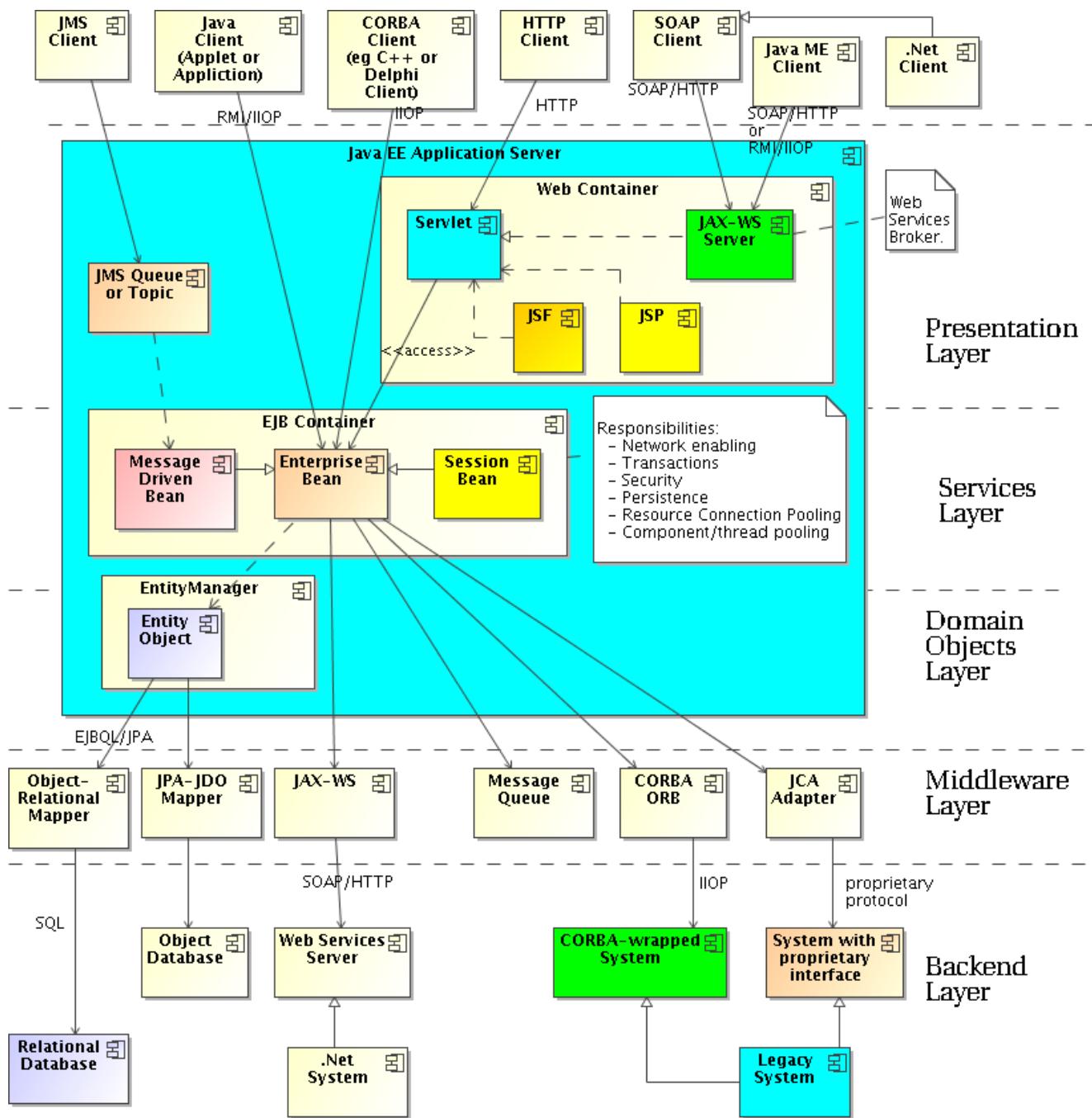


Figure 1.1: The Java EE architecture

#### 1.1.4.1 Application-server generated layers

In addition to the five layers used by client applications, the application server generates some internal layers which host certain responsibilities.

##### 1.1.4.1.1 The enterprise services layer

The application server generates an `EJBObject` for every enterprise bean, which acts as an interception layer at which the following enterprise services are applied:

- network enabling,
- mapping physical objects to virtual objects in order to support higher scalability,
- activation/deactivation of components at appropriate times,
- transaction- ,
- security- , and
- persistence support.

#### 1.1.4.1.2 Database mapping layer

The Java EE specification provides two standard APIs for database mapping layers, *Java data objects* (JDO) and the *Java Persistence API* (JPA). Both are extremely easy to use and are structurally very similar. JPA has been designed specifically with relational databases in mind while JDO is more general supporting also other persistence technologies like object databases.

In addition Java EE provides a technology neutral, object oriented query language, EJB-QL (The EJB Query Language) which is mapped ultimately on queries in the chosen persistence technology (e.g. appropriate SQL queries for your choice of relational database).

#### 1.1.5 The Java EE APIs

Java EE defines a range of standard APIs which aim to

- simplify the development of distributed enterprise applications, and
- introduce standards which enable architects and developers to avoid vendor locking in the architecture, design and development of the system -- components which implement the standard APIs can be exchanged.

The standard JEE APIs are:

- **EJB** Enterprise JavaBeans is the core Java EE specification for the business logic layer. It supports component-based business objects which are managed by the EJB container.
- **CORBA** The non-proprietary CORBA API has been absorbed within the Java EE standard APIs. It provides the integration/middleware support basis for EJBs, enabling object-oriented messaging with other (including non-Java) environments.
- **Java Persistence API** The Java Persistence API is a general-purpose framework to persist the state of objects to an underlying (typically relational) data store. It draws on the experience gained from frameworks such as *Hibernate* to bridge the gap between object-oriented design, and relational storage. Typically used to completely abstract the developer from SQL-based storage for business domain objects, it consists of:
  - The Java Persistence API
  - The query language
  - Object/relational mapping metadata
- **JDBC** The *Java Database Connectivity* API has been around for much longer than the Java EE specification. It provides a standard, low-level API for SQL-based data repository queries and statements. Historically the preferred mechanism for interaction with relational databases, the only reason why it should still be used directly (in the face of the Java Persistence API) is to cater for unsupported or non-standard operations, or as extreme performance tuning.
- **Web Presentation Layer** The Java EE Presentation Layer consists of the Servlets API (encompassing *Servlets*, *Java Server Pages (JSP)*, *Filters* and *Event Listeners*) and JSF (*Java Server Faces*), a Model-View-Controller and Data Binding framework. The many components of the presentation layer allows for the creation of maintainable and standards-compliant web interfaces.
- **JNDI** JNDI provides a standard API for name-based querying of object references. It provides the mechanism to decouple from specific implementations of naming and directory services like COSNaming, the RMI registry or LDAP.

- **JAAS/JCE** The *Java Authentication and Authorization* and the *Java Cryptography Extension* API's are standard APIs for security related services supporting the standard security requirements including authentication, authorization, confidentiality, data integrity and non-repudiation.
- **JTA** Transaction support is central for enterprise systems. The *Java Transaction API* defines a standard API for transaction demarcation and transaction control. It implements the Java mapping of the OMG's *Object Transaction Service* (OTS). The *Java Transaction Service* is an implementation of the JTA.
- **JMS** The *Java Message Service* provides a standard API for asynchronous messaging which decouples architects and developers from proprietary technologies like IBM and Microsoft MQ, Tuxedo and others.
- **JCA** The *Java EE Connector Architecture* specifies how application components can access connection-based resources in a standard way, by building adaptors that transform these external resources into *managed resources*. Typically used to interact with legacy systems.
- **JMX** The *Java Management eXtension* provides a standard API for managing deployed components and services including, starting, stopping and querying the status of a service provider.
- **XML** Most of the standard Java XML APIs play an important part in some or other aspect of the Java EE environment. All the functionality, from simple DOM and SAX parsing, to XSLT transformation, XML Schema-based data binding and SOAP web services are available throughout the environment. XML is viewed as the preferred protocol for exchanging and processing data.
  - **JAXP** The Java API for XML Processing provides a standard API for accessing parsing, validation and XSLT transformation engines.
  - **JAXB** The *Java API for XML Binding* which provides a standard interface for mapping between Java objects and XML elements. Typically used in the context of web services (as a sub-component of JAX-WS) or stand-alone, to transfer object state to or from XML.
  - **JAX-WS** The *Java API for XML Web Services* provide an intuitive mechanism to expose your services as SOAP web services, with a automatic data-binding, as well as WS-I compliance. JAX-WS replaces the previous JAX-RPC standard.
  - **JAXM** The *Java API for XML-based Messaging* provides a standard API for libraries which marshall and demarshall SOAP messages. It is usually used in conjunction with JAXB.
  - **JAXR** The *Java API for XML Registries* which is used to register or discover web-services providers in a web-services registry like UDDI or ebXML.

## 1.1.6 Roles defined for within JavaEE

### 1.1.6.1 Introduction

One of the aims of the Java EE reference architecture is the clear separation of many of the responsibilities across roles different roles with typically different skills requirements.

### 1.1.6.2 JavaEE Container/Application Server Provider

This is typically a vendor who develops Web and/or EJB containers/servers. The servers/containers implement standard the standard API's of the Java EE specification.

### 1.1.6.3 Tool Providers

Tool providers typically provide

- development tools,
- deployment tools,
- application assemblers,

- management tools for
    - reporting,
    - performance monitoring and tuning,
- and other management tasks.

#### **1.1.6.4 Enterprise Bean Developer**

Enterprise bean developers are programmers who develop individual business logic components. For each enterprise bean they develop

- The remote and home interfaces.
- The bean implementation.
- A core ejb deployment descriptor where the bean developer provides the specification of the bean elements and the resources required by the bean.
- Packages the bean elements in a ejb-jar file.

#### **1.1.6.5 Client Application Developer**

Client application developers collaborate closely with users in order to develop a usable client application which uses the business logic components. They may also supply deployment descriptors for clients.

#### **1.1.6.6 Web Component Developer**

Web component developers typically develop XHTML, JSPs and servlets for a web-based presentation layer to applications. Often a model-view-controller framework like Java Server Faces (JSF) or the Jakarta Struts framework is used by web-component developers.

#### **1.1.6.7 Application Assembler**

Application assemblers typically have extensive business knowledge as well as reasonable technical skills. They have to understand the enterprise bean remote and home interfaces and assemble applications from

- ejb-jar files containing enterprise bean elements, and
- ejb-war files containing web components like servlets and JSPs.

They create Java EE application (EAR) files which package the client side as well as the server side elements of an application in a single jar file.

#### **1.1.6.8 Application Deployer**

Application deployers configure and deploy Java EE applications setting security attributes, specifying component and connection pool sizes, mapping EJB bean attributes onto database elements, specifying transaction controls and so on.

#### **1.1.6.9 System Administrator**

System administrators manage Java EE-based systems by, among others,

- ensuring availability,
- supporting the network infrastructure, managing user and user group mappings onto EJB security roles, and
- monitoring and tuning application performance.

## 1.1.7 What does the EJB container provide?

### 1.1.7.1 Introduction

As the EJB container is responsible for hosting shared business objects within the application server, it is responsible for transparently applying a number of enterprise and middleware services to beans. Had the developer needed these services in a traditional CORBA/DCOM or Java RMI deployment, several explicit API calls would have to be made from within the business objects. In other words, the business logic would be polluted with a number of complex technicalities.

### 1.1.7.2 Concurrency support

The application server automatically supports concurrent service requests via multiple bean instantiation, i.e. concurrent service requests are processed by different bean instances. Bean developers need thus not worry about writing multi-threaded servers -- in fact they are forbidden to do so because that would interfere with the containers concurrency support.

Furthermore, in the case where the different client bean instances share common data resources the container takes over the responsibility of synchronizing the different data views.

### 1.1.7.3 Component pooling

Component pooling is possible because, as we shall see, clients do not obtain a direct reference to the enterprise bean instance.

As one deploys enterprise beans within a container, the container typically creates a pool of instances of the bean. The algorithm used is specific to the container -- this is one of the many areas where application server vendors compete.

Pooling is particularly simple for stateless session beans which provide a set of client services but which do not maintain information across service requests. In this case the session bean is returned to the pool upon completion of the service and if the same or another client requests a service from that same enterprise bean one of the bean instances in the pool is allocated to the client.

Containers also provide component pooling and life-cycle management for other enterprise beans. For message-driven beans it is basically as simple as for stateless session beans. For stateful session beans and entity beans the state has to be persisted before the same bean instance can be used for another client.

If, say during peak hours, the demand for a particular enterprise bean increases, the container can dynamically increase the pool size and at a later stage, when the load decreases, the pool size can be reduced again.

A relatively small number of beans can thus serve a large number of clients.

### 1.1.7.4 Network enabling

Bean developers need not make the beans network-enabled. The container automatically supports distributed architectures by wrapping the bean instances by bean objects which are network enabled. These bean objects *intercept* bean service requests, in order to provide the other services like transaction, security and persistence. Every EJB is published as both an RMI and a CORBA object, and can easily (through the use of Java annotations) be published as a Web Service as well.

### 1.1.7.5 Persistence

The application server automatically loads bean information from persistent storage upon bean activation and saves the bean state automatically onto persistent storage upon deactivation.

Furthermore, the bean developer need not know the details of the structure of the persistent storage (e.g. which object fields are stored in which columns of which tables or whether the persistent storage is a relational or object database). The mapping to persistent storage can be done declaratively by a database administrator who need not be a Java developer. It can be even left to the EJB container to create the required database tables and to define the mapping implicitly.

---

#### Note

If one uses JPA based persistence, then one is locked into object-relational mapping and hence to relational databases.

---

EJB also maps specialization relationships onto relational databases and allows you to choose from a set of standard mappings for specialization.

#### 1.1.7.6 Component location transparency

EJB Application Servers must provide a naming and directory service which implements the Java Naming and Directory Interface (JNDI). The JNDI is a generic API for interfacing with general naming and directory services (such as, for example, LDAP).

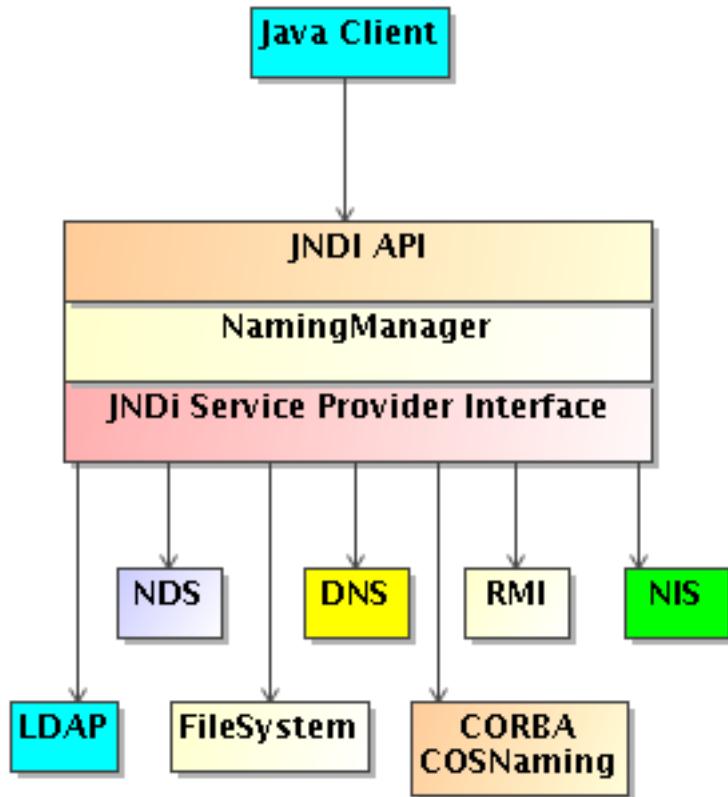


Figure 1.2: The JNDI Architecture

As is shown can be seen in Figure 1.2, JNDI wraps concrete naming and directory services. Some of the naming and directory services which can currently be accessed through JNDI are

- **Local File Systems** Files and directories
- **COSNaming** CORBA's standard naming service is meant to enable CORBA clients to look up a reference to a CORBA object from a name.
- **RMI registry** Java's RMI naming service which fulfills the same purpose as the CORBA naming service for Java RMI objects.
- **LDAP** The *Lightweight Directory Access Protocol* was developed in the early 1990's as a standard directory protocol which would be used by a wide range of applications. It facilitated, for example, that the particulars (e.g. personal details, phone numbers, e-mail addresses, network and device access particulars, ...) of a new employee could be entered into one central location. Applications like phone and e-mail number search applications, answering machine services, network administration applications etc. would all obtain the required information from one central LDAP. LDAP data is structured as a hierarchical database which allows multiple entries for a specific item. Sun's iPlanet directory server and OpenLDAP are the most well known LDAP servers but most other directory services provide an LDAP interface. LDAP version has support for referrals -- i.e. it makes it possible that LDAP service requests are referred on to other service providers. This enables large-scale clustering and distribution.

- **DNS** Internet *domain name servers* which map a domain name (the host name) onto a IP address (the message path) can also be accessed through JNDI.
- **NIS** NIS, Sun's *Network Information Service*, which acts as a yellow pages service for network resources.
- **NDS** The *Novell Directory service*.

JNDI thus provides a standard interface which decouples the application from the physical naming and directory service provided by the environment. Naming services are typically used to obtain a reference to an object in a distributed environment. Directory services are really sophisticated naming services which include metadata describing the objects they reference. This enables clients to make more sophisticated searches for objects -- i.e. for example to query all printers in a particular building which can print color onto A3 sized paper.

#### 1.1.7.7 Transaction support

A transaction is a set of operations that must be processed as a single unit and if that unit was not successful in its entirety the entire transaction must be rolled back. Transaction boundaries are used as instants where object states are synchronized with the database.

The EJB application server/container provides implicit support for distributed transaction management freeing developers of the burden of either including API calls to managing transactions themselves.

#### 1.1.7.8 Security support

When you open your systems across an intranet or even internet, security becomes of vital importance. The EJB application server facilitates a declarative support for authentication and authorization which removes the burden of making calls to a security API from the bean developer.

The authentication is usually done at the persistence layer using normal JAAS (Java Authentication and Authorization Service) based security.

Confidentiality is usually transparently configured within the architecture resulting in secure communication (via, for example, SSL).

At the services layer, the main focus is on authorization. The bean deployer defines security roles for entire enterprise beans or for individual services supplied by enterprise beans. A security administrator maps users and user groups onto security roles.

#### 1.1.7.9 Session management

The client session is solidly managed across presentation and services layers (the session beans). The session context and state is propagated across session beans.

#### 1.1.7.10 Interception

Enables you to intercept both, business logic and bean management services in order to add further responsibilities around a set of base responsibilities addressed by the bean logic itself and by the application server. This makes enterprise beans externally extensible.

#### 1.1.7.11 Resource connection pooling

In a similar way, the container is typically responsible for resource connection pooling like, for example, database connection pooling. Establishing these connections is typically expensive. Enterprise beans should typically not establish connections themselves to resources. Instead they obtain a connection from the container who maintains a connection pool.

### 1.1.7.12 Implicit monitoring

EJB containers typically monitor the usage of beans in order to

- Optimize bean and thread pools.
- Perform load balancing across multiple machines
- Support reporting for administration and maintenance purposes.

## 1.1.8 Enterprise beans as business-logic components

### 1.1.8.1 Introduction

Enterprise beans are meant to be *pure server side business logic components which can be deployed on application servers within different business processes (work flows) requiring different security and transactional support, as well as different persistence mappings*. To achieve this a bean implementation should not contain any deployment information and should focus purely on business logic.

### 1.1.8.2 Types of enterprise beans

#### 1.1.8.2.1 Introduction

The EJB standard defines two types of beans:

- **Session Beans** Session beans provide expose services (tasks, business work flows) for clients. They may be
  - **Stateless** Stateless session beans do not maintain state across service requests. They are usually used as facades to provide a consistent, client-centric services interface on top of a potentially complex application.
  - **Stateful** Stateful session beans maintain state across service requests, i.e. they maintain the state of the interaction or conversation throughout the client session. (for example, a shopping cart or workflow manager)
- **Message Driven Beans** Message driven beans provide asynchronous service offerings to clients. They process service requests off a message queue or topic.

Prior to version 3, a third type of bean called *entity beans* existed. Though the concept of persistent entities (containing the ‘state’ of the system) still exist, they exist now as part of a separate, simplified standard called JPA (the Java Persistence API).

#### 1.1.8.2.1.1 Beans as lightweight components

Since version 3 of the EJB specification, nothing in the Java source code is specific to EJB. The source code is that of a simple Java class, devoid of any architecture information.

The information which is relevant if the Java class is deployed in an EJB container is provided as meta-data using Java annotations (or XML files). This information is used by the application server to manage the component. In the case of entities, it may include information on how the information in the Java objects must be mapped to the database.

#### 1.1.8.2.2 Session Beans

Session beans are objects which perform some operations on behalf of a single client. They are really like objects delivering use cases to clients. Session beans are only accessible by one client at a time, i.e. they are not shared between clients.

In response to client service requests the container creates a new instance. The new instance is a new object for the new session and has a unique identity from the client’s perspective.

---

**Note**

In actual fact one of the session bean instances in the method ready pool is activated and assigned to process a service request for the new conceptual object. Based on CORBA, enterprise beans *separate the client view of the object from the physical realization of the object.*

---

Conceptually a session bean lives as long as the client maintains the session -- hence the name *session bean*. They can be seen as a server object which exists only to serve a single client over the period of a session.

The EJB specification requires support for two flavors of session beans, stateless and stateful session beans.

#### 1.1.8.2.2.1 Stateless Session Beans

Stateless session beans *do not maintain state across service requests*, even if those requests were made from the same client session. For example, the tasks of crediting one account, querying the balance of another and making a payment from a third do not require any information to be held across these service requests. A stateless session bean could provide these services.

---

**Note**

Stateless session beans facilitate *very simple object pooling*. Any bean instance can be used from the method-ready pool. No state needs to be persisted across de- and re-activations.

---

#### 1.1.8.2.2.2 Stateful Session Beans

Stateful session beans *maintain state information across service requests within a single session*. For example, a bean handling a customer during an on-line shopping spree would typically be a stateful session bean with the shopping cart content as well as later on the payment method and delivery address details being part of the state of the session bean.

---

**Note**

*To preserve state and identity across deactivations and reactivations, stateful session beans must be serializable.* The state is typically serialized onto a file or onto a blob in a database.

---

#### 1.1.8.2.3 Message-Driven Beans

MDBs are beans which can process JMS (Java Messaging Service) messages. JMS messages are asynchronous messages fed by a messaging service that facilitates reliable asynchronous communication. If a message recipient is temporarily not available, the message will typically be stored until it can be delivered.

##### 1.1.8.2.3.1 Synchronous versus asynchronous bean requests

Session beans receive synchronous service requests -- i.e. the calling thread's execution is paused until a reply or an exception is received. In the case of message driven beans the calling thread simply dispatches the message and, upon confirmation of receipt from the message queue or topic, the client thread continues processing while the server may still be processing the message.

##### 1.1.8.2.3.2 The onMessage() service

Unlike entity and session beans, message driven beans provide only a single `onMessage()` service. If different services are to be provided by a single message driven bean (typically this is not recommended), then the different service requests must be encoded in the message itself.

---

### 1.1.8.2.3.3 Providing a response

Message-driven beans do not support return values (it would not make sense in the context of processing asynchronous service requests.) If a message-driven bean should make a response available to the client, it again should place a response message on a message queue which will route messages to the client.

## 1.1.8.3 Elements of an enterprise bean

### 1.1.8.3.1 Introduction

There are a number of elements which contribute to an enterprise bean. Here we discuss these elements and their purpose.

### 1.1.8.3.2 The Remote Interface

The remote interface is the contract that specifies the services offered by the enterprise bean. Remote clients (entities outside the application server) request services from beans as specified through the remote interface. This is arguably the most important aspect of bean development, and is also usually the first deliverable produced when following a design process such as URDAD.

This contract is the only aspect of the bean which is visible from the outside world, which means that it is the most difficult to change without impacting clients.

---

#### Note

As of EJB 3.1 onwards the remote interface is optional. We generally recommend contract driven development and hence the separate specification of a services from any particular implementation class.

---

### 1.1.8.3.3 Local Interfaces

If another component in the same (local) application server - such as another bean or a web component - uses a session bean through its remote interface, it is very slow compared to local Java calls because the service request is marshaled onto RMI/I-IOP/TCP/IP as if it was a remote call (i.e. communication occurs through local networking). Furthermore, exchanged objects are passed by value (and not by reference, as per local Java method calls).

The bean developer may choose to, however, expose either the same (or a different set of) services through another interface, the *local* interface. Services requested through this interface are realised as local Java method calls, with greatly improved performance and pass-by-reference semantics.

### 1.1.8.3.4 The enterprise bean implementation class

The bean implementation contains the realisation of the services published in the remote (and/or local) interfaces. They should contain pure business logic which is devoid of any transactional, security or persistence logic. This is the only class written by the developer, and this class will effectively be ‘wrapped’ by the various enterprise services the container provides.

The bean implementation must implement all remote and/or home interfaces through which it is published, i.e. it must realise all the services it claims to be able to provide.

### 1.1.8.3.5 The EJBObject

If incoming service requests were dispatched directly to a bean, all of the enterprise services provided by the EJB container would be bypassed. To this end, beans which are deployed in a EJB container are never directly accessible: The EJB container generates a class (which implements both the remote interface of the bean, as well as the `javax.ejb.EJBObject` interface). The EJB container uses the information specified either as Java annotations, or in XML deployment descriptor, to generate an EJBObject which enforces the qualities desired by the developer (security, transactional behaviour, web services publication, etc.).

The EJB object thus intercepts the call to the enterprise bean, enabling the container to provide standard support to your beans, such as

---

- concurrency,
- transactions,
- security, and
- persistence.

The way in which a service request is intercepted and ultimately processed is shown in the following figure.

#### Note

Since EJB uses RMI/IOP/TCP/IP as protocol, enterprise beans can be accessed from CORBA through the standard CORBA protocol, IOP. This means they are directly accessible to, for example, CORBA or C++ clients.

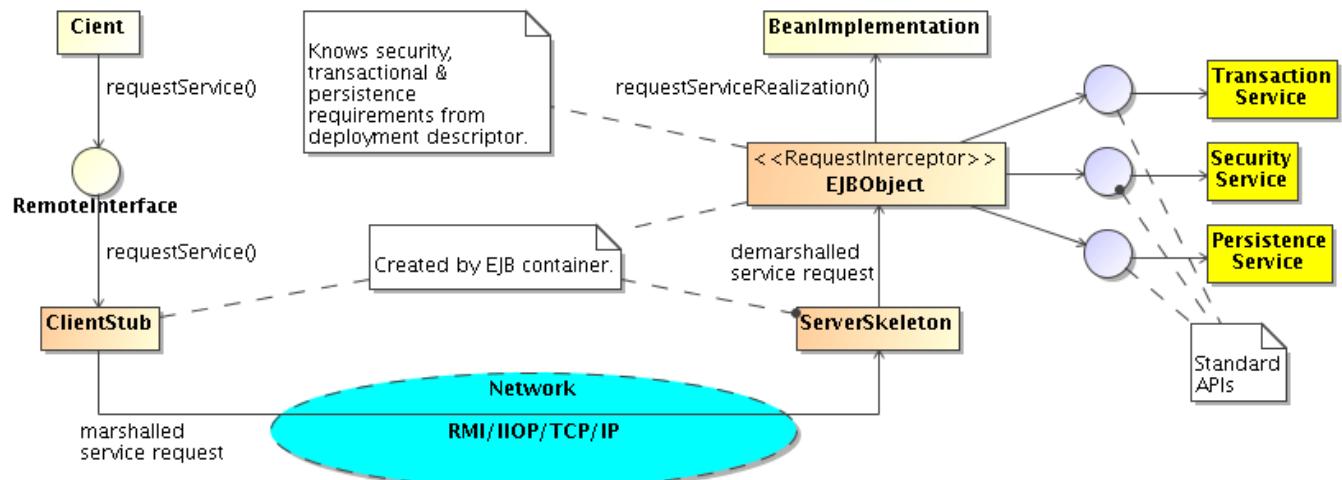


Figure 1.3: Bean Service Request Processing

#### 1.1.8.3.6 Deployment Descriptors

The deployment descriptor represents the configuration of an EJB module (a deployable set of enterprise beans), and is used to specify the bean implementation details (i.e. the bean class and its home and remote interfaces) as well as the middle ware support required from the container (e.g. required support for transaction, security and persistence) as well as the security roles in which the beans themselves should run in.

The deployment descriptor is written in XML and can be developed with any text or XML editor. However, usually application servers supply specific tools to create and maintain deployment descriptors.

#### Note

Since EJB version 3, XML-based deployment descriptors are an optional feature, as most configuration can be applied directly to the beans using Java annotations.

#### 1.1.8.4 The deployment package

Your Enterprise Beans artifacts (i.e. those which are not generated by the application server) are packaged within a JAR file called an EJB Module, which is deployed in the EJB container.

The EJB module file is a compressed container of the user-defined bean specific elements (the implementation class, the home and remote interfaces, and the deployment descriptor), and sometimes application server specific files (such as server-specific persistence configuration).



Figure 1.4: The EJB-Jar package

#### 1.1.8.5 How enterprise bean elements collaborate?

In the following figure, we show the structural relationships between the bean elements and how they collaborate in the context of delivering a service to a client.

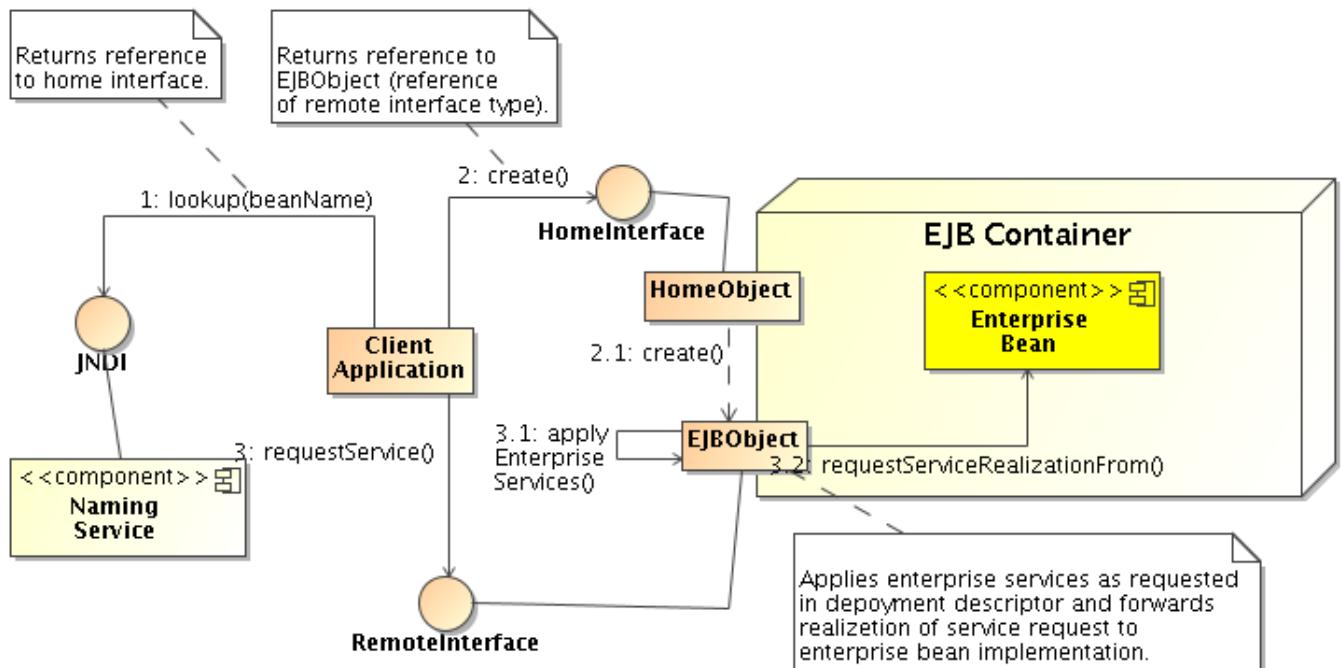


Figure 1.5: How do the bean elements collaborate

#### 1.1.8.6 Enterprise bean restrictions

The container takes over a lot of responsibilities which would otherwise have resided with the bean developer. Consequently the bean itself is prevented from doing certain things which interfere with the container operations.

##### 1.1.8.6.1 Beans can't give clients direct access to the bean instance

The client should never interface with the bean instance directly, but should instead interface only with the EJB object which is generated by the container and acts as a portal to the bean. This architecture enables the container to intercept bean service requests and take over the responsibility of container managed services like transactions, concurrency, security and persistence.

Though you can't pass a handle of the bean instance to the client, you can, of course pass the handle to bean helper classes (which are, from the client's point of view part of the bean implementation).

#### 1.1.8.6.2 Enterprise beans may not accept network server connections

Beans should not act as servers themselves. Once again, this is the responsibility of the EJB object. The bean can, however, open client connections to other network servers (e.g. CORBA servers or other enterprise beans).

#### 1.1.8.6.3 Enterprise beans should be single-threaded

In order to be able to effectively handle concurrent service requests from a large number clients the application server uses thread pooling. For this reason the application classes should not create their own threads. If they do want to have a piece of work done in a separate thread, then they should submit the piece of work to the application server which will assign a thread from a thread pool for processing that piece of work in a managed way.

The bean should also not use synchronization itself. This too will be handled by the application server within the transactions framework.

#### 1.1.8.6.4 Enterprise beans should not create a user interface

The whole idea behind Enterprise Java Beans is to separate the presentation layer from the business logic layer. No direct interaction via a GUI (AWT or Swing) or keyboard input is allowed for enterprise beans. Of course, the latter follows from the fact that enterprise beans may not use the `java.io` package. Bean developers may also not assume that the bean host has any form of GUI support.

#### 1.1.8.6.5 Stateful session beans can't have persistent class fields

The idea of maintaining persistent class information (static fields) for an EJB goes largely against OO concepts and warrants a redesign with perhaps introducing further entity beans.

There are several problems with maintaining class state for an EJB. Firstly, the container will not manage concurrent access to such fields and you as bean developer may not. Furthermore, the EJB service requests may be distributed by the container across Java Virtual Machines (JVMs) and the class state would not be available across these JVMs.

Thus, you should only use constant (`final`) class attributes for EJBs.

#### 1.1.8.6.6 Enterprise beans may not use any native libraries

The reasons for this are firstly security and secondly portability. If you really need to obtain access to native libraries you should wrap them as a CORBA component and use them via standard CORBA service requests.

### 1.1.9 Entity objects

The Java EE specification supports the concepts of entity objects, which are managed by an entity manager. Entity objects exist from when they are created until they are deleted. The entity manager manages the persistence to durable storage (some form of database).

Entity objects have persistent object identity and can be looked up on that object identity. Hence, these objects exist, from the user's side, across client sessions and server restarts. The entity manager will typically use JPA (the Java Persistence API) to interface with an object-relational mapper (such as Hibernate or Toplink) in order to persist the state of objects to a relational database.

---

#### Note

Though possible (via JDBC), Java EE proposes that developers no longer work with relational databases using relational elements such as SQL and result sets. Instead, it is proposed to stay within a fully object-oriented realm, and let the infrastructure manage the mapping between the two worlds.

---

### 1.1.10 Enterprise bean as flyweights

The EJB model uses the flyweight pattern where a few physical objects (from the bean pool) represent many conceptual objects. For example, a bank may manage many accounts, all of which are conceptually available at any stage. At any moment in time there may actually be more accounts being manipulated than there are entity objects for accounts. If a client requests a service from her account, the application server will grab any available account object from the bean pool, populate it with the object identity and state from the database and the user uses it.

Note that across invocations from the same user reference the physical object providing the service may change, but it represents the same user (conceptual) object. To facilitate this the instance specific state as well as the object identity need to be externalised (for example residing in the data base).

---

#### Note

The separation of the conceptual or user object from the physical object is also core in the non-proprietary object-oriented integration technology, CORBA. In CORBA, which in many ways forms the basis for EJB, the physical realization of a user object may even be non-object oriented (e.g. a ANSI-C or Cobol realization) and may change location and realization technologies, while representing the same user object.

---

### 1.1.11 Clustering

Clustering is a group of machines working transparently together to provide enterprise services with a higher level of reliability and scalability.

#### 1.1.11.1 What needs to be clustered

For a fail-over safe system, one would preferably not have any single point of failure, i.e. a situation where the failure in one system component results in a failure. To this end one would usually want redundancy and fail over safety in all of the following:

- firewalls,
- load balancers,
- web containers with session replication,
- EJB containers with session replication,
- JNDI repositories with synchronization,
- data bases with synchronization and fail-safe data base connections or SAN switches,
- messaging services, and
- web services.

#### 1.1.11.2 Availability versus reliability

In a high availability system a redundant server is available when a server falls over. The user may, however, have to establish a new session and the session state of the previous session will typically get lost.

In a high reliability system the state of the various components is replicated across the cluster or at least across parts of the cluster such that, upon a server falling over, the user can transparently continue with his/her work flow without realizing that the machine hosting the session has been switched.

---

### 1.1.11.3 Using Storage Area Networks (SAN)

Normally shared disk clusters where software elements on different nodes use shared persistent storage would be avoided due to the single point of failure at the persistence level. However, SAN can provide a single logical interface into a redundant storage medium to provide failover and scalability.

Using shared fail-over safe storage as in SAN systems reduces the complexity around the replication and synchronization

### 1.1.11.4 Replication algorithms

Some application servers like perform cluster-wide or partition-wide replication with automatic cluster configuration via node discovery mechanisms, while others like perform single-node replication with spawning of another replicated bean on another available node upon node failure.

## 1.1.12 Client containers

The Java EE specification introduces the concept of a client container which provides the required environment for a Java EE client.

The services provided and the actual configuration differ a lot from application server to application server. In some the client container is simply a set of required libraries while in others the client container may be an environment within which the client is executed.

Client containers facilitate suitable environment for the client provides and manages the gateway between a Java EE client application and the Java EE server components.

In particular it supports

- The client environment provides the environment enabling the Java EE client to communicate with the Java EE server components. This may include support for different communication protocols like RMI/SSL.
- The client container provides the client the infrastructure and configuration for looking up server side components.
- The client container makes the Java EE annotations available to the client.
- Application servers may, but are not required to, provide a client container which is distributable via Java WebStart.

## 1.1.13 Java EE and quality attributes

To evaluate the Java EE architecture, we look at the realisation of quality attributes.

### 1.1.13.1 Flexibility/modifiability

The clean layer and component-based approach with each component implementing an appropriate interface results in

- encapsulation,
- decoupling,
- responsibility localisation,
- run-time deployment and registration, and
- the enforced use of a naming service

results typically in a flexible system which can be modified cost-effectively.

### 1.1.13.2 Reliability and availability

Using clustering with state replication and the transactional support provided by J2EE can result in a very reliable system with a high level of availability.

### 1.1.13.3 Scalability

The

- thread-pool based concurrency support,
- resource connection pooling, and
- clustering

all contribute to make typical Java EE applications very scalable.

### 1.1.13.4 Manageability

This domain is typically tool-vendor dependent. Nevertheless, with some application servers it is easy to hot-deploy enterprise beans and to manage performance via bean poll sizes and other measures.

### 1.1.13.5 Security

The front-end supports authentication, confidentiality and non-repudiation (typically via SSL), while the business logic layer supports role-based authorization with automatic forwarding of the security context across bean calls.

### 1.1.13.6 Performance

Performance is a two-sided sword in the Java EE architecture. Already by the nature of being a reference architecture for distributed enterprise systems, a performance issue is usually introduced due to the nature of distributed systems.

Furthermore, the layering and the decoupling with the requirement that home object references are obtained via JNDI from a naming service adds further performance pressures.

On the other hand, the support for local interfaces, thread and resource connection pooling, database caching opportunities introduced with CMP2 and of course the clustering can all contribute to achieving an acceptable level of performance.

### 1.1.13.7 Integrability

Java EE scores very well on integrability with support for CORBA and web services integration standard, and the possibility of plugging in JCA adapters for native integration.

### 1.1.13.8 Time-to-market

Time-to-market is supposed to be good for Java EE. One of the problems is, however, that designers and developers are typically faced with a steep learning curve in both, the technology and the tools supporting the technologies. Furthermore, many of the tools hide the underlying concepts making the understanding and consequent problem solving at times more difficult.

### 1.1.14 Java EE best practices

Even though the architecture is to a large extend prescribed with the J2EE reference architecture, there is still ample room for developing applications which do not perform well or which are not modifiable, reliable or secure.

---

**Note**

Many of these practices are equally applicable for other distributed systems like CORBA or web-services based systems.

---

- *Minimize the number of network roundtrips* required by the application, perhaps using the *session facade* or *message facade* pattern.
- If network bandwidth is a problem, consider *compressing the data* which is transferred.
- *Work object-oriented within the application server and services oriented outside.* Thus, publish high-level user services in a *services facade* (e.g. a session or web-services facade) from which your entity and other domain objects are accessed.

---

**Simple example**

Instead of clients executing a transfer between accounts by interacting with the account objects directly via `acc1.debit(x)` and `acc2.credit(x)`, we rather publish a higher-level `transfer` service receiving the two account numbers and the transfer amount as arguments.

---

- In some cases you may want to reduce overheads incurred when looking up the bean home interface from the naming repository by *caching home interface references*.
- You may want to use *local interfaces* for local calls in order to reduce the marshalling and communication overheads. In practice application servers will automatically map calls to local beans as local calls. The decision should of whether to make a local or remote call should preferably be left to the application server.

---

**Note**

It is usually safe to map remote calls onto local calls, but not vice versa due to the differences in parameter passing between local and remote calls.

---

- In stateful session beans, reduce the serialization overhead by declaring working data which does not need to maintain state as *transient*.
- Declare non-transactional methods with either *Never* or *NotSupported* transactional attribute.

### 1.1.15 Java EE and SOA

Java EE is the most widely used reference architecture for enterprise systems based on the layered architectural pattern. It provides the basis for multi-tiered enterprise systems and is particularly useful for interactive systems.

Another widely used reference architecture for enterprise systems is the Services Oriented Architecture (SOA). The Java community has provided a standard API specification for SOA based systems, the *Java Business Integration* (JBI) specification.

SOA provides an infrastructure for

- defining, maintaining and executing business processes defined across enterprise systems,
- decoupling enterprise systems such that they just provide services to or request services from the enterprise bus.

SOA is not meant to replace Java EE. It does not provide an infrastructure for effectively hosting business logic and does not provide a presentation layer. The individual systems integrated through the enterprise bus may very often be based on the Java EE architecture.

---

### 1.1.16 Exercises

1. Discuss the criteria you would use to decide whether Java EE is a suitable architecture for your project.
2. Consider the following systems and discuss the benefits and disadvantages of using a Java EE architecture:
  - Video shop application.
  - System for processing insurance claims.
  - Surveillance system that processes streaming video.
3. Provide an example of where you would use
  - a stateless session bean,
  - a stateful session bean (other than the typical e-commerce shopping cart scenario!),
  - an entity bean and a
  - message driven bean.

What criteria would you use to decide whether to use a normal class or an enterprise bean in a Java EE-based system?

# Chapter 2

## Maven

### 2.1 Introduction

Apache Maven grew out of a realization that different Ant-based builds tended to be quite different making it difficult for developers to move across projects. Furthermore, Ant provided no standard way to

- make the outputs of the project visible and globally reusable nor to
- effectively manage project dependencies

#### 2.1.1 Guiding principles of Maven

Apache Maven is more than just a build tool. It suggests and enforces proven work and structural patterns for developers, testers, project managers and users to be able to effectively collaborate in order to develop, test, observe and deploy/install the software system.

Apache Maven was developed in the context of the following guiding principles:

1. Enforce proven patterns including patterns for
  - file/project structure,
  - development process,
  - dependency management and installation.
2. Increase visibility including
  - code and component visibility for project tracking and reuse,
  - test visibility for bug tracking and fixing,
  - visibility of any other metadata like project vision, participating developers, ...
3. Effectively manage dependencies and resources

### 2.1.2 What does Maven provide?

- **Dependency management** First of all Maven requires that the direct project dependencies are explicitly declared. It then facilitates the retrieval of dependencies from local and remote repositories including transitive dependencies.
- **Standard project structure**
- **Standard processes for building, testing, installing, ...**
- **Project tracking**
- **Project visibility** Reporting, metrics
- **Repositories** Remote repositories Both large repositories of standard libraries and local repositories. All artifacts in repository versioned. Metadata.
- **Speed and maintainability** can setup project fast and less environment to maintain
- **Simpler modularization of projects** Maven encourages modularization
- **Distributability** standard installs smaller distributions with dependencies pulled in when not locally available

### 2.1.3 Maven versus Ant

- **Declarative versus operational approach**
- **Project structure / metadata versus explicit build instructions**
- **Dependency management**
- **Project inheritance**
- **Modularization**
- **Project reporting and measurement**
- **Repositories** Global and local repositories for plugins, libraries and applications.

### 2.1.4 What is Maven really?

Ultimately Maven is a project management tool which provides

- a project object model (POM) which encapsulates project metadata including dependencies, structure, distribution information and build and reporting customizations,
- standard life cycles for build, clean and documentation generation/reporting,
- a plugin framework which enables plugins to publish goals which can be bound to life cycle phases of either the standard built-in life cycle or to phases of its own life cycle,
- a dependency management system,
- project repositories and mechanisms sourcing dependencies from local and remote repositories,
- some special plugins with their own life cycle like that for cleaning projects and generating reports for them.

Through this Maven encourages

- universal reuse of plugins and projects through a managed infrastructure.
- uniform project structures favouring convention over configuration.

## 2.2 Maven's Project Object Model (POM)

### 2.2.1 Introduction

The *Project Object Model* (POM) encapsulates the project metadata. It may contain

- the project identifier, name, version, ...
- information on the project contributors and licensing,
- the structural organization (directories) of the project,
- the project dependencies,
- relationships between this project and other projects, and
- information about the build environment for the project.

### 2.2.2 POM structure

The POM is encoded in XML and the structure of the POM is defined by an XML schema. The UML class diagram of figure Figure 2.1.

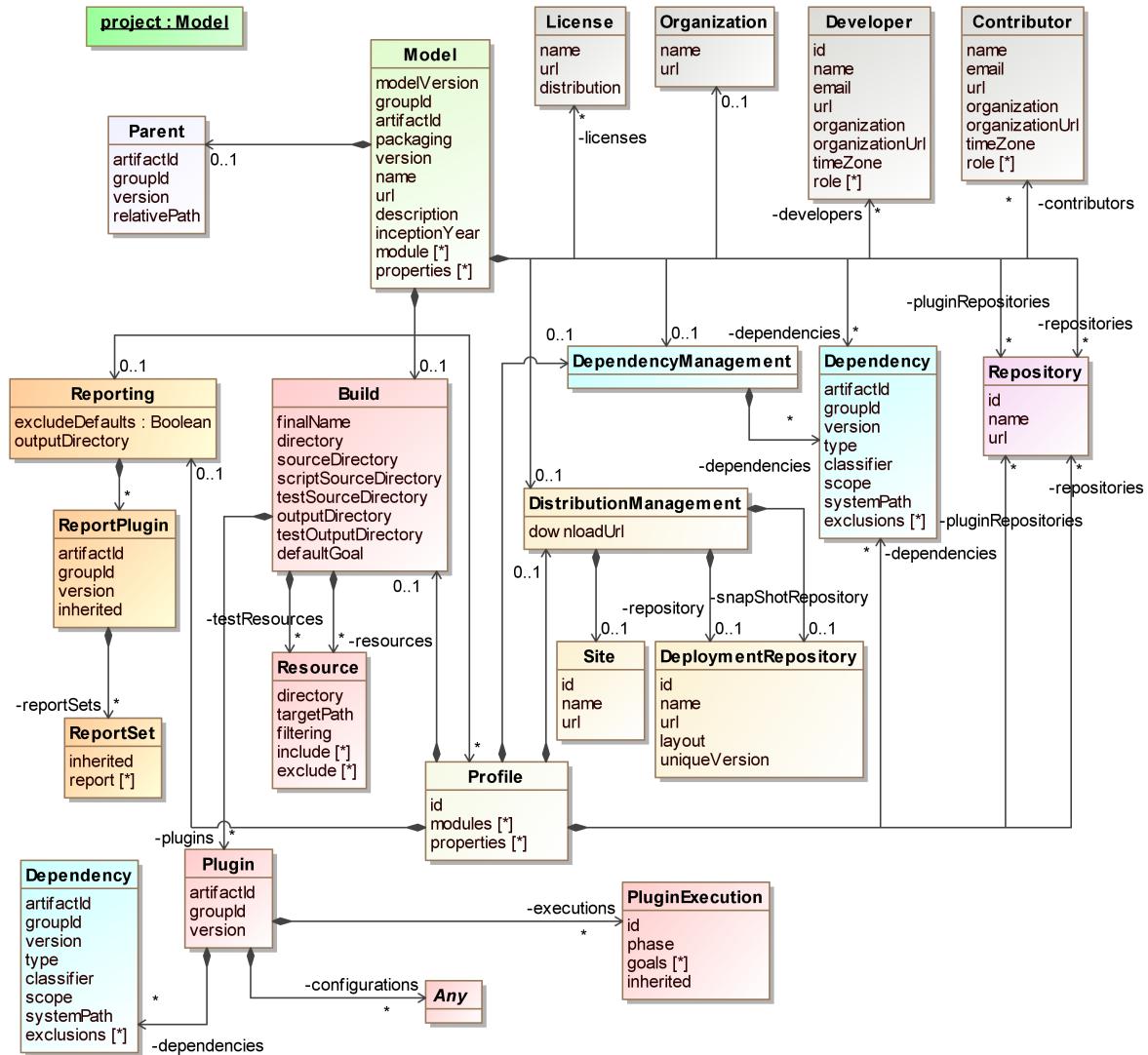


Figure 2.1: The structure of Maven's POM

At the high level the project object model may specify

- a project identifier assembled from artifactId, groupId and version,
  - the version of the POM model used,
  - general project metadata like project name and description, licensing and contributor information and information about the organization which claims ownership of the project,
  - relationships the project has with other projects,
  - the project dependencies,
  - repositories from which dependencies and plugins are to be sourced,
  - build information,
  - information on how the outputs need to be distributed,
  - information of different build profiles like development, testing and production profiles.

- information on how the project measurement and reporting should be done,
- as well as a collection of properties which can be used throughout the POM.

### 2.2.3 Project Identifiers

Projects, dependencies and plugins are identified by

```
<groupId>:<artifactId>:<packaging>:<version>
```

Strictly speaking the packaging itself is not part of the project ID, but it is still used to source the appropriately packaged version of a resource.

For example, a minimal POM

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>za.co.solms.mavenCourse</groupId>
  <artifactId>simpleExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <name>simple</name>
  <url>http://www.solms.co.za</url>
</project>
```

which has as project identifier

```
za.co.solms.mavenCourse:SimpleExample:1.0-SNAPSHOT
```

It would, by default, be packaged in a jar named

```
simpleExample-1.0-SNAPSHOT.jar
```

#### 2.2.3.1 Group ID

The `groupId` is to identify the group or organization which takes ownership of the project. The convention is to use the domain name of the group/organization in reverse.

#### 2.2.3.2 ArtifactId

The `artifactId` is a project identifier which should be unique within the group or organization which owns the project.

#### 2.2.3.3 Version

This represents an identifier for the version of a project. During active development the version is usually designated as a SNAPSHOT version.

#### 2.2.3.4 Packaging

This specifies the packaging to be used. Examples include `jar`, `war`, `ear`, `pom`, and `zip`.

## 2.2.4 POM inheritance

In order to ensure some level of uniformity across projects you typically will want to define common project parameters in a parent POM which will be inherited across the various child POMs.

### 2.2.4.1 Declaring the parent POM

A POM declares another POM the parent POM by specifying the project identifier and optionally a relative path. For example

```
<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/maven-v4_0_0.xsd">

    <parent>
        <groupId>za.co.solms.mavenCourse</groupId>
        <artifactId>simpleParent</artifactId>
        <version>1.0</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.mavenCourse</groupId>
    <artifactId>simplestExample</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>

    <name>simple</name>
    <url>http://www.solms.co.za</url>
</project>
```

A child project will inherit all definitions defined in the parent project and will only have to either

- add additional definitions which are not inherited,
- override inherited definitions with its own definitions.

---

#### Note

The inheritance is not at class but at instance level. Thus, a particular instance of a POM inherits from another instance of a POM which has been declared it's parent POM.

---

### 2.2.4.2 The Super POM

Maven enforces a common ultimate parent POM from which all projects inherit. This POM is called the Super POM. It comes packaged with any Maven installation.

The Super POM encapsulates some defaults shared across all projects including

- The default remote Maven repository from which dependencies are, by default, obtained.
- The default remote Maven plugin repository from which plugins are sourced.
- Default build information including
  - a default directory structure form Maven projects including the source and output directories for the code and tests which are set to

```
src/main/java  
src/test/java  
src/main/scripts  
target  
target/test-classes
```

- the default target directory and default build filename which is

```
 ${pom.artifactId}-${pom.version}
```

which is ultimately appended with the packaging type.

- A default list of plugins which are available for the default Maven build cycle including plugins for
  - compiling,
  - creating jars, wars, ears, rars and javadoc,
  - resolving dependencies,
  - deploying and installing, and
  - cleaning the project.
- the default report output directory which is set to

```
target/site
```

•

#### 2.2.4.3 The effective POM

The effective POM is the sum-total of all inherited POM elements, some of which may be overwritten at some level of the parent-child hierarchy for the POM. You can use the `effective-pom` goal of the `help` plugin to query the effective POM for a specific child/concrete POM by executing

```
mvn help:effective-pom
```

from within the directory which contains the child POM.

### 2.2.5 Project dependencies

The POM requires that the direct dependencies of a project are explicitly specified. Project dependencies may be either

- *external dependencies* on libraries or other resources developed by other groups or organizations,
- libraries or other resources developed within other internal projects.

#### 2.2.5.1 Specifying project dependencies

Project dependencies are specified in a separate dependencies section. The dependency is identified through the standard

```
groupId:artifactId:version
```

project identifier:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>solms.co.za.mavenCourse</groupId>
      <artifactId>testLibrary</artifactId>
      <version>1.4.2</version>
    </dependency>
  </dependencies>
  ...
</project>
```

#### 2.2.5.1.1 Specifying version ranges

Instead of locking the dependency into a particular version, Maven allows you to specify a dependency on a version range, specifying the lowest and highest version numbers which would be acceptable. This can be done inclusively via square brackets `[1.51, 1.9]` or exclusively via round brackets `(1.5, 2)`.

You can leave one boundary open ended. For example,

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[, 4)</version>
</dependency>
```

defines a dependency on any version of JUnit up to, but excluding version 4.0.

#### 2.2.5.1.2 Specifying the scope of a dependency

By default dependencies are available in all classpaths for all build phases and are packaged within the resultant archive. Maven allows you, however, to reduce the scope of a dependency via the `scope` attribute which can have the following values

- **compile** The `compile` scope is the default scope. A dependency which is assigned the `compile` scope is available in all class paths and is included in the package for the project (main) artifact. For example, a library which you are compiling against and whose classes are required at run time would typically be specified using the

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>solms.co.za.mavenCourse</groupId>
      <artifactId>testLibrary</artifactId>
      <version>1.4.2</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

- **provided** The `provided` dependency is used to specify a dependency which would be provided by the deployment/execution environment for the project artifact. For example, the EJB, JTA, JPA, ... APIs would be provided by the application server into which an application is to be deployed into. Dependencies with `provided` scope are still included in the class paths for compilation purposes, but are not packaged within the resultant project artifact.

```
<project>
  ...
  <dependencies>
```

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax-servlet-api</artifactId>
    <version>[2.2,2.3)</version>
    <scope>provided</scope>
</dependency>
</dependencies>
...
</project>
```

- **runtime** Dependencies specified with `runtime` scope are required during execution and testing, but not for compilation. For example, you might need a particular API/contract jar at compile time, but the actual implementation classes for that API only at run time.
- **test** Dependencies declared with `test` scope are only required for the compilation and execution of tests. For example, a JUnit dependency would typically be scoped as a `test` dependency:

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>[4,)</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    ...
</project>
```

- **system** The `system` scope is meant to be used to specify native system dependencies. If you declare a dependency with `system` scope, you will have to specify the `systemPath` element. System scope should be used only in very exceptional cases.

### 2.2.5.2 Transitive dependencies

Maven resolves transitive dependencies, i.e. recursively dependencies of dependencies. Maven does this by building a dependency graph, resolving any conflicts if possible.

For example, your project may have a dependency on the HP *jena* framework for processing RDF and OWL from Java. *Jena* itself depends on *jena.ark* which is its implementation of the SparQL query language for semantic knowledge repositories. *jena.ark* in turn depends on projects like *lucene* and *xerces* and so on. Maven resolves the recursive dependencies and includes them in the application assembly.

### 2.2.5.3 Dependency management

Across the various projects and modules one may accumulate dependencies across large version ranges of other projects. One often would like to standardize the dependencies to particular versions and then manage a controlled process to evolve ones dependencies to later versions. This one does not, however, want to do project for project or even module for module. It is for this purpose that Maven introduces the concept of dependency management.

Dependencies themselves are, of course, not inherited by child projects. One can, however, define in the `<dependencyManagement>` section of the parent project the preferred versions for libraries which are used across child projects. In the child project one still needs to specify the dependency on a particular library, but one can omit to specify a version. In such cases the version will be determined by the version specified in the `<dependencyManagement>` element of the parent project.

Consider, for example, the following top-level POM used by an organization:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>za.co.solms</groupId>
  <artifactId>approvedLibs</artifactId>
  <version>2.1</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.0.1</version>
      </dependency>
      <dependency>
        <groupId>com.hp.hpl.jena</groupId>
        <artifactId>jena</artifactId>
        <version>2.6.0</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

All projects within the organization could now inherit the approved versions for standard libraries used within the organization from the parent POM. The child POMs for the actual projects would not specify the version numbers for these standard libraries (except if they need to override the approved version numbers with project specific version numbers):

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>za.co.solms</groupId>
  <artifactId>courseNotesAssembly</artifactId>
  <version>0.8-SNAPSHOT</version>
  ...
  <dependencies>
    <dependency>
      <groupId>com.hp.hpl.jena</groupId>
      <artifactId>jena</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

#### 2.2.5.4 Exclusions

At times one would like to explicitly exclude particular library versions. The reason for that would be to resolve a conflict between two dependencies caused typically by a transitive dependency being on a different version of a library than what is required by the project itself. This is typically caused by an unnecessary locking into a particular version, i.e. when a specific version is specified in a dependency but the dependency should have really been a dependency range as later and/or earlier versions of that library would be substitutable.

For example, the following POM excerpt specifies a dependency on a *testLibrary*, but

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>solms.co.za.mavenCourse</groupId>
      <artifactId>testLibrary</artifactId>
      <version>1.4.2</version>
      <exclusions>
```

```
<exclusion>
    <groupId>xerces</groupId>
    <artifactId>xercesImpl</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
...
</project>
```

## 2.2.6 Project modules

Whilst project inheritance is used to inherit metadata from parent projects, modules are used to specify a part-of or aggregation relationship between projects. This is used to aggregate builds of sub-projects into a single higher-level build which builds all sub-projects of a project.

One specifies a POM for the aggregate object which lists a number of sub-projects as modules. These sub-projects are, however, not identified via the standard groupId-artifactId-version identifiers but are assumed to be of the same groupId. The module name maps onto the artifactId of the sub-project and the subproject needs to be contained in a sub-directory of the directory for the aggregate project.

Whilst the inheritance relationship is specified from the child projects perspective, aggregation is done from the perspective of the aggregate projects with the sub-projects being unaware that they are modules of an aggregate project.

For example, one could specify different modules for the web client, swing client, web services portal, business logic layer and the reporting module of an enrollment system:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>za.co.solms</groupId>
    <artifactId>enrollmentSystem</artifactId>
    <version>2.3</version>

    <modules>
        <module>webClient</module>
        <module>swingClient</module>
        <module>webServicesPortal</module>
        <module>businessLogicLayer</module>
        <module>reporting</module>
    </modules>
    ...
</project>
```

these sub-projects would be contained in corresponding sub-directories of the enrollment system base directory. They would each have their own POM which could be unaware of the higher-level aggregate enrollment-system project.

The aggregate project need not be the parent of the sub-projects. It is, however, very common to declare the aggregate project the parent project of the sub-projects as that enables one to specify commonalities across sub-projects and in particular introduce some uniformity around the version dependencies via the *dependencyManagement* construct.

## 2.2.7 Build customization

The build section of the POM is used to customize the standard Maven build life cycle. The customization usually involves adding additional build steps to the standard build life cycle for the artifact type built by the project.

One typically defines for a custom build step

- the source and output directories for the sources and tests,
- the directories containing any resources required for the build or for the tests,

- the plugins to be used, the goals (services) to be executed and the build phase to which they should be bound, and any dependencies and configurations.

### 2.2.7.1 Customizing/configuring plugin behaviour

One can use the build section of the POM to customize the behaviour of a defined goal. This can be done in the configuration sub-element for the plugin element of the POM.

For example, it is relatively common to have to customize the behaviour of the *clean* plugin, having to specify additional directories and/or file patterns which need to be removed during the clean phase.

```
<project>
    ...
    <plugins>
        <plugin>
            <artifactId>maven-clean-plugin</artifactId>
            <configuration>
                <filesets>
                    <fileset>
                        <directory>temp/generatedClasses</directory>
                        <includes>
                            <include>*.java</include>
                        </includes>
                    </fileset>
                </filesets>
            </configuration>
        </plugin>
    </plugins>
</project>
```

### 2.2.7.2 Adding a goal to a life cycle phase

For example, in order to add a step which creates Java classes from XML schemas one can specify a build customization which requests the generate goal of the *jaxb2* plugin to generate java classes from all XML schemas located in a `src/main/resources/schemas` directory.

In order to be able to use the Java 6 built-in JAXB support we specify the source and target Java version for the Maven compile step to 1.6.

Finally we also need to include the repositories from which the jaxb2 libraries can be sourced from. The resultant POM is shown 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/maven-4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.example</groupId>
    <artifactId>jaxb-maven-sample-java6</artifactId>
    <packaging>jar</packaging>
    <version>0.1</version>

    <name>JAXB / Maven Sample (Java 6+)</name>
    <description>
        A Sample Maven 2.x project that illustrates usage of the
        JAXB Maven plugin to compile XML Schema resources to Java.
        This is suitable for isolated work on XML documents. For
    </description>
```

```
    web services, usage of the JAX-WS is recommended.
</description>

<developers>
    <developer>
        <organization>Solms TCD</organization>
        <organizationUrl>http://www.solms.co.za/</organizationUrl>
        <email>info@solms.co.za</email>
    </developer>
</developers>

<build>
    <plugins>
        <!-- Configuration to compile all schemas in the
            resources/schemas directory. Automatically
            invoke during the 'generate-sources' phase -->
        <plugin>
            <groupId>org.jvnet.jaxb2.maven2</groupId>
            <artifactId>maven-jaxb2-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <schemaDirectory>src/main/resources/schemas</schemaDirectory>
                <schemaIncludes>
                    <include>*.xsd</include>
                </schemaIncludes>
            </configuration>
        </plugin>
        <!-- Assume a Java SE 6 environment, which includes JAXB.
            For Java 5, extra dependencies on the JAXB Implementation
            should be specified -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
            </configuration>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>[4.1,]</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<repositories>
    <repository>
        <id>maven2-repository.dev.java.net</id>
        <name>Java.net Maven 2 Repository</name>
        <url>http://download.java.net/maven/2</url>
    </repository>
</repositories>
```

```
</repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>maven2-repository.dev.java.net</id>
        <url>http://download.java.net/maven/2</url>
    </pluginRepository>
</pluginRepositories>

</project>
```

## 2.2.8 Distribution Information

One can include distribution information in the `distributionManagement` section of the project's object mode. This provides information which is typically used for deployment or to make the resultant artifact available for other projects. In addition one can specify the URL which should be used by other projects when downloading the deployed artifact. This is done in the `downloadURL` element. Maven itself adds a `status` element which specifies the status of the distribution.

### 2.2.8.1 Specifying distribution Info

The distribution information contains an id, name and url of the distribution repositories for the project's artifacts as well as a flag whether unique versions should be created by adding a time stamp to the file names. The distribution management section allows for a main and a snapshot repository.

For example,

```
<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">
    ...
    <distributionManagement>
        <repository>
            <uniqueVersion>false</uniqueVersion>
            <id>solmsMain</id>
            <name>Solms Main Repository</name>
            <url>scp://solms.co.za/repository/main</url>
            <layout>default</layout>
        </repository>
        <snapshotRepository>
            <uniqueVersion>true</uniqueVersion>
            <id>solmsDev</id>
            <name>Solms Development Repository</name>
            <url>sftp://propellers.net/maven</url>
            <layout>default</layout>
        </snapshotRepository>
        ...
    </distributionManagement>
    ...
</project>
```

specifies a main and a development repository. The development repository adds time stamp information to the artifact names in order to ensure unique versions.

### 2.2.8.2 Specifying login credentials

Obviously one should not distribute the server login credentials (username, password, path to private key, ...) with the POM. These should remain on the build server and are specified in the `settings.xml` file:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>solms.co.za</id>
      <username>myUserName</username>
      <password>myPassword</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>myPassphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
    </server>
  </servers>
  ...
</settings>
```

The permissions are used when creating new files and directories, i.e. these are the permissions to be used for any new files.

## 2.3 Maven's life cycles

### 2.3.1 Introduction

Maven provides a declarative approach to project builds, specifying the build requirements in the projects object model (POM) and then using this information to build, clean or document the project. To this end Maven defines three standard life cycles which are not plugin specific:

- The *build life cycle* is used to compile, test, package, install and deploy a project.
- The *clean life cycle* is used to remove any generated directories and files so that only the sources remain.
- The *site life cycle* measures the project (e.g. collect results of test) and generates a documentation site which contains general project metadata and reporting on project status.

The project is built following a life cycle which is a pipeline of phases which is executed sequentially. Maven defines a standard build life cycle, which is generally followed when building a project. The default build life cycle contains the following core phases (this is not a complete list of phases):

1. validate
2. compile
3. test
4. package
5. deploy
6. install

The build process follows the phases in the order of the life cycle, executing any goals/operations which are bound to these phases.

Plugin goals are bound to life cycle phases in one of the following ways

1. The plugin for the project packaging (project type) binds goals/operations to its life cycle phases which are either phases of one of the standard life cycles or phases of its custom life cycle.

- 
2. In your build customization you can bind different goals/operations to phases of either of the three life cycles.

The life cycle which is used is determined by the plugin for the specified package type (e.g. jar or war). A plugin may hence choose to either

- bind goals to the standard life cycle phases,
- inherit a standard life cycle and add phases, or
- define a new life cycle with its own phases.

In either case, the plugin will bind its operations (goals) to life cycle phases which are either phases of its own life cycle or phases of the standard build life cycle.

### 2.3.2 Maven's default build life cycle

Maven defines a default build life cycle. This is a generic life cycle which is abstract enough to work for the vast majority of builds. Instead of modifying the life cycle itself, different plugins and project may bind different goals/operations to the phases of the default life cycle.

Maven's default build life cycle contains the following phases:

1. **validate** The *validate* phase is meant to be used to validate that the project has all information required to build the project.
2. **generate-sources** The *generate-sources* phase is used if sources need to be generated from other artifacts. This can include goals like generation of Java classes from XML schemas and the generation of code from a design model.
3. **process-sources** The *process-sources* phase is used to pre-process the sources before compilation. This could be used, for example, for filtering and obfuscation.
4. **generate-resources** The *generate-resources* phase is meant to be used if one needs to generate any resources which should be included in the final assembly.
5. **process-resources** The *process-resources* phase for any processing of the generated resources including copying, renaming, setting permissions and so forth.
6. **compile** During the *compile* phase the source code is meant to be compiled.
7. **process-classes** The *process-classes* phase is meant for any post-compile processing like byte-code enhancements, annotation processing, and so forth.
8. **generate-test-sources** The *generate-test-sources* phase is meant to be used if one generates test sources from other artifacts. Typically this would be done from some form of contract specifications (e.g.\ the UML-based services contracts coming from an URDAD analysis and design process).
9. **generate-test-resources** The *generate-test-resources* phase can be used to generate resources for the testing. This could include generating test data.
10. **process-test-resources** The *process-test-resources* phase is used to do any post-creation processing. This typically may involve filtering the resources or copying them into appropriate locations for packaging.
11. **test-compile** The *test-compile* phase is meant for operations which compile the test sources.
12. **test** The *test* phase is the phase where the actual testing of the compiled sources is to be done.
13. **prepare-package** The *prepare-package* phase is a pre-processing phase which is meant to perform operations preparing the organization of resources for packaging before the actual packaging is done. This may include unpackaging certain packaged resources.
14. **package** The *package* phase is meant to create the distributable package, e.g.\ the jar or war.

15. **pre-integration-test** The *pre-integration-test* phase is a preparation phase for the *integration-test* phase. This phase is commonly used to set up the environment for the integration test.
16. **integration-test** The *integration-test* phase is meant to perform the testing in an environment which mirrors the environment within which the artifact is to be deployed. It may include deploying the package into the integration testing environment.
17. **post-integration-test** The *post-integration-test* phase is meant to be a clean-up phase which restores the environment within which the integration test was done.
18. **verify** Whilst the *test* and *integration-test* phases verify the functionality of the package components and that they do not break the functionality offered by the environment, the *verify* phase provides the opportunity to verify whether the package meets certain quality criteria.
19. **install** The *install* phase is meant to install the package in a local repository so that other projects which have a dependency on it can use it.
20. **deploy** The *deploy* phase is used to copy the final package to a remote repository for sharing with other groups and projects. Usually only production-ready packages are deployed.

### 2.3.3 Maven's clean life cycle

Maven's *clean* defines the following phases

1. pre-clean
2. clean
3. post-clean

By default only the *clean* plugin's `clean` goal/operation is bound to the *clean* phase.

### 2.3.4 Maven's site life cycle

The *site* life cycle is used for generating project documentation. It has the following phases:

1. pre-site
2. site
3. post-site
4. site-deploy

#### 2.3.4.1 Default goal bindings for the site life cycle

The *site* plugin binds the

- `site:site` goal to the *site phase*, and the
- `site:deploy` goal to the *site-deploy* phase.

### 2.3.5 Package-based goal bindings

The POM enables one to specify, for a project, the packaging. This identifies the main plugin which determines the build for the project type. Many plugins simply specify goal bindings to standard life cycle phases. Others may define their own life cycle and corresponding goal bindings.

For example, the *jar ejb* and *war* plugins all bind goals to a subset of the default build life cycle. The goal bindings for each of these package/project types are, in fact, the same except for the bindings for the package phase:

1. *process-resources* -> resources:resources
2. *compile* -> compiler:compile
3. *process-test-resources* -> resources:testResources
4. *test-compile* -> compiler:testCompile
5. *test* -> surefire:test
6. *package* -> respectively to jar:jar, ejb:ejb and war:war
7. *install* -> install:install
8. *deploy* -> deploy:deploy

### 2.3.6 Project based life cycle goals

The initial goal binding is determined by the packaging which represents the project type. Additional goal bindings can be specified on a per project or a per parent project level. In the latter case these bindings are inherited by all child projects.

The project or parent-project specific goal bindings are specified in the <>build<> customization element of the POM:

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>pluginGroupId</groupId>
                <artifactId>pluginArtifactId</artifactId>
                <executions>
                    <execution>
                        <phase>somePhase</phase>
                    </execution>
                    <goals>
                        <goal>
                            someGoal1
                        </goal>
                        <goal>
                            someGoal2
                        </goal>
                    </goals>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

## 2.4 Executing Maven

The syntax for executing Maven is

```
mvn [option]* [plugin:goal]* [phase]*
```

specifying at least one plugin goal or life cycle phase and zero or more options.

### 2.4.1 Executing a plugin goal

When executing a plugin goal only the operation/service representing that goal for the plugin is executed. This is done via

```
mvn [option]* pluginName:goalName
```

For example,

```
mvn eclipse:eclipse
```

executes the *eclipse* goal of the *eclipse* plugin which creates an eclipse project for the Maven project.

### 2.4.2 Executing a life cycle phase

When executing Maven against a life cycle phase, all preceding phases of the chosen life cycle are executed. The life cycle which is used is determined by the main plugin which is chosen on the package type for the project (e.g. jar or war). It may be either one of the standard life cycles (default build, clean or site) or its own custom life cycle. In either case, the plugin will bind its own goals as well as the goals of other plugins to the life cycle phases.

Maven thus

1. determines the package type from the POM,
2. finds the corresponding plugin for that package type,
3. obtains from the plugin the life cycle phases and the goals/operations bound to each of the phases up to the requested life cycle phase, and
4. executes the life cycle phases up to the requested phase in sequence by executing any bound plugin goals for each of the life cycle phases.

For example,

```
maven test
```

for a *jar* project executes all phases prior to and including the test phase, i.e. all goals which were bound to these build cycle phases.

In general it is preferable to execute life cycle phases over plugin goals as the dependencies for the build will be resolved.

## 2.5 Maven repositories

### 2.5.1 Introduction

The Maven repository is a central component of Maven, facilitating universal reuse of project artifacts including Maven plugins.

A Maven repository is simply a storage space for reusable artifacts generated from various projects and the Project Object Model (POM) describing how the artifacts can be built and what the artifact dependencies are.

## 2.5.2 Repository structure

The Maven repository is simply a file system with a tree structure conforming to

```
<groupId>.<artifactId>.<version>
```

hierarchy. That directory will contain

- the actual artifact,
- the POM for the project with which the artifact was created,
- and a hashing key which is used to verify that the received artifact is not corrupted.

## 2.5.3 Repository locations

Maven uses both, remote and local repositories. The local and remote repositories are both structured in the same way and hence can be processed using the same logic. Furthermore, since the local and remote repositories have the same structure, they can be simply synchronized.

### 2.5.3.1 Remote repositories

The remote repositories on uses typically include

- the central Maven repository containing the core maven plugins and globally published maven projects,
- an internal remote repository where one publishes one's internal projects across the organization, or
- group/organization specific repositories which publish their projects seperately from the central Maven repository.

There is no structural difference between internal and external remote repositories. Hence the same logic is used to process them and they can be synchronized. Typically the central repository is very large and one does not usually want to synchronize the full repository.

#### 2.5.3.1.1 Accessing remote repositories

Files can be retrieved from remote repositories either via the HTTP or via a mechanism like scp using the `file://URL` protocol. In the former case the directory needs to be, of course, served by the web server. Security can be done via HTTPS or simply by restricting the user access to the directory.

Files can be uploaded to a repository using scp, ftp or other file copy mechanisms.

#### 2.5.3.1.2 Specifying the location of remote repositories

The location of the official central Maven artifact and plugin repositories are specified in the Super-POM:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <name>defaultProject</name>

    <repositories>
        <repository>
            <id>central</id>
            <name>Maven Repository Switchboard</name>
            <layout>default</layout>
            <url>http://repo1.maven.org/maven2</url>
            <snapshots><enabled>false</enabled></snapshots>
        </repository>
    </repositories>
</project>
```

```
</repositories>

<pluginRepositories>
    <pluginRepository>
        <id>central</id>
        <name>Maven Plugin Repository</name>
        <url>http://repo1.maven.org/maven2</url>
        <layout>default</layout>
        <snapshots><enabled>false</enabled></snapshots>
    </pluginRepository>
</pluginRepositories>

...
</project>
```

You can add further repository and plugin repository specifications in either the POM for your project or in a common parent POM for your projects.

#### 2.5.3.2 The local repository

The local repository

- acts as a cache for remote repositories, and
- hosts the artifacts of local projects.

It has the same structure as the remote repositories and can be synchronized with them.

The location of your local repository is

```
 ${home}/.m2/repository
```

#### 2.5.4 How Maven uses Respositories

When Maven resolves dependencies, it sources the the metadata (POM) for the dependency from either the local cache repository or one of the remote repositories. From the POM it obtains the transitive dependencies. This is done recursively building up a dependency tree.

Maven will attempt to resolve any conflicts and then will source the actual artifacts for the dependencies, looking first in the local cache before trying to source them remotely. The integrity of any sourced dependency is validated using the hash which is also obtained from the repository.

#### 2.5.5 Repository tools

To create a Maven repository, it is sufficient to simply create the appropriate directory structure containing the artifacts, poms and hash keys and then to serve the file system by some mechanism (typically via HTTP).

However, Maven repositories typically provide additional functionality like

- searching,
- providing tools for conveniently navigating the repository, and
- publishing the metatdata in a convenient way.

## 2.6 Hello World via Maven

### 2.6.1 Introduction

A hello-world application is commonly used to test and demonstrate an infrastructure without introducing any significant functionality. The purpose is to start getting comfortable with the environment and tools.

### 2.6.2 Creating a project via the Archetype plugin

When creating a project via the `generate` goal of the `archetype` plugin, one needs to minimally specify the `groupId` and `artifactId`:

```
mvn archetype:generate -DgroupId=za.co.solms.training.maven -DartifactId=helloWorld
```

Maven will ask you which archetype it should generate. You will see that there are templates for a wide variety of project types like, for example, JavaEE, JSF and Spring projects. For a basic Java application you can use `maven-archetype-quickstart` which is the default option.

This generates a project directory of the same name as the artifact identifier defines a POM with the corresponding group and artifact identifiers using the default `jar` packaging and defaulting the version to `1.0-SNAPSHOT`. It defaults the project name to the artifact id and sets the url to `http://www.maven.org`. In addition Maven adds a dependency on JUnit:

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>za.co.solms.training.maven</groupId>
  <artifactId>helloWorld</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloWorld</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

In addition to the POM, the `create` `generate` of the `archetype` plugin creates the expected directory structures for the sources

```
-src
  -main
    -java
      -za
        -co
          -solms
            -training
              -maven
                *App.java
```

and test sources of the project:

```
-src
  -test
```

```
-java  
-za  
-co  
-solms  
-training  
-maven  
*AppTest.java
```

as well as the source files for a minimal (hello-world) application and its corresponding test application.

### 2.6.3 Executing default life cycle phases

One can now go ahead and execute any of the default life-cycle phases. For example,

```
mvn compile
```

will execute all life cycle phases up to and including the compile phase. The compiled classes are stored in

```
target/classes
```

Executing the compile phase will result in executing all phases prior to the compile phase in sequential order. This includes, for example, the process-resources phase which attempt to copy any resources from

```
src/main/resources
```

into the target

```
target/classes
```

so that they are in the class path and that they will be included in the packaging.

Similarly we can use

```
mvn test
```

to run the stages up to and including that of compiling and executing the test programs. The test results are saved in

```
target/surefire-reports
```

To execute the life cycle phases up to and including that of creating the jar one can execute

```
mvn package
```

The resultant jar package is simply stored in

```
target
```

If you run

```
mvn install
```

it will execute all the preceding phases including the package phase and will then go ahead and install the jar into the local repository.

#### 2.6.3.1 Deploying onto a server

The last phase of the default build life cycle is the deploy phase. In order to be able to execute the deploy phase one needs to set up the distribution information in the POM and specify the authentication details in a separate `settings.xml` which is not distributed when the project is deployed.

### 2.6.3.1.1 Specifying distribution info

In order to be able to deploy the jar onto a remote server, the server details need to be specified in the distributionManagement section of the POM:

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>za.co.solms.training.maven</groupId>
  <artifactId>helloWorld</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloWorld</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <distributionManagement>
    <snapshotRepository>
      <uniqueVersion>true</uniqueVersion>
      <id>solmsRepository</id>
      <name>Solms Internal Repository</name>
      <url>scp://solms.co.za/var/maven/repository</url>
    </snapshotRepository>
  </distributionManagement>
</project>
```

### 2.6.3.1.2 SSH key for login without password

It is common for systems to have to log automatically into an ssh server without requesting a user to provide a password. This is, for example, required for cron-scheduled backups via rsync or even via scp and for automated deploys within build scripts.

This can be achieved by generating a private-public key pair and appending the public key to the server's authorized keys.

To do this, log into the client machine (the one which should log in) and generate a pair of authentication keys specifying the encryption algorithm to be used (here DSA):

```
ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/clientUser/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/clientUser/.ssh/id_dsa.
Your public key has been saved in /home/clientUser/.ssh/id_dsa.pub.
The key fingerprint is:
4e:c4:73:12:a0:5e:de:40:85:df:4d:bc:1c:f5:15:b3 clientUser@clientMachine
The key's randomart image is:
+--[ DSA 1024]----+
|       o+o . .oo|
|       o... . + .+|
|       . o.=..+ o E.|
|       . o +.+ . + |
```

```
|   . . S      |
|     o       |
|     .       |
|           |
|           |
+-----+
```

Now use ssh to create, on the server, a .ssh in the user's home directory:

```
ssh serverUser@serverMachine mkdir -p .ssh
Password:
```

where serverMachine is the URL of the server machine. It will still ask for the user's password.

Finally append the client's public key to authorized\_keys file in the .ssh directory in the server user's home directory. You will be once again asked for the password, but this is the last time:

```
cat .ssh/id_dsa.pub | ssh serverUser@serverMachine 'cat >> .ssh/authorized_keys'
Password:
```

From now onwards the clientUser can log into the serverMachine under user serverUser without having to supply a password. You can test this via

```
ssh serverUser@serverMachine
```

#### 2.6.3.1.3 Specifying authentication settings

Finally the authentication settings can be added to the settings.xml file, enabling the deploy process to authenticate itself via the public key:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <servers>
    <server>
      <id>repository</id>
      <username>fritz</username>
      <privateKey>~/.ssh/id_dsa</privateKey>
    </server>
  </servers>
</settings>
```

#### 2.6.4 Executing specific plugin goals

When building a project you will generally want to execute Maven specifying the target life cycle phase. This ensures that all preceding phases are executed so that all requirements for your target phase are met. Maven will execute any plugin goals which are bound to the executed phases of the resolved life cycle.

However, at times you may want to execute a particular plugin goal. The reasons for this may be that you want to

- execute a goal which is not bound to any of the default life cycle phases, or
- you want to only execute a particular goal without any of the goals bound to previous life cycle phases.

For example, you could want to just execute the compiler's compile goal you can run

```
mvn compiler:compile
```

This will execute only the compile operation of the *compiler* plugin and not any other goals bound to either the *compile* phase or any of the preceding phases.

#### 2.6.4.1 Executing a program from Maven

One can use the `:java` goal of the `exec` plugin to execute a Java application, specifying the main class in the `exec.mainClass` parameter:

```
mvn exec:java -Dexec.mainClass=za.co.solms.training.maven.App
```

In a similar way one can execute a binary executable via

```
mvn exec:exec -Dexec.executable="binaryExecutable" -Dexec.workingdir="~/temp" -Dexec.args ←
  ="-arg1 -arg2"
```

For example, the following asks Maven to provide a directory listing using long info and showing the hidden files:

```
fritz@fritzLaptop ~/temp/maven/helloWorld $ mvn exec:exec -Dexec.executable="ls" -Dexec. ←
  args="-l -a"
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
[INFO] -----
[INFO] Building helloWorld
[INFO]   task-segment: [exec:exec]
[INFO] -----
[INFO] [exec:exec]
[INFO] total 8
[INFO] drwxr-xr-x 5 fritz fritz 176 Dec 13 22:19 .
[INFO] drwxr-xr-x 3 fritz fritz 80 Dec 13 06:28 ..
[INFO] -rw-r--r-- 1 fritz fritz 936 Dec 13 21:51 pom.xml
[INFO] -rw-r--r-- 1 fritz fritz 470 Dec 13 22:07 settings.xml
[INFO] drwxr-xr-x 4 fritz fritz 96 Dec 13 14:24 src
[INFO] drwxr-xr-x 6 fritz fritz 216 Dec 13 06:39 target
[INFO] drwxr-xr-x 3 fritz fritz 72 Dec 13 22:19 ~
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Sun Dec 13 22:20:45 SAST 2009
[INFO] Final Memory: 5M/82M
[INFO] -----
```

#### 2.6.5 Generating documentation

The `site` plugin can be used to generate documentation for a project:

```
mvn site
```

The `index.html` file for the project is stored in the `target/site` directory. The documentation can be deployed onto the project documentation server via

```
mvn site-deploy
```

This does, however, require that the site distribution information is provided in a `site` sub-element of the `distributionManagement` section of the POM:

```
<project>
  ...
  <distributionManagement>
    <site>
      <id></id>
      <name></name>
      <url></url>
```

```
</site>
</distributionManagement>
</project>
```

## 2.6.6 Cleaning the project

The *clean* plugin is used to remove all temporary and output files and directories, leaving only the sources including any resources, the POM and any settings and properties files. It is executed via

```
mvn clean
```

## 2.7 Maven JAXB Sample

### 2.7.1 Introduction

This simple project uses the JAXB compiler to generate the Java binding classes for an XML schema and then uses a JUnit test to execute the bindings.

This example shows

- adding further dependencies,
- adding and customizing plugins,
- specifying additional repositories and plugin repositories,
- adding further metadata to the project,
- and generating project documentation.

### 2.7.2 The schema

The schema is saved in

```
src/main/resources/schemas/accounts.xsd
```

It defines a few classes with relationships between them:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsschema
    targetNamespace="http://example.solms.co.za/accounts"
    xmlns:a="http://example.solms.co.za/accounts"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

    <xs:complexType name="Account" abstract="true">
        <xs:sequence>
            <xs:element name="balance" type="xs:double" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="accountNumber" type="a:AccountNumber" use="optional"/>
    </xs:complexType>

    <xs:simpleType name="AccountNumber">
        <xs:annotation>
            <xs:documentation>
                10-digit account number
            </xs:documentation>
        </xs:annotation>
    </xs:simpleType>
</xsschema>
```

```
</xs:documentation>
</xs:annotation>
<xs:restriction base="xs:string">
    <xs:pattern value="\d{10}"/>
</xs:restriction>
</xs:simpleType>

<xs:complexType name="CreditAccount">
    <xs:complexContent>
        <xs:extension base="a:Account">
            <xs:sequence>
                <xs:element name="minBalance" type="xs:double" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="ChequeAccount">
    <xs:complexContent>
        <xs:extension base="a:Account">
            <xs:sequence>
                <xs:element name="chequeFee" type="xs:double" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="Client">
    <xs:sequence>
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="account" type="a:Account" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>

<xs:element name="client" type="a:Client">
    <xs:annotation>
        <xs:documentation>
            An instance document containing a single client.
            Accounts must be unique by account number.
        </xs:documentation>
    </xs:annotation>
    <xs:key name="uniqueAccounts">
        <xs:selector xpath="a:account"/>
        <xs:field xpath="@accountNumber"/>
    </xs:key>
</xs:element>

</xs:schema>
```

### 2.7.3 The Test Application

```
package za.co.solms.example;

import java.io.StringWriter;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
```

```
import org.junit.Test;
import org.xml.sax.SAXException;
import za.co.solms.example.accounts.ChequeAccount;
import za.co.solms.example.accounts.Client;
import za.co.solms.example.accounts.CreditAccount;
import za.co.solms.example.accounts.ObjectFactory;

/**
 * Illustrates XML marshalling
 */
public class JAXBTest
{
    @Test
    public void testGenerateXML() throws Exception
    {
        // Create some data using the generated classes
        Client client = new Client();
        client.setName("Jack Black");

        ChequeAccount a1 = new ChequeAccount();
        a1.setAccountNumber("1007657643");
        a1.setBalance(100.00);
        a1.setChequeFee(0.75);
        client.getAccount().add( a1 );

        CreditAccount a2 = new CreditAccount();
        a2.setAccountNumber("1007657642");
        a2.setBalance(-735.18);
        a2.setMinBalance(-10000.00);
        client.getAccount().add( a2 );

        // Using JAXB we marshal the client to XML (in this case, just to a
        // string - we validate the output during marshalling)
        JAXBContext ctx = JAXBContext.newInstance("za.co.solms.example.accounts");
        Marshaller marshaller = ctx.createMarshaller();
        // Pretty-print output
        marshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, true );
        // Perform validation on marshalling using our schema
        marshaller.setSchema( getAccountsSchema() );
        // Marshal to string
        StringWriter writer = new StringWriter();
        marshaller.marshal( new ObjectFactory().createClient(client), writer );

        // Display output
        System.out.println( writer.getBuffer() );
    }

    /** Gets the accounts Schema */
    private Schema getAccountsSchema()
    {
        try
        {
            SchemaFactory sf = SchemaFactory.newInstance( XMLConstants.W3C_XML_SCHEMA_NS_URI );
            return sf.newSchema( getClass().getClassLoader().getResource("schemas/accounts.xsd") );
        }
        catch (SAXException e)
        {
            throw new RuntimeException("Failed to read accounts schema", e);
        }
    }
}
```

{}

## 2.7.4 Java-5 POM

JAXB is not included in Java 5 and we hence need to add the appropriate dependencies and repositories:

```
<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/maven-4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>za.co.solms.example</groupId>
  <artifactId>jaxb-maven-sample-java5</artifactId>
  <packaging>jar</packaging>
  <version>0.1</version>

  <name>JAXB / Maven Sample (Java 5)</name>

  <description>
    A Sample Maven 2.x project that illustrates usage of the
    JAXB Maven plugin to compile XML Schema resources to Java.
    This is suitable for isolated work on XML documents. For
    web services, usage of the JAX-WS is recommended.

    In Java SE 5, JAXB is not part of the platform, so we need
    to explicitly construct a run-time dependency on the
    Reference Implementation of JAXB.
  </description>

  <developers>
    <developer>
      <organization>Solms TCD</organization>
      <organizationUrl>http://www.solms.co.za/</organizationUrl>
      <email>info@solms.co.za</email>
    </developer>
  </developers>

  <build>
    <plugins>
      <!-- Configuration to compile all schemas in the
          resources/schemas directory. Automatically
          invoke during the 'generate-sources' phase -->
      <plugin>
        <groupId>org.jvnet.jaxb2.maven2</groupId>
        <artifactId>maven-jaxb2-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <schemaDirectory>src/main/resources/schemas</schemaDirectory>
          <schemaIncludes>
            <include>*.xsd</include>
          </schemaIncludes>
        </configuration>
      </plugin>
    </plugins>
  </build>

```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>

</plugins>
</build>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>[4.1,]</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>[2.1,]</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>maven2-repository.dev.java.net</id>
    <name>Java.net Maven 2 Repository</name>
    <url>http://download.java.net/maven/2</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>maven2-repository.dev.java.net</id>
    <url>http://download.java.net/maven/2</url>
  </pluginRepository>
</pluginRepositories>

</project>
```

## 2.7.5 Java-6 POM

JAXB is included in Java 6:

```
<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/maven- ↵
  v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>za.co.solms.example</groupId>
  <artifactId>jaxb-maven-sample-java6</artifactId>
  <packaging>jar</packaging>
  <version>0.1</version>

  <name>JAXB / Maven Sample (Java 6+)</name>
```

```
<description>
  A Sample Maven 2.x project that illustrates usage of the
  JAXB Maven plugin to compile XML Schema resources to Java.
  This is suitable for isolated work on XML documents. For
  web services, usage of the JAX-WS is recommended.
</description>

<developers>
  <developer>
    <organization>Solms TCD</organization>
    <organizationUrl>http://www.solms.co.za/</organizationUrl>
    <email>info@solms.co.za</email>
  </developer>
</developers>

<build>
  <plugins>
    <!-- Configuration to compile all schemas in the
        resources/schemas directory. Automatically
        invoke during the 'generate-sources' phase -->
    <plugin>
      <groupId>org.jvnet.jaxb2.maven2</groupId>
      <artifactId>maven-jaxb2-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>generate</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <schemaDirectory>src/main/resources/schemas</schemaDirectory>
        <schemaIncludes>
          <include>*.xsd</include>
        </schemaIncludes>
      </configuration>
    </plugin>
    <!-- Assume a Java SE 6 environment, which includes JAXB.
        For Java 5, extra dependencies on the JAXB Implementation
        should be specified -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>[4.1,]</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<repositories>
```

```
<repository>
  <id>maven2-repository.dev.java.net</id>
  <name>Java.net Maven 2 Repository</name>
  <url>http://download.java.net/maven/2</url>
</repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>maven2-repository.dev.java.net</id>
    <url>http://download.java.net/maven/2</url>
  </pluginRepository>
</pluginRepositories>

</project>
```

## 2.7.6 Executing goals

When running the tests via `mvn test`

- any dependencies are downloaded,
- the XJC compiler is used to compile the schema, generating Java binding classes,
- all generated and written Java classes are compiled,
- and the tests are executed.

Running

```
mvn site
```

generates documentation for the site showing

- the project description,
- the project dependencies,
- the test results,
- the project team,

and a range of other information.

## 2.8 Maven JAXWS Sample

### 2.8.1 Introduction

This is an example of a Maven script for a simple Java web client in the form of a unit test. It

- sources the web services contract (WSDL) from the service host,
- generates the Java adapter and binding classes (jaxws runs jaxb under the hood),
- compiles the test class, and
- executes the test.

## 2.8.2 Java-6 POM

```
<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/maven-4.0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>za.co.solms.example</groupId>
  <artifactId>jaxws-maven-sample-java6</artifactId>
  <packaging>jar</packaging>
  <version>0.1</version>

  <name>JAX-WS / Maven Sample (Java 6)</name>

  <description>
    A Sample Maven 2.x project that illustrates usage of the
    JAX-WS Maven plugin to compile and use web services
    from Java.
  </description>

  In Java SE 6, JAX-WS is part of the platform, so this project
  only requires the JAX-WS WSDL compiler plugin to do initial
  WSDL to Java compilation.
  </description>

  <developers>
    <developer>
      <organization>Solms TCD</organization>
      <organizationUrl>http://www.solms.co.za/</organizationUrl>
      <email>info@solms.co.za</email>
    </developer>
  </developers>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>wsimport</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <target>2.1</target>
          <wsdlUrls>
            <wsdlUrl>http://www.webservicex.net/CurrencyConvertor.asmx?WSDL</wsdlUrl>
          </wsdlUrls>
          <sourceDestDir>target/generated-sources</sourceDestDir>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>com.sun.xml.ws</groupId>
      <artifactId>jaxws-tools</artifactId>
      <version>2.1.7</version>
    </dependency>
  </dependencies>
</project>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
    <source>1.6</source>
    <target>1.6</target>
</configuration>
</plugin>

</plugins>
</build>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>[4.1,]</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<repositories>
    <repository>
        <id>maven2-repository.dev.java.net</id>
        <name>Java.net Maven 2 Repository</name>
        <url>http://download.java.net/maven/2</url>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>maven2-repository.dev.java.net</id>
        <url>http://download.java.net/maven/2</url>
    </pluginRepository>
</pluginRepositories>
</project>
```

### 2.8.3 The Test Application

```
package za.co.solms.example;

import net.webservicex.Currency;
import net.webservicex.CurrencyConvertor;
import net.webservicex.CurrencyConvertorSoap;
import org.junit.Test;

public class JAXWSTest
{
    @Test
    public void testCurrency()
    {
        Currency from = Currency.USD;
        Currency to = Currency.ZAR;

        System.out.printf("Getting conversion rate from %s to %s...\n", from, to);
        CurrencyConvertorSoap cc = new CurrencyConvertor().getCurrencyConvertorSoap();

        double rate = cc.conversionRate(from, to);
```

```
        System.out.printf("From web service: Conversion rate from %s to %s is %f\n", from, to, ←
            rate);
    }
}
```

## 2.8.4 Executing the web service test

Running

```
mvn test
```

sources the WSDL, compiles the adapter and binding classes as well as the test and then runs the test:

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building JAX-WS / Maven Sample (Java 6)
[INFO]   task-segment: [test]
[INFO] -----
[INFO] [jaxws:wsimport {execution: default}]
[INFO] Processing: http://www.webservicex.net/CurrencyConvertor.asmx?WSDL
[INFO] jaxws:wsimport args: [-s, /home/fritz/solmsRoot/resources/information/technologies/ ←
  java/tools/maven/jaxwsSamples/jaxws-sample-java6/target/generated-sources, -d, /home/ ←
  fritz/solmsRoot/resources/information/technologies/java/tools/maven/jaxwsSamples/jaxws- ←
  sample-java6/target/classes, -target, 2.1, -Xnocompile, http://www.webservicex.net/ ←
  CurrencyConvertor.asmx?WSDL]
parsing WSDL...

[WARNING] Ignoring SOAP port "CurrencyConvertorSoap12": it uses non-standard SOAP 1.2 ←
  binding.
You must specify the "-extension" option to use this binding.
  line 271 of http://www.webservicex.net/CurrencyConvertor.asmx?WSDL

[WARNING] ignoring port "CurrencyConvertorHttpGet": no SOAP address specified. try running ←
  wsimport with -extension switch.
  line 274 of http://www.webservicex.net/CurrencyConvertor.asmx?WSDL

[WARNING] ignoring port "CurrencyConvertorHttpPost": no SOAP address specified. try running ←
  wsimport with -extension switch.
  line 277 of http://www.webservicex.net/CurrencyConvertor.asmx?WSDL

generating code...

[INFO] [resources:resources]
[WARNING] Using platform encoding (ANSI_X3.4-1968 actually) to copy filtered resources, i.e. ←
  . build is platform dependent!
[INFO] skip non existing resourceDirectory /home/fritz/solmsRoot/resources/information/ ←
  technologies/java/tools/maven/jaxwsSamples/jaxws-sample-java6/src/main/resources
[INFO] [compiler:compile]
[INFO] Compiling 7 source files to /home/fritz/solmsRoot/resources/information/technologies/ ←
  /java/tools/maven/jaxwsSamples/jaxws-sample-java6/target/classes
[INFO] [resources:testResources]
[WARNING] Using platform encoding (ANSI_X3.4-1968 actually) to copy filtered resources, i.e. ←
  . build is platform dependent!
[INFO] skip non existing resourceDirectory /home/fritz/solmsRoot/resources/information/ ←
  technologies/java/tools/maven/jaxwsSamples/jaxws-sample-java6/src/test/resources
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
[INFO] Surefire report directory: /home/fritz/solmsRoot/resources/information/technologies/ ←
  java/tools/maven/jaxwsSamples/jaxws-sample-java6/target/surefire-reports
```

```
-----
T E S T S
-----
Running za.co.solms.example.JAXWSTest
Getting conversion rate from USD to ZAR...
From web service: Conversion rate from USD to ZAR is 7.431000
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.544 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 11 seconds
[INFO] Finished at: Thu Dec 17 08:02:34 SAST 2009
[INFO] Final Memory: 12M/80M
[INFO] -----
```

# Chapter 3

## A simple JavaEE-6 example: WeatherBuro

### 3.1 Overview

The purpose of this simple example is to provide an overview of the typical elements of a Java EE 6 enterprise application, showing the purpose of each element as well as how the elements tie up with one another. It also emphasizes the spirit of decoupling and clean layering as well as component reuse across the various layers.

#### 3.1.1 Core elements of the example

The example is a simple example which aims to contain many of the common elements of typical enterprise systems. In particular it demonstrates

- Maven based build and testing environment.
- A typical 5 tired architecture containing
  1. A relational database for persistence using JPA-based object-relational mapping for the persistence onto the database using JPQL as an object-oriented query language.
  2. A domain objects layer containing the object representation of the business entities.
  3. An EJB-based services layer making use of the application server's resource management including thread, connection and object pooling, transaction management and security.
  4. A JSF 2-based integration layer to the web-based presentation layer.
  5. A component-based JSF-2 Facelets layer with embedded Ajax calls.
- Unit testing using an embedded container and an embedded database.

### 3.2 Environment setup

You will have to setup a database as well as the application server itself. Most application servers (e.g. Glassfish, Apache Geronimo, JBoss, ...) are these days downloaded as a zip file which is simply unzipped. You will typically create a configuration for your system. This is done in Glassfish via

```
./asadmin create-domain <your-domain>
```

You may have to setup a database of your choice or at least setup the connectivity for the database. To this end you will have to copy the relevant JDBC driver into the `lib` directory for your application server and then setup the database descriptor which contains information on where the database is running and the user id and password which the application server should use when connecting to the database. In addition you will have to configure a connection pool for the database. Both of the latter can be done either through the application server's management console or by editing the relevant configuration files.

### 3.2.1 Database server

We chose *Postgresql* as an example of a very solid implementation of an open-source database. Postgres is known for its excellent performance, stability and maturity as well as its true open licensing.

---

#### Alternative: Use Derby database

For development you will often want to simply use the Derby database which ships with Glassfish. The database is started via

```
asadmin start-database
```

In order to inspect the database created via the object-relational mapping you can use

- a minimalistic front-end like `i j` which is packaged with Glassfish (`glassfishv3/javadb/bin`),
  - a generic front-end like the Java based Squirrel database front end,
  - a front-end like SQLExplorer which integrates into your IDE (e.g. into Eclipse).
- 

#### 3.2.1.1 Installation

Install your database as per instructions for your operating system.

##### 3.2.1.1.1 Installing PostgreSQL on Gentoo Linux

On Gentoo Linux the installation is particularly simple. You download and install the software from source via

```
emerge dev-db/postgresql-server
```

and configure the database server via

```
emerge --config postgresql-server
```

Typically you can use the default port as well as the standard database and configuration directory. The databases will be stored in `/var/lib/postgresql`.

The database server is started via

```
/etc/init.d/postgresql start
```

Normally you would want to start the database automatically upon system startup. This is done by adding the start script to the default run-level via

```
rc-update add postgresql default
```

#### 3.2.1.2 Creating the database, users and permissions

You can create a database from the PostgreSQL user interface, `psql` via

```
CREATE DATABASE <appServerDB>
```

The `psql` interface should be run from the `postgres` user:

```
su - postgres
psql
```

### 3.2.1.2.1 Creating the system user

Your application server will maintain a connection pool of connections established from the system. To this end you want to create a system user. This is done on PostgreSQL via

```
CREATE USER <appServerUsername> ENCRYPTED PASSWORD '<appServerPassword>' NOCREATEDB ←  
NOCREATEUSER;
```

You can list the users via

```
select * from pg_shadow;
```

and alter the user password via

```
alter user <appServerUsername> with password '[newPassword]';
```

The sbss user was given full permission on the sbss database via

```
GRANT ALL PRIVILEGES ON DATABASE <applicationDB> to <appServerUsername>;
```

## 3.2.2 Application server

Glassfish was chosen for the application server because

- it is the reference implementation for many of the Java EE standards, and
- it has good support for the latest Java EE standards, in particular Facelets and JSF-2.

Whatever we develop is, however, developed in an application server neutral way, using, wherever possible only public standards.

### 3.2.2.1 Installing the Glassfish JavaEE application server

Simply download the latest Glassfish application server and unzip the zip file into your directory of choice.

### 3.2.2.2 Creating a domain for your system domain

A sbss domain was created for the sbss system via

```
./asadmin create-domain --instanceport 80 <domainName>
```

setting the username and password for the admin and master users.

The admin port would be running on the default port, 4848.

### 3.2.2.3 Setting up the database connection pool and JDBC resource

The JDBC driver for your database needs to be copied into the `lib` directory of the application server (Note: not the `lib` directory of your domain). If you use PostgreSQL, then the driver can be obtained from <http://jdbc.postgresql.org/download.html>.

You will need to restart the application server for the new driver to be loaded. You can then go to the admin console to create the connection pool:

```
http://localhost:4848
```

Under *Resources -> JDBC -> Connection Pools* create a new connection pool with pool name sbssConnectionPool, resource type javax.sql.DataSource and database vendor Postgresql. On the next screen under additional properties one populates the server and database name, database user and password using

```
servername: localhost
database name: <yourDbName>
```

and the database user authentication credentials (username and password). This adds the following JDBC connection pool descriptor to the JDBC connection pool resources of the config/domain.xml domain configuration file:

```
<jdbc-connection-pool datasource-classname="org.postgresql.ds.PGSimpleDataSource"
    res-type="javax.sql.DataSource"
    name="sbssConnectionPool">
    <property name="DatabaseName" value="<yourDbName>" />
    <property name="LoginTimeout" value="0" />
    <property name="Password" value="<dbUserPassword>" />
    <property name="PortNumber" value="0" />
    <property name="PrepareThreshold" value="5" />
    <property name="ServerName" value="localhost" />
    <property name="SocketTimeout" value="0" />
    <property name="Ssl" value="false" />
    <property name="TcpKeepAlive" value="false" />
    <property name="UnknownLength" value="2147483647" />
    <property name="User" value="<dbUser>" />
</jdbc-connection-pool>
```

You can use the *ping* facility to see whether the connection pool is working

Finally we have to create the JDBC resource for the connection pool by going to *Resources -> JDBC Resources*. Select the connection pool and assign yourJdbcResource as JNDI name. This adds the JNDI the corresponding *JDBCResource* in the resources section of the domain.xml domain descriptor

```
<jdbc-resource pool-name="sbssConnectionPool" jndi-name="yourJdbcResource" />
```

as well as a resource reference in the server descriptor of the servers section:

```
<servers>
    <server name="server" config-ref="server-config">
        <application-ref ref="__admingui" virtual-servers="__asadmin" />
        ...
        <resource-ref ref="yourJdbcResource" />
    </server>
</servers>
```

### 3.2.2.4 Setting up a JDBC security realm for container based authentication

Typically you would want to configure your enterprise application such that the application server will request authentication as soon as a user tries to access a protected resource, e.g. a service which should only be accessible by users who have certain security roles assigned to them.

To this end the application server will have to compare the provided authentication credentials (e.g. username and password) against those stored either in your system or in some other organization wide user repository.

In JavaEE you would typically configure a security realm for your system. In Glassfish this can be done from the administration console by selecting

```
Configuration -> Security -> Realms
```

and selecting to add a new security realm. For a JDBC realm which authenticates users against a relational database containing usernames, passwords and roles you will have to specify

- `jdbcRealm as JAASContext,`
- a JNDI name,
- the userstable name and the column names containing the usernames and passwords,
- the group table and the column name containing the group or role names (this table must also contain a username column which is a foreign key to the user table), and
- the digest algorithm (use `none` if you store the passwords non-encrypted in your database).

### 3.3 The build and testing environment

We chose Apache Maven to manage our build and testing environment. To this end we create a main project with two sub-modules for the business logic (domain object and services layers) and the web front-end (facelets view and JSF binding layers). The sub-modules are separate child projects which are compiled and packaged separately. Unit testing is done against the services layer, i.e. in the business logic module.

#### 3.3.1 The parent project

The parent POM specified

- the project identifier (`groupId`, `artifactId` and `version`),
- the `businessLogic` and `web` modules for the project,
- a selection of Maven repositories where the dependencies are sourced from including those required for Glassfish,
- since this is a JavaEE project, a dependency on the JavaEE6 API, and
- a requirement that the project is to be compiled against Java 6:

```
<?xml version="1.0" encoding="UTF-8"?>
<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/maven- -->
  v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>za.co.solms.ejbCourse</groupId>
  <artifactId>weatherBuro</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>

  <name>Parent POM for WeatherBuro system</name>
  <url>http://www.solms.co.za</url>

  <modules>
    <module>weatherBuro-businessLogic</module>
    <module>weatherBuro-web</module>
  </modules>

  <repositories>
    <repository>
      <id>maven-repo</id>
      <name>Maven repository</name>
      <url>http://repo1.maven.org/maven2/</url>
    </repository>
  </repositories>
```

```
<repository>
    <id>java.net</id>
    <url>http://download.java.net/maven</url>
</repository>

<repository>
    <id>Glassfish</id>
    <name>Glassfish repository</name>
    <url>http://download.java.net/maven/glassfish/</url>
</repository>
</repositories>

<dependencies>

    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>6.0</version>
        <scope>provided</scope>
    </dependency>

</dependencies>

<build>
    <!-- Tell Maven to compile for Java 6 -->
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>
```

### 3.3.2 The business logic module

The business logic module contains the domain objects and session-bean based services layers as well as the unit testing for the business services.

The POM specifies

- the module artifact id and packaging,
- the parent project,
- additional dependencies on *Embedded Glassfish* and *JUnit*, both with `test` scope as both are required for unit testing the business logic layer, and
- a specification requesting that the module should be built against the EJB 3.1 specification.

In addition, we have a fudge which uses the Apache Ant plugin to copy the persistence descriptors. This is required to make certain that we are using the persistence descriptor for the embedded Derby database during unit testing and to copy the persistence descriptor for the production system which is run against a PostgreSQL server back after unit testing:

```
<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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.ejbCourse</groupId>
    <artifactId>weatherBuro-businessLogic</artifactId>
    <packaging>ejb</packaging>

    <description>Business Logic layer with setup for out of Container Testing of EJB 3.0</description>

    <parent>
        <groupId>za.co.solms.ejbCourse</groupId>
        <artifactId>weatherBuro</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <dependencies>

        <dependency>
            <groupId>org.glassfish.extras</groupId>
            <artifactId>glassfish-embedded-all</artifactId>
            <version>3.0</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>[4.1,]</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Tell Maven to build for EJB3.1 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-ejb-plugin</artifactId>
                <configuration>
                    <ejbVersion>3.1</ejbVersion>
                </configuration>
            </plugin>

            <plugin>
                <artifactId>maven-antrun-plugin</artifactId>
                <version>1.3</version>
                <executions>
                    <execution>
                        <id>copy-test-persistence</id>
                        <phase>process-test-resources</phase>
                        <configuration>
                            <tasks>
                                <echo message="Copying ${project.build.outputDirectory}/META-INF/persistence.xml to ${project.build.outputDirectory}/META-INF/persistenceDeploy.xml"/>
                                <move file="${project.build.outputDirectory}/META-INF/persistence.xml" tofile="${project.build.outputDirectory}/META-INF/persistenceDeploy.xml" />
                            </tasks>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

```

```
<echo message="Copying ${project.build.testOutputDirectory}/ ↔
    META-INF/persistence.xml to ${project.build. ↔
        outputDirectory}/META-INF/persistence.xml" />
<copy file="${project.build.testOutputDirectory}/META-INF/ ↔
    persistence.xml" tofile="${project.build.outputDirectory ↔
        }/META-INF/persistence.xml" />
</tasks>
</configuration>
<goals>
    <goal>run</goal>
</goals>
</execution>
<execution>
    <id>restore-persistence</id>
    <phase>prepare-package</phase>
    <configuration>
        <tasks>
            <echo message ←
                ="#####">
<delete file="${project.build.outputDirectory}/META-INF/ ↔
    persistence.xml" />
<echo message="Copying ${project.build.outputDirectory}/META- ↔
    INF/persistenceDeploy.xml to ${project.build. ↔
        outputDirectory}/META-INF/persistence.xml" />" />
<move file="${project.build.outputDirectory}/META-INF/ ↔
    persistenceDeploy.xml" tofile="${project.build. ↔
        outputDirectory}/META-INF/persistence.xml" />
        </tasks>
    </configuration>
    <goals>
        <goal>run</goal>
    </goals>
    </execution>
    </executions>
</plugin>
</plugins>
</build>

</project>
```

### 3.3.3 The web module

The web module contains the Facelets based view layer as well as a JSF-2 based binding layer. We are also making use of the *PrimeFaces* facelets tag library for certain more snazzy UI components.

The POM specifies

- the module artifact id and packaging,
- the parent project,
- a dependency on the business logic layer module, and
- the repository where the *PrimeFaces* tag library is sourced from as well as a dependency on the tag library.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.ejbCourse</groupId>
    <artifactId>weatherBuro-web</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>A web front-end for the WeatherBuro system</name>

    <parent>
        <groupId>za.co.solms.ejbCourse</groupId>
        <artifactId>weatherBuro</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <url>http://www.solms.co.za</url>

    <repositories>
        <!-- For PrimeFaces Tag repository -->
        <repository>
            <id>prime-repo</id>
            <url>http://repository.prime.com.tr</url>
            <layout>default</layout>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>za.co.solms.ejbCourse</groupId>
            <artifactId>weatherBuro-businessLogic</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

        <dependency>
            <groupId>org.primefaces</groupId>
            <artifactId>primefaces</artifactId>
            <version>2.0.0</version>
        </dependency>
    </dependencies>
</project>
```

## 3.4 The domain objects layer

The domain objects layer contains the business entities which need to be persisted to the database. In JavaEE this is commonly done using JPA based object-relational mappers like *EclipseLink* or *Hibernate*. All development and querying is done in the object domain (and not in the relational domain).

### 3.4.1 The domain objects model

Figure Figure 3.1 shows a UML class diagram the domain model of our simple weather buro system.

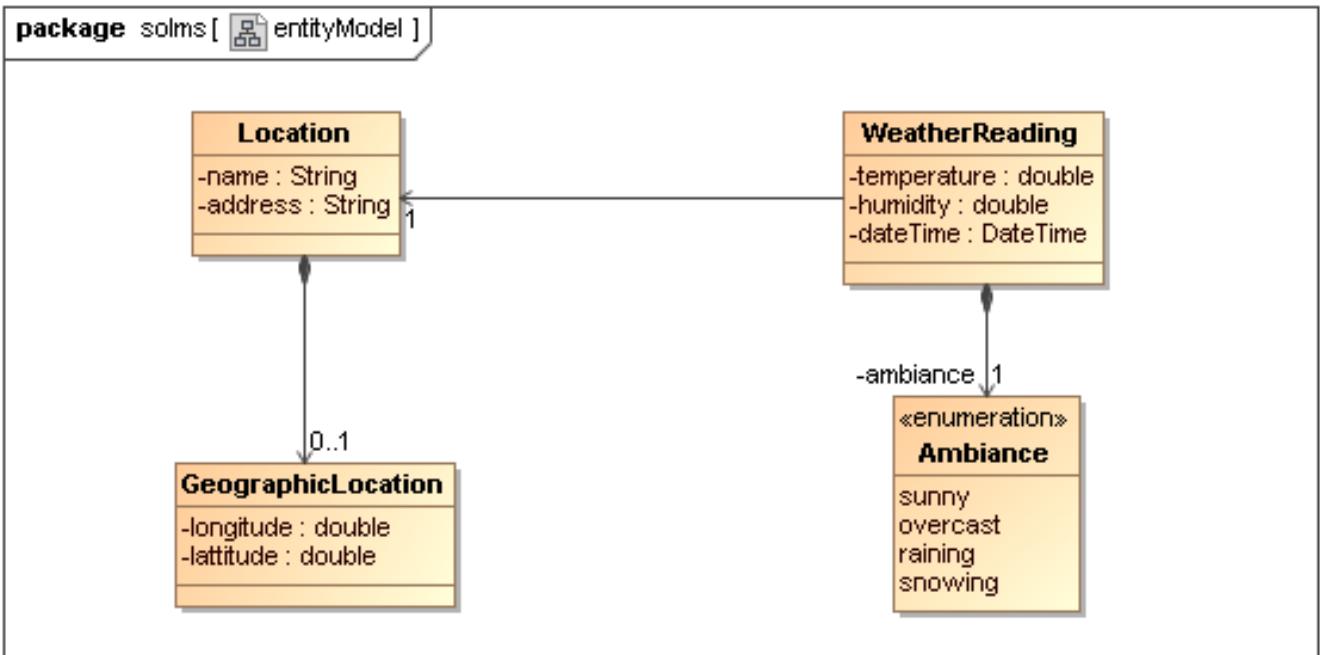


Figure 3.1: A UML class diagram for the weather buro domain model

We have locations which have a name, an address and geographic coordinates which belong to the location. Weather readings are associated with a location and have a date, the temperature, humidity and ambiance which is an enumeration which may have the values sunny, cloudy, raining or snowing.

### 3.4.2 Implementation mapping of the domain objects layer

Classes in the domain/entity model are mapped onto either Java entities or onto embedded classes, i.e. classes whose state is embedded within the persistence unit (e.g. table) of the owning entity - embedded classes should only be used for the component side of UML composition relationships. Note that all entity classes have been declared serializable and hence that the entities can be detached and used as value objects. At a later stage they can be merged back into a managed state

#### 3.4.2.1 Geographic coordinates

The geographic coordinates can be mapped onto either

- an entity which is persisted in a separate table, or onto
- an embedded class whose fields are embedded as columns in the table of the owner class.

We chose to model Geographic Coordinates as a separate entity (one can make a strong argument for using an embedded class instead). The class is annotated as an entity. A domain-neutral object identifier, id, has been added and annotated as an @Id which should be auto-generated by the application server:

```

/**
 *
 */
package za.co.solms.location;

import java.io.Serializable;
  
```

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

/**
 * Represents a position on earth in the form of degrees longitude and latitude.
 *
 * @author fritz@solms.co.za
 */
@Entity
public class GeographicCoordinates implements Serializable
{
    public GeographicCoordinates() {}

    public GeographicCoordinates(double longitude, double latitude)
    {
        this.longitude = longitude;
        this.latitude = latitude;
    }

    @Id
    @GeneratedValue
    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
        this.id = id;
    }

    public double getLongitude()
    {
        return longitude;
    }

    public void setLongitude(double longitude)
    {
        this.longitude = longitude;
    }

    public double getLatitude()
    {
        return latitude;
    }

    public void setLatitude(double latitude)
    {
        this.latitude = latitude;
    }

    public boolean equals(Object o)
    {
        try
        {
            GeographicCoordinates arg = (GeographicCoordinates)o;
            return (this.latitude == arg.latitude)
                && (this.longitude == arg.longitude);
        }
        catch (ClassCastException e)
    }
}
```

```
    {
        return false;
    }
}

public int hashCode()
{
    return new Double(longitude).hashCode() + new Double(latitude).hashCode();
}

private int id;
private double longitude, latitude;

private static final long serialVersionUID = 1L;
}
```

### 3.4.2.2 Locations

Locations are entities which contain geographic coordinates via composition. We chose to map this onto a one-to-one unidirectional entity relationship (which can be queried from only the owner, i.e. the location side) and have requested all-cascading which includes cascading merges and deletes. This ensures that when a Location is deleted or updated, its geographic location is deleted or updated as well (in composition the component may not survive the owner). This has been done via the @OneToOne annotation on the getter for the geographic coordinates.

```
package za.co.solms.location;

import java.io.Serializable;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistenceNamedQuery;
import javax.persistence.OneToOne;

/**
 * Represents a location with a name, address and geographic location.
 *
 * @author fritz@solms.co.za
 */
@Entity
@NamedQueries({
    @NamedQuery(name="findAllLocations",
        query="Select l from Location l"),
    @NamedQuery(name="findLocationByName",
        query="select l from Location l where l.name = :locationName")
})
public class Location implements Serializable
{
    public Location() {}

    public Location(String name, String address, GeographicCoordinates geographicCoordinates)
    {
        this.name = name;
        this.address = address;
        this.geographicCoordinates = geographicCoordinates;
    }
}
```

```
}

@Id
@GeneratedValue
public int getId()
{
    return id;
}

public void setId(int id)
{
    this.id = id;
}

@Column(unique=true, nullable=false)
public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}

public String getAddress()
{
    return address;
}

public void setAddress(String address)
{
    this.address = address;
}

@OneToOne(cascade=CascadeType.ALL)
public GeographicCoordinates getGeographicLocation()
{
    return geographicCoordinates;
}

public void setGeographicLocation(GeographicCoordinates geographicCoordinates)
{
    this.geographicCoordinates = geographicCoordinates;
}

public boolean equals(Object o)
{
    try
    {
        Location arg = (Location)o;
        return this.getName().equals(arg.getName())
            && this.getAddress().equals(arg.getAddress())
            && this.getGeographicLocation().equals(arg.getGeographicLocation());
    }
    catch (ClassCastException e)
    {
        return false;
    }
}

public int hashCode()
```

```
{  
    return name.hashCode() + address.hashCode() + geographicCoordinates.hashCode();  
}  
  
public String toString() {return name;}  
  
private int id;  
private String name, address;  
private GeographicCoordinates geographicCoordinates;  
  
private static final long serialVersionUID = 1L;  
}
```

We did not want to make a domain attribute like the location name the primary key, even though business required that location names need to be unique - after all, this business rule could change. So we added a domain-neutral object identifier and assigned `unique=true` and `nullable=false` constraints to the name column.

In addition this class defines some pre-compiled JPA queries for this entity. These queries are available to the services layer (e.g. session beans) to find appropriate objects using the entity manager to create and execute the queries. Here we have two simple queries finding all locations and the locations with a specified name respectively. We could add queries to find all locations within a particular geographic area or within a certain radius from certain other locations.

### 3.4.2.3 Weather readings

Weather readings are associated via a many-to-one relationship to locations. However, since this is an association and not a composition relationship, we do not specify and cascading.

```
/**  
 *  
 */  
package za.co.solms.weather;  
  
import java.io.Serializable;  
import java.util.Date;  
  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
import javax.persistence.JoinColumn;  
import javax.persistence.ManyToOne;  
import javax.persistence.NamedQueries;  
import javax.persistence.NamedQuery;  
import javax.persistence.Temporal;  
import javax.persistence.TemporalType;  
  
import za.co.solms.location.Location;  
  
/**  
 * @author fritz@solms.co.za  
 *  
 */  
@Entity  
@NamedQueries({  
    @NamedQuery(name="findAllWeatherReadings",  
        query="Select wr from WeatherReading wr"),  
    @NamedQuery(name="findAllWeatherReadingsForLocation",  
        query="select wr from WeatherReading wr where wr.location = :location"),  
    @NamedQuery(name="findAllWeatherReadingsForLocationAndPeriod",  
        query="select wr from WeatherReading wr where wr.location = :location")  
})
```

```
        + " and wr.dateTime > :after and wr.dateTime <= :onOrBefore")  
    })  
  
public class WeatherReading implements Serializable  
{  
    public WeatherReading() {}  
  
    public WeatherReading(Location location, Date dateTime,  
                          double temperature, double humidity, Ambiance ambiance)  
    {  
        this.location = location;  
        this.dateTime = dateTime;  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.ambiance = ambiance;  
    }  
  
    @Id  
    @GeneratedValue  
    public int getId()  
    {  
        return id;  
    }  
  
    public void setId(int id)  
    {  
        this.id = id;  
    }  
  
    public double getTemperature()  
    {  
        return temperature;  
    }  
  
    public void setTemperature(double temperature)  
    {  
        this.temperature = temperature;  
    }  
  
    public double getHumidity()  
    {  
        return humidity;  
    }  
  
    public void setHumidity(double humidity)  
    {  
        this.humidity = humidity;  
    }  
  
    @Temporal(TemporalType.TIMESTAMP)  
    @Column(nullable=false)  
    public Date getDateTIme()  
    {  
        return dateTime;  
    }  
  
    public void setDateTIme(Date dateTime)  
    {  
        this.dateTime = dateTime;  
    }  
  
    public Ambiance getAmbiance()
```

```
{  
    return ambiance;  
}  
  
public void setAmbiance(Ambiance ambiance)  
{  
    this.ambiance = ambiance;  
}  
  
@ManyToOne  
@JoinColumn(nullable=false)  
public Location getLocation()  
{  
    return location;  
}  
  
public void setLocation(Location location)  
{  
    this.location = location;  
}  
  
public boolean equals(Object o)  
{  
    try  
    {  
        WeatherReading arg = (WeatherReading)o;  
        return (this.id == arg.id) && (this.temperature == arg.temperature)  
            && (this.humidity == arg.humidity) && (this.ambiance == arg.ambiance);  
    }  
    catch (ClassCastException e)  
    {  
        return false;  
    }  
}  
  
public int hashCode()  
{  
    return id + dateTime.hashCode() + new Double(temperature).hashCode()  
        + new Double(humidity).hashCode();  
}  
  
private int id;  
private double temperature, humidity;  
private Date dateTime;  
private Ambiance ambiance;  
private Location location;  
  
private static final long serialVersionUID = 1L;  
}
```

Each weather reading has a date. The date could be persisted onto a SQL date which has no time information or the full date/time could be persisted. We require the latter and specify this via `@Temporal(TemporalType.TIMESTAMP)` annotation. In addition, each reading requires a date/time stamp and hence we specify that the corresponding column may not be null via a `@Column(nullable=false)` annotation.

Each reading has an ambiance which is an enumeration:

```
package za.co.solms.weather;  
  
/**  
 * An enumeration for accepted ambiance types.  
 */
```

```
/*
 * @author fritz@solms.co.za
 */
public enum Ambiance
{
    sunny,
    overcast,
    raining,
    snowing
}
```

The enumeration is embedded by the object relational mapper.

Finally we define some pre-compiled queries for the entity which will be used by the services layer. For example

```
@NamedQuery(name="findAllWeatherReadingsForLocationAndPeriod",
    query="select wr from WeatherReading wr where wr.location = :location"
    + " and wr.dateTime > :after and wr.dateTime <= :onOrBefore")
```

finds all weather readings for a specified location which are after some provided after date and on or before some provided onOrBefore date.

---

**Note**

These queries are object-oriented queries which traverse the object graph. They are mapped by the chosen object-relational mapper onto the appropriate query for the chosen database (e.g. onto the appropriate flavour of SQL).

---

### 3.4.3 The persistence descriptor

The persistence descriptor is stored in the `src/main/resources/META-INF` directory. The file has the name `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0" 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_1_0.xsd">
    <persistence-unit name="persistence_ejbCourse">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <jta-data-source>jdbc/ejbCourse</jta-data-source>
        <non-jta-data-source>jdbc/ejbCourse</non-jta-data-source>
        <properties>
            <!-- drop-and-create-tables -->
            <property name="eclipselink.ddl-generation" value="create-tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

The file specifies which JDBC resource to use - we chose the JDBC resource which we defined for our connection pool to our database as well as the object-relational mapper (in our case EclipseLink) and any specific attributes around the persistence mapping (we chose the tables to be created by the O/R mapper, but not that they should be deleted when the application is undeployed).

## 3.5 The services layer

The domain objects are not directly accessed - they are accessible only through a services layer. The services layer is provided by two stateless session beans which provide services around locations and weather readings respectively. Each implements an appropriate contract encoded as a Java interface.

### 3.5.1 Location services

This services contract defines a set of persistence and lookup services as well as the exceptions used to specify that a service cannot be provided due to some pre-condition not having been met:

```
package za.co.solms.location;

import java.util.List;

/**
 * The contract for the location services to be provided.
 *
 * @author fritz@solms.co.za
 */
public interface LocationServices
{
    /**
     * Persists the provided location in the database.
     *
     * @param location the location to be persisted
     * @return the entity object for the persisted location with its generated id.
     * @throws LocationNameInUseException if there is already a location in persistent
     * storage which has the same name as the provided location.
     */
    public Location persistLocation(Location location) throws LocationNameInUseException;

    /**
     * Removes the provided location from persistent storage.
     *
     * @param location the location to be deleted from storage.
     */
    public void removeLocation(Location location);

    /**
     * Removes all locations from persistent storage.
     */
    public void removeAllLocations();

    /**
     * Provide a list of all locations stored in the database.
     *
     * @return a list of all locations.
     */
    public List<Location> getAllLocations();

    /**
     * Returns the location with the specified name. Since names
     * are enforced to be unique there will be at most one matching
     * location.
     *
     * @return the location with the specified name
     * @throws NoSuchLocationException if there is no location with the specified name.
     */
}
```

```
public Location getLocationByName(String locationName) throws NoSuchLocationException;  
  
/**  
 * This exception is used to notify a requester of a service that the service is  
 * not going to be provided because a specified location did not exist.  
 *  
 * @author fritz@solms.co.za  
 *  
 */  
public class NoSuchLocationException extends Exception  
{  
    private static final long serialVersionUID = 1L;  
}  
  
/**  
 * Thrown if a service cannot be provided due to a particular location  
 * existing already.  
 *  
 * @author fritz@solms.co.za  
 *  
 */  
public class LocationExistsException extends Exception  
{  
    private static final long serialVersionUID = 1L;  
}  
  
/**  
 * This exception is thrown if the request service cannot be provided because  
 * a certain location name is already in use, i.e. there exists already a location  
 * with a particular name.  
 *  
 * @author fritz@solms.co.za  
 *  
 */  
public class LocationNameInUseException extends Exception  
{  
    private static final long serialVersionUID = 1L;  
}  
}
```

---

**Note**

Nothing in the services contract refers to EJBs or even JavaEE. The contract is architecture neutral.

---

### 3.5.2 LocationServicesBean

The implementation class has been annotated as `@Stateless`. We use the `@Local` annotation to specify that the `LocationServices` interface should be used as a local interface for the session bean. Our clients (including our web-based presentation layer) will have no direct dependencies on the bean implementation class - only on the services contract.

It makes liberal use of the entity manager to persist, remove, update/merge and find entities as well as to create and execute instances of the pre-compiled JPQL queries which we defined with the entity classes. The entity manager is injected into our session bean via a `@PersistenceContext` annotation.

```
/**  
 *  
 */  
package za.co.solms.location;
```

```
import java.util.List;
import java.util.logging.Logger;

import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

/**
 * A services implementation which will be manifested as a stateless session
 * bean when deployed in an EJB application server.
 *
 * @author fritz@solms.co.za
 */
@Local(LocationServices.class)
@Remote(LocationServices.class)
@Stateless
public class LocationServicesBean implements LocationServices
{

    /* (non-Javadoc)
     * @see solms.co.za.location.LocationServices#getAllLocations()
     */
    @SuppressWarnings("unchecked")
    @Override
    public List<Location> getAllLocations()
    {
        return entityManager.createNamedQuery("findAllLocations").getResultList();
    }

    /* (non-Javadoc)
     * @see solms.co.za.location.LocationServices#getLocationByName()
     */
    @SuppressWarnings("unchecked")
    @Override
    public Location getLocationByName(String locationName) throws NoSuchLocationException
    {
        Query query = entityManager.createNamedQuery("findLocationByName");
        query.setParameter("locationName", locationName);
        List<Location> matchingLocations = query.getResultList();
        assert matchingLocations.size()<2
            : "uniqueness constraint on location name seems not enforced";
        if (matchingLocations.size() == 0)
            throw new NoSuchLocationException();
        else
            return matchingLocations.get(0);
    }

    /* (non-Javadoc)
     * @see solms.co.za.location.LocationServices#persistLocation(solms.co.za.location. Location)
     */
    @Override
    public Location persistLocation(Location location) throws LocationNameInUseException
    {
        try
        {
            getLocationByName(location.getName());
        }
        catch (NoSuchLocationException e)
        {
            return location;
        }
        EntityManager em = getEntityManager();
        em.persist(location);
        return location;
    }
}
```

```
// if previous request does not throw an exception, the location name is used already ←
.
throw new LocationNameInUseException();
}
catch (NoSuchLocationException e)
{
    entityManager.persist(location);
    return location;
}
}

@Override
public void removeLocation(Location location)
{
    logger.info("***** Removing location " + location);
    entityManager.remove(entityManager.merge(location));
    logger.info("***** done removing location.");
}

@Override
public void removeAllLocations()
{
    for (Location location: this.getAllLocations())
        this.removeLocation(location);
}

@PersistenceContext
private EntityManager entityManager;

private static Logger logger = Logger.getLogger(LocationServicesBean.class.getName());
}
```

### 3.5.3 WeatherServices

The WeatherServices interface represents the contract publishing the weather services. This includes the standard entity creation, deletion, lookup and update services. In addition to this the contract also specifies a business service for calculating the average temperature over a specified period.

```
package za.co.solms.weather;

import java.util.Date;
import java.util.List;

import za.co.solms.location.Location;

/**
 *
 * @author fritz@solms.co.za
 *
 */
public interface WeatherServices
{
    /**
     * Returns the weather reading which has the specified id.
     * @param id the id of the required weather reading
     * @return the weather reading with the specified id.
     * @throws NoWeatherReadingsException if there is no weather reading with the
     *         specified id.
     */
}
```

```
public WeatherReading getWeatherReading(int id) throws NoWeatherReadingsException;

/**
 * This service stores the provided weather reading in persistent storage (a database).
 * The primary key will have been set for the returned entity.
 *
 * @param reading the value object which should be persisted.
 * @return the entity which has been persisted (with its primary key)
 */
public WeatherReading persistWeatherReading(WeatherReading reading);

/**
 * Returns a list of all weather readings for a specified location
 * which fall in a particular period.
 * @param location the location for which weather readings are requested
 * @param after the date-time after which readings are required
 * @param onOrBefore the date-time up to which readings are required.
 * @return the list of weather readings within the requested period
 * for the specified location
 *
 */
public List<WeatherReading> getWeatherReadingsForLocation(Location location,
    Date after, Date onOrBefore);

/**
 * Returns the average temperature for the specified location and specified
 * period. This is a simplistic average looking simply at the average of all
 * readings within the period.
 *
 * @param location the location for which the average temperature is required
 * @param after the start of the period for which the average temperature is required
 * @param onOrBefore the end of the period for which the average temperature is required
 * @return the average temperature for the specified location and period
 * @throws NoWeatherReadingsException if there are no weather readings for
 * the specified location over the specified period
 */
public double getAverageTemperature(Location location, Date after, Date onOrBefore)
    throws NoWeatherReadingsException;

/**
 * Returns a list of all weather stored weather readings.
 * @return a list of all weather readings contained in persistent storage.
 */
public List<WeatherReading> getAllWeatherReadings();

/**
 * Removes the provided weather reading from persistent storage.
 *
 * @param weatherReading
 */
public void removeWeatherReading(WeatherReading weatherReading);

/**
 * Removes all weather readings from persistent storage.
 */
public void removeAllWeatherReadings();

/**
 * An exception used to notify a client that the request service could not
 * be provided because there were no applicable weather readings.
 *
 * @author fritz@solms.co.za
```

```
/*
 */
public class NoWeatherReadingsException extends Exception
{
    private static final long serialVersionUID = 1L;
}

/**
 * Updates the persisted weather reading to the state of the provided value object.
 * @param weatherReading the weather reading to be updated in persistent storage
 * @return the updated weather reading entity.
 */
public WeatherReading updateWeatherReading(WeatherReading weatherReading);
}
```

### 3.5.4 WeatherServicesBean

The bean implementation class is similar to that of the location services. Here, however, we have some business logic for calculating the average temperature for a location over some period. This could have been done with a query, but is done here as some logic implemented in Java.

```
/**
 *
 */
package za.co.solms.weather;

import java.util.Collection;
import java.util.Date;
import java.util.List;

import javax.ejb.Local;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import za.co.solms.location.Location;

/**
 * @author fritz@solms.co.za
 */
@Local(WeatherServices.class)
@Stateless
public class WeatherServicesBean implements WeatherServices
{

    /* (non-Javadoc)
     * @see za.co.solms.weather.WeatherServices#getAverageTemperature(za.co.solms.location. ↴
     *      Location, java.util.Date, java.util.Date)
     */
    @Override
    public double getAverageTemperature(Location location, Date onOrBefore, Date after)
        throws NoWeatherReadingsException
    {
        Collection<WeatherReading> readings
            = this.getWeatherReadingsForLocation(location, onOrBefore, after);

        if (readings.isEmpty())
```

```
        throw new NoWeatherReadingsException();

    double sum = 0;
    for (WeatherReading weatherReading : readings)
        sum += weatherReading.getTemperature();
    return sum / readings.size();
}

/* (non-Javadoc)
 * @see za.co.solms.weather.WeatherServices#getWeatherReadingsForLocation(za.co.solms. ←
 *      location.Location, java.util.Date, java.util.Date)
 */
@Override
@SuppressWarnings("unchecked")
public List<WeatherReading> getWeatherReadingsForLocation(Location location,
    Date after, Date onOrBefore)
{
    Query query = entityManager.createNamedQuery("←
        findAllWeatherReadingsForLocationAndPeriod");
    query.setParameter("location", location);
    query.setParameter("after", after);
    query.setParameter("onOrBefore", onOrBefore);
    return query.getResultList();
}

/* (non-Javadoc)
 * @see za.co.solms.weather.WeatherServices#persistWeatherReading(za.co.solms.weather. ←
 *      WeatherReading)
 */
@Override
public WeatherReading persistWeatherReading(WeatherReading weatherReading)
{
    entityManager.persist(weatherReading);
    return weatherReading;
}

@Override
public WeatherReading updateWeatherReading(WeatherReading weatherReading)
{
    return entityManager.merge(weatherReading);
}

@Override
public void removeAllWeatherReadings()
{
    for (WeatherReading weatherReading: this.getAllWeatherReadings())
        this.removeWeatherReading(weatherReading);
}

@Override
public void removeWeatherReading(WeatherReading weatherReading)
{
    entityManager.remove(entityManager.merge(weatherReading));
}

@Override
@SuppressWarnings("unchecked")
public List<WeatherReading> getAllWeatherReadings()
{
    return entityManager.createNamedQuery("findAllWeatherReadings").getResultList();
}
```

```
@PersistenceContext  
EntityManager entityManager;  
  
@Override  
public WeatherReading getWeatherReading(int id)  
    throws NoWeatherReadingsException  
{  
    return entityManager.find(WeatherReading.class, id);  
}  
}
```

## 3.6 Unit testing

The services layer should be unit-tested. We use here JUnit as our unit testing framework and perform the testing in an embedded container and embedded database, both of which are running in the JavaSE virtual machine within which the test is executed.

---

**Note**

JavaEE-6 specifies the requirement for supporting embedded containers and JavaSE-6 specifies the requirement for supporting embedded databases and object-relational mappers.

---

In a Maven project the unit tests are, by default, stored in a `src/test/java` directory and all tests found here are automatically executed.

### 3.6.1 Initializing the embedded container and registering the enterprise beans

The embedded container needs to be initialized and the session beans need to be registered in the embedded container. For this we annotate a service with `@BeforeClass` which specifies that the service should be executed once when the test class is loaded:

```
@BeforeClass  
public static void initialize()  
{  
    context = EjbContainerUtility.getContainer().getContext();  
    lookupServiceBeans();  
}
```

This makes use of our `EjbContainerUtility` as well as of a separate service to load the beans. The container utility simply creates the `EJBContainer` if it has not yet been created:

```
package za.co.solms.javaee.glassfish.container.embedded;  
  
import java.util.logging.Logger;  
  
import javax.ejb.embeddable.EJBContainer;  
  
/**  
 * A little utility class which returns a handle to the embedded EJB container, starting ←  
 * the latter if it is not running.  
 *  
 * @author fritz@solms.co.za  
 */  
public class EjbContainerUtility
```

```

{
    /**
     * Preventing instantiation of this utility class.
     */
    private EjbContainerUtility() {}

    public static EJBContainer getContainer()
    {
        if (container == null)
        {
            logger.info("***** Starting embedded EJB container");
            container = EJBContainer.createEJBContainer();
        }
        else
            logger.info("***** Embedded EJB container already running - only returning handle.");

        return container;
    }

    private static EJBContainer container;

    private static Logger logger = Logger.getLogger(EjbContainerUtility.class.getName());
}

```

The `lookupServiceBeans()` service looks up the required beans using the global context and uses an assertion to check that the beans have indeed been found:

```

private static void lookupServiceBeans()
{
    try
    {
        locationServices = (LocationServices)context.lookup
            ("java:global/classes/LocationServicesBean");
        assertNotNull(locationServices);
    }
    catch (NamingException e)
    {
        fail("Could not lookup session bean " + e.toString());
    }
}

```

### 3.6.2 LocationServicesTest

The test classes have, in addition to the above, the various test services annotated as a JUnit test. A static import ensures that the static assertion and fail services do not have to be qualified with the class name:

```

package za.co.solms.location;

import javax.ejb.EJB;
import javax.naming.Context;
import javax.naming.NamingException;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import za.co.solms.javaee.glassfish.container.embedded.EjbContainerUtility;
import za.co.solms.location.LocationNameInUseException;

```

```
import za.co.solms.location.LocationServices.NoSuchLocationException;
import static junit.framework.Assert.*;



/**
 * A unit test for the location services.
 *
 * @author fritz@solms.co.za
 *
 */
public class LocationServicesTest
{
    @BeforeClass
    public static void initialize()
    {
        context = EjbContainerUtility.getContainer().getContext();
        lookupServiceBeans();
    }

    private static void lookupServiceBeans()
    {
        try
        {
            locationServices = (LocationServices)context.lookup(
                "java:global/classes/LocationServicesBean");
            assertNotNull(locationServices);
        }
        catch (NamingException e)
        {
            fail("Could not lookup session bean " + e.toString());
        }
    }

    @Before
    public void setupTestData()
    {
        location1 = new Location("Soccer City", "1a Pirates Ave\nSoweto",
            new GeographicCoordinates(29.8, 19.3));

        location2 = new Location("Gamadoelas CBD", "1 Krokodil Str\nGamadoelas",
            new GeographicCoordinates(26.1, 27.6));

        location3 = new Location("Soccer City", "15 Nyala Rd\nGamadoelas",
            new GeographicCoordinates(29.9, 19.5));
    }

    @After
    public void cleanupTestData()
    {
        locationServices.removeAllLocations();
    }

    @Test
    public void testPersistence()
    {
        int numLocations = locationServices.getAllLocations().size();

        try
        {
```

```
locationServices.persistLocation(location1);
locationServices.persistLocation(location2);
assertEquals(numLocations+2, locationServices.getAllLocations().size());

try
{
    Location location = locationServices.getLocationByName("Soccer City");
    assertEquals(location1, location);
}
catch (NoSuchLocationException e)
{
    fail("Could not retrieve persisted location");
}

catch (LocationNameInUseException e)
{
    fail("Location name already used????");
}
}

@Test
public void testNameUniqueness()
{
try
{
    locationServices.persistLocation(location1);
}
catch (LocationNameInUseException e)
{
    fail("Location name already used????");
}

try
{
    locationServices.persistLocation(location3);
    fail("Location name uniqueness not enforced.");
}
catch (LocationNameInUseException e)
{
}
}

@Test
public void testGetLocationByName()
{
try
{
    locationServices.persistLocation(location1);
}
catch (LocationNameInUseException e)
{
    fail("Location name already used????");
}

try
{
    assertEquals(location1, locationServices.getLocationByName("Soccer City"));
}
catch (NoSuchLocationException e)
{
    fail("Could not retrieve location by name.");
}
```

```
}

@EJB
private static LocationServices locationServices;

private Location location1, location2, location3;

private static Context context;
}
```

### 3.6.3 WeatherServicesTest

The weather services test is a little more involved as it requires both session beans and needs to construct and test more complex data objects and business services:

```
/***
 *
 */
package za.co.solms.weather;

import java.util.Date;
import java.util.GregorianCalendar;
import java.util.logging.Logger;

import javax.ejb.EJB;
import javax.naming.Context;
import javax.naming.NamingException;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import za.co.solms.javaee.glassfish.container.embedded.EjbContainerUtility;
import za.co.solms.location.GeographicCoordinates;
import za.co.solms.location.Location;
import za.co.solms.location.LocationServices;
import za.co.solms.location.LocationServices.LocationNameInUseException;
import za.co.solms.weather.WeatherServices.NoWeatherReadingsException;

import static junit.framework.Assert.*;

/***
 * @author fritz@solms.co.za
 *
 */
public class WeatherServicesTest
{
    @BeforeClass
    public static void initialize()
    {
        context = EjbContainerUtility.getContainer().getContext();
        lookupServiceBeans();
    }

    private static void lookupServiceBeans()
    {
        try
        {
```

```
locationServices = (LocationServices)context.lookup
    ("java:global/classes/LocationServicesBean");
assertNotNull(locationServices);

weatherServices = (WeatherServices)context.lookup
    ("java:global/classes/WeatherServicesBean");
assertNotNull(weatherServices);
}
catch (NamingException e)
{
    fail("Could not lookup session bean " + e.toString());
}
}

@Before
public void setupTestData()
{
    d1 = new GregorianCalendar(2010,0,20,6,0,0).getTime();
    d2 = new GregorianCalendar(2010,2,20,17,30,0).getTime();
    d3 = new GregorianCalendar(2010,3,20,12,0,0).getTime();
    d4 = new GregorianCalendar(2010,5,5,6,0,0).getTime();

    l1 = new Location("l1","addr1",new GeographicCoordinates(1,2));
    l2 = new Location("l2","addr2",new GeographicCoordinates(3,4));

    try
    {
        locationServices.persistLocation(l1);
        locationServices.persistLocation(l2);

        r1 = new WeatherReading(l1, d1, 22, 85, Ambiance.overcast);
        r2 = new WeatherReading(l2, d1, 27, 95, Ambiance.sunny);
        r3 = new WeatherReading(l1, d2, -12, 70, Ambiance.snowing);
        r4 = new WeatherReading(l1, d3, 15, 100, Ambiance.raining);
        r5 = new WeatherReading(l2, d3, 20, 85, Ambiance.overcast);
        r6 = new WeatherReading(l1, d4, 24, 65, Ambiance.sunny);
    }
    catch (LocationNameInUseException e)
    {
        fail("Claims location name already in use.");
    }
}

@After
public void cleanupTestData()
{
    weatherServices.removeAllWeatherReadings();
    locationServices.removeAllLocations();
}

@Test
public void testPersist()
{
    r1 = weatherServices.persistWeatherReading(r1);
    try
    {
        WeatherReading r = weatherServices.getWeatherReading(r1.getId());

        assertEquals(r1, r);
    }
    catch (NoWeatherReadingsException e)
```

```
{  
    fail("Could not retrieve persisted weather reading.");  
}  
}  
  
@Test  
public void testGetAverageTemperature()  
{  
    weatherServices.persistWeatherReading(r1);  
    weatherServices.persistWeatherReading(r2);  
    weatherServices.persistWeatherReading(r3);  
    weatherServices.persistWeatherReading(r4);  
    weatherServices.persistWeatherReading(r5);  
    weatherServices.persistWeatherReading(r6);  
  
    Date after = d1;  
    Date onOrBefore = d4;  
    try  
    {  
        double avgTemp = weatherServices.getAverageTemperature(l1, after, onOrBefore);  
  
        logger.info("***** avg temp = " + avgTemp);  
        assertEquals(9, avgTemp, doubleEps);  
    }  
    catch (NoWeatherReadingsException e)  
    {  
        fail("Supposedly no weather readings within period");  
    }  
  
}  
  
@EJB  
private static WeatherServices weatherServices;  
  
@EJB  
private static LocationServices locationServices;  
  
private Date d1, d2, d3, d4;  
private Location l1, l2;  
private WeatherReading r1, r2, r3, r4, r5, r6;  
  
private static Context context;  
  
private final static double doubleEps = 1e-9;  
  
Logger logger = Logger.getLogger(WeatherServicesTest.class.getName());  
}
```

## 3.7 JSF2/Facelets based web presentation layer

JSF-2/Facelets is the preferred presentation layer technology for web-based user interfaces in Java. They provide an the ability to efficiently and cleanly develop presentation layers from reusable components with different redeling pipelines which are bound to a services layer.

### 3.7.1 Core elements of JSF2/Facelets based web presentation layers

The core elements of a JSF-2/Facelets based presentation layer are

- A XHTML based view layer in the form of Facelets representing reusable views which are rendered by a pluggable rendering engine.
- JSF binding components which provide to the presentation layer
  - state management,
  - integration with service providers, e.g. the EJB-based services layer containing our business logic,
- a simple expression language providing intuitive connectivity to the JSF binding components and their services and the state they manage,
- simple, non-centralized navigation rules which can be embedded directly within the facelets or the JSF services, or alternatively a more centralized set of navigation rules specified in a `FacesConfig` configuration file,
- validation and data conversion,
- reusable components in the form of facelet/binding component pairs which are reused by higher-level facelets and binding components respectively,
- support for conditional rendering of UI components which is useful for profiling (e.g. showing only those components which are relevant for the user, i.e. only those which are usable by a person who has certain security roles assigned,
- lightweight interactive interfaces via built-in Ajax support, and
- simple environment for resource-bundle based internationalization.

### 3.7.2 Configuration

The configuration files are stored in `src/main/web-app/WEB-INF`. They contain minimally

- an empty beans.xml file,
- a virtually empty facesConfig.xml file which in our case contains specifications for the supported locales and resource bundles,
- a web.xml file specifying
  - the faces servlet,
  - that all \*.jsf requests are mapped onto the faces servlet,
  - the welcome (default) file, and
  - the default suffix (.xhtml) used for the faces.

In addition we register the prime faces servlet in order to be able to use the PrimeFaces tag library. If security is added then one defines here the security realm specification as well as the protected domains.

- a sun-web.xml file which has the applications context root. If security is used, this file will also hold the security role mappings (roles onto groups).

#### 3.7.2.1 faces-config

```
<?xml version="1.0"?>
<faces-config version="2.0"
    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/web-facesconfig_2_0.xsd">
<application>
    <locale-config>
        <default-locale>en-ZA</default-locale>
        <supported-locale>af-ZA</supported-locale>
```

```
<supported-locale>zu-ZA</supported-locale>
</locale-config>

<resource-bundle>
    <base-name>messages</base-name>
    <var>msgs</var>
</resource-bundle>
</application>
</faces-config>
```

### 3.7.2.2 faces-config

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5"
    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/ web-app_2_5.xsd">

    <display-name>WeatherBuro</display-name>

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>Resource Servlet</servlet-name>
        <servlet-class>org.primefaces.resource.ResourceServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Resource Servlet</servlet-name>
        <url-pattern>/primefaces_resource/*</url-pattern>
    </servlet-mapping>

    <!-- Required for PrimeFaces -->
    <context-param>
        <param-name>com.sun.faces.allowTextChildren</param-name>
        <param-value>true</param-value>
    </context-param>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <session-config>
        <session-timeout>10</session-timeout>
    </session-config>

    <context-param>
        <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
        <param-value>.xhtml</param-value>
    </context-param>
```

```
</context-param>
</web-app>
```

### 3.7.2.3 faces-config

```
<?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>/weatherBuro</context-root>

    <class-loader delegate="true"/>

    <jsp-config>
        <property name="keepgenerated" value="true">
            <description>Keep a copy of the generated servlet class' java code.</description>
        </property>
    </jsp-config>
</sun-web-app>
```

### 3.7.2.4 Directory structure

in src/main/webapp we have the index.html file which refers to the home facelet. We have chosen to store all facelets in a faces sub-directory of web-app with the location related facelets in a locations sub-directory, the weather-related facelets in a weather sub-directory and the date-time related facelets in a date-time sub-directory.

### 3.7.3 The index.html file

This file simply redirects to the home-facelet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
        <meta http-equiv="Refresh" content="0; URL=/weatherBuro/faces/home.jsf"/>
        <title>Solms TCD</title>
    </head>
<html>
```

### 3.7.4 The facelet template

This file defines the high level layout including a region for the navigation, a left, right and content panels and a footer. The file also specifies that the application's main menu should be inserted into the navigation panel.

Note that the facelets, jsf and prime faces name spaces are imported to appropriate name space prefixes. We also attach the default styling to be used across all pages to the main template.

The file is stored in src/main/webapp/faces/mainTemplate.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:p="http://primefaces.prime.com.tr/ui">
<h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta http-equiv="PRAGMA" content="no-cache" />
    <meta http-equiv="EXPIRES" content="-1" />
    <meta http-equiv="Cache-Control" content="no-cache"/>
    <meta http-equiv="Cache-Control" content="no-store"/>
    <meta http-equiv="Cache-Control" content="must-revalidate"/>
    <meta http-equiv="Expires" content="Mon, 8 Aug 2006 10:00:00 GMT"/>
    <link href="/sbss/resources/css/default.css" rel="stylesheet" type="text/css" />
    <link href="/sbss/resources/css/cssLayout.css" rel="stylesheet" type="text/css" />

    <title>Facelets Template</title>
</h:head>
<body>
    <h:outputStylesheet library="css" name="cssLayout.css" target="head"/>

    <div id="header" class="outerFrame">
        <ui:insert name="header">
        </ui:insert>
    </div>

    <div id="navigation" class="outerFrame">
        <ui:insert name="navigation">
            <ui:include src="mainMenu.xhtml"/>
        </ui:insert>
    </div>

    <div id="leftMargin" class="outerFrame">
        <ui:insert name="leftMargin">
            Left margin
        </ui:insert>
    </div>

    <div id="rightMargin" class="outerFrame">
        <ui:insert name="rightMargin">
            Right margin
        </ui:insert>
    </div>

    <div id="content" class="content">
        <ui:insert name="content">
            Center content goes here
        </ui:insert>
    </div>

    <div id="footer" class="outerFrame">
        <ui:insert name="footer">
            Footer goes here
        </ui:insert>
    </div>

</body>
</html>
```

### 3.7.5 The main menu

We use a simple buttons-based main menu which will be inserted for every page into the navigation panel of that page. Notice that for JSF based event processing the elements are preceded with the appropriate namespace prefix (e.g. h:):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.prime.com.tr/ui">
<h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta http-equiv="PRAGMA" content="no-cache" />
    <meta http-equiv="EXPIRES" content="-1" />
    <meta http-equiv="Cache-Control" content="no-cache"/>
    <meta http-equiv="Cache-Control" content="no-store"/>
    <meta http-equiv="Cache-Control" content="must-revalidate"/>
    <meta http-equiv="Expires" content="Mon, 8 Aug 2006 10:00:00 GMT"/>
    <link href="/sbss/resources/css/default.css" rel="stylesheet" type="text/css" />
    <link href="/sbss/resources/css/cssLayout.css" rel="stylesheet" type="text/css" />

    <title>Facelets Template</title>
</h:head>
<body>
    <h:outputStylesheet library="css" name="cssLayout.css" target="head"/>

    <div id="header" class="outerFrame">
        <ui:insert name="header">
        </ui:insert>
    </div>

    <div id="navigation" class="outerFrame">
        <ui:insert name="navigation">
            <ui:include src="mainMenu.xhtml"/>
        </ui:insert>
    </div>

    <div id="leftMargin" class="outerFrame">
        <ui:insert name="leftMargin">
            Left margin
        </ui:insert>
    </div>

    <div id="rightMargin" class="outerFrame">
        <ui:insert name="rightMargin">
            Right margin
        </ui:insert>
    </div>

    <div id="content" class="content">
        <ui:insert name="content">
            Center content goes here
        </ui:insert>
    </div>

    <div id="footer" class="outerFrame">
        <ui:insert name="footer">
            Footer goes here
        </ui:insert>
    </div>
```

```
</body>
</html>
```

### 3.7.6 Facelets for adding a new location

We use a simple buttons-based main menu which will be inserted for every page into the navigation panel of that page. Notice that for JSF based event processing the elements are preceded with the appropriate namespace prefix (e.g. h :):

#### 3.7.6.1 The AddLocation facelet page

This page simply defines the header and sets the addLocationPanel as the content. There is no binding object for the page facelet.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml11 <
  /DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.prime.com.tr/ui">

  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>Add location</title>
  </h:head>
  <h:body>
    <ui:composition template="/faces/mainTemplate.xhtml">

      <ui:define name="content">
        <ui:include src="addLocationPanel.xhtml"/>
      </ui:define>
    </ui:composition>

  </h:body>
</html>
```

#### 3.7.6.2 The AddLocationPanel

The AddLocationPanel.xhtml facelet assembles a facelets based form from lower level facelet components which here is simply the data capture facelet and a panel of buttons executing the use cases (add and cancel) for the captured location. Note the biding to the binding bean's addLocation service. It also defines a tag for rendering any messages sent from the JSF binding component.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml11 <
  /DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.prime.com.tr/ui">

  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>Add location</title>
  </h:head>
  <h:body>
    <ui:composition template="/faces/mainTemplate.xhtml">
```

```
<ui:define name="content">
    <ui:include src="addLocationPanel.xhtml"/>
</ui:define>
</ui:composition>

</h:body>
</html>
```

### 3.7.6.2.1 Navigation

Navigation in the view can be done using relative or absolute links to other view components (the .xhtml postfix will automatically be appended).

### 3.7.6.3 AddLocation binding object

In JSF a managed bean is used to store the state of the view component and to perform the integration with our services layer. It is

- annotated as a @ManagedBean,
- specify a scope (request, session, view, application or none),
- must be serializable and have a default constructor,
- must have getters and setters for any data objects used to store the view's state, and
- implement the services requested which may integrate with a session bean based services layer by having those session beans injected.

In addition it will have to get access to the binding components for the facelet components to get hold of the state they manage. This is done by annotating those binding components as @ManagedProperty.

Note that the binding component sends any relevant messages which should be shown in the view to the faces context, e.g. the message that the name for a location is already in use.

```
/***
 *
 */
package za.co.solms.locations;

import java.io.Serializable;

import javax.ejb.EJB;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;

import za.co.solms.location.LocationServices;
import za.co.solms.location.LocationServices.LocationNameInUseException;

/***
 * @author fritz@solms.co.za
 *
 */
@ManagedBean
@RequestScoped
public class AddLocationBinding implements Serializable
```

```
{  
    public AddLocationBinding() {}  
  
    public String addLocation()  
{  
        try  
{  
            locationServices.persistLocation(locationBinding.getLocation());  
            return "locations";  
        }  
        catch (LocationNameInUseException e)  
{  
            FacesMessage msg = new FacesMessage(FacesMessage.SEVERITY_ERROR,  
                "Location name already used.", e.getMessage());  
            FacesContext.getCurrentInstance().addMessage("nameField", msg);  
  
            return "";  
        }  
    }  
  
    public LocationBinding getLocationBinding()  
{  
        return locationBinding;  
    }  
  
    public void setLocationBinding(LocationBinding locationBinding)  
{  
        this.locationBinding = locationBinding;  
    }  
  
    @EJB  
    LocationServices locationServices;  
  
    @ManagedProperty(value="#{locationBinding}")  
    private LocationBinding locationBinding;  
  
    private static final long serialVersionUID = 1L;  
}
```

### 3.7.6.3.1 Navigation

The binding object navigates to subsequent views simply by returning a string for the relative or absolute path to the view. The .xhtml postfix is, once again, automatically appended. For example, from the add location view we navigate back to the locations view after a location has been added.

### 3.7.6.4 The LocationPanel

The LocationPanel.xhtml has the labels and fields capturing the name and address of the location and includes a lower level presentation layer component for capturing the geographic coordinates:

The binding to the properties managing the state for the view is done using EL expressions referring to properties of its binding bean.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1 ←  
    /DTD/xhtml1-transitional.dtd">  
<fieldset xmlns="http://www.w3.org/1999/xhtml"  
    xmlns:ui="http://java.sun.com/jsf/facelets"  
    xmlns:h="http://java.sun.com/jsf/html"  
    xmlns:f="http://java.sun.com/jsf/core"
```

```
xmlns:p="http://primefaces.prime.com.tr/ui">

<legend>Location</legend>

<h:panelGrid columns="2">
    <h:outputLabel id="nameLabel" for="nameField" value="#{msgs.name}" />
    <h:inputText id="nameField" value="#{locationBinding.location.name}" />
    <h:outputLabel id="addressLabel" for="addressField" value="#{msgs.address}" />
    <h:inputTextarea rows="5" id="addressField" value="#{locationBinding.location.address}"/>
        }"/>
</h:panelGrid>

<ui:include src="geographicCoordinatesPanel.xhtml"/>

</fieldset>
```

### 3.7.6.5 Location binding object

The LocationBinding component simply has the fields and getters and setters for the state of the location view (the name and address). Note that it in turn obtains the binding component for the lower level view component as a @ManagedProperty.

```
/***
 *
 */
package za.co.solms.locations;

import java.io.Serializable;

import javax.enterprise.context.RequestScoped;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;

import za.co.solms.location.Location;

/***
 * @author fritz@solms.co.za
 *
 */
@ManagedBean
@RequestScoped
public class LocationBinding implements Serializable
{
    public LocationBinding() {}

    public Location getLocation()
    {
        location.setGeographicLocation(geographicCoordinatesBinding.getGeographicCoordinates());
        ;
        return location;
    }

    public void setLocation(Location location)
    {
        this.location = location;
    }

    public GeographicCoordinatesBinding getGeographicCoordinatesBinding()
    {
        return geographicCoordinatesBinding;
    }
}
```

```
public void setGeographicCoordinatesBinding(
    GeographicCoordinatesBinding geographicCoordinatesBinding)
{
    this.geographicCoordinatesBinding = geographicCoordinatesBinding;
}

private Location location = new Location();

@ManagedProperty(value="#{geographicCoordinatesBinding}")
private GeographicCoordinatesBinding geographicCoordinatesBinding;

private static final long serialVersionUID = 1L;
}
```

### 3.7.6.6 The GeographicCoordinatesPanel

The `GeographicCoordinatesPanel.xhtml` simply has the fields for the longitude and latitude coordinates and binds them to the properties of the binding bean:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml11 <-
 /DTD/xhtml11-transitional.dtd">
<fieldset xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:p="http://primefaces.prime.com.tr/ui">

    <legend>Geographic coordinates</legend>

    <h:panelGrid columns="2">
        <h:outputLabel id="longitudeLabel" for="longitudeField" value="#{msgs.longitude}"/>
        <h:inputText id="longitudeField" value="#{geographicCoordinatesBinding. <-
            geographicCoordinates.longitude}"/>
        <h:outputLabel id="latitudeLabel" for="longitudeField" value="#{msgs.latitude}"/>
        <h:inputText id="latitudeField" value="#{geographicCoordinatesBinding. <-
            geographicCoordinates.latitude}"/>
    </h:panelGrid>

</fieldset>
```

### 3.7.6.7 GeographicCoordinatesBinding object

The `GeographicCoordinatesBinding` simply manages the state for its corresponding view

```
/***
 *
 */
package za.co.solms.locations;

import java.io.Serializable;

import javax.enterprise.context.RequestScoped;
import javax.faces.bean.ManagedBean;

import za.co.solms.location.GeographicCoordinates;
```

```
/***
 * @author fritz@solms.co.za
 *
 */
@ManagedBean(eager=true)
@RequestScoped
public class GeographicCoordinatesBinding implements Serializable
{
    public GeographicCoordinatesBinding() {}

    public GeographicCoordinates getGeographicCoordinates()
    {
        return geographicCoordinates;
    }

    public void setGeographicLocation(GeographicCoordinates geographicCoordinates)
    {
        this.geographicCoordinates = geographicCoordinates;
    }

    private GeographicCoordinates geographicCoordinates = new GeographicCoordinates();

    private static final long serialVersionUID = 1L;
}
```

### 3.7.7 Facelets for viewing and the locations location

We use a prime-faces based table to show the various locations. The view provides the ability to add and remove locations

#### 3.7.7.1 The Locations facelet page

This page simply defines the header and sets the LocationsPanel as the content. There is no binding object for the page facelet.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml11 <-
     /DTD/xhtml11-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.prime.com.tr/ui">

    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
        <title>Locations</title>
    </h:head>
    <h:body>
        <ui:composition template="/faces/mainTemplate.xhtml">
            <ui:define name="content">
                <ui:include src="locationsPanel.xhtml"/>
            </ui:define>
        </ui:composition>
    </h:body>
</html>
```

### 3.7.7.2 The LocationsPanel

The LocationsPanel.xhtml uses a prime-faces table to show the locations. The table uses binds the elements to a collection obtained from the binding component and binds the selected object to a corresponding property in the binding object.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml11 <!--
   /DTD/xhtml11-transitional.dtd">
<h:form xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:p="http://primefaces.prime.com.tr/ui">

  <legend>Locations</legend>

  <h:messages style="color:red"/>

  <ui:include src="locationsTable.xhtml"/>

  <h:panelGrid columns="3" >
    <h:commandButton id="addLocation" value="#{msgs.add}" action="/faces/locations/ <--
      addLocation"/>
    <h:commandButton id="removeLocation" value="#{msgs.remove}" action="#{locationsBinding.removeLocation}"/>
  </h:panelGrid>

</h:form>
```

### 3.7.7.3 LocationsBinding object

The locations binding object maintains the state for the view and makes the integration to our services layer.

```
package za.co.solms.locations;

import java.io.Serializable;
import java.util.List;
import java.util.logging.Logger;

import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

import za.co.solms.location.Location;
import za.co.solms.location.LocationServices;

/**
 *
 * @author fritz@solms.co.za
 *
 */
@ManagedBean
@RequestScoped
public class LocationsBinding implements Serializable
{

  public String removeLocation()
  {
    logger.info("Requested to remove " + selectedLocation);
    locationServices.removeLocation(selectedLocation);
    return "";
  }
}
```

```
}

public List<Location> getAllLocations()
{
    return locationServices.getAllLocations();
}

public Location getSelectedLocation()
{
    return selectedLocation;
}

public void setSelectedLocation(Location selectedLocation)
{
    this.selectedLocation = selectedLocation;
}

private Location selectedLocation;

@EJB
private LocationServices locationServices;

private static final long serialVersionUID = 1L;

private static Logger logger = Logger.getLogger(LocationsBinding.class.getName());
}
```

## Part II

# Persistence via the Java Persistence API (JPA)

# Chapter 4

## JPA overview

### 4.1 What is JPA?

JPA, the *Java Persistence API* aims to provide a standard persistence framework to be used when persisting entities to relational databases. It is a standard API for Java based object-relational (O/R) mappers. It allows you to abstract from

- any specific O/R framework (e.g. Hibernate, EclipseLink, ...)
- from any specific database and the flavour of SQL it may be using.

Even though JPA is commonly used from within enterprise applications which often run in enterprise application servers, JPA can also be used within Java-SE applications.

### 4.2 What does JPA provide?

JPA provides

- object relational mapping which maps an object graph onto a structure of linked database tables,
- a standard API for entity managers which enable one to
  - persist and remove entities,
  - retrieve persisted entities either eagerly or lazily,
  - provide entities as value objects which can be merged back into the persistence context (into managed state),
  - to create interpreted and pre-compiled JPQL queries which are queries across the object graph which are ultimately mapped onto the query language supported by the underlying physical persistence technology,
  - to create transactions which have the persistence context as an enlisted resource,
  - managing the state of the persistence context itself including flushing, clearing and refreshing of the persistence context.

### 4.3 JPA providers

JPA, the Java Persistence API, is a standard API for object relational mappers and persistence managers. It does not provide an implementation.

Commonly used examples of JPA providers include

- EclipseLink / TopLink
- Hibernate

# Chapter 5

## Persistence contexts

### 5.1 Overview persistence contexts

Even though a persistence context is one of the central concepts of JPA, it is often not very well understood.

#### 5.1.1 What is a persistence context?

A persistence context is a cache of objects whose persistence is managed. The cache maintains objects in memory which can be efficiently manipulated without everytime having to consult the database.

The persistence context typically maintains a set of non-shared database connections. Only one instance with the same object identity exists within a persistence context.

#### 5.1.2 What is an entity manager?

An entity manager is an entity resource manager which is associated with a persistence context. It maintains a cache for the persistence context and the life cycle of the entity instances contained within that persistence context. The entity manager interacts with the object-relational mapper and uses a connection pool to interact with the persistence provider (e.g. database).

The entity manager is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Entities which are managed by an Entity Manager will automatically propagate these changes to the database when a transaction is committed. Entities which have been detached can be merged back into managed state resulting in any modifications made outside the persistence context being ultimately persisted to database upon commit.

#### 5.1.3 Optimistic concurrency control

For highly scalable systems one usually requires optimistic concurrency control with versioning. Version or timestamp checking is used to detect

- conflicting updates across transactions, and to
- prevent lost updates within a transaction.

The behaviour is really the same as in concurrent version control systems like subversion or CVS. Different persistence contexts obtain their own detached copy of the entities. Which ever persistence context commits first will merge their changes into the global persistence context and persist its domain to the database.

Subsequent entity managers who commit may encounter a conflict when they merge their changes back into the global context. If this is the case, an exception is thrown. Otherwise the changes made within that transaction are persisted through to the database.

## 5.1.4 The life span of an entity manager

A persistence context may be either a

- transaction scoped persistence context, or a
- extended persistence context.

### 5.1.4.1 Transaction-scoped persistence contexts

In the case of transaction-scoped persistence context, one will obtain (in a managed application) or have to create (in a non-managed application) a new persistence context per transaction. Any entities which have been enlisted within the cache will be detached at the end of the transaction and any changes made after detachment will no longer be propagated into the database.

Transaction-scoped persistence contexts do not support optimistic concurrency control. They are thus largely used in non-managed applications which usually do not have high concurrency demands.

### 5.1.4.2 Extended persistence contexts

Extended persistence contexts maintain a cache across transactions. They provide thus more efficient caching and support optimistic concurrency control. Extended transaction contexts are typically used in managed applications where the caching and optimistic concurrency control are important to achieve the required scalability.

Note that the cache of an extended persistence context may fill up, over time, with old objects. These need to be flushed from time to time. This is done automatically in a managed environment, but needs to be done manually in a non-managed environment.

## 5.1.5 The transaction management for a persistence context

The transaction type for a persistence context may be either RESOURCE\_LOCAL or JTA. In the case of RESOURCE\_LOCAL the transaction management is provided by JPA which typically delegates it to the local resource (e.g. database). In the case of JTA a transaction manager implementing the *Java Transaction API* is used. Such entity managers can enlist multiple resources within a transaction. In managed environments transaction boundaries are usually managed by the application server deducing the relevant transaction boundaries from the more abstract transaction requirements annotations (e.g. `requires`, `requires-new`, ...).

RESOURCE\_LOCAL is often used in non-managed applications where transaction control may be required only for resources from a single database. When using RESOURCE\_LOCAL, you must use the entity transaction API to begin/commit around every call to your entity manager:

```
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();
entityManager.persist(myEntity);
transaction.commit();
```

The entity manager for a managed environment is provided by the application server. It will usually use JTA transaction management, allowing for multiple resources (e.g. databases, message queues, external systems, ...) to be enlisted within a transaction.

## 5.1.6 The scope of a persistence context

The scope of a persistence context is the domain of entities which are managed by it. The scope is effectively defined by specifying the collection of entities which fall within the scope of the persistence context. This can be done in one of the following means:

- A persistence unit may refer to a `orm.xml` file defining the entities and how they should be mapped onto a relational database. This is specified in a `<mapping-file>` element in the `persistence.xml`.

- You can use one or more <jar-file> elements to specify that the entity classes in those jar files need to be included in the persistence context.
- You can have a list of <class> elements listing the entity classes to be managed within the persistence context.
- The annotated managed persistence classes contained in the root of the persistence unit which is the jar file or directory, whose META-INF directory contains the persistence.xml file. This approach is the common approach when defining the persistence context for managed applications.

### 5.1.7 How are entity managers obtained?

Depending on whether one performs persistence from a managed or non-managed application, the entity manager is either provided/injected by the environment (i.e. by the application server) or needs to be created manually.

#### 5.1.7.1 Obtaining an entity manager in a container managed environment

In a container managed environment the entity manager is provided by the container and is obtained either via dependency injection by annotating an EntityManager field or via a JNDI lookup.

To obtain a JTA based entity manager you need to annotate the entity manager field with a @PersistenceContext annotation

```
@PersistenceContext  
EntityManager entityManager;
```

To obtain a entity manager using RESOURCE\_LOCAL transaction suppoer, you annotate the entity manager fied with a @PersistenceUnit annotation

```
@PersistenceUnit  
EntityManager entityManager;
```

#### 5.1.7.2 Manual creation of an entity manager in a JavaSE application

In a JavaSE application, the entity manager is not injected from a container, but must be created explicitly. For this you will

- define the persistence context descriptor in a META-INF directory (or construct the properties in code),
- obtain a entity manager factory for your persistence context from the general persistence environment, providing the entity manager factory a suitable name, and
- create an entity manager for your persistence context.

```
EntityManager em = Persistence.createEntityManagerFactory("myPersistenceContext").  
createEntityManager();
```

Generally you should only have a single entity manager per persistence context active at any time.

---

#### Note

Calling EntityManagerFactory.createEntityManager() twice results in two separate EntityManager instances and therefor two separate PersistenceContexts/Caches.

---

### 5.1.8 Detaching objects from a persistence context

If an object leaves the cache/persistence context, it is detached from it. The entity is then in a value object state. This will, for example, happen when an object is serialized (e.g. by being passed as parameter in a remote service request). An object will also become detached if it exists beyond the life span of the entity manager (and hence cache). Any updates made to a detached object are not reflected in the PersistenceContext/Cache.

You cannot call request an entity manager to persist or remove a detached entity (value object). Once the value object has been re-attached to the persistence context via `entityManager.merge(myValueObject)`, it can be persisted and removed again.

---

**Note**

Due to lazy loading, detached objects (e.g. serialized parameters) may not have all the information populated.

---

## 5.2 Persistence context configuration

The persistence unit for non-managed applications uses `RESOURCE_LOCAL` for the transaction type and needs to specify the database and driver which should be used as well as the entities which should be managed by the persistence unit.

### 5.2.1 Typical persistence context for non-managed Java applications

For non-managed Java applications one needs to specify the database, database driver and login credentials which should be used as well as the set of entity classes which should be managed by the persistence context. The latter can be specified as a list of classes, in a separate `orm.xml` file or by specifying the jar-file(s) which contains the entity classes. The latter is often the most convenient approach:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

<persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jar-file>myEntities.jar</jar-file>
    <properties>
        <property name="eclipselink.target-database" value="DERBY"/>
        <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
        <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
        <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/myDB" />
        <property name="javax.persistence.jdbc.user" value="myApp"/>
        <property name="javax.persistence.jdbc.password" value="myApp"/>
    </properties>
</persistence-unit>
</persistence>
```

### 5.2.2 Typical persistence context for managed applications

A typical persistence unit configuration for a managed environment uses JTA-based transactions and refers to a data source defined for the container. In addition it can specify some properties for the object-relational mapper:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0" 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_1_0.xsd">
<persistence-unit name="myEnterpriseApp" transaction-type="JTA">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<jta-data-source>jdbc/myDataSource</jta-data-source>
<properties>
    <property name="eclipselink.ddl-generation" value="create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

# Chapter 6

## Entities

### 6.1 Overview

Entities are data classes which exist from when they are created up to the point where they are explicitly removed. They are persisted through to persistent storage (e.g. database) and may survive the life span of a session or application.

### 6.2 Simple entities

We define a simple entity here as an entity which does not have any relationships to other entities.

#### 6.2.1 Declaring an entity

An Entity is defined by annotating them with `javax.persistence.Entity`

```
import javax.persistence.Entity;

@Entity
public class Account
{
    ...
}
```

One may customize the persistence by specifying, for example, the database table to which the entity should be persisted via

```
@Entity(name="ACCOUNTS")
public class Account
{
    ...
}
```

#### 6.2.2 Requirements for entities

Entities must satisfy a number of requirements:

- **Constructors** Entities must have a public or protected default (no-argument) constructor. They may have other constructors. If the default constructor is declared protected, it is only available for the entity manager and users are forced to use the publicly available constructors.

- **Primary key** Every entity requires a primary key, which may be a simple primary key represented by a bean field, or a composite key. The primary key is specified by annotating the relevant field with `javax.persistence.Id`

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Account
{

    // Services and constructors here
    //...

    @Id private int accountNumber;
    private double balance;
}
```

- **Support for serialization** Entities which are meant to be detachable in order to pass them around as value objects (i.e. sent to a client through a remote interface) must be serializable. These temporarily detached value objects can be re-attached to the entity manager at a later stage.
- **Final** Neither the class, nor any of its methods, may be final. The JPA provider must be able to subclass your class, in order to provide natural interception points and to access protected fields not published via public access methods.
- **Entities may be abstract** Shared data may be encapsulated in abstract entities which are subclassed by various concrete entities.
- **Entities and inheritance** Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- **Fields accessed only via accessors or business methods** Persistent instance fields must be declared with private, package or protected scope (preferably private), and can only be accessed directly by the entity class's methods. Clients must access the entity's state through accessor or business methods.
- **All annotations either at getter or field level** The entity manager will access the fields either directly or via getters and setters. The method used depends on whether the annotations are done on the fields or on the getters. They should, however, not be mixed - i.e. they should be either all at the field level or all at the getter level. Alternatively the access type can be explicitly specified on an entity by annotating the entity with either `@Access(AccessType.FIELD)` or `@Access(AccessType.Property)`.

### 6.2.3 What is persisted?

The persistent state of an entity is defined by its fields. The fields are accessed by the entity manager either

- via *accessors* (`getXXX()` methods) following the JavaBeans specification, or
- via *direct access to fields*.

The schema used for a particular object is inferred, based on whether the *primary key* (the `@javax.persistence.Id` annotation) has been indicated on a field, or an accessor method.

For example, state management would be performed through the get/set service of the following class. The entity manager will call the services in order to extract the state to be stored in the database, or to populate an instance with information from the database.

```
@Entity
public class Account implements Serializable
{
```

```
@Id  
public int getAccountNumber()  
{  
    return accountNumber;  
}  
  
public void setAccountNumber( int accountNumber )  
{  
    this.accountNumber = accountNumber;  
}  
  
private int accountNumber;  
private double balance;  
}
```

whereas the state would be managed directly through the fields in the following case. The entity manager will use the serialisation mechanism to extract / populate the object's state.

```
@Entity  
public class Account implements Serializable  
{  
    public int getAccountNumber()  
    {  
        return accountNumber;  
    }  
  
    public void setAccountNumber( int accountNumber )  
    {  
        this.accountNumber = accountNumber;  
    }  
  
    @Id  
    private int accountNumber;  
    private double balance;  
}
```

---

**Note**

Though the latter case 'breaks encapsulation' (as the entity manager can use privately declared fields), this is usually not a problem, as this effectively applies only during object creation. Encapsulation usually needs to protect the state of an object from clients, not from I/O frameworks.

---

### 6.2.3.1 Valid persistent field types

The following are valid data types for persistent fields:

- Java primitives and primitives wrappers,
- The following built-in classes:
  - `java.lang.String`,
  - `java.util.Date` (requiring the `@Temporal` annotation),
  - `java.math.BigDecimal` and `BigInteger`,
  - `java.sql.Time` and `Timestamp` (requiring the `@Temporal` annotation),
  - `byte[]`, `Byte[]`, `char[]` and `Character[]`,

- `java.sql.Blob` and `Clob`,
- embedded classes,
- other entities,
- collections of primitives, and
- any other serializable objects.

#### 6.2.3.2 Collection variables

Collection variables are persisted naturally, which allows for the standard usage of `java.util.Collection`, `java.util.List`, or `java.util.Set` data types.

Generics should be used (i.e. `List<Account>`), and the various relationship annotations (such as `@OneToMany`) are required to control the mapping, such as putting bi-directional mappings in place.

#### 6.2.3.3 Transient fields

Transient fields (fields which do not need to be persisted, and hence do not form part of the object's persistent state) are specified by

- in the case of property access annotating the getter or setter as `@javax.persistence.Transient`, or
- by declaring the field itself as `transient` using the Java language keyword, in the case of field access.

#### 6.2.3.4 Field validation

Field validation may be done in the *setter* methods, which may throw an exception. An exception will cause the controlling transaction (if any) to be rolled back.

---

##### Note

It is typically questionable whether use-case specific validation should be performed on the entity object at all: This should rather be enforced by controller objects (e.g. the session beans managing the entities) and user interface(s).

---

#### 6.2.4 Embedded classes

Entities may have as components finer grained objects which are persisted, not as separate entities, but are expanded as a set of columns within the tables created for the entities within which they are embedded.

For example, a location may have a name, an address and geographic coordinates which include the degrees longitude and degrees latitude. Embedding the `GeographicLocation` class within a `Location` entity would add the `degreesLongitude` and `degreesLatitude` columns to the `Location` table.

As such embedded objects have no persistent identity. Their identity is the role in the context of the owner entity.

---

##### Note

Embedded classes are only used for composition relationships between classes, i.e., no other object may obtain a reference to an embedded object.

---

### 6.2.4.1 Defining an embeddable class

An class which is meant to be embeddable within entity beans must be annotated as such using the `@Embeddable` annotation. Furthermore, it must be serializable:

```
@Embeddable
class GeographicLocation implements Serializable
{
    // getters & setters

    private double degreesLongitude, degreesLatitude;
}
```

#### 6.2.4.1.1 Specifying the access type

By default, the access type of an embeddable class is determined by the access type of the entity within which it is embedded. This can be changed by annotating the embeddable class with an `@Access` annotation whose value is either `AccessType.Field` or `AccessType.Property`.

---

#### Note

It is generally recommended to specify the access type of the embeddable explicitly in order to prevent potential object-relational mapping errors caused by the entity manager loosing track of the state due to access through both channels. This can happen when the embedded class is contained in an entity with one access type which is, in turn, part of an entity which uses another access type.

---

### 6.2.4.2 Embedding and embeddable class within an entity

To embed an embeddable class within an entity one has to add a field for the embedded class and annotate it or the getter as `@Embedded`

```
@Entity
class Location implements Serializable
{
    ...

    @Embedded
    public GeographicCoordinates getCoordinates() { return coordinates; }

    private String name;
    ...
    private GeographicCoordinates coordinates;
}
```

### 6.2.5 Primary keys

For every entity one must specify a primary key which may be

- a simple primary key, or
- a composite primary key.

#### 6.2.5.1 Simple primary keys

Simple primary keys are persisted into a single database column which will be assigned a primary key constraint.

---

### 6.2.5.1.1 Valid types for simple primary keys

The following are valid data types for persistent fields:

- Java primitives and primitives wrappers, and
- `java.lang.String`

Although approximate numeric types like `float` or `double` are permitted, they should generally not be used due to their inability to represent absolute values.

### 6.2.5.1.2 Specifying the primary key field

A primary field is specified for an entity by annotating either

- an accessor method, or
- an instance field

with `@javax.persistence.Id`.

For example, the following code snippet specifies that the `accountNo` is to be used as a primary key for accounts:

```
@Entity
public class Account
{
    @Id
    public getAccountNo() { return accountNo; }
    ...
    private int accountNo;
}
```

### 6.2.5.1.3 Automatically generating primary keys

If required, one can request the entity manager/database to automatically generate the value of a primary key (which will always be a unique value) by annotation the key with the `@GeneratedValue` annotation:

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Account implements Serializable
{

    ...

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long accountNumber;
}
```

The annotation takes parameters, which allows the developer to indicate the generator (such as a particular database table), and/or to indicate the strategy to be used (typically realised by the underlying database). When one ‘doesn’t care, as long as it is unique’ the `AUTO` strategy usually works well.

### 6.2.5.2 Primary key classes (composite keys)

The JPA specification supports composite keys via primary key classes. A primary key class is defined as an embeddable class whose properties form the primary key fields. It must have a default constructor as well as setters and getters for the primary key fields.

#### 6.2.5.2.1 Interface for the primary key class

```
package za.co.solms.partsCatalog;

/**
 * Interface for a part identifier.
 */
public interface PartId
{
    public String getCode();

    public String getManufacturerId();

    public interface Mutable extends PartId
    {
        public void setCode(String newCode);

        public void setManufacturerId(String newManufacturer);
    }
}
```

#### 6.2.5.2.2 Implementation of a primary key class

```
package za.co.solms.partsCatalog;

import java.io.Serializable;
import javax.persistence.Embeddable;
import za.co.solms.partsCatalog.PartId;

/**
 * Primary key class for part id which has been annotated as
 * embeddable such that it can be used as the primary key class
 * for an entity persisted via an JPA entity manager.
 */
@Embeddable
public class PartPK implements PartId.Mutable, Serializable
{
    public PartPK(String code, String manufacturerId)
    {
        setCode(code);
        setManufacturerId(manufacturerId);
    }

    protected PartPK() {}

    public void setCode(String newCode)
    {
        this.code = newCode;
    }

    public void setManufacturerId(String newManufacturer)
    {
        this.manufacturerId = newManufacturer;
    }

    private String code;
    private String manufacturerId;
}
```

```

}

public String getCode()
{
    return code;
}

public String getManufacturerId()
{
    return manufacturerId;
}

private String code, manufacturerId;
private static final long serialVersionUID=200609211600L;
}

```

#### 6.2.5.2.3 Entity bean using the primary key class

```

package za.co.solms.partsCatalog;

import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class PartBean implements Part.Mutable
{

    public PartPK getPartId() {return partPk; }

    public String getDescription() {return description; }

    public String getName() {return name; }

    public void setDescription(String newDescription)
    {
        this.description = newDescription;
    }

    public void setName(String newName) {this.name = newName; }

    public void setPartId(PartId newPartId)
    {
        this.partPk = new PartPK(newPartId.getCode(), newPartId.getManufacturerId());
    }

    @EmbeddedId
    private PartPK partPk;
    private String name, description;
}

```

#### 6.2.6 Specifying column mappings

The object-relational mapping can be customized in the `orm.xml` descriptor files or via annotations. For example, the column name, length and precision can specified via the `@Column` annotation:

```

@Entity(name="VHCL")
public class Vehicle
{

```

```
@Column(name="REG_NO" length="10")
public String getRegistrationNumber()
{
    ...
}
```

### 6.2.7 Column constraints

Commonly column constraints include a specification on whether a column is required or not and whether the entries in the column need to be unique:

```
@Entity(name="VHCL")
public class Vehicle
{
    @Column(name="REG_NO", length="10", nullable=false, unique=true)
    public String getRegistrationNumber()
    {
        ...
    }
}
```

### 6.2.8 Primitive collections and maps

Collections and maps of basic types need to be annotated with an `@ElementCollection` annotation. If the storage provider is a relational database, collections of primitives are mapped onto a separate table with a link column and a value. Maps of primitives onto primitives are mapped onto a separate table with one link column, one key column and one value column.

The mapping can be customized using the `@CollectionTable` annotation which allows you to specify the name of the table. Maps can be additionally annotated with a `@MapKeyColumn( name="..." )` annotation.

```
@Entity
public class CarPriceList implements Serializable
{
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "UnavailableCars")
    public List<String> withdrawnCars;

    @ElementCollection(fetch = FetchType.EAGER)
    public Map<String, double> activeCarPrices;
}
```

## 6.3 Relationships

JPA supports persistent relationships which are mapped down to database layer. The supported include the standard object-oriented relationships:

- association and aggregations,
- composition, and
- specialization.

### 6.3.1 Summary of UML relationships

Figure 6.1 summarizes the UML relationships. It shows that these are conceptually specializations of each other and that we have weak and strong variants of ‘is a’, ‘has a’ and ‘uses’.

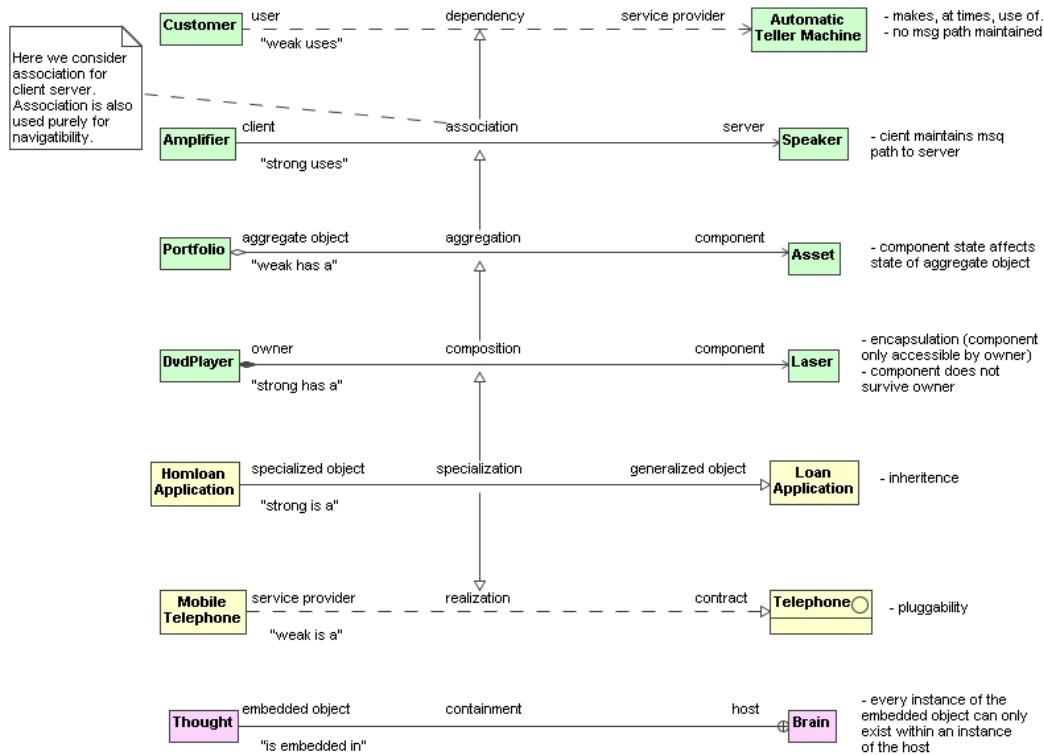


Figure 6.1: Summary of UML relationships

#### 6.3.1.1 Dependency

Instances of the one class, the user, make, at times, use of instances of the other class, the service provider. The latter is often modelled as an interface in order to decouple the user from any particular implementation of a service provider. For example, clients of the bank, upon spotting an ATM, may decide to use it in order to withdraw some cash from their account, but they do not maintain a message path to any particular ATM.

A dependency is called a ‘weak uses’ because the user does not maintain a message path and is not in a position to, at any stage, send further service requests to the service provider.

#### 6.3.1.2 Association

Association is used for two purposes. On the one side it is used purely for navigability. In the second case it is used for a client server relationship (or peer-to-peer in the case of binary associations). In either case, the object which has the association maintains a message path to the associated object.

It is conceptually a special form of dependency where the client still, at times, makes use of the service provider, but now the client maintains a message path to the service provider. For example, an amplifier has a message path to the speakers (the cables) in order to send service requests to them.

An association is called a ‘strong uses’ because the client maintains the relationship and is in a position to send, at any stage, further service requests to the service provider.

### 6.3.1.3 Aggregation

Aggregation is a special form of association. The aggregate object still maintains a message path to the component. It still can make use of the components. For example, in the context of a portfolio calculating its value, it will request the value of each asset and sum them up.

However, in aggregation a state transition in the component may imply a state transition in the aggregate object, i.e. aspects of the component state are part of the state of the aggregate object. In our example, a change in the value of any of the assets results in a change in the value of the portfolio.

Aggregation is a weak has a relationship because it does not take exclusive control of the component. The component can be accessed directly and may be part of other aggregate objects. Furthermore, the asset can survive the portfolio. For example, a particular asset may be part of a number of different portfolios. A change in its value results in the value of multiple portfolios changing. Furthermore, one may decide to remove a portfolio (a particular grouping view onto one's assets), but the assets would still survive.

### 6.3.1.4 Composition

Composition is a special type of aggregation (and hence also a special type of association and a special type of a dependency). If the component state changes, the state of the owner also changes. The owner also maintains the message path and may, at any stage, issue further service requests to the component.

Now we have, however, a ‘*strong has a*’ relationship where the owner takes full responsibility for the component and encapsulates the component. If a user of the DVD player wants to send a service request to its laser, it will have to do so via the services offered by the DVD player itself. If the laser is broken, the DVD player is broken too (it is responsible for the laser). Finally, should we decide to scrap the DVD player, the laser will be scrapped also.

### 6.3.1.5 Realisation

Realisation is a weak is a relationship. It is used to show that a service provider implements an interface (and often a complete contract). This facilitates substitutability of one service provider with any other realising the same contract.

### 6.3.1.6 Specialisation

Specialisation is a very strong relationship which should be used with care. It is commonly used for data or value objects. Specialisation can be conceptually seen as special form of realisation in that the sub-class is a specialised realisation of the super-class. One can say, specialisation inherits substitutability from realisation.

It can also be seen as a special form of composition as every sub-class instance will create an encapsulated super-class instance through which it obtains the superclass attributes, services and relationships. The super-class instance for the sub-class cannot be accessed directly from outside the sub-class instance. It will also not survive the sub-class instance.

The superclass instance is part of the state of the sub-class instance. If the state of the superclass instance changes, the state of the sub-class instance changes too. For example, assume a home loan application inherits a loan amount from loan application. If the loan amount changes the state of the home loan application changes.

The sub-class instance also maintains a message path to the super class instance (*super* in Java and *base* in C#). It is thus also a special for of association. It may, for example make use of a superclass service via *super.serviceRequest()*.

### 6.3.1.7 Containment

Containment is a separate relationship where instances of one class can only exist in instances of another. There are examples of such relationships in nature.

### 6.3.1.8 Shopping for relationships

In order to determine the correct relationship between two classes one can take a requirements driven approach - similar to a shopping list for relationships. In either case one should *always choose the weakest relationship* which fulfils one's requirements.

The process of determining the correct relationship goes along two legs. On the one side you are trying to establish the type of dependency between the two classes. On the other side you will assess the level of substitutability and inheritance required.

First we assess whether there is a dependency between the classes. If instances of one class, A, never make use of instances of another class, B, and if one also does not need to be able to navigate from an A to a B, then there is not much of a relationship between these classes. Otherwise there is at least a dependency of A on B.

Next ask yourself whether instances of A should maintain a message path to instances of B. If so, upgrade the dependency to an association. If not, leave the relationship as a dependency.

If we reached this point, we have at least an association from A to B. Next you can ask yourself whether any change in the state of an instance of B results in a change of state in the instance of A which maintains an association to it. If the answer is yes, then upgrade the association to an aggregation relationship. Otherwise leave it as an association.

If we reached this point we have at least an aggregation relationship from A to B. Next, you can ask yourself whether the aggregate object needs to take full control of the component, or whether other objects should be allowed to access the component directly. If full control is required, then upgrade the relationship to a composition relationship. Otherwise leave it as an aggregation relationship.

---

**Note**

If you decided on composition, you can do the following test to check whether you perhaps made an error. Check whether it would make sense for the component to outlast (survive) the owner. If the answer is yes, then the relationship could not have been a composition relationship.

---

Next let us look at the plug-ability requirements. If the class should be pluggable (i.e. if the service provider should be substitutable), then one should introduce a contract for the service requirements. In the bare form, the contract is simply an interface and we have a realisation relationship.

In order to assess whether you should upgrade the realisation relationship to a specialisation relationship, assess whether you want to inherit common properties and services.

---

**Note**

In general we would recommend to favour interfaces and realisation above inheritance and specialisation. The latter tends to result in very rigid designs which are difficult to modify. One may choose to use specialisation only for value or data objects which do not perform significant functionality.

---

### 6.3.2 Composition between entity beans

In a composition relationship the component may not survive the owner. This is supported in EJB via the *cascading* relationship attribute. Cascading is supported for *create*, *merge* and *remove* operations.

---

**Note**

Cascading-delete enforces that the component entity bean is removed when the owner is removed.

---

### 6.3.3 Relationship types

JPA supports uni-directional and bi-directional one-to-one, one-to-many, many-to-one and many-to-many relationships.

### 6.3.3.1 Relationship owner

For each relationship there is a *relationship owner* who maintains the pointer (e.g. foreign key) of the relationship. In the case of bi-directional relationships the related entity also provides a message path to the relationship owner.

### 6.3.3.2 Unidirectional single-valued relationships

Consider the uni-directional single-valued relationship shown in the following figure:

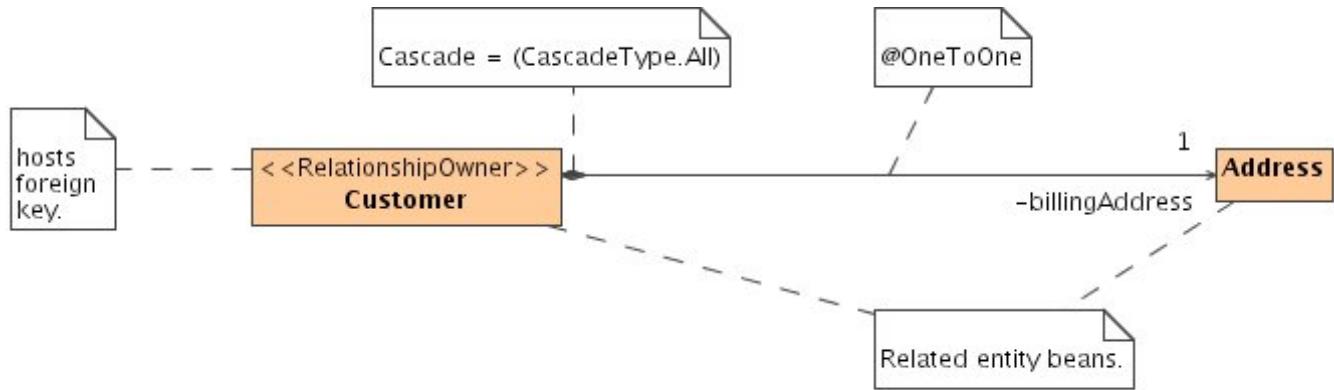


Figure 6.2: A unidirectional one-to-one relationship

The relationship owner simply maintains the message path as well as the foreign key. The mapping onto entity beans would be as follows:

```

@Entity
public class Address {...}

@Entity
public class Customer {
{
    public Address getAddress() { return address; }

    public void setAddress(Address addr) { address = addr; }

    @OneToOne(cascade=CascadeType.DELETE)
    private Address address;
}
  
```

#### Note

Cascading delete is specified to request the composition behaviour, i.e. that the component should not outlast the owner.

### 6.3.3.3 Bidirectional one-to-one relationships

Consider the bi-directional one-to-one relationship shown in the following figure:

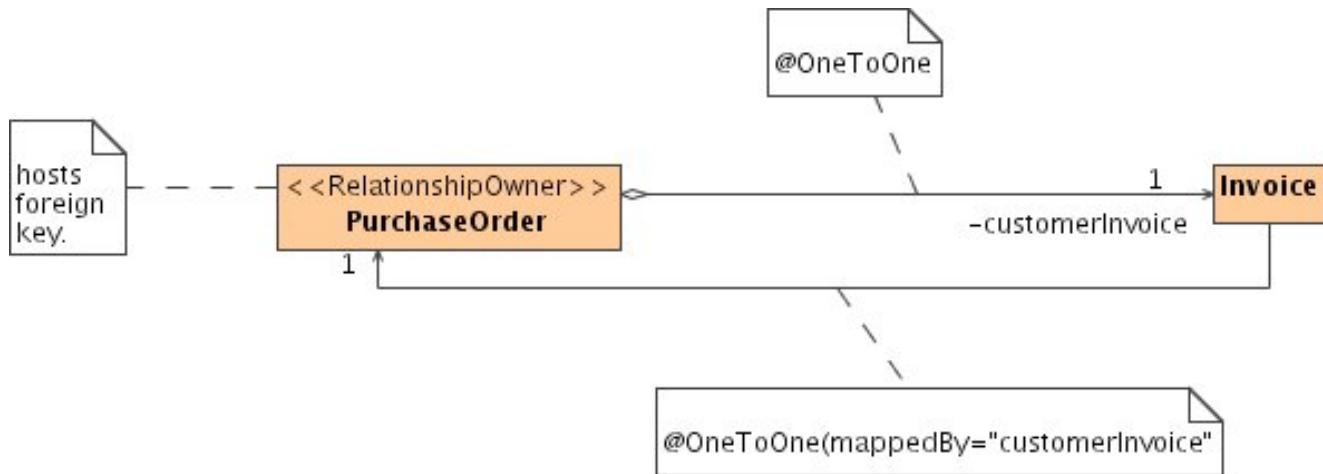


Figure 6.3: A bidirectional one-to-one relationship

Here both entities maintain message paths to one another. At database level, there is, however, only one foreign key maintained, i.e. only one relationship owner.

For the reverse relationship we specify a `mappedBy` attribute which ensures that this relationship is not implemented at storage level:

```

@Entity
public class PurchaseOrder
{
    public Invoice getIssuedInvoice() {return invoice; }

    public void setIssuedInvoice(Invoice inv) {invoice = inv;}
    ...

    @OneToOne
    private Invoice issuedInvoice;
}

@Entity
public class Invoice
{
    public PurchaseOrder getPurchaseOrder() {return purchaseOrder; }

    public void setPurchaseOrder(PurchaseOrder order)
    {
        purchaseOrder = order;
    }
    ...

    @OneToOne (mappedBy="issuedInvoice")
    private PurchaseOrder order;
}

```

#### 6.3.3.4 Bidirectional many-to-one relationships

To implement one-to-many or many-to-many relationships we need to use either one of the `java.util` collection types:

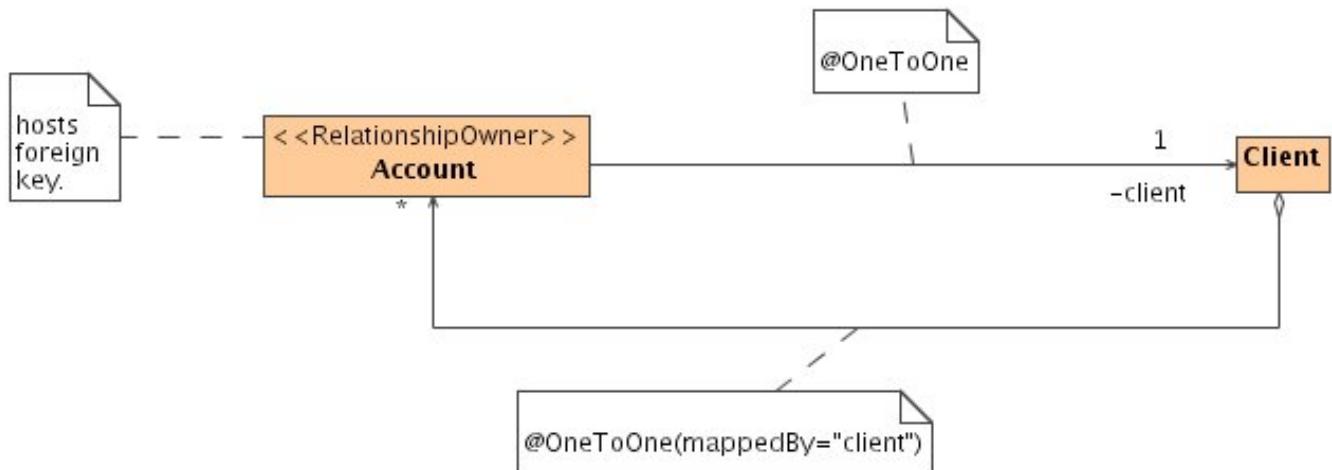


Figure 6.4: A bidirectional many-to-one relationship

The mapping onto entity beans would be as follows:

```

@Entity
public class Client
{
    public Collection<Account> getAccounts()
    {
        return accounts;
    }
    ...

    @OneToMany(mappedBy="client")
    private Collection<Account> accounts;
}

@Entity
public class Account
{
    public Client getClient() {return client;}

    public void setClient(Client clnt) {client = clnt;}

    @ManyToOne
    private Client client;
}

```

### 6.3.3.5 Cascading Operations

Depending on your design, you may desire that operations on a parent component (e.g. a Client) *cascade* to its constituent components. (for example, deleting a client may or may not cause all the client's accounts to be deleted). This is specified with the `cascade` parameter of any of the relationship annotations, with a set of enumerated values provided by the enumeration `javax.persistence.CascadeType`. For example, to cause all operation (including deletion) to be cascaded to the constituent component:

```

public class Client
{
    ...

```

```
@OneToOne(cascade=CascadeType.ALL)
private Portfolio portfolio;
}
```

The allowable values are:

- **ALL** Cascade all operations
- **MERGE** Cascade merge (update) operation
- **PERSIST** Cascade persist operation
- **REFRESH** Cascade refresh operation
- **REMOVE** Cascade remove operation

### 6.3.4 Fetching strategies

The JPA specification supports the definition of fetching strategies which enables one to perform some optimisation based on domain and usage knowledge.

#### 6.3.4.1 Eager fetching

Associated entities and embedded objects can be eagerly loaded or lazily fetched. The fetch strategy for a bean relationship can be specified as a relationship attribute:

For example

```
@Entity
public class Order
{
    ...
    @ManyToOne(fetchType=FetchType.EAGER)
    public Client getClient()
    {
        ...
    }
}
```

specifies that the client information should be retrieved when the order is retrieved.

In the case of lazy fetching, the associated bean is only fetched when it is required. It is specified via the `FetchType.LAZY` value for the `fetchType` relationship attribute.

#### 6.3.4.2 Defaults fetching strategies

The default fetching strategy is

- *EAGER* for one-to-one and many-to-one bean relationships and for embedded classes.
- *LAZY* for one-to-many and many-to-many relationships.

### 6.3.5 Specialization relationships

JPA supports persistent specialization relationships (which are mapped through to the database) as well as polymorphism on entities.

### 6.3.5.1 Mapping onto relational databases

Relational databases do not intrinsically support specialization. The JPA specification supports 3 types of mappings:

1. Joined subclass.
2. Single table per class hierarchy.
3. Table per class.

### 6.3.5.2 Joined subclass

This is often the preferred mapping. The ultimate superclasss is a simple entity bean mapping. Each subclass has a reference (foreign key) to its direct superclass.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person {...}

@Entity
public class Employee extends Person {...}

@Entity
@InheritanceJoinColumn(name="EMPLOYEE_REF")
public class Contractor extends Employee {...}
```

### 6.3.5.3 Single table per class hierarchy

This mapping results in a non-normalized database structure which requires table changes whenever another subclass is added. It does, however, have the advantage of persisting the fact that it is a specialization relationship through to database which would allow for reconstruction of the specialization relationships through reflection on the database structure.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE
              discriminatorType=DiscriminatorType.STRING
              discriminatorValue="Person"
              // default: fully qualified class name
)
@DiscriminatorColumn(name="Type") // default: "TYPE"
public class Person {...}

@Entity
@Inheritance(discriminatorValue="Employee")
public class Employee extends Person {...}

@Entity
@Inheritance(discriminatorValue="Contractor")
public class Contractor extends Employee {...}
```

### 6.3.5.4 Single table per class

This is only seldomly used -- iff, then mainly for the top level of the specialization hierarchy:

```
@Entity  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
public class Person {...}  
  
@Entity  
public class Employee extends Person {...}  
  
@Entity  
public class Contractor extends Employee {...}
```

## Chapter 7

# The Java Persistence Query Language (JPQL)

### 7.1 Introduction

The Java Persistence Query Language (JPQL) is a storage technology-neutral object-oriented query language enabling users to formulate queries across object graphs. As such the conceptual queries specified in EJB QL are mapped onto the query language for the chosen persistence technology like the SQL for the relational database you've chosen or OQL for an object database.

### 7.2 JPQL versus SQL

The structure of an JPA query is in many ways similar to a traditional SQL query. It is generally of the form

```
SELECT |UPDATE|DELETE <selection> FROM <source> WHERE <condition>
```

where

- **SELECT** specifies the type of objects or values to be selected which may be
  - an entity,
  - a value object or
  - a primitive data type
- **FROM** specifies the domain to which the query applies and
- **WHERE** specifies constraints which restrict the result collection.

For example, if we have an `Account` entity with a `balance` field, we can issue the following query:

```
SELECT a FROM Account a WHERE a.balance > 0
```

#### 7.2.1 Result Collections in JPQL

A core difference between JPQL and SQL is that the result collection in JPQL will be a collection of references to one of

- entities,
- other Java objects which are expanded within the same table (embedded classes),

- Java primitives,
- new instances of Java result objects whose fields are populated from the query,

while in SQL the result is a new conceptual table with column entries sourced potentially from different tables, i.e. it can contain elements from different tables and hence elements extracted from different entities.

Ultimately the result collection will be either a standard `java.util.Collection` or a `java.util.Set`.

### 7.2.2 Selecting entity Attributes

We can use the element access operator to select specific attributes of an entity. For example

```
SELECT a.balance FROM Account a
```

returns a collection of all account balances. In this case the `Object(..)` phrase is dropped. EJB QL specification requires that you wrap your result with an `Object()` phrase only in that case where a stand-alone variable is returned without navigating along a path.

## 7.3 JPQL statement types

The JPQL is syntactically similar to the Standard Query Language (SQL) in that it supports 3 types of statements:

- **Select statements** Select statements are used access to selected data in persistent storage,
- **Update statements** Update statements are used to modify information maintained in persistent storage.
- **Delete statements** Delete statements are used to remove information currently held in persistence storage.

### 7.3.1 Elements of query statements

The elements of a `select` statement are

- a `select clause` which determines the type of the objects or values returned (in JPQL the result set is always a collection of objects or values), where the objects are either retrieved entities which match the query or new objects whose fields were populated from the query,
- a `FROM clause` which constrains the domain from which the selection is done,
- an optional `WHERE clause` which may be used to constrain the collection of objects selected from that domain,
- an optional `GROUP BY clause` which enables one to group query results in terms of groups,
- an optional `HAVING clause` used in conjunction with the GROUP BY clause in order to filter over aggregated groups, and
- an optional `ORDER BY clause` enabling one to request an ordering from the returned result objects/values.

### 7.3.2 Elements of Update and delete statements

The update and delete statements contain only the `UPDATE/DELETE clause` and an optional `WHERE clause`.

Update/Delete exists for greater performance and scalability in certain use-cases, but have their limitations. Entities retrieved from persistent storage before issuing the update will not have updated state until the next time they retrieved or refreshed via the EntityManager.

## 7.4 Polymorphism

JPQL statements are in essence polymorphic in that all statement elements which apply to a target bean also apply to all its specializations.

## 7.5 Navigating object graphs

Before we can discuss navigation through object graphs, we have to discuss single and multi-valued paths.

### 7.5.1 Simple paths

Assume we have a Bond entity bean which has the structure shown in Figure 7.1. One of the strengths of JPQL is its ability to smoothly navigate across relationships, i.e. through object graphs. Consider, as an example, the UML diagram for a bond shown in Figure 7.1

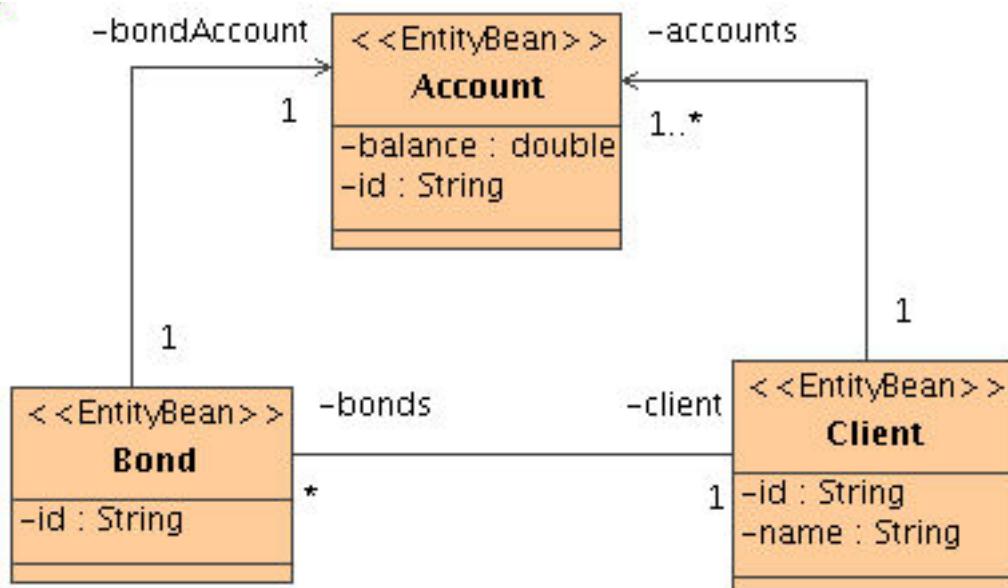


Figure 7.1: UML class diagram for a bond

In JPQL one can traverse relationships in an object-oriented fashion using the Java element access operator. For example, we could specify the following SELECT statement to select all bond accounts

```
SELECT b.bondAccount FROM Bond b
```

returns a collection of accounts, each of which is a bond account. The Bond entity bean must supply an abstract accessor method to query the related bond account. The equivalent SQL statement would look something like this

```
SELECT account from Account, Bond
WHERE Bond.bondAccount = Account.id
```

Our query may span multiple nodes like in

```
SELECT bond.bondAccount.balance FROM Bond bond
```

### 7.5.2 Single-Valued versus Multi-Valued Paths

A single valued path is a path without any branching below the highest layer (i.e. the layer connected to the result objects). SELECT clauses and most WHERE clauses require a single-valued path.

For example, all the queries discussed in the previous section use single valued paths in the SELECT statement and are hence valid JPQL statements. On the other hand, querying all the bond accounts of all the clients via

```
SELECT client.bonds.account FROM Client client      -> INVALID
```

resembles a multi-valued path because `c` refers to a collection of clients each of which has a collection of bonds which each has an account.

As a second example, consider the UML diagram for a course schedule shown in figure Figure 7.2. In a relational database this object graph could be represented as 4 tables, one for each entity.

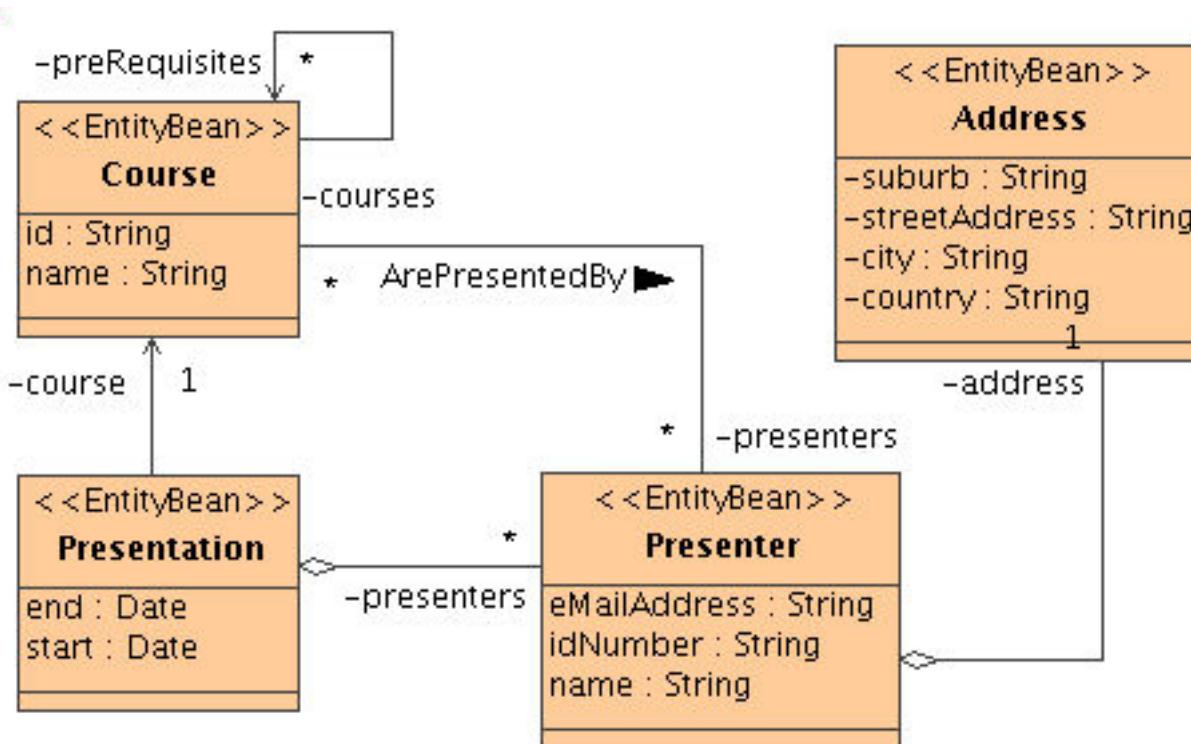


Figure 7.2: UML class diagram for presentations of courses for a course schedule

If we wanted to extract all course names which are currently scheduled i.e. for which there exists a presentation), we could do this via the following SQL query:

```
SELECT Course.name from Course, Presentation
WHERE Presentation.course = Course.id
```

To achieve the same in JPQL we can specify the following query:

```
SELECT p.course.name FROM Presentation p
```

This is a single-valued path and hence the query is valid. On the other hand, the query

```
SELECT p.presenters.course.name FROM Presentation p      -> INVALID
```

is incorrect because we have, once again, a multi-valued path.

## 7.6 Specifying the source of a query

The FROM clause specifies and constrains the domain of the query by specifying the domain as a particular entity type.

### 7.6.1 Selecting from multiple domains

The FROM clause supports selecting from multiple domains delimited by commas.

For example, should you wish to find all election results which had a greater turnout than South Africa's 1994 election you could specify the following query:

```
SELECT DISTINCT election
FROM Election election, Election election94
WHERE election.turnout > election94.turnout AND
      election.country = 'South Africa' AND
      election94.year = '1994'
```

### 7.6.2 Joins

Joins combine

#### 7.6.2.1 Inner joins

Inner joins are used to select from an inclusion set obtained by a join condition over different objects. They can be specified implicitly via the cartesian product or explicitly.

##### 7.6.2.1.1 Implicit inner joins

Implicit inner joins join multiple paths from multiple entities implicitly. For example, the following query performs an implicit inner join to determine those companies which are both customers and service providers:

```
SELECT DISTINCT c
FROM Customer c, ServiceProvider sp
WHERE c.companyRegistrationNo = sp.id
```

##### 7.6.2.1.2 Explicit inner joins

The following explicit INNER JOIN returns a collection of all bond accounts of clients living in Johannesburg:

```
SELECT bonds.account
FROM Client c INNER JOIN c.bonds bonds
WHERE c.address.city = 'Johannesburg'
```

Here the INNER JOIN can be abbreviated to JOIN (INNER is optional).

This is equivalent to

```
SELECT bonds.account
FROM Client c IN (c.bonds) bonds
WHERE c.address.city = 'Johannesburg'
```

### 7.6.2.2 Left outer joins

While an inner join retrieves only those objects which satisfy the join condition, an outer join does the same thing but with the addition of returning objects from the left collection for which there were no matching objects in the right collection.

For example, assume we have a one-to-zeroOrOne relationship between book and publisher, i.e a book may or may not be published by a publisher. Now assume you want to retrieve all information about all books and load the publishers for those books which have publishers into the cache. We thus use an outer join to retrieve the set of entities where matching values in the join condition may be absent.

```
SELECT b FROM Book b LEFT JOIN b.publisher p WHERE p.address.country = South Africa
```

gets all books, irrespective of whether they do or do not have publishers and also loads all those publishers of books who are in South Africa into the cache.

## 7.7 Collapsing Multi-Valued Paths into Single-Valued Paths via Collection Variables

SELECT clauses are restricted to single-valued paths. The same is largely true for WHERE clauses. So, how do we handle queries along multi-valued paths?

In JPQL this is done by defining collection variables via an IN clause. Consider, for example the invalid query

```
SELECT client.bonds.account FROM Client client      -> INVALID
```

The correct form of this query in JPQL is

```
SELECT bonds.account FROM Client c, IN(c.bonds) bonds      -> VALID
```

Here the IN-clause defines a collection variable, bonds, which, for each client, resembles the client's bonds.

In a similar way we can fix the following invalid EJB-QL statement

```
SELECT p.presenters.course.name FROM Presentation p      -> INVALID
```

by defining a collection variable, ps , via an IN clause

```
SELECT ps.course.name FROM Presentation p, IN(p.presenters) ps      -> VALID
```

## 7.8 Constraining a result set via a WHERE clause

Analogous to SQL, EJB-QL uses a WHERE clause to restrict the elements returned in the result collection. For example, we can select only those courses to which one or more presenters have been allocated via

```
SELECT Object(c)
  FROM Course c
 WHERE c.presenters NOT EMPTY
```

### 7.8.1 Comparison operators which can be used in WHERE clauses

JPQL supports a relatively extensive set of comparison operators which can be used in where clauses:

- =, <, >, <=, >=, <>
- BETWEEN, LIKE, IN, IS NULL, EMPTY, MEMBER OF which can all be inverted by combining them with a NOT.

For example

```
SELECT a FROM album a WHERE a.year NOT BETWEEN 1980 AND 2005  
SELECT s FROM soccer_club s where s.home.city IN ('London', 'Madrid', 'Rio de Janeiro')  
select c FROM customer c WHERE c.email LIKE '%ac.za'
```

### 7.8.2 Calculation and string operators

JPQL supports the following calculation and string operators to be used within WHERE and HAVING clauses:

- **Calculation operators** MAX, MIN, SUM, AVG, ABS, COUNT, SQRT
- **String operators** LENGTH, SUBSTRING, UPPER, LOWER, CONCAT

### 7.8.3 Using collection variables in WHERE clauses

We often have to define collection variables for multi-valued path constraints in WHERE clauses. For example

```
SELECT Object(c)  
  FROM Course c  
 WHERE c.prerequisites.name = 'Programming in Java'
```

is invalid because of the match against a multi-valued path, while

```
SELECT Object(c)  
  FROM Course c, IN(c.prerequisites) p  
 WHERE p.name = 'Programming in Java'
```

## 7.9 Constructing result objects

Even though the result of any JPQL query is always a single object/primitive or a collection of objects/primitives, the language provides a mechanism to assemble a new objects from queries via SELECT NEW

```
SELECT NEW za.co.academics.UniversityInfo (u.name, u.address, c.name c.registeredStudents)  
  FROM University u JOIN u.course c WHERE c.registeredStudents > 100
```

creates a list of result objects which are populated from elements of the university and course entities.

## 7.10 Nested queries

JPQL supports nested queries, i.e. queries which have sub-queries embedded within the conditional expression of a WHERE or HAVING clause.

For example, to select the best student on a course, one could use

```
SELECT s FROM student s where s.courseResults.average = (SELECT MAX(s.courseResults.average ←  
 ) from student s)
```

## 7.11 Ordering

One can use the ORDER BY clause followed by either ASC or DESC to request that the result set should be ordered in ascending or descending order of some field.

For example, to return a list of soccer stadiums which can seat at least 10 000 spectators in the order of the number of spectators they can accommodate, one can use

```
SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY numSEATS DESC
```

To refine the sort order, one can use multiple commands separated sort criteria which will be applied in the order in which they are defined:

```
SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY s.numSEATS DESC, s.age ASC
```

## 7.12 Grouping

The GROUP BY construct enables the aggregation of values according to a set of properties. The HAVING construct enables conditions to be specified that further restrict the query result.

For example,

```
SELECT j.publisher, count(j.circulation)
FROM journal j
GROUP BY j.publisher
HAVING COUNT(j.circulation) > 100000
```

selects all journals with a circulation of more than 100000

---

**Note**

The expression which appears in the GROUP BY clause must appear in the SELECT clause.

---

## 7.13 The Java Persistence Query Language (JPQL)

One may specify query inputs either as positional or as named parameters. The query input can only be used in the WHERE clause or HAVING clause of a query.

### 7.13.1 Positional parameters

Positional parameters are specified with a question mark (?) prefix followed by an integer designating the position of the parameter. Input parameters are automatically numbered, starting from 1. The same parameter can be used multiple times within a the same query.

### 7.13.2 Named parameters

Named parameters are case sensitive and their identifier is prefixed by the ":" symbol. For example

```
SELECT sf FROM soccerFixture sf WHERE (sf.date >= :date1) AND (sf.date <= :date2)
```

## Chapter 8

# Constructing and executing queries

Queries are constructed in JPQL via the entity manager and are executed by requesting a result list:

```
List<Product> products = entityManager.createQuery
    ("SELECT p FROM Product p WHERE p.description like :descr")
    .setParameter("descr", description)
    .setMaxResults(30)
    .setFirstResult(pageNo*30)
    .getResultList();
```

## 8.1 Named queries

Named queries are annotated with a @NamedQuery

```
@NamedQuery(name="bonds.getAllAbove"
    query="select b from Bond b where b.balance >= :amount")
```

Instances of named queries are created via

```
Query query = entityManager.createNamedQuery("bonds.getAllAbove");
query.setParameter(0, new Double(500000));
List<Bond> list = (List<Bond>)query.getResultList() ;
```

## **Part III**

# **Enterprise beans**

# Chapter 9

## Session Beans

### 9.1 Introduction

In EJB 3, session beans are the only beans which can be directly accessed by the client. They are typically used to publish a services facade to clients and to manage workflow and workflow state for the client session.

There are stateful and stateless session beans. Both exist thus only for the duration of a session. However, while stateful session beans maintain session state across service requests, stateless do not. The latter incur less overhead across activation/passivation, i.e. they exist solely for performance reasons.

### 9.2 Business Interfaces

An EJB container is essentially a business logic component container, and as such it supports component based software development.

#### 9.2.1 What is a component?

A component is a reusable and deployable object realising a *contract* defined by

1. an interface publishing the services offered by the component and the messages through which these services are requested,
2. the preconditions which must be satisfied before the service provider is willing to provide the service, and
3. the post-conditions (i.e. the deliverables) which are going to be provided by the service provider upon successful completion of the service.

#### 9.2.2 EJBs as business logic components

Session Beans can expose two interfaces (which are often the same), the Local interface, and the Remote interface. The only way to access a Session Bean is through one of these interfaces; we are thus forced into component-based development as the interface enforces a contract-driven approach with precondition specification (via exceptions) and the loose coupling that results from this approach.

#### 9.2.3 Remote interfaces

A remote interface provides access to an enterprise bean from outside the run-time environment in which the bean itself is deployed. This enables remote clients to directly access the business services and processes offered by session beans.

Remote clients can be client applications or applets, servlets deployed in a servlet container running in a separate Java run-time environment or other enterprise beans deployed in another EJB container (potentially deployed within the same cluster).

### 9.2.3.1 Defining a remote interface

There are two ways of specifying which interface is the `Remote` interface for a session bean: Either by classifying the interface itself, or by referring to an interface from within the bean, and indicating the role this interface plays (local or remote). In either case, the annotation `javax.ejb.Remote` is applied

In the former scenario, the remote interface is a standard Java interface which has been annotated with `@Remote`. For example

```
import javax.ejb.*;  
  
@Remote  
public interface OrderProcessorRemote  
{  
    public Invoice processOrder(Order order)  
        throws InsufficientFundsException;  
}
```

The Bean realising the interface may now simply state that it *implements* it:

```
import javax.ejb.*;  
  
@Stateless  
public class OrderProcessorBean implements OrderProcessorRemote  
{  
    ...  
}
```

In the latter scenario, the interface itself is unclassified/plain, which means it is not intrinsically tied to realisation within EJB:

```
public interface OrderProcessor  
{  
    public Invoice processOrder(Order order)  
        throws InsufficientFundsException;  
}
```

in which case the implementing bean must now contain the `Remote` annotation, indicating the `class` of the interface which should act as remote:

```
import javax.ejb.*;  
  
@Stateless  
@Remote({OrderProcessor.class})  
public class OrderProcessorBean implements OrderProcessor  
{  
    ...  
}
```

The latter approach is preferable, as the interface is now a pure contract, which could even be realised using technology other than EJB, without affecting existing clients.

### 9.2.3.2 A bean realising the business contract

Remote or external clients access session beans via the remote interface. These can be *stateless* or *stateful*. In either case the bean may implement a remote or a local interface, or both:

```
import javax.ejb.*;  
  
@Stateless  
@Remote({OrderProcessor.class})  
@Local({OrderProcessor.class})  
public class OrderProcessorBean implements OrderProcessor
```

```
{  
    public Invoice processOrder(Order order)  
        throws InsufficientFundsException  
    {  
        ...  
    }  
}
```

### 9.2.3.3 Access path when using a remote interface

Service requests provided via the remote interface go through a number of steps:

1. Remote clients will use the bean services via a local stub (which may be a RMI or a CORBA stub).
2. The client stub will marshall the service request onto IIOP (or RMI/IIOP in the case of a Java client).
3. An RMI server tie will de-marshal the IIOP message and map it onto a call onto the EJBObject.
4. EJBObject provides the interception layer which
  - associates a physical bean instance with the conceptual user object,
  - applies enterprise services like security and transaction (e.g. checking whether the user has the security roles required to access the requested service, starting a transaction and enlisting resources within that transaction, ...)
  - delegates the business logic responsibilities to the bean instance, and
  - applies further enterprise services upon completion (e.g. commit a transaction on successful completion or roll it back on failure).
5. The server skeleton will then marshal any responses (return values or exceptions) back onto the IIOP stream.
6. The client stub will de-marshal the response and pass it on to the client object.

## 9.2.4 Local interfaces

A local interface provides access to an enterprise bean from within the same run-time environment in which the bean itself is deployed. This enables local clients like other enterprise beans and servlets deployed within a servlet container running in the run-time environment of the application server to obtain efficient access to the bean instance, bypassing the marshaling of the request onto an IIOP message.

### 9.2.4.1 Defining a local interface

As with remote interfaces, there are two ways of indicating that a particular interface should be used in the role of local interface. The first is to classify the interface itself with the @Local EJB annotation. For example

```
import javax.ejb.*;  
  
@Local  
public interface ShippingRequestProcessor  
{  
    public TrackingNumber ship(Order order);  
}
```

The second is to leave the interface unclassified, and classify the role the interface plays to the bean within the bean itself, using the Local annotation:

```
import javax.ejb.*;  
  
@Stateless  
@Local({ShippingRequestProcessor.class})  
public class ShippingBean implements ShippingRequestProcessor  
{  
    public TrackingNumber ship(Order order)  
    {  
        //...  
    }  
}
```

#### 9.2.4.2 Access path when using a local interface

Service requests provided via the local interface still need to be intercepted by the server-generated EJBObject, in order to still apply enterprise services. However, the service request messages need no longer be marshaled onto the RMI/IOP protocol:

1. Using JNDI, Local clients will obtain a direct reference to the EJBObject for your bean.
2. The EJBObject provides the interception layer which
  - associates a physical bean instance with the conceptual user object,
  - applies enterprise services like security and transaction (e.g. checking whether the user has the security roles required to access the requested service, starting a transaction and enlisting resources within that transaction, ...)
  - delegates the business logic responsibilities to the bean instance, and
  - applies further enterprise services upon completion (e.g. commit a transaction on successful completion or roll it back otherwise).
3. The client receives the response, if any, directly from the EJBObject.

---

#### Note

Bypassing the stub and skeleton and the message marshalling/demarshalling they perform will typically result in a significant performance benefit within the application server - for example accessing the beans from a local web application.

---

#### 9.2.5 Switching between local and remote interfaces

EJB does not support automatic switching between local and remote interfaces. The reason for this is that it would be unsafe, as the different parameter handling may result in changes in logic.

##### 9.2.5.1 Parameter handling in plain Java objects

Java only supports input parameters, i.e. they are copied from the client to the server upon service request and never back. Method parameters may be either primitives (like int, boolean or double) or object references. In either way they are copied to the client. In the case of an object reference, the client using that reference (without changing the value of the reference itself) will access the actual object and potentially change its state. It is through this mechanism that Java simulates output and input/output parameters.

##### 9.2.5.2 Parameter handling in remote Java objects (via RMI)

When accessing remote Java objects, primitive parameters are also treated as input arguments. For object parameters, there are, however, two scenarios.

---

1. If the argument is itself a remotely accessible object (i.e. a RMI server which implements `java.rmi.UnicastRemoteObject`, then the client will receive a remote reference through which the object itself can be manipulated.
2. Otherwise the class for that object must implement `java.io.Serializable` and the object will be serialized onto the RMI stream upon service request. The object will not be sent back upon service completion.

From the above we see that objects which are RMI servers are treated the same across local and remote interfaces, but other objects are not. Changing a local to a remote interface and vice versa is only safe if the client only requires read access.

---

**Note**

An effective way of enforcing this is by making the argument an interface which provides clients read-only access to the underlying object.

---

### 9.2.5.3 Providing access locally and remotely

A bean implementation, if deemed safe, may implement both a local and a remote interface. In this case one often uses interface extension to avoid code duplication and promote consistency:

```
public interface OrderProcessor
{
    public Invoice processOrder(Order order)
        throws InsufficientFundsException;
}
```

```
@javax.ejb.Local
public interface OrderProcessorLocal
    extends OrderProcessor {}
```

```
@javax.ejb.Remote
public interface OrderProcessorRemote
    extends OrderProcessor {}
```

```
@javax.ejb.Stateless
public class OrderProcessorBean
    implements OrderProcessorLocal, OrderProcessorRemote
{
    public Invoice processOrder(Order order)
        throws InsufficientFundsException
    {
        ...
    }
}
```

Again, if one does not want to classify the roles the interfaces play within the interfaces themselves, one may use the alternative method of performing the classification within the Bean implementation itself. In this scenario, you may still choose to implement different interfaces for remote and local access, but it is quite common to now refer to the same interface:

```
import javax.ejb.*;

@Local({OrderProcessor.class})
@Remote({OrderProcessor.class})
@Stateless
public class OrderProcessorBean implements OrderProcessor
{
```

```
public Invoice processOrder(Order order)
    throws InsufficientFundsException
{
    ...
}
```

## 9.2.6 Automatically generated business interfaces

In the case where the developer only writes the bean implementation class, EJB supports automatic generation of either a local or a remote interface (but not both.)

### 9.2.6.1 Automatic interface naming

Beans for which interfaces are to be generated automatically must be named with the one of the following suffixes:

```
***Bean, ***Impl, ***Implementation, ***EJB
```

The application server will generate an interface with the same name as the bean implementation class, but with the suffix dropped. Thus, the automatically generated business interface for `AccountBean` would be named `Account`.

### 9.2.6.2 Which services are published in the automatically generated interface?

The application server will, by default, publish all public methods (except those used by the application server for dependency injection.)

This default can, however, be overridden by annotating those methods which should appear in the interface with the `@BusinessMethod` annotation. As soon as any method is annotated using `@BusinessMethod`, all other non-annotated public methods will no longer be included in the application server generated business interface.

### 9.2.6.3 Automatically generated local interfaces

The application server will generate automatically a local interface for any session bean class which is not annotated as `@Remote`.

### 9.2.6.4 Automatically generated remote interfaces

To request an automatically generated remote interface, one must annotate the bean implementation class with the `@Remote` annotation. Note that the local interface will no longer be generated if a remote interface is requested, i.e. that you will have to define the interfaces yourself for any bean class for which you want *both* local and remote interfaces.

### 9.2.6.5 Automatically generated interfaces: A good idea?

Though a useful feature, if one follows a typical design process such as URDAD, the *contracts* are usually the first coding deliverable, which means that the Java interfaces should already be derived even before design decisions are made such as the types of components to be used (e.g. EJB).

In this spirit, it is usually a better idea to always write (or generate) your contracts as the result of the *design process*, and not by ‘reverse engineering’ them from an implementation.

## 9.3 A generic Ant build script for EJB projects

One can use a generic ant script which enforces a standard structure on EJB projects. This follows the *integrated* approach, where there is direct code-sharing at a software project level, and for example both the business layer and the presentation layer are packaged in one Enterprise Archive for deployment.

### 9.3.1 Source packaging

Our generic ant script enforces a packaging structure on the source repository which facilitates easy building and deployment of the various components of an EJB based system including

- the business logic layer,
- the server side presentation layer, and
- client applications.

To this end the ant script separates the client and server side source bases and enforces that artifacts which are common to both, client and server side are contained in a `common` directory. On the server side it differentiates between business logic layer and presentation layer components and defines `server/common` directory for those server side artifacts which are required by both, business logic layer and presentation layer. The directory structure specified in the ant script is illustrated in Figure 9.1.

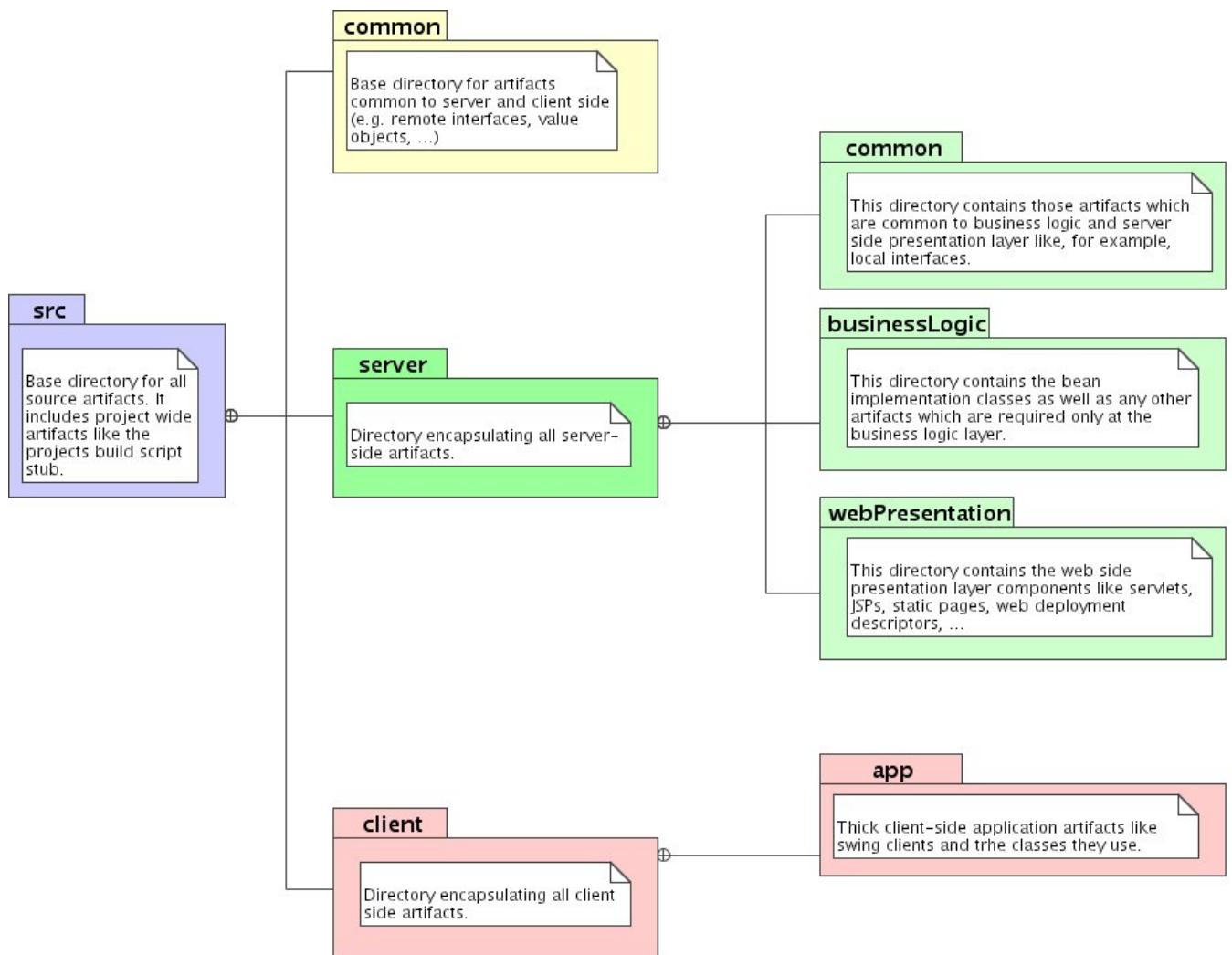


Figure 9.1: Directory structure for source artifacts of our EJB projects

### 9.3.2 Common targets

Common targets across EJB projects include

- compilation of any of the source units,
- archiving of the business logic layer, presentation layer and client application,
- generation of an enterprise archive which packages the server side business logic and presentation layers within a single enterprise archive,
- deployment of the presentation layer, business logic layer or the enterprise application as a whole, and
- running the client application.

### 9.3.3 The common ant build targets

```
<!-- GENERIC BUILD TARGETS FOR EJB3 PROJECTS -->
<!-- ===== -->

<property environment="env"/>

<!-- Archive name: -->
<property name="jar.server.businessLogic"
          value="${archive.name.base}.jar"/>
<property name="jar.client.app"
          value="${archive.name.base}Client.jar"/>
<property name="jar.server.webInterface"
          value="${archive.name.base}.war"/>
<property name="ear"
          value="${archive.name.base}.ear"/>

<!-- GENERIC PROJECT DIRECTORIES STRUCTURE -->
<!-- ===== -->

<!-- Shelf for project libraries -->
<property name="lib.dir" value="shelf" />

<!-- Source directories -->
<property name="src" value="src"/>
<property name="src.common"
          value="src/common" />
<property name="src.server"
          value="src/server" />
<property name="src.server.common"
          value="src/server/common" />
<property name="src.server.businessLogic"
          value="src/server/businessLogic" />
<property name="src.meta-inf"
          value="src/META-INF" />
<property name="src.server.webInterface"
          value="src/server/webInterface" />
<property name="src.client.app"
          value="src/client/app" />

<!-- Output directories -->
<property name="build.common"
          value="build/common" />
<property name="build.server.common"
          value="build/server/common" />
<property name="build.server.businessLogic"
          value="build/server/businessLogic" />
<property name="build.server.webInterface"
          value="build/server/webInterface" />
<property name="build.client.app"
```

```
    value="build/client/app" />

<property name="dist.dir" value="dist"/>

<property name="jboss.home" value="${env.JBOSS_HOME}"/>

<property name="jboss.server.config" value="all"/>

<property name="deploy.dir"
  value="${jboss.home}/server/${jboss.server.config}/deploy"/>

<!-- COMPILER ENVIRONMENT -->
<!-- ===== -->

<!-- Javac Compiler flags -->
<property name="compile.debug" value="true"/>
<property name="compile.optimize" value="false"/>
<property name="compile.deprecation" value="true"/>

<!-- Paths -->

<!-- shelf contains the libraries required at compile time -->
<path id="shelf">
  <fileset dir="../common/shelf">
    <include name="**/*.jar" />
    <include name="**/*.rar" />
  </fileset>
</path>

<!-- JBOSS libraries for executing client -->
<path id="libs.jboss">
  <fileset dir="../common/lib">
    <include name="jbossall-client.jar"/>
  </fileset>
</path>
<!--path id="libs.jboss">
  <fileset dir="${jboss.home}/lib">
    <include name="**/*.jar"/>
  </fileset>
  <fileset dir="${jboss.home}/server/${jboss.server.config}/lib">
    <include name="**/*.jar"/>
  </fileset>
  <fileset dir="${jboss.home}/server/${jboss.server.config}/deploy/ejb3.deployer">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${jboss.home}/server/${jboss.server.config}/deploy/jboss-aop-jdk50.deployer" -->
    <!--
      <include name="*.jar"/>
    </fileset>
</path-->

<path id="build.path">
  <path refid="shelf"/>
  <pathelement path="${build.common}"/>
  <pathelement path="${build.server.common}"/>
  <pathelement path="${build.server.businessLogic}"/>
  <pathelement path="${build.server.webInterface}"/>
  <pathelement path="${build.client.app}"/>
</path>

<!-- TARGETS -->
<!-- ===== -->
```

```
<!-- Removes all generated artifacts -->
<target name="clean">
  <delete dir="${build.common}" />
  <delete dir="${build.server.common}" />
  <delete dir="${build.server.businessLogic}" />
  <delete dir="${build.server.webInterface}" />
  <delete dir="${build.client.app}" />
  <delete dir="${dist.dir}" />
  <delete file="${deploy.dir}/jar.server.businessLogic" />
  <delete file="${deploy.dir}/jar.server.webInterface" />
</target>

<!-- Prepares the required directories for the build -->
<target name="prepare">
  <mkdir dir="${build.common}" />
  <mkdir dir="${build.server.common}" />
  <mkdir dir="${build.server.businessLogic}" />
  <mkdir dir="${build.server.webInterface}" />
  <mkdir dir="${build.client.app}" />
  <mkdir dir="${dist.dir}" />
</target>

<!-- Compiles the files common to both, client and server side -->
<target name="compile.common" depends="prepare">
  <javac srcdir="${src.common}" destdir="${build.common}"
    deprecation="${compile.deprecation}"
    optimize="${compile.optimize}"
    debug="${compile.debug}">
    <classpath>
      <path refid="build.path"/>
    </classpath>
  </javac>
</target>

<!-- Compiles those files common to both, server side business logic
     and server side presentation layers. -->
<target name="compile.server.common" depends="compile.common">
  <javac srcdir="${src.server.common}"
    destdir="${build.server.common}"
    deprecation="${compile.deprecation}"
    optimize="${compile.optimize}"
    debug="${compile.debug}">
    <classpath>
      <path refid="build.path"/>
    </classpath>
  </javac>
</target>

<!-- Compiles the server side business logic layer -->
<target name="compile.server.businessLogic"
  depends="compile.server.common">
  <javac srcdir="${src.server.businessLogic}"
    destdir="${build.server.businessLogic}"
    deprecation="${compile.deprecation}"
    optimize="${compile.optimize}"
    debug="${compile.debug}">
    <classpath>
      <path refid="build.path"/>
    </classpath>
  </javac>
</target>
```

```
<!-- Compiles the client application -->
<target name="compile.client.app" depends="compile.common">
    <javac srcdir="${src.client.app}" destdir="${build.client.app}"
        deprecation="${compile.deprecation}"
        optimize="${compile.optimize}"
        debug="${compile.debug}">
        <classpath>
            <path refid="build.path"/>
        </classpath>
    </javac>
</target>

<!-- Compiles the server side presentation layer -->
<target name="compile.server.webInterface"
    depends="compile.server.common">
    <javac srcdir="${src.server.webInterface}"
        destdir="${build.server.webInterface}">
        <classpath>
            <path refid="build.path"/>
        </classpath>
    </javac>
</target>

<!-- Generates the Java archive for the server side business logic layer -->
<target name="jar.server.businessLogic"
    depends="compile.server.businessLogic">
    <delete file="${dist.dir}/${jar.server.businessLogic}"/>
    <zip zipfile="${dist.dir}/${jar.server.businessLogic}">
        <zipfileset dir="${build.common}"
            includes="**/*.class"/>
        <zipfileset dir="${build.server.common}"
            includes="**/*.class"/>
        <zipfileset dir="${build.server.businessLogic}"
            includes="**/*.class"/>
        <zipfileset dir="${src.meta-inf}"
            prefix="META-INF"
            includes="*.xml"/>
        <zipfileset dir="${src.server.businessLogic}"
            includes="*.properties"/>
    </zip>
</target>

<!-- Generates the Web archive containing the server side presentation layer -->
<target name="jar.server.webInterface"
    depends="compile.server.webInterface">
    <delete file="${dist.dir}/${jar.server.webInterface}"/>
    <zip zipfile="${dist.dir}/${jar.server.webInterface}">
        <zipfileset dir="${build.common}"
            prefix="WEB-INF/classes">
            <include name="**/*.class"/>
        </zipfileset>
        <zipfileset dir="${build.server.common}"
            prefix="WEB-INF/classes">
            <include name="**/*.class"/>
        </zipfileset>
        <zipfileset dir="${build.server.webInterface}"
            prefix="WEB-INF/classes">
            <include name="**/*.class"/>
        </zipfileset>
        <zipfileset dir="${src.server.webInterface}/java"
            <include name="*.jsp"/>
```

```
<include name="*.jspx"/>
</zipfileset>
<zipfileset dir="${src.server.webInterface}"
    prefix="WEB-INF">
    <include name="*.xml"/>
</zipfileset>
</zip>
</target>

<!-- Creates the enterprise archive containing both,
    server-side presentation and business logic layers -->
<target name="ear"
    depends="jar.server.businessLogic,jar.server.webInterface">
    <delete file="${dist.dir}/${ear}"/>
    <jar jarfile="${dist.dir}/${ear}">
        <fileset dir="${dist.dir}"
            includes="${jar.server.businessLogic},${jar.server.webInterface}"/>
        <fileset dir="${src.server}"
            includes="META-INF/application.xml"/>
    </jar>
</target>

<!-- Archives the client application -->
<target name="jar.client.app" depends="compile.client.app">
    <delete file="${dist.dir}/${jar.client.app}"/>
    <jar jarfile="${dist.dir}/${jar.client.app}">
        <fileset dir="${build.common}" includes="**/*.class"/>
        <fileset dir="${build.client.app}" includes="**/*.class"/>
        <fileset dir="${src.client.app}" includes="*.properties"/>
    </jar>
</target>

<target name="deploy.server.businessLogic"
    depends="jar.server.businessLogic">
    <copy file="${dist.dir}/${jar.server.businessLogic}"
        todir="${deploy.dir}" />
</target>

<target name="deploy.server.webInterface"
    depends="jar.server.webInterface">
    <copy file="${dist.dir}/${jar.server.webInterface}"
        todir="${deploy.dir}" />
</target>

<target name="deploy.ear" depends="ear">
    <copy file="${dist.dir}/${ear}"
        todir="${deploy.dir}" />
</target>

<target name="undeploy.server">
    <delete file="${deploy.dir}/${jar.server.businessLogic}" />
    <delete file="${deploy.dir}/${jar.server.webInterface}" />
    <delete file="${deploy.dir}/${ear}" />
</target>

<target name="run.client.app">
    <echo message="Running the ${client.app} application:" />
    <java fork="on" classname="${client.app}" >
        <classpath>
            <pathelement location="${dist.dir}/${jar.client.app}"/>
            <path refid="libs.jboss"/>
        </classpath>
    </java>
</target>
```

```
</java>
</target>
```

This file may be included (e.g. using the XML Entities mechanism) into any other ant build script.

### 9.3.4 Location of projects, common ant script and common libraries

All project directories must be at the same level in the directory hierarchy. A common directory containing the `buildCommon` script as well as a `shelf` directory with all libraries common across projects is placed at the same level as the project directories.

## 9.4 Stateless session beans

Stateless session beans provide users a services (functional) interface without maintaining state across service requests.

Stateless session beans exist for the duration of a user session (that is the conceptual/virtual user object -- the physical bean object continues existing in the object pool).

### 9.4.1 Stateless session beans as a services facade

Since stateless session beans maintain no state, they do not as such manage a workflow. Instead they are usually used to provide a higher-level services facade onto an underlying object-oriented system.

### 9.4.2 Life cycle of a stateless session bean

Stateless session beans do not maintain state across service requests. That makes it possible to use a very simple mapping between the user and the physical object from the bean pool.

For every service request the application server activates any bean from the bean pool by binding the bean instance to the user session bean object. After having applied the enterprise services, the application server delegates the realisation of that request to the bean instance.

Upon completion the bean instance is committed back to the method ready pool. The next request made on the virtual user object will be, once again, be delegated to any available physical object from the pool. This process is particularly light-weight as no state needs to be maintained across service requests.

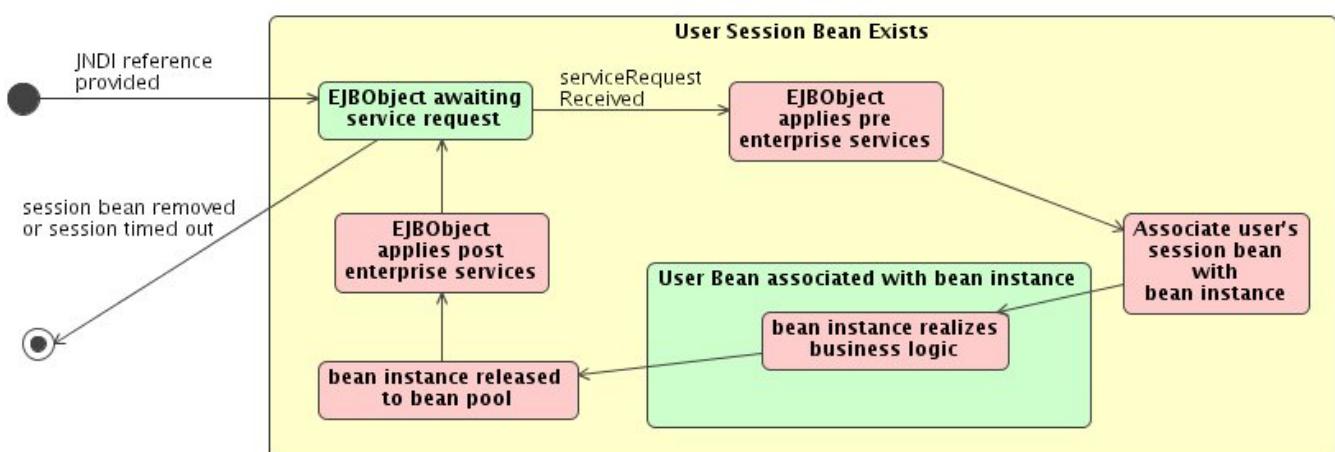


Figure 9.2: Life cycle of a stateless session bean

The figure above illustrates the life cycle of a stateless session bean including the binding and unbinding of the bean instance to/from the user's virtual session bean, i.e. the usage of few physical objects for many conceptual objects.

### 9.4.3 Writing the business interfaces

We shall write a simple average calculating bean which receives a collection of floating point numbers as parameter. Since the parameter is only an input parameter which is not modified within the bean body, we can safely provide both, local and remote interfaces. These will extend a plain Java interface which encapsulates the commonalities across the local and remote interfaces.

#### 9.4.3.1 Average.java

```
package za.co.solms.average;

import java.util.Collection;

/**
 * A interface for java classes calculating the average of
 * a collection of floating point numbers.
 */
public interface Average
{
    public double average(Collection<Double> data);
}
```

#### 9.4.3.2 AverageRemote.java

The remote interface is packaged within the `common` source directory because it must be available for both, client and server. It simply extends the plain Java interface and adds the `@Remote` annotation:

```
package za.co.solms.average;

import javax.ejb.Remote;

/**
 * A remote interface for session beans calculating
 * the average of a collection of floating point numbers.
 */
@Remote
public interface AverageRemote extends Average {}
```

#### 9.4.3.3 AverageLocal.java

In order to enable local components (such as servlets running in the embedded web container) to access the enterprise bean through the local interface, we package the local interface in the `server/common` directory. As with the remote interface, the local interface also simply extends the plain Java interface and adds the `@Local` annotation:

```
package za.co.solms.average;

import javax.ejb.Local;

/**
 * A local interface to a session beans calculating the
 * average of a collection of floating point numbers.
 */
@Local
public interface AverageLocal extends Average {}
```

#### 9.4.3.4 AverageBean.java

The bean implementation class is a very simple class which contains only the business logic. Otherwise it is annotated as a stateless session bean and implements a local and a remote interface.

```
package za.co.solms.average;

import java.util.Collection;
import javax.ejb.Stateless;
import za.co.solms.average.AverageLocal;
import za.co.solms.average.AverageRemote;

/** A stateless session bean calculating the average of a
 * collection of floating point numbers. The bean is
 * accessible via local and remote interfaces.
 */
@Stateless
public class AverageBean
implements AverageRemote, AverageLocal
{
    public double average(Collection<Double> data)
    {
        if (data.size() == 0)
            return 0;

        double result = 0;
        for (Double d : data)
        {
            result += d;
        }
        return result / data.size();
    }
}
```

#### 9.4.3.5 Building and deploying the bean

To build and deploy the bean you only need to run the ant task, `deploy.server.businessLogic`. This will compile the common code, the server-side common code, the business logic layer code, create the Java archive, `Average.jar` and copy it to the JBoss deployment directory. For this to work you will need to have the `JBOSS_HOME` environment variable set.

## 9.5 A simple session bean client

The following illustrates a stand-alone Java client which will access the EJB across the network. Stand-alone clients are, to some degree, bound to the application server which hosts the EJB, as the client needs to be configured to connect to the specific (type and location) JNDI naming service of the application server in question. This limitation has caused the introduction of the *client container* in Java EE 5, a self-contained environment which is used to launch a client application which now no longer needs to contain such configuration information.

#### 9.5.1 Client.java

The client initialises its naming context and then looks up the application server generated remote object using the JNDI name for the remote interface. Once the reference to the remote session bean has been obtained, the client may use the business services offered by the session bean.

```
package za.co.solms.average;
```

```
import java.util.Collection;
import java.util.LinkedList;
import java.util.Random;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import za.co.solms.average.AverageRemote;

/**
 * Client for the stateless session bean Average via its
 * remote interface.
 */
public class Client
{
    public static void main(String args[])
    {
        Random rng = new Random();
        try
        {
            int numDataPoints = 10000;
            Collection<Double> data = new LinkedList<Double>();
            for (int i = 0; i < numDataPoints; ++i)
                data.add(new Double(rng.nextDouble() * 100));

            /*
             * Initialize the reference to the naming service using the
             * naming service specified in a jndi.properties file available
             * within the class path.
             */
            InitialContext ctx = new InitialContext();

            System.out.println("Got Naming Service");

            /*
             * Look up the bean's EJB remote object via JNDI via the default
             * JNDI name (the name of the remote interface) and cast the
             * received object reference to AverageRemote.
             */
            String jndiName = "AverageBean/remote";
            System.out.println("Now looking up "
                + jndiName+ " in JNDI tree.");
            AverageRemote avg
                = (AverageRemote) ctx.lookup(jndiName);

            System.out.println("Got remote handle to session bean");

            /*
             * Here we are calling the remote session bean:
             */
            double average = avg.average(data);

            System.out.println("The average is " + average);
        }
        catch (NamingException e)
        {
            System.out.println("Could not find bean with specified JNDI name.");
            e.printStackTrace();
        }
    }
}
```

### 9.5.2 Packaging the JNDI properties file with the client

```
# Configures the host and type of JNDI naming service to connect to
# when EJBs are looked up
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost
```

### 9.5.3 Running the client application

To build the client application you only need to run the ant-task, `jar.client.app`. This will compile the common and client code and archive them into `AverageClient.jar`. In addition to this the JBoss client-side library files need to be included in the classpath.

### 9.5.4 Providing a fully self-contained client

For the client to run on a remote machine one needs to make some of the EJB libraries available on the client side. One can package these within the client jar, but one needs to first unpack the library jars before inserting the library components into the client jar (because extraction from nested jars is not currently supported in Java).

## 9.6 A simple web client

Web clients are typically servlets and JSPs deployed in a web container. Most application servers provide a comprehensive J2EE solution by including both, an EJB container and a web container. JBoss, for example, comes packed with an embedded Tomcat web container.

### 9.6.1 Should servlets use local or remote interfaces?

If the servlet is running in an embedded web container, it may use local interfaces in order to reduce the communication overheads. This does, however, imply that moving the servlets container to another machine or wanting to support that service requests arising from a servlet deployed within a cluster should be processible by any machine in a cluster will result in having to make coding changes. Still, in many cases including certain clustering configurations, using local interfaces is preferable to using remote interfaces.

### 9.6.2 Client input form: average.jsp

The JSP encapsulates a view. It has a single entry field and a single button whose action command is associated with the controller implemented as a servlet:

```
<%@ page contentType="text/html; charset=ISO-8859-1" %>
<html>
    <title>Average Calculator</title>
    <body>
        <h1><center>Average Calculator</center></h1>
        <center>
            <form action="AverageCalculator" method="post">
                Data (space or comma separated):
                <input type="text" name="data" value="" size="80">
                <p>
                    <input type="submit" name="action" value="Average">
                </p>
            </form>
        </center>
    </body>
</html>
```

```
</form>
</center>
</p>
</body>
</html>
```

### 9.6.3 Client controller: AverageServlet.java

The controller

- demarshals the input fields,
- looks up a session bean instance via JNDI,
- delegates the business logic to the session bean, and
- delegates the responsibility of showing the result view to an appropriate JSP.

```
package za.co.solms.average;

import java.io.*;
import java.util.*;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.log4j.Logger;
import za.co.solms.average.*;

/**
 * A servlet client for the Average stateless session bean
 */
public class AverageServlet extends HttpServlet
{

    /**
     * Gets handle to bean and looks up bean from naming service.
     */
    public void init() throws ServletException
    {
        super.init();
        try
        {
            InitialContext ctx = new InitialContext();
            logger.info("Got handle to naming context.");
            averageCalculator
                = (AverageLocal)ctx.lookup("AverageBean/local");

            logger.info("Looked up average bean");
        }
        catch (NamingException e)
        {
            throw new RuntimeException(e);
        }
    }

    /**
     * Upon POST request, demarshalls the HTTP request parameters,
     * calls the session bean and generates the HTTP response.
     */
}
```

```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException
{
    /*
     * De-marshalling request:
     */
    String dataString = request.getParameter("data");
    Collection<Double> data = new LinkedList<Double>();
    StringTokenizer tokenizer
        = new StringTokenizer(dataString, " ,;");
    while (tokenizer.hasMoreTokens())
        data.add(new Double(tokenizer.nextToken()));

    /*
     * Delegating business logic to session bean:
     */
    Double result = averageCalculator.average(data);

    /*
     * Storing result in request context:
     */
    request.setAttribute("average", result);

    /*
     * Delegating response view to JSP:
     */
    RequestDispatcher dispatcher
        = getServletContext( ).getRequestDispatcher("/averageResult.jsp");
    dispatcher.forward(request, response);
}

private AverageLocal averageCalculator;
private Logger logger = Logger.getLogger(AverageServlet.class);
}
```

#### 9.6.4 Client result view: averageResult.jsp

The response JSP extracts the calculated value and shows it:

```
<%@ page contentType="text/html; charset=ISO-8859-1" %>
<html>

    <title>Averaging Result</title>

    <body>
        <h1><center>Averaging Result</center></h1>

        <p>
            The averag of <%= request.getParameter("data") %>
            is <%= request.getAttribute("average") %>.
        </p>

        <p>
            Press <a href="/average">here</a> to return to average calculator.
        </p>

    </body>
</html>
```

### 9.6.5 Web deployment descriptor

The web deployment descriptor specifies

- mappings between servlet names used in the deployment descriptor and the servlet implementation classes,
- the default (welcome) page for the web application, and
- mappings between the URLs used to access servlets and the corresponding servlet names.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <servlet>
        <servlet-name>AverageServlet</servlet-name>
        <servlet-class>
            za.co.solms.average.AverageServlet
        </servlet-class>
    </servlet>

    <welcome-file-list>
        <welcome-file>average.jsp</welcome-file>
    </welcome-file-list>

    <servlet-mapping>
        <servlet-name>AverageServlet</servlet-name>
        <url-pattern>/AverageCalculator</url-pattern>
    </servlet-mapping>

</web-app>
```

### 9.6.6 Using a web client

The application root URL is determined by the name of the war file, average.war. Thus using the URL

`http://localhost:8080/average`

will access the default page for the application which we specified to be `average.jsp`.

## 9.7 Enterprise application archives

The EJB specification supports the packaging of the business logic and presentation layers of an application in a single deployable unit, the enterprise application archive (EAR). The archive is a standard Java archive (jar) file which will typically contain

1. an enterprise java beans archive, a JAR file, for the business logic layer components,
2. a web archive archive, a WAR file, for the presentation layer components, and
3. an application deployment descriptor, `application.xml`.

### 9.7.1 The application deployment descriptor

An application archive packages multiple modules in a single unit. Typically this will include one or more presentation layer modules and one or more business logic layer modules. For each presentation layer module, one can specify the context root (base URI). This replaces the default context root provided by the name of the web archive file.

Since Java EE 1.4, all the deployment descriptors are based on XML Schema, which greatly simplifies editing and validation. The relevant schemas for the particular version of Java EE application being developed can always be found at <http://java.sun.com/xml/ns/javaee/>

```
<?xml version="1.0" encoding="UTF-8"?>
<application
    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/application_5.xsd"
    version="5">
    <display-name>Average EAR</display-name>
    <module>
        <web>
            <web-uri>average.war</web-uri>
            <context-root>/average</context-root>
        </web>
    </module>
    <module>
        <ejb>average.jar</ejb>
    </module>
</application>
```

## 9.8 Stateful session beans

Stateful session beans maintain state across service requests. The state services only for the duration of the session and is only accessible from within the user session.

### 9.8.1 Stateful session beans as workflow managers

Since stateful session beans exist and maintain state for the duration of a user session, they may be used to as workflow managers.

### 9.8.2 Life cycle of a stateful session bean

Stateful session beans maintain state for the duration of the user session across bean passivation and re-activation. That requires that the state of the bean's object identity and state needs to be persisted upon bean passivation (i.e. when the bean is released to the bean pool) and reconstructed upon re-activation.

For every service request the application server activates any bean from the method ready pool by binding the flyweight object to the user session bean object. After having applied the enterprise services, the application server delegates the realisation of that request to the flyweight.

The following image illustrates the life cycle of a stateful session bean including the passivation and re-activation of the physical bean instance.

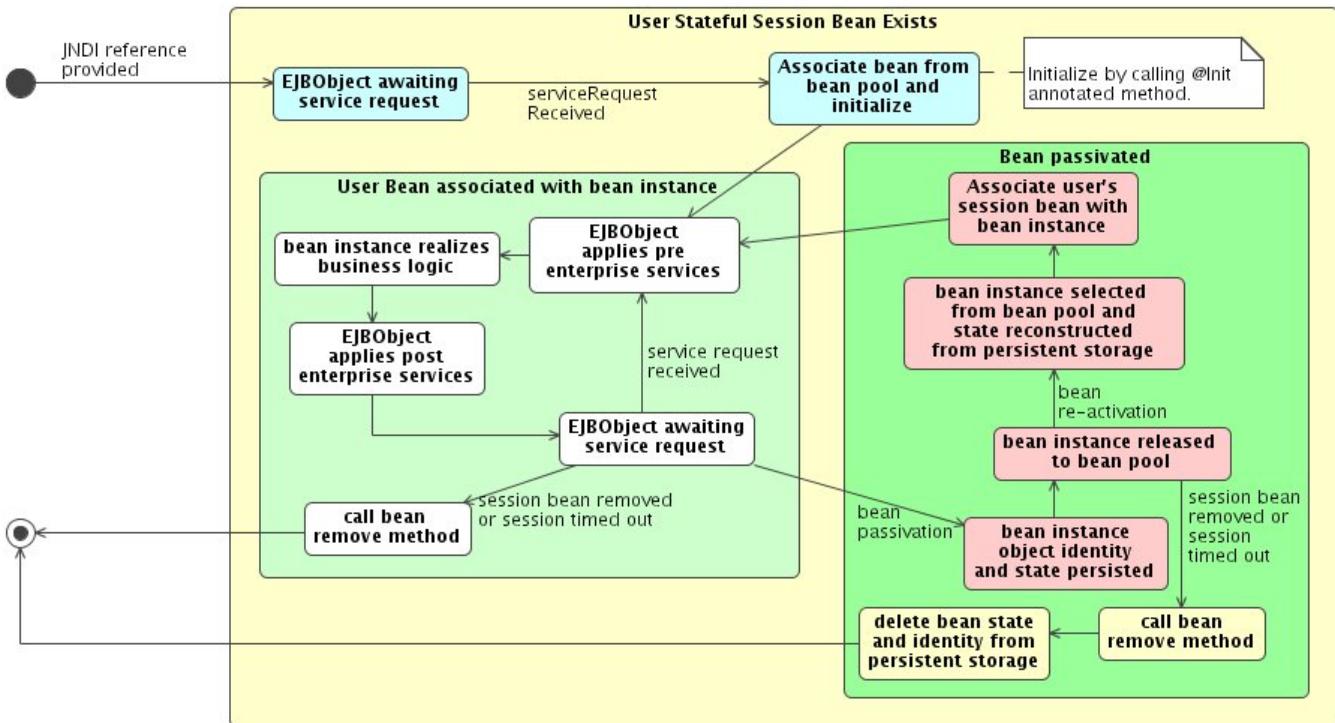


Figure 9.3: Life cycle of a stateful session bean

#### 9.8.2.1 Stateful session beans are realised using flyweight and memento patterns

In order to facilitate the representation of potentially many stateful session bean objects by a limited number of physical bean instances, EJB makes use of the flyweight pattern. Object identity and state is externalised; the EJB-Object maintains an external identity for the conceptual user session bean which may be realised by different physical bean instances over time and the persistent storage maintains the beans state externally.

Object serialisation is used to grab a memento (the encapsulated state of the bean instance) without breaking encapsulation as the serialised state can only be used to reconstruct the bean instance state.

#### 9.8.3 Writing the business interfaces

As an example, let us implement a stateful shopping cart session bean which has only a remote interface.

##### 9.8.3.1 ShoppingCart.java

```

package za.co.solms.ejb3.shopping.cart;

import java.util.Collection;
import za.co.solms.ejb3.shopping.products.Product;
import javax.ejb.Remote;

/** Business interface for a shopping cart. */
@Remote
public interface ShoppingCart
{
    public void add(Product product, int quantity);

    public CartContent getCartContent();
}
  
```

```
public Collection<Product> getProductsList();

public void clear();
}
```

This interface must be available to both, server and client side and hence it is packed in the common source folder.

#### 9.8.3.2 Value objects: CartContent.java and Product.java

Similarly, the value objects transferred between the stateful session bean. Furthermore, the value object classes must be serializable so that they can be serialized across the RMI/IOP connection. The product class is very simple:

```
package za.co.solms.ejb3.shopping.cart;

import java.io.Serializable;
import java.text.DecimalFormat;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import org.apache.log4j.Logger;
import za.co.solms.ejb3.shopping.products.Product;

/**
 * A value object representing the content of a shopping cart,
 * i.e. entries of quantities of various products.
 */
public class CartContent implements Serializable
{
    public CartContent() {}

    public void add(Product product, int quantity)
    {
        if (entries.containsKey(product))
        {
            logger.info("Product already in cart: updating...");
            entries.get(product).add(quantity);
            if (entries.get(product).getValue() == 0)
                entries.remove(product);
        }
        else
        {
            logger.info("Product not yet in cart: adding ...");

            if (quantity > 0)
                entries.put(product, new Count(quantity));
        }
    }

    public void clear() {entries.clear();}

    public String toString()
    {
        String s = "Cart:\n";
        for (Product product : entries.keySet())
        {
            s += entries.get(product) + " of ";
            s += product + " at ";
            s += priceFormatter.format(product.getPrice()) + "\n";
        }
        return s;
    }
}
```

```
}

private class Count implements Serializable
{
    public Count() {}

    public Count(int num){value = num; }

    public void increment() {++value; }

    public void add(int n) {value += n; }

    public int getValue() {return value; }

    public String toString()
    {
        return Integer.toString(value);
    }

    private int value = 0;

    private static final long serialVersionUID = 87328763287632L;
}

private Map<Product,Count> entries =
    new HashMap<Product, Count>();

private static final DecimalFormat priceFormatter =
    new DecimalFormat("R#####0.00");

private static final Logger logger =
    Logger.getLogger(CartContent.class);
}
```

The cart content maintains a mapping of products onto number of instances included in the cart:

```
package za.co.solms.ejb3.shopping.products;

import java.io.Serializable;

/**
 * Encapsulates products which can be added
 * to a shopping cart.
 */
public class Product implements Serializable
{
    public Product(String name, double price)
    {
        this.name = name;
        this.price = price;
    }

    public String getName()
    {
        return name;
    }

    public double getPrice()
    {
        return price;
    }
}
```

```
public int hashCode()
{
    return name.hashCode();
}

public boolean equals(Object o)
{
    try
    {
        Product arg = (Product)o;
        return arg.getName().equals(name);
    }
    catch (ClassCastException e) {return false;}
}

public String toString() {return name;}
private String name;
private double price;
private static final long serialVersionUID = 314159269876342876L;
}
```

### 9.8.3.3 ShoppingCartBean.java

Stateful session bean implementations must be annotated as `@Stateful`. Furthermore, since object serialization is used for bean passivation and re-activation, the bean class must implement `java.io.Serializable`.

```
package za.co.solms.shopping.cart;

import java.util.Collection;
import java.util.LinkedList;
import za.co.solms.ejb3.shopping.cart.CartContent;
import za.co.solms.ejb3.shopping.cart.ShoppingCart;
import za.co.solms.ejb3.shopping.products.Product;
import javax.ejb.Stateful;

/**
 * The stateful session bean realisation for the shopping cart.
 */
@Stateful
public class ShoppingCartBean implements ShoppingCart
{
    public void add(Product product, int quantity)
    {
        cartContent.add(product, quantity);
    }

    public CartContent getCartContent()
    {
        return cartContent;
    }

    public void clear()
    {
        cartContent.clear();
    }

    public Collection<Product> getProductsList()
    {
        return productsList;
    }
}
```

```
}

private CartContent cartContent = new CartContent();

private static final Collection<Product> productsList =
    new LinkedList<Product>();

static
{
    productsList.add(new Product(
        "Programming in Java", 5950));
    productsList.add(new Product(
        "Business Analysis using UML", 5950));
    productsList.add(new Product(
        "Design Patterns", 4760));
    productsList.add(new Product(
        "Object-Oriented Analysis and Design using UML", 5950));
    productsList.add(new Product(
        "Architecture", 5950));
    productsList.add(new Product(
        "Enterprise Java Beans", 5950));
    productsList.add(new Product(
        "XML and Web Services via Java", 5950));
}
}
```

#### 9.8.3.4 Building and deploying the bean

To build and deploy the bean you only need to run the ant task, `deploy.server.businessLogic`.

## 9.9 Reacting to life cycle events

Because EJB represents a move from physical objects (as per Java SE) to virtual, pooled objects, the developer can no longer rely on the standard lifecycle events of Java objects (creation via the constructor, destruction via the garbage collector after calling `finalize()`).

### 9.9.1 The life cycle of standard Java objects

Recall that, in order to execute code in normal Java objects (such as object set-up or clean-up) when object are *created* or *destroyed*, one uses the constructor, as well as the `finalize` service:

```
public class Something
{
    public Something()
    {
        // This code runs right after the object has been created
        // ...
    }

    public void finalize()
    {
        // This code runs right before the object is destroyed
        // ...
    }
}
```

### 9.9.2 The life cycle of session beans

Recall that session beans are pooled, and that, from the user perspective, a session bean is *created* when a client looks it up (the session starts) and that it is destroyed as soon as the session ends. These events do not correspond with the physical creation or destruction of the object (via the constructor and `finalize()`) so we need different ‘hooks’ for these life-cycle events.

### 9.9.3 Callback listener methods

Any method in an enterprise bean (which takes no arguments, has no return value, and throws no checked exceptions) may be annotated with one of the following annotations (in the `javax.ejb` package), to have the container automatically call that method when the indicated event occurs:

- **@PostConstruct** called at the start of the session, at the point when the conceptual object (from the user’s perspective) is created. For EJB development, this effectively replaces the constructor as the place where any initial setup regarding the state of the object is performed. Recall that, since the application server will create and maintain a pool of session beans at its own will, the constructor no longer has any real meaning with regards to the set-up of the object’s state, and indeed, most enterprise beans have no more than a default constructor.
- **@PreDestroy** called at the end of the session, at the point when the conceptual object (from the user’s perspective) is destroyed. For EJB development, this effectively replaces the `finalize()` service as the point where any ‘last actions’ are performed before the object is garbage-collected. In the case of a session bean, however, the object is usually not garbage-collected, but merely returned to the method-ready pool.
- **@PostActivate** called directly after the bean has been removed from the method-ready pool, and bound to a user session. This may, of course, occur several times during a single session as the bean is re-used across several user sessions if necessary.
- **@PrePassivate** called directly before the bean is unbound from a particular user session, and placed back in the method-ready pool.

## 9.10 Asynchronous session bean services

In order to support processing of asynchronous service requests you could always make use of message driven beans. The client thread returns as soon as the message has been successfully delivered to the queue or topic.

As of EJB 3.1, there is support for session beans processing asynchronous requests without the need of transmitting a message via a queue or topic.

### 9.10.1 Why should one not create threads within code executed within an application server?

Of course, instead of making an asynchronous request via a message queue or via some other asynchronous support infrastructure, one could simply create a new thread and make the request from that newly spawned thread. This is, however, not allowed in code running within application servers.

The reasons why one should not create threads within code running in an application server are mainly three: resource management, security and issues around thread-specific storage:

- **Resource management** If a thread creates its own threads then it interferes with the resource management task of the application server. The application server manages processing resources by controlling the execution threads.
- **Security** When threads are spawned by users, it is difficult for the application server to protect itself from an effective (often unintentional) denial-of-service attack, where accidentally the creation of a large number of threads brings down the services offered by the application server.
- **Thread-specific storage issues** Application servers often use thread specific storage to propagate metadata with the request. This could include things like your session and transactions contexts, user principal and roles and so on. This vital information would not be available to threads spawned by the user.

### 9.10.2 Asynchronous processing of requests

For a simple asynchronous request the processing is done in the background. The application server assigns a separate, managed thread to the piece of work which needs to be executed asynchronously whilst the original thread returns, allowing the user code to perform other tasks in the mean time.

A service is declared an *asynchronous service* by annotating it with an `@asynchronous` annotation. For example, the `OrderProcessorBean` shown below provides a normal `processOrder` service which is processed synchronously (within the calling thread), but it makes use of an asynchronous `shipOrder` service. The `processOrder` service will typically return with the `processOrderResult` prior to the `shipOrder` service having been completed:

```
@Stateless
public class OrderProcessorBean
{
    @Asynchronous
    private void shipOrder(OrderShipmentRequest orderShipmentRequest)
    {
        ...
    }

    public ProcessOrderResult processOrder(ProcessOrderRequest processOrderRequest)
    {
        ...
        shipOrder(shipOrderRequest);

        return processOrderResult;
    }
}
```

### 9.10.3 Deferred synchronous processing of requests

The `@asynchronous` annotation can also be used for processing of deferred synchronous requests, i.e. for services which are requested asynchronously, but for which, at a later stage the return value needs to be obtained. In Java this is done by having the asynchronous service return a `Future` from which the result is obtained later stage.

Consider, for example, a process which needs to obtain prices for multiple service providers (e.g. airlines, hotels, ...) and then assemble an optimal itinerary for a client (optimized say first on price, then travel time and so on). In this case the process of booking an optimal itinerary would source multiple quotes through deferred synchronous requests made to different service providers, block until sufficient quotes have been obtained and then assemble the optimal itinerary. In such an example, the `provideServiceProviderQuote` service would be annotated as `@asynchronous` returning a `Future` (i.e. be a deferred synchronous service):

```
@Stateless
public class BookingServicesBean
{
    @asynchronous
    public Future<QuoteResult> provideQuote(QuoteRequest quoteRequest)
    {
        ...
    }

    public BookItineraryResult bookItinerary(BookItineraryRequest request)
    {

        ...
        Future<QuoteResult> futureQuoteResult = provideQuote(quoteRequest);
        ...
        // do some other things (eg request further quotes)
        ...
    }
}
```

```

        // Now block process until result is available
        QuoteResult result = futureQuoteResult.get();
        ...
    }
}

```

#### 9.10.4 Asynchronous beans

Asynchronous beans are beans for which all services are processed either asynchronously or in a deferred synchronous way, i.e. byt annotating a session bean as `@asynchronous` one specifies that all its services are asynchronous (the ones which return a future are deferred synchronous).

For example, we could annotate the `BookingServicesBean` as `@asynchronous`, thereby specifying that both, the `provideQuote` and `bookItinerary` services be asynchronous (or, in this case, deferred synchronous services):

```

@asynchronous
@Stateless
public class BookingServicesBean
{

    public Future<QuoteResult> provideQuote(QuoteRequest quoteRequest)
    {
        ...
    }

    public BookItineraryResult bookItinerary(BookItineraryRequest request)
    {

        ...
        Future<QuoteResult> futureQuoteResult = provideQuote(quoteRequest);
        ...
        // do some other things (eg request further quotes)
        ...
        // Now block process until result is available
        QuoteResult result = futureQuoteResult.get();
        ...
    }
}

```

### 9.11 A web adapter keeping the control of business processes in the business logic layer

#### 9.11.1 Introduction

In a normal URDAD-based technology neutral process specification a user requests a service and the controller assembles a business processes from services sourced from different service providers (realizing appropriate contracts) as well as potentially from services sourced from the user itself.

When processing an order, the customer might have to accept a quote. So, in that case, the `processOrder` service would be assembled from lower level services sourced from Finance, StockManagement, Deliveries as well as the `acceptQuote` service which is requested from the client. Note that you cannot return for the process order service with a quote. The customer did not ask you to request a quote, but to process an order and you can only return once that service has been provided. Instead, provide quote needs to be a call-back to the customer.

In an URDAD based design this would be cleanly done with the understanding that you have the appropriate adapters to each of the concrete service providers you are ultimately using as well as an adapter to your customer. The customer could request the

service from their systems via a web service or from a human user interface which could be a application or web client. For each integration channel you would have the appropriate adapter.

In the case of a web client one is facing the technical challenge that you cannot make a call-back to the processor. You can only respond with a HTTP response, not with a request. The adapter thus needs to deliver the acceptQuote request as a return, i.e. we need to decouple the technicality of making a call or a return from the conceptual level where we are making a request or a response. The infrastructure needs to thus be able to send both, a request or a response, as either a call or a return.

## 9.11.2 A Call-Return mapping adapter

For web front ends as well as any other front-end where you either cannot or do not want to make an active call-back to the access layer you require an adapter which maps call to the client onto a response send to the client and a call from the client onto a return to the service controller.

### 9.11.2.1 Implementations of Call-Return mapping adapter

Here we discuss some concrete implementations of call-return mapping adapters.

#### 9.11.2.1.1 JavaEE implementation of a Call-Return mapping adapter

The minimalistic test example is a service provider who offers a service doX which requires at some stage the user to provideA. The conceptual workflow is shown below:

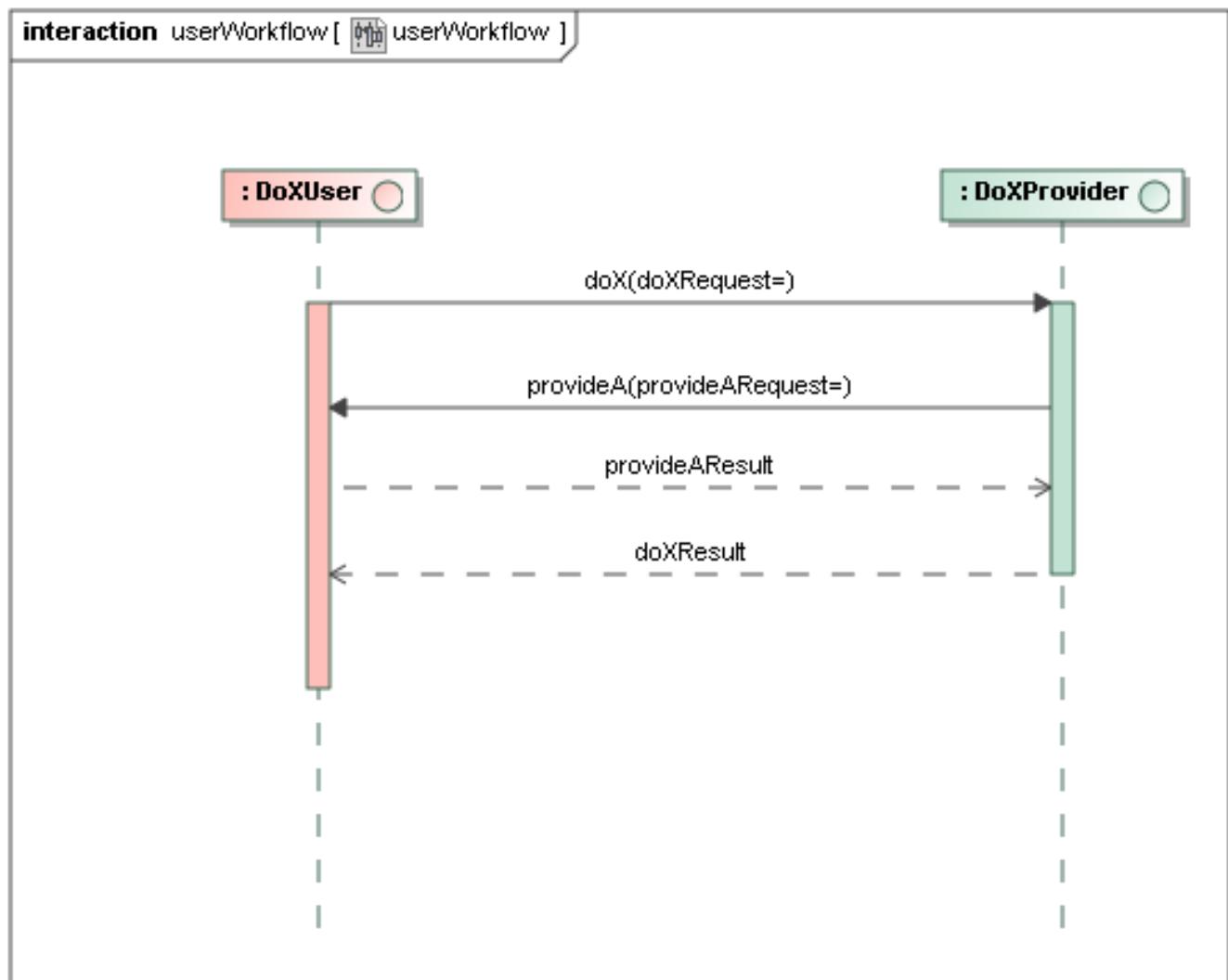


Figure 9.4: Workflow as per technology neutral design (URDAD)

Because we cannot make a call-back when having a web front-end, we need to map the request back to the client onto a return:

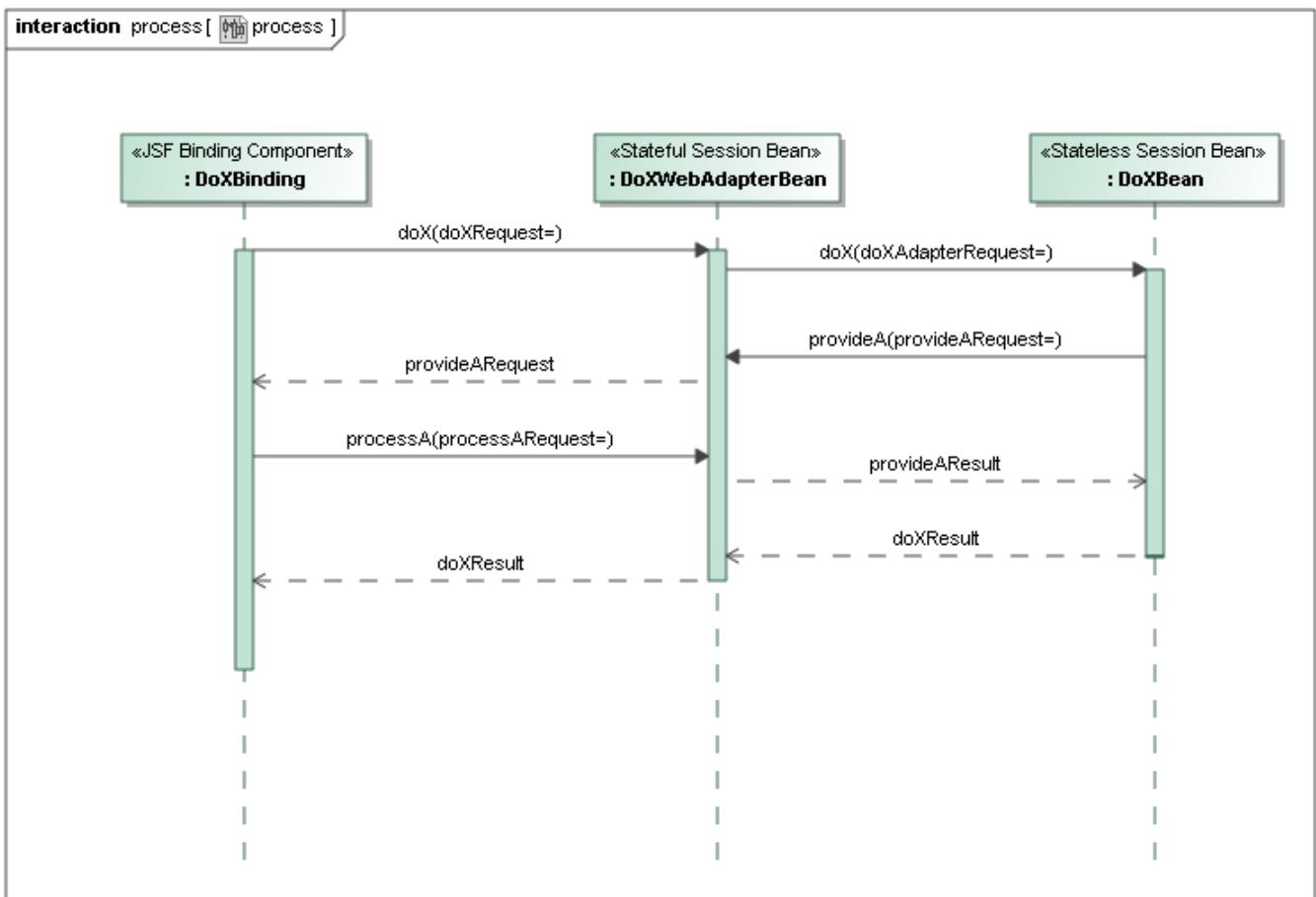


Figure 9.5: Workflow using a call-return adapter

#### 9.11.2.1.1.1 The service layer

The service itself is independent of the adapter used to access the service. It is based on a simple URDAD based design. The interface for the service is simply

```

/**
 *
 */
package za.co.solms.test.webAdapter.businessLogic;

import java.io.Serializable;

/**
 * @author fritz@solms.co.za
 */
public interface ServiceX
{
    public DoXResult doX(DoXAdapterRequest request);

    public interface ServiceXInterface extends ServiceXInterfaceBusinessLogic,
        ServiceXInterfaceUserInterface
    {
    }
}

```

```
public interface ServiceXInterfaceBusinessLogic
{
    public ProvideAResult provideA(ProvideARequest request);
}

public interface ServiceXInterfaceUserInterface
{
    public ProvideARequest doX(DoXRequest request);

    public DoXResult processA(ProvideAResult provideAResult);
}

public class ProvideARequest implements Serializable
{
    private static final long serialVersionUID = 1L;
}
public class ProvideAResult implements Serializable
{
    private static final long serialVersionUID = 1L;
}

public class DoXRequest implements Serializable
{
    public DoXRequest()
    {
    }

    private static final long serialVersionUID = 1L;
}

public class DoXAdapterRequest extends DoXRequest implements Serializable
{
    public DoXAdapterRequest(ServiceXInterface serviceXInterface)
    {
        this.serviceXInterface = serviceXInterface;
    }

    public ServiceXInterface getInterface()
    {
        return serviceXInterface;
    }

    private ServiceXInterface serviceXInterface;

    private static final long serialVersionUID = 1L;
}

public class DoXResult implements Serializable
{
    private static final long serialVersionUID = 1L;
}
```

The services itself is implemented as a stateless session bean:

```
/***
 *
 */
package za.co.solms.test.webAdapter.businessLogic;
```

```
import javax.ejb.Stateless;

/**
 * @author fritz@solms.co.za
 *
 */
@Stateless
public class ServiceXBean implements ServiceX
{

    @Override
    public DoXResult doX(DoXAdapterRequest request)
    {
        ProvideAResult provideAResult = request.getInterface().provideA(new ProvideARequest());

        return new DoXResult();
    }
}
```

#### 9.11.2.1.1.2 Stateful Session Bean adapter using asynchronous session bean call

For the adapter an asynchronous call to a session bean (EJB 3.1 +) we need the stateless session bean which offers the asynchronous service

```
/**
 *
 */
package za.co.solms.test.webAdapter.adapter;

import java.util.concurrent.Future;

import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.EJB;
import javax.ejb.Stateless;

import za.co.solms.test.webAdapter.businessLogic.ServiceX;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.DoXAdapterRequest;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.DoXResult;

/**
 * @author fritz@solms.co.za
 *
 */
@Asynchronous
@Stateless
public class ServiceXAsync
{
    public ServiceXAsync() {}

    @Asynchronous
    public Future<DoXResult> doXAsync(DoXAdapterRequest request)
    {
        DoXResult result = serviceX.doX(request);
        return new AsyncResult(result);
    }

    @EJB
    private ServiceX serviceX;
}
```

as well as the adapter which transfers control across services using Futures as communication mechanism:

```
/***
 *
 */
package za.co.solms.test.webAdapter.adapter;

import java.util.concurrent.Future;
import java.util.logging.Logger;

import javax.ejb.EJB;
import javax.ejb.Stateful;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

import com.google.common.util.concurrent.ValueFuture;

import za.co.solms.test.webAdapter.businessLogic.ServiceX.DoXAdapterRequest;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.DoXRequest;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.DoXResult;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.ProvideARequest;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.ProvideAResult;
import za.co.solms.test.webAdapter.businessLogic.ServiceX.ServiceXInterface;

/***
 * @author fritz@solms.co.za
 *
 */
@Stateful
public class ServiceXAdapter implements ServiceXInterface
{
    public ServiceXAdapter() {}

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public ProvideARequest doX(DoXRequest request)
    {
        DoXAdapterRequest adapterRequest = new DoXAdapterRequest(ServiceXAdapter.this);
        doXResultFuture = serviceXAsync.doXAsync(adapterRequest);

        try
        {
            return aRequested.get();
        }
        catch (Exception e)
        {
            logger.severe("FAILED");
            return null;
        }
    }

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public DoXResult processA(ProvideAResult provideAResult)
    {
        aProvided.set(provideAResult);

        try
        {
            DoXResult doXResult = doXResultFuture.get();

            return doXResult;
        }
```

```
        }
        catch (Exception e)
        {
            logger.severe("process execution failed");
            doXResultFuture.cancel(true);
            return null;
        }
    }

/* (non-Javadoc)
 * @see za.co.solms.test.webAdapter.businessLogic.ServiceX.ServiceXInterface#provideA(za. ←
 *      co.solms.test.webAdapter.businessLogic.ServiceX.ProvideAResult)
 */
@Override
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public ProvideAResult provideA(ProvideAResult request)
{
    aRequested.set(request);
    try
    {
        return aProvided.get();
    }
    catch (Exception e)
    {
        logger.severe("process execution failed");
        doXResultFuture.cancel(true);
        return null;
    }
}

@EJB
private ServiceXAsync serviceXAsync;

private ValueFuture<ProvideAResult> aRequested = ValueFuture.create();
private ValueFuture<ProvideAResult> aProvided = ValueFuture.create();

private Future<DoXResult> doXResultFuture;

private static Logger logger = Logger.getLogger(ServiceXAdapter.class.getName());
}
```

## 9.12 Sending an email

All resources including connection resources should be managed by the application server. Hence one should not open directly a connection to a mail server, but request a mail resource from the application server instead.

### 9.12.1 Registering an email server

This is typically done through the management console of your application server. On Glassfish the resultant mail service descriptor is added to the `config/domain.xml` domain descriptor:

```
<mail-resource host="solms.co.za" store-protocol="pop3" store-protocol-class="com.sun.mail. ←
imap.POP3Store"
    jndi-name="mail/solms" from="appServer@solms.co.za" user="appServer">
<property name="mail.smtp.auth" value="true" />
<property name="mail.smtp.password" value="appServerPassword" />
<property name="mail.smtp.user" value="sbss" />
```

```
<property name="mail.smtp.port" value="425" />
</mail-resource>
```

### 9.12.2 Sending an email from an enterprise bean

Sending an email from a session or message-driven bean is no different than sending one from any other Java code except that the email server session is not to be created within the code, but rather to be provided (via dependency injection) by the application server:

```
@Stateless
public class PersonServices
{

    @Override
    public void emailAuthenticationCredentials(String emailAddress)
        throws NoPersonWithThatEmailAddressException, MessagingException
    {
        List<Person> persons = this.getPersonsWithEmailAddress(emailAddress);

        if (persons.size() == 0)
            throw new NoPersonWithThatEmailAddressException();

        Transport transport = mailSession.getTransport();

        for (Person person:persons)
        {
            User user = person.getUser();

            MimeMessage message = new MimeMessage(mailSession);
            message.setSubject("Password notification");
            message.setText("Your username and password for solms.co.za are \n" +
                "username: " + user.getUsername() + "\npassword:" + user.getPassword());
            message.setFrom(new InternetAddress("info@solms.co.za"));
            message.addRecipient(Message.RecipientType.TO,
                new InternetAddress(emailAddress));

            transport.connect();
            transport.sendMessage(message,
                message.getRecipients(Message.RecipientType.TO));
            transport.close();
        }
    }

    @PersistenceContext
    private EntityManager entityManager;

    @EJB
    private UserServices userServices;

    @Resource(name = "mail/solms")
    private Session mailSession;
```

### 9.13 Web services access

The Enterprise Java Beans specification requires that application servers must support the publication of session beans as web service endpoints, using the standard JAX-WS framework.

Through this mechanism, incoming web services (SOAP) messages are de-marshalled to Java object using the JAXB framework, and returned types are marshalled back to SOAP, supporting interoperability with as great number of services and clients.

To publish a session bean as a web service, the bean must be annotated with the `@javax.jws.WebService` annotation. Using only this annotation, the JAX-WS framework will use its default interpretation to generate a full web services contract from the session bean, although the bean author may typically wish to customise this process via further annotations.

For example, consider the Java services contract for the following (incomplete) book shop:

```
package za.co.solms.example;

/** A contract for a simplistic book shop */
public interface BookShop
{
    /** Enquires as to the availability of a particular book.
     * @param request Contains information that identifies the book in question
     * @return Information about the availability of the book
     * @throws UnknownBookException If the request identifies an unknown book
     * @throws InsufficientInformationException If the request does not identify a book
     */
    public BookAvailabilityResponse enquireBookAvailability( BookAvailabilityRequest request
    )
        throws UnknownBookException, InsufficientInformationException;

    // etc. ...
}
```

```
package za.co.solms.example;

/** A request to enquire about the availability of a book, which may be
 * partially or fully identified by the attributes.
 */
public class BookAvailabilityRequest
{
    public String getBookName() {
        return bookName;
    }

    public void setBookName(String bookName) {
        this.bookName = bookName;
    }

    public String getBookAuthor() {
        return bookAuthor;
    }

    public void setBookAuthor(String bookAuthor) {
        this.bookAuthor = bookAuthor;
    }

    public String getPublisherName() {
        return publisherName;
    }

    public void setPublisherName(String publisherName) {
        this.publisherName = publisherName;
    }

    public int getYearOfPublication() {
        return yearOfPublication;
    }
}
```

```
public void setYearOfPublication(int yearOfPublication) {
    this.yearOfPublication = yearOfPublication;
}

private String bookName, bookAuthor, publisherName;
private int yearOfPublication;
}

package za.co.solms.example;

/** A response to an enquiry for book availability, indicating the number
 * of books available, as well as the estimated delivery time.
 */
public class BookAvailabilityResponse
{
    public int getNumberInStock() {
        return numberInStock;
    }

    public void setNumberInStock(int numberInStock) {
        this.numberInStock = numberInStock;
    }

    public double getEstimatedDeliveryTimeInDays() {
        return estimatedDeliveryTimeInDays;
    }

    public void setEstimatedDeliveryTimeInDays(double estimatedDeliveryTimeInDays) {
        this.estimatedDeliveryTimeInDays = estimatedDeliveryTimeInDays;
    }

    private int numberInStock;
    private double estimatedDeliveryTimeInDays;
}
```

A stateless session bean implementation, which has been annotated for web services access (using all-defaults) would look as follows:

```
package za.co.solms.example.impl;

import javax.ejb.*;
import javax.jws.*;
import za.co.solms.example.*;

@Stateless
@Remote({BookShop.class})
@WebService
public class BookShopBean implements BookShop
{
    public BookAvailabilityResponse enquireBookAvailability( BookAvailabilityRequest request -->
    )
    throws UnknownBookException, InsufficientInformationException
    {
        // TODO Implement...
        return null;
    }
}
```

**Note**

Web Service publication is only supported for *stateless* session beans, as services are stateless by definition. The contract is also only generated from the *remote* interface.

### 9.13.1 The generated web services contract

When deployed, the application server will automatically publish the service under a default URL, usually derived from the bean name (for example, `http://localhost:8080/BookShopBeanService/BookShopBean`)

Following the industry standard convention, requesting the contract for this service (via `http://192.168.1.142:8080/BookShopBeanService/BookShopBean?WSDL`) we find the following generated contract:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by JAX-WS RI -->
<definitions
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility" -->
    -1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://impl.example.solms.co.za/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://impl.example.solms.co.za/" name="BookShopBeanService">

    <types>
        <xsd:schema>
            <xsd:import namespace="http://impl.example.solms.co.za/"
                schemaLocation="http://192.168.1.142:8080/BookShopBeanService/BookShopBean?xsd=1"/>
        </xsd:schema>
    </types>

    <message name="enquireBookAvailability">
        <part name="parameters" element="tns:enquireBookAvailability"/>
    </message>
    <message name="enquireBookAvailabilityResponse">
        <part name="parameters" element="tns:enquireBookAvailabilityResponse"/>
    </message>
    <message name="UnknownBookException">
        <part name="fault" element="tns:UnknownBookException"/>
    </message>
    <message name="InsufficientInformationException">
        <part name="fault" element="tns:InsufficientInformationException"/>
    </message>

    <portType name="BookShopBean">
        <operation name="enquireBookAvailability">
            <input message="tns:enquireBookAvailability"/>
            <output message="tns:enquireBookAvailabilityResponse"/>
            <fault message="tns:UnknownBookException" name="UnknownBookException"/>
            <fault message="tns:InsufficientInformationException"
                name="InsufficientInformationException"/>
        </operation>
    </portType>

    <binding name="BookShopBeanPortBinding" type="tns:BookShopBean">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <operation name="enquireBookAvailability">
            <wsp:PolicyReference URI="# BookShopBeanPortBinding_enquireBookAvailability_WSAT_Policy"/>
            <soap:operation soapAction="" />
        </operation>
    </binding>

```

```
<input>
    <soap:body use="literal"/>
</input>
<output>
    <soap:body use="literal"/>
</output>
<fault name="UnknownBookException">
    <soap:fault name="UnknownBookException" use="literal"/>
</fault>
<fault name="InsufficientInformationException">
    <soap:fault name="InsufficientInformationException" use="literal"/>
</fault>
</operation>
</binding>

<service name="BookShopBeanService">
    <port name="BookShopBeanPort" binding="tns:BookShopBeanPortBinding">
        <soap:address location="http://192.168.1.142:8080/BookShopBeanService/" ↵
            BookShopBean"/>
    </port>
</service>
</definitions>
```

And the generated schema for the structure of the exchanged messages:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Published by JAX-WS RI -->
<xss:schema xmlns:tns="http://impl.example.solms.co.za/" xmlns:xss="http://www.w3.org/2001/ ←
    XMLSchema" →
    version="1.0" targetNamespace="http://impl.example.solms.co.za/">

    <xss:element name="InsufficientInformationException" type="←
        tns:InsufficientInformationException"/>

    <xss:element name="UnknownBookException" type="tns:UnknownBookException"/>

    <xss:element name="enquireBookAvailability" type="tns:enquireBookAvailability"/>

    <xss:element name="enquireBookAvailabilityResponse" type="←
        tns:enquireBookAvailabilityResponse"/>

    <xss:complexType name="enquireBookAvailability">
        <xss:sequence>
            <xss:element name="arg0" type="tns:bookAvailabilityRequest" minOccurs="0"/>
        </xss:sequence>
    </xss:complexType>

    <xss:complexType name="bookAvailabilityRequest">
        <xss:sequence>
            <xss:element name="bookAuthor" type="xs:string" minOccurs="0"/>
            <xss:element name="bookName" type="xs:string" minOccurs="0"/>
            <xss:element name="publisherName" type="xs:string" minOccurs="0"/>
            <xss:element name="yearOfPublication" type="xs:int"/>
        </xss:sequence>
    </xss:complexType>

    <xss:complexType name="enquireBookAvailabilityResponse">
        <xss:sequence>
            <xss:element name="return" type="tns:bookAvailabilityResponse" minOccurs="0"/>
        </xss:sequence>
    </xss:complexType>
```

```
<xs:complexType name="bookAvailabilityResponse">
    <xs:sequence>
        <xs:element name="estimatedDeliveryTimeInDays" type="xs:double"/>
        <xs:element name="numberInStock" type="xs:int"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="UnknownBookException">
    <xs:sequence>
        <xs:element name="message" type="xs:string" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="InsufficientInformationException">
    <xs:sequence>
        <xs:element name="message" type="xs:string" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

### 9.13.2 Customising the web services mapping

The full range of JAX-WS and JAXB annotations may be used to customise the generated contract, such as mapping Java attributes to either XML Schema elements or attributes, customising the names of arguments and return types, and so on.

For more details, refer to the JAX-WS and JAXB documentation.

## 9.14 Exercises

1. Write a stateless session bean which publishes through both local and remote interfaces, a service to perform unit conversion. For now, only *distance* needs to be supported.
  - The primary service should be something like: `public Distance convertDistance( Distance d, DistanceUnit u )`. A distance is something with a unit, and a magnitude.
  - The service should throw an appropriate exception (e.g. `UnsupportedUnitException`) if an unsupported unit is requested. The service should support at least a few units (e.g. meter, kilometer, mile, foot, inch) and should be able to convert between any of them.
  - It would probably be a good idea to provide a service that indicates the supported units for clients.
2. Write a stateful session bean that allows a user to play the ‘guess the number’ game. When the session starts, it randomly picks a number between 1 and 20, with a prize money of R1,000,000.00. It provides the user a service to guess the number: If the user guesses correctly, he should be indicated to have won the prize money. For every incorrect guess, the prize money that can be won during the next guess is halved.

# Chapter 10

## EJB Interception

### 10.1 Introduction

Interception is a powerful feature that, for any managed component in a Java EE container, provides an extension point for future pluggable extensions around the services these components provide. This results in improved maintainability, as the classes themselves can be kept simple (the extra logic is not absorbed within the class itself).

The EJB specification caters for two features through interception

- *dependency injection* where the container injects dependencies on resources it manages, and
- *interceptor methods and classes* used to intercept a bean's business services

### 10.2 Dependency injection

Dependency injection helps to manage complexity by removing certain responsibilities out of the application logic, by delegating it to the container. This improves decoupling, re-usability and plug-ability.

#### 10.2.1 Dependency injection as a form of inversion of control

To explain *inversion of control*, recall the original non-graphical software applications. There the application code was under full control, asking the user for one input, then the next, then processing that, etc -- all decisions on where to go next was done by the application logic.

Inversion of control was the key change when moving from console based applications to graphical user interfaces, the control was transferred from the application logic to the user interface. *Inversion of control thus involves relinquishing control to something which is outside the application logic.*

In EJB applications, your components often need references to objects managed by the application server. These may include *session contexts, timer services, transaction contexts, entity managers*, as well as other beans. The responsibilities of providing references to these resources should reside in the application server. Inversion of control enables one to delegate the population of fields (or of method arguments) to the container.

#### 10.2.2 Which dependencies are injected ?

Typically one should only inject those dependencies which one should not, or does not want to, embed in the application logic. This includes dependencies on infrastructure objects and services managed by the container including

- any resources managed by the container (such as database or mail server connections), and

- any support objects managed by the container for your enterprise beans including session contexts, transaction contexts, entity managers and timer services.

In addition to the above one can also inject dependencies between your own objects, reducing the strength of the coupling between components, improving maintainability and re-usability. This may include

- dependencies on other enterprise beans, and
- dependencies on web services published in JNDI,
- dependencies on resources managed within deployment descriptors (e.g. text strings for configuration or messages).

### 10.2.3 Setter injection

EJB supports setter injection, where the container will automatically call one or more ‘setter’ methods on your component, automatically supplying the desired input argument. The setter methods for which injections are defined will be called prior to any business service being called. If a dependency injection fails, the deployment itself will fail (i.e. the container will not ignore the event of not being able to satisfy a dependency).

For container-managed infrastructure objects, the `@javax.ejb.Resource` annotation is used.

For example

```
@Resource(mappedName="clientDB" type="javax.sql.DataSource")
protected void setClientDB(DataSource db)
{
    this.clientDB = db;
}
```

will specify that the `setClientDB` is to be called using as parameter the resource published in the JNDI naming service as `clientDB`.

If the type of the argument can be inferred, the type parameter can be omitted. For example

```
@Resource(mappedName="clientDB")
protected void setClientDB(DataSource db)
{
    this.clientDB = db;
}
```

would suffice for the above example.

If one does not care about a *particular* instance (with a particular name) being provided, one may also omit the JNDI name, leaving the decision of which resource to inject entirely up to the container:

```
@Resource
protected void setClientDB(DataSource clientDB)
{
    this.clientDB = clientDB;
}
```

To inject enterprise beans, the `@javax.ejb.EJB` annotation is used. It is common practise to omit the JNDI name, since we typically only care about the available functionality (i.e. the interface which it implements), and would want the container to provide any available implementation.

- only the parameter type

```
@EJB
protected void setTaxCalculator( VatCalculatorRSALocal taxCalculator )
{
    this.taxCalculator = taxCalculator;
}
```

- Alternatively, one may request that a particular bean be injected, based on its *bean name*:

```
@EJB (name="VatCalculatorRSA")
protected void setTaxCalculator( VatCalculatorRSALocal taxCalcultor )
{
    this.taxCalculator = taxCalculator;
}
```

or, more commonly, based on its JNDI *mapped name*:

```
@EJB (mappedName="ejb/VATCalculator")
public void setTaxCalculator( VatCalculatorRSALocal taxCalcultor )
{
    this.taxCalculator = taxCalculator;
}
```

in both cases, as per the bean's deployment configuration (i.e. annotations or deployment descriptor file).

#### 10.2.3.1 Method access level modifier

The setter methods do not have to be `public`. Even though it would be safe to make them public (since a client will not be able to invoke them, as clients are never exposed to the *class* of an enterprise bean, only the remote or local interface) the container also supports `protected` and even `private` setters.

#### 10.2.4 Field injection

The EJB specification supports dependency injection directly to fields, reducing the amount of code in a component with dependencies by removing the need to write a setter method, unless you want to take additional action when a resource is injected.

Consider a sports events planning service, which may need to make use of a booking engine, as well a weather forecasting service. One has merely to declare object references to these objects (via their interfaces) and annotate them with `@javax.ejb.EJB`, and the container will provide implementations before any of the bean's business services are invoked.

```
import javax.ejb.*;

@Stateless
@Remote({SportsEventPlanner.class})
public class SportsEventPlannerBean implements SportsEventPlanner
{
    ...
    @EJB
    private WeatherService weatherService;

    @EJB
    private ResourceBookingService bookingEngine;
}
```

##### 10.2.4.1 Injecting context objects

Dependency injection is also commonly used to provide a bean access to the context objects maintained by the application server for that bean. This is done by injecting variables into the class:

```
@Resource javax.ejb.SessionContext sessionContext;
@Resource javax.ejb.UserTransaction userTransaction;
@Resource javax.ejb.TimeService timer;
```

These objects provide access to the details of the current user session, the current transaction, time-scheduling services, etc.

#### 10.2.4.2 Field access level modifier

The container also supports the injection of fields with all access levels, including `protected` and even `private` fields.

#### 10.2.5 How are injection resources resolved?

- **Resolving by bean property or field type** The resource will be looked up by name via the default JNDI name generated for that type (local or remote interface). For example,

```
@EJB  
public void setTaxCalculator(TaxCalculatorLocal taxCalculator)  
{  
    this.taxCalculator = taxCalculator;  
}
```

will be looked up under the ejb-name, ‘TaxCalculatorLocal’, within the default name space for that resource type, `java:comp/env/ejb`.

- **Resolving by variable name**

```
@Resource  
public void setClientDB(DataSource clientDB)  
{  
    this.clientDB = clientDB;  
}
```

The EJB specification requires that predefined web service references can also be looked up, based on web service name. The default name space searched when resolving web services is `java:comp/env/service`.

- **Resolving by injector name**

```
@EJB (name="vatCalculatorRSA" beanInterface="TaxCalculatorLocal")  
public void setTaxCalculator(TaxCalculator taxCalculator)  
{  
    this.taxCalculator = taxCalculator;  
}
```

The ejb-name is a logical name which may be mapped in a deployment descriptor via and `<ejb-ref>` tag onto an actual JNDI name. If this mapping is not provided, the logical name will be used directly in a JNDI lookup.

#### 10.2.6 Linking logical EJB names to physical JNDI names

If an EJB is given a name (called the ejb-name, this name is used as the basis of the bean’s JNDI name. The mapping is done either in the `<enterprise-beans>` section of an `ejb-jar.xml` deployment descriptor, or as part of the annotation to indicate that a bean is a stateless or stateful session bean:

```
<ejb-jar 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_0.xsd"  
    version="3.0">  
    <enterprise-beans>  
        <session>  
            ...  
            <ejb-name>EmployeeService</ejb-name>  
            <ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>  
            ...  
            <ejb-ref>  
                <description>
```

```
This is a reference to an EJB 2.1 entity bean that
encapsulates access to employee records.
</description>
<ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>com.wombat.empl.EmployeeRecordHome</home>
<remote>com.wombat.empl.EmployeeRecord</remote>
</ejb-ref>
<ejb-local-ref>
<description>
This is a reference to the local business interface
of an EJB 3.0 session bean that provides a payroll
service.
</description>
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
<local>com.aardvark.payroll.Payroll</local>
</ejb-local-ref>
<ejb-local-ref>
<description>
This is a reference to the local business interface
of an EJB 3.0 session bean that provides a pension
plan service.
</description>
<ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
<local>com.wombat.empl.PensionPlan</local>
</ejb-local-ref>
...
</session>
...
</enterprise-beans>
</ejb-jar>
```

or, using the name attribute of the annotation:

```
import javax.ejb.*;

@Stateless(name="SAHomeLoanCalculator")
@Remote({HomeLoanCalculator.class})
public class HomeLoanCalculatorBean implements HomeLoanCalculator
{
    ...
}
```

Alternatively, an absolute JNDI name may be given to the bean via the mappedName attribute:

```
import javax.ejb.*;

@Stateless(mappedName="calculators/HomeLoan")
@Remote({HomeLoanCalculator.class})
public class HomeLoanCalculatorBean implements HomeLoanCalculator
{
    ...
}
```

## 10.3 Interceptors

Interceptors enable one to intercept any of the business services offered by session and message-driven beans. They provide a mechanism for pluggable add-on responsibilities around business services. They may be used for filtering, custom security solutions, validation, or any other responsibility in the spirit of aspect-oriented programming.

### 10.3.1 When should one use interceptors

Interceptor classes are useful when one wants to add additional responsibilities to a service, which you do not want to include in the responsibility domain of the class(es) whose services you are intercepting.

Interceptors thus add additional optional work flow steps around some core service or across a range of services. Interception provides a mechanism to weave these additional work flow steps into your code, applying it across the application domain of the interceptor. In that context they are similar to simple aspects and can often be used instead of aspects.

### 10.3.2 Defining interceptors

Interceptors may be defined as interceptor methods within a bean class or as separate interceptors which can be applied across beans. In either case the interceptor is provided an interception context which it can inspect to obtain information about the service request which is being intercepted.

#### 10.3.2.1 The invocation context

The interceptor invocation runs within the *transaction and security context* of the business service it intercepts, and it may access those objects just like any session bean (i.e. by having them injected). The interceptor method, however, receives an instance of `javax.interceptor.InvocationContext`, which it may use to:

- get a handle to the method it currently intercepts via the `getMethod() : Method` service,
- get a handle to the parameters of the service it intercepts via `getParameters() : Object[]`,
- change the parameters which are passed on to the actual business service via `setParameters( Object[] params )`,
- get a handle to target object, i.e. the object which would have received the service request message had it not been intercepted, via `getTarget() : Object`,
- get a handle to the context data (a map which interceptors may use to place any information to be shared between interceptors, and other components) via `getContextData() : Map<String, Object>`, and
- pass control to the next interceptor (if there is one) or alternatively to the business service itself via `proceed() : Object`.

#### 10.3.2.2 Interceptor methods

Interceptor methods can be defined directly in the bean class itself. This provides switchable additional functionality around the core bean service. They must have the following signature:

```
import javax.ejb.*;
import javax.interceptor.*;

@Stateless
@Remote({SomeInterface.class})
public class SomeBean implements SomeInterface
{

    ...

    /** This is an interceptor method */
    @AroundInvoke
    public Object intercept( InvocationContext ctx )
    {
        // Here the interception logic
        ...
        //

        // delegate control to next object in chain of interceptors:
    }
}
```

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

An interceptor method, irrespective of whether it is defined within a bean class or in a separate interceptor class, will thus obtain an invocation context as parameter and must return `invocationContext.proceed()`.

Calling `proceed` on the invocation context passes control to the next interceptor (if there is one) or alternatively to the business service being intercepted.

#### 10.3.2.2.1 Aborting an interception

An interceptor method can choose to abort the service request being intercepted by simply not returning `invocationContext.proceed()`, but directly returning the return value for the service it is intercepting (without the service which is intercepted actually being called) or by throwing an exception.

#### 10.3.2.3 Interceptor classes

The most common way of defining an interceptor is by defining a separate class which contains one or more interception methods in it (each annotated with `@javax.interception.AroundInvoke`).

##### 10.3.2.3.1 Interceptor life span and state

An interceptor has the same life span as the context whose service is intercepted. An interceptor for a session bean has hence the life span of the session while an interceptor applied to a message driven bean exists while the service request is being processed.

###### 10.3.2.3.1.1 Interceptor state

An interceptor may have state in the form of instance members and this state is guaranteed to survive across invocations.

###### 10.3.2.3.1.2 Activation and de-activation

If an interceptor is applied to a stateful session bean it is activated and passivated upon bean instance activation and passivation.

###### 10.3.2.3.1.3 Life cycle callback services

In order to perform specific tasks on creation, destruction, passivation or re-activation, one may provide callback handlers similar to the enterprise bean callback handlers, i.e. methods which have been annotated with `@PostConstruct`, `@PreDestroy`, `@PostActivate` or `@PrePassivate` annotations.

##### 10.3.2.3.2 Requirements for interceptor classes

Interceptor classes

- must provide a default constructor, and
- must provide one method annotated with `@AroundInvoke` which takes an invocation context as parameter and returns an object which will be generated via `invocationContext.proceed()` or directly the return value of the method being intercepted (in this case all further decorators and the method would be skipped).

### 10.3.2.3.3 Interceptor exceptions

Interceptors may throw

- any checked exception thrown by the business service it intercepts, and
- any runtime exceptions in order to indicate an error the inability to realize the contractual obligations.

Any exception thrown within a transactional context which is not caught and handled prior to reaching the transaction boundary will result in *transaction roll-back*.

### 10.3.2.3.4 Declaring an interceptor class in a deployment descriptor

Instead of annotating a class as an interceptor class in code, we can declare it in a deployment descriptor of a module or application. This can be done by adding an `interceptors` section to a `ejb-jar.xml` deployment descriptor included in the `META-INF` directory of your deployment:

```
<interceptors>
  <interceptor>
    <interceptor-class>
      za.co.solms.comms.email.EmailInterceptor
    </interceptor-class>
    <aroundInvoke-method>
      <method-name>sendMail</method-name>
    </aroundInvoke-method>
    ...
  </interceptor>
</interceptors>
```

### 10.3.2.3.5 Interceptor specialization

One can only specify a single interception method per class (either via annotating the method as `@AroundInvoke` or by declaring an interceptor in a deployment descriptor). However, an interceptor may inherit further interception services from superclasses which have themselves been declared interceptor classes.

Thus, applying an interceptor which is derived from another interceptor results in multiple interceptions. The superclass interceptor method is applied prior to the specialized interceptor's interceptor method.

## 10.3.3 Applying Interceptors

### 10.3.3.1 Application domain for interceptors

Interceptors are used to provide pluggable add-on responsibility around business level services. As such they can be applied to stateful and stateless session beans and message driven beans.

An interceptor may be

- a *method level interceptor* which intercepts a particular business service of a particular bean,
- a *class level interceptor* which intercepts all business services of a particular bean, or
- a *default interceptor* which intercepts all business services of all enterprise beans within a particular deployment.

### 10.3.3.2 Method level interceptors

Method level interceptors are used when one wants to apply an interceptor to only one specific business service.

### 10.3.3.2.1 Applying method level interceptors via annotations

One can simply annotate the relevant business method of an enterprise bean if one wants to apply an interceptor to only that business service:

```
@Interceptors({MyInterceptor.class})
public String generateForecast(PersonInfo personInfo) throws WontSayException
{
    ...
}
```

### 10.3.3.2.2 Applying method level interceptors in the deployment descriptor of a bean

Instead of applying an method-level interceptor in the source code of a bean via an annotation, we can also apply the interceptor in a separate deployment descriptor. This is done in a `interceptor-bindings` section of the `assembly-descriptor` of the `ejb-jar.xml` deployment descriptor.

For example, below we specify that `za.co.example.MyInterceptor` should intercept the `generateForecast( :- PersonInfo )` business service of the `AstrologyFeeder`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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_0.xsd"
  version="3.0">

  <!-- Declare it -->
  <interceptors>
    <interceptor>
      <interceptor-class>
        za.co.example.MyInterceptor
      </interceptor-class>
    </interceptor>
  </interceptors>

  <!-- Map it -->
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>AstrologyFeeder</ejb-name>
      <interceptor-class>
        za.co.example.MyInterceptor
      </interceptor-class>
      <method>
        <method-name>generateForecast</method-name>
        <method-params>
          <method-param>
            za.co.solms.astrology.AstrologyFeeder.PersonInfo
          </method-param>
        </method-params>
      </method>
    </interceptor-binding>
  </assembly-descriptor>

</ejb-jar>
```

### 10.3.3.3 Class level interceptors

If an interceptor is assigned to a bean class, it will intercept all business services of the bean.

### 10.3.3.1 Applying class level interceptors via annotations

To assign a single interceptor class to a bean, one can use the `@javax.interceptor.Interceptors` annotation:

```
@Interceptors ({MyInterceptor.class})
public class ShoppingBean implements Shopping
{
    ...
}
```

Multiple interceptors are assigned simply by listing multiple classes:

```
@Interceptors({MyInterceptor.class, LoggingInterceptor.class})
public class ShoppingBean implements Shopping
{
    ...
}
```

If multiple interceptors have been specified for a bean, they will intercept in the order in which they are declared.

### 10.3.3.2 Applying class level interceptors in the deployment descriptor of a bean

Instead of applying a class-level interceptor in the source code of a bean via an annotation, we can also apply the interceptor in a separate deployment descriptor. This is done in a `interceptor-bindings` section of the `assembly-descriptor` of the `ejb-jar.xml` deployment descriptor.

For example, below we specify that the `RequestLogger` is applied as an interceptor to all business services of the `ProcessOrder` session bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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_0.xsd"
  version="3.0">

  <!-- Declare it -->
  <interceptors>
    <interceptor>
      <interceptor-class>
        za.co.solms.logging.RequestLogger
      </interceptor-class>
    </interceptor>
  </interceptors>

  <!-- Map it -->
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>za.co.solms.retail.ProcessOrder</ejb-name>
      <interceptor-class>
        za.co.solms.logging.RequestLogger
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>

</ejb-jar>
```

---

#### Note

An interceptor is configured as a class-level interceptor simply by omitting any method declarations in the mapping in `ejb-jar.xml`.

---

#### 10.3.3.4 Default interceptors

At times one may want to weave in additional work flow steps across all business services of all session and message-driven beans within a module or application, i.e. across all business services published within a particular domain. This could be useful, for example, for auditability (e.g. logging). This is where default interceptors become useful.

Default interceptors cannot be specified via annotations. They must be declared in the `ejb-jar.xml` deployment descriptor for the domain, and this is done by specifying `*` (all) as the `ejb-name` which is targeted.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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_0.xsd"
  version="3.0">

  <!-- Declare it -->
  <interceptors>
    <interceptor>
      <interceptor-class>
        za.co.solms.auditing.logging.RequestLogger
      </interceptor-class>
    </interceptor>
  </interceptors>

  <!-- Map it -->
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>
        za.co.solms.auditing.logging.RequestLogger
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>

</ejb-jar>
```

#### 10.3.3.5 Interception order

If multiple interceptors are applicable for a particular service request, then they are applied in the following order

- Default interceptors are applied first in order of their specification in the deployment descriptor.
- Class level interceptors are applied prior to method level interceptors.
- Interceptor classes are applied prior to interception methods. They are applied in the order they are requested in either the bean annotation or the deployment descriptor.
- If the interception context (e.g. the bean to which the interception is applied) has a superclass then the superclass interceptions take preference to the subclass.
- The `@AroundInvoke` method of a superclass of an interceptor class is applied prior to that of the subclass.

---

##### Note

There may be only a single `@AroundInvoke` method in any class, though an interceptor class. However, applying an interceptor class can still result in multiple interceptions for a single service request as the interceptor methods of the superclass of the interceptor class are also applied. These are applied in order of the class hierarchy with superclass interceptor methods taking preference above subclass interceptor methods.

---

### 10.3.3.6 Specifying exclusions for class and default interceptors

At times one would like to specify that certain business services should not be intercepted by interceptors which have been applied across the class or the domain (i.e. class-level or default interceptors).

#### 10.3.3.6.1 Specifying exclusions via annotations

If the class or default interceptors have been applied via annotations, it is likely that you will want to specify the exclusions also via annotation:

```
@ExcludeClassInterceptors  
@ExcludeDefaultInterceptors  
public void someSpecialBusinessService()  
{  
    ...  
}
```

#### 10.3.3.6.2 Specifying exclusions in a deployment descriptor

If the class or default interceptors have been applied outside the implementation code in a separate deployment descriptor, it is likely that you will want to specify the exclusions also in that deployment descriptor. This is done as follows:

```
<assembly-descriptor>  
    ...  
    <interceptor-binding>  
        <ejb-name>za.co.solms.SomeSpecialSessionOrMessageDrivenBean</ejb-name>  
        <exclude-class-interceptors/>  
        <exclude-default-interceptors/>  
        <method-name>someSpecialBusinessService</method-name>  
    </interceptor-binding>  
    ...  
</assembly-descriptor>
```

### 10.3.3.7 Injecting dependencies into interceptors

Interceptors themselves may use dependency injection to obtain references to any other components: EJBs, resources, etc. For example:

```
public class MyInterceptor  
{  
  
    @AroundInvoke  
    public Object intercept( InvocationContext ctx )  
    {  
        ...  
    }  
  
    @Resource(mappedName="java:ConnectionFactory")  
    QueueConnectionFactory cf;  
  
    @Resource(mappedName="queue/forecastNotification")  
    Queue queue;  
}
```

---

#### Note

Interceptors should be written in such a way as to minimally disrupt the request flow (unless that was the intention of the interceptor). To that end, optimal use should be made of messaging services etc, to not block the calling thread with interception work. Messages should rather be dispatched to other beans which will asynchronously take action upon intercepted messages.

---

### 10.3.4 A simple example: AstrologyFeeder

This examples provides a simple astrology forecast service. It is intercepted by an email interceptor which sends an e-mail to parties which have made use of this service in order to sell other services to them (e.g. to bring them into conversation with diseased family members).

---

**Note**

Instead of sending the e-mail in the interceptor directly, one would normally rather put a message on a queue and have a message driven bean process those messages in order to send the emails. This results in less client latency and provides a further level of decoupling.

---

#### 10.3.4.1 The context contract

```
package za.co.solms.astrology;

import java.io.Serializable;
import java.util.Date;

/** A contract for an astrology forecast service */
public interface AstrologyFeeder
{
    /**
     * Generates a forecast based on the provided personal information.
     * @param personInfo the information about the person for which the
     * forecast is requested.
     * @return text representing the forecast.
     */
    public String generateForecast( PersonInfo personInfo)
    throws WontSayException;

    /**
     * The personal information required for a forecast.
     */
    public class PersonInfo implements Serializable
    {
        public PersonInfo( String name,
                           String eMailAddress, Date dateOfBirth)
        {
            this.name = name;
            this.eMailAddress = eMailAddress;
            // To enforce encapsulation (composition
            // relationship) we clone the given date
            this.dateOfBirth = (Date)dateOfBirth.clone();
        }

        public String getName()
        {
            return name;
        }

        public String getEMailAddress()
        {
            return eMailAddress;
        }

        public Date getDateOfBirth()
        {
            return (Date) dateOfBirth.clone();
        }
    }
}
```

```
}

private String name, eMailAddress;
private Date dateOfBirth;
private static final long serialVersionUID = 200609192036L;
}

/***
 * This exception is thrown if the Astrology service is not prepared to
 * publish the forecast for the person for which the forecast was requested.
 */
public class WontSayException extends Exception {}
}
```

#### 10.3.4.2 The interceptor

```
package za.co.solms.comms.email;

import java.util.Date;
import javax.interceptor.*;
import javax.mail.*;
import javax.mail.internet.*;
import org.apache.log4j.Logger;
import za.co.solms.astrology.AstrologyFeeder.PersonInfo;

/***
 * This interceptor is used to add the responsibility of send an email to
 * somebody who can extract value from being notified of a person having
 * requested an astrological forecast:
 */
public class NotifyForecastRequestInterceptor
{

    @AroundInvoke
    public Object notifyOfForecastRequest( InvocationContext ctx)
        throws Exception
    {
        logger.info("Intercepted " + ctx.getMethod().getName());

        try
        {

            // Use the JavaMail API to hard-code a message
            // (this should ideally not be performed in the
            // same thread as here, but rather by another EJB
            // in response to a JMS message dispatched from this
            // interceptor)
            MimeMessage message = new MimeMessage(session);
            message.setRecipient(
                Message.RecipientType.TO,
                new InternetAddress(recipient));
            message.setSubject("Astrological forecast requested.");
            String eMailAddress = ((PersonInfo) ctx
                .getParameters()[0])
                .getE-MailAddress();
            message.setContent("Requester: "
                + eMailAddress, "text/plain");
            message.setSentDate(new Date());
            Transport.send(message);
        }
    }
}
```

```
        }
        catch (AddressException e)
        {
            logger.error("Invalid mail recipient", e);
        }
        catch (MessagingException e)
        {
            logger.error("Mail sending failed", e);
        }

        // Continue
        return ctx.proceed();
    }

    // A JavaMail session (injected)
    @Resource(mappedName="Mail")
    private Session session;

    // Who should receive the mail
    private static final String recipient = "fritz@solms.co.za";

    // Logging, using Log4J
    private static final Logger logger = Logger
        .getLogger(NotifyForecastRequestInterceptor.class);
}
```

#### 10.3.4.3 The bean implementation class

```
package za.co.solms.astrology;

import java.util.Random;
import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateful;
import javax.interceptor.Interceptors;
import za.co.solms.astrology.AstrologyFeeder;

@Stateful
@Remote({ AstrologyFeeder.class })
public class AstrologyFeederBean implements AstrologyFeeder
{

    @Interceptors(
    { za.co.solms.comms.email.NotifyForecastRequestInterceptor.class })
    public String generateForecast(PersonInfo personInfo)
    throws WontSayException
    {
        String period = forecastPeriods[rng
            .nextInt(forecastPeriods.length)];

        String outcome = forecastOutcomes[rng
            .nextInt(forecastOutcomes.length)];

        if (outcome
            .endsWith("no longer with us"))
            throw new WontSayException();

        return "During the next " + period
    }
}
```

```
        + " you will " + outcome + ".";
    }

private final static String[] forecastPeriods =
{ "week", "fortnight", "month", "quarter",
  "year" };

private static final String[] forecastOutcomes =
{
    "find new love",
    "find more love",
    "make an important career move which will change your life",
    "make a career move which will not change anything",
    "experience some personal loss," +
        " but come out of this as a stronger person",
    "step in front of a bus and be no longer with us" };

private Random rng = new Random();
}
```

#### 10.3.4.4 A simple RMI client

```
package za.co.solms.astrology;

import java.util.GregorianCalendar;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import za.co.solms.astrology.AstrologyFeeder.WontSayException;

public class Client
{
    public void run()
    {
        // Construct input to astrology service
        AstrologyFeeder.PersonInfo personInfo = new AstrologyFeeder.PersonInfo(
            "Nostradamus",
            "nostradamus@prophetsUnlimited.com",
            new GregorianCalendar(1503, 11, 14)
                .getTime());

        try
        {
            // Look up astrology service bean
            InitialContext ctx = new InitialContext();
            String jndiName = "AstrologyFeederBean/remote";
            AstrologyFeeder feeder = (AstrologyFeeder) ctx
                .lookup(jndiName);

            // Request business service
            String foreCast = feeder
                .generateForecast(personInfo);
            System.out.println("Forecast: "
                + foreCast);
        } catch (WontSayException e)
        {
            System.out
                .println("This is bad, they are too scared to tell.");
        } catch (NamingException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
    }

}

public static void main(String[] args)
{
    new Client().run();
}

}
```

### 10.3.5 Exercises

The intelligence agency of the western world's foremost economical power is very paranoid about news agencies publishing bad publicity regarding their president.

- Discuss three potential uses of EJB filters which have not been mentioned in the course notes thus far.
- Write a stateless session bean which offers a service of accepting news articles from journalists working remotely in the field. A news article should at least have a headline, a body of text, and a date. The bean does not have to do anything with the article for now, but a real implementation may store it persistently (using JPA) or place it on a message queue for approval.
- Write an interceptor which is applied as a method-level interceptor to the news submission service: If a submitted news article contains any mention of the country's president, it should not prevent the article from being submitted, but it should indicate that it has logged this incident (and probably that the journalist in question will receive a visit from an unmarked, black government vehicle and be taken away for questioning).
- Write a default interceptor (applied to all beans in the module) which, for any news article being sent as a parameter (or return value) to/from any service, will perform bad language filtering: Any occurrence of a bad word (from a list of bad words) should silently be replaced by ‘\*\*\*’ once it reaches its destination.

# Chapter 11

## Messaging

### 11.1 Introduction

For many application domains the synchronous service request-response model is unsatisfactory. For example, when you place an order with one of your business partners you may be content to submit the order to a message queue for processing at your business partner's convenience. Your business process may now continue immediately, irrespective of whether the order has been processed already (even if your business partner's systems are currently unavailable.)

#### 11.1.1 Support for JMS as a Managed Resource

To address this the EJB specification requires that access to messaging services which implement the Java Messaging Service (JMS) API must be supported via JMX -- i.e. the JMS must be managed by application servers as a managed resource. The messaging service is then looked up via JNDI, or injected via dependency injection, just as would be any other resource, such as a database connection or an EJB.

#### 11.1.2 Message-Driven Beans

The EJB specification requires that application servers support message driven beans (MDB)s. Message Driven Beans are enterprise beans which process asynchronous service requests coming in from a JMS compliant messaging service.

#### 11.1.3 How does a Message-Driven Bean Work

Unlike with typical service providers such as Session beans, clients do not send service requests directly to a Message-driven bean. Instead, a message is sent to a messaging service which takes over the responsibility of delivering the service request to a service provider.

A message-driven bean (a Java class implementing the `javax.jms.MessageListener` interface) is configured to receive messages from one or more messaging services, either through configuration in a XML deployment descriptor, or (more commonly) by annotating it with the `javax.ejb.MessageDriven` annotation.

All the benefits of the EJB component model, such as object pooling for high concurrency, continue to be applied.

#### 11.1.4 Robustness features

##### 11.1.4.1 Guaranteed Message Delivery

With guaranteed message delivery the MOM (Message-Oriented Middle ware) persists your message to a database or some other persistent storage resource until

- the message has been sent AND
- the message consumer has acknowledged consumption and successful processing of a message OR
- until the message expires.

If acknowledgement of receipt is not received within some set time period the message is re-delivered.

#### 11.1.4.2 Certified Message Delivery

Certified message delivery is a special case of guaranteed message delivery where, in addition to the steps followed for guaranteed message delivery, the message originator is notified of the consumption of a message.

## 11.2 Why Message Driven Beans?

Message driven beans provide a level of decoupling and reliability which is useful for many business applications. Furthermore, they may improve the perceived performance of client applications by performing certain services in the background.

### 11.2.1 Typical Applications of Message-Driven Beans

Below we list a few applications which may be suitable for implementation as message-driven beans:

- *Supplying information* to information processors, such as
  - Weather stations supplying information to Weather bureaus.
  - Financial information providers supplying market information to be used for valuation and pricing purposes.
  - Entertainment providers supplying information to booking organisations such as Computicket.
- *Loan applications* like home-loan applications which require a number of steps to be processed like credit checks, property valuation and others. The response may be sent at a later stage via an e-mail or a call-back through some other technology.
- The *ordering of items* from an on-line vendor like amazon. The client does not need to wait for credit-card verification and product availability checks and may, once again, be notified via e-mail or a call-back.
- General *action messages* requesting certain actions to be performed like
  - Payment orders.
  - Requests for suppliers to deliver supplies to refresh stock.
  - Instructions to a stock broker to purchase shares in a stock (company) at some specified price.

### 11.2.2 When Should You Consider Using MDBs?

There many scenarios which may prompt you to consider using a message-driven bean:

- **Non-blocking clients** Clients simply send the service request message and continue processing. This may improve the performance and usability quality attributes of client applications.
- **Reliability** Clients can issue service requests even if the service provider is currently not available. When using guaranteed service delivery the message is delivered when the service provider becomes available again. If, furthermore, the client itself maintains a message spool with messages stored and forwarded to the messaging service and removed from the spool on receipt of message delivery from the messaging service, then system reliability is very solid indeed. Failure will only occur in the very unlikely event of all three tiers failing.
- **Decoupling clients from service providers** Clients do not interface directly with service providers and the latter may be replaced by alternative service providers without affecting client applications.

- **Smooth load balancing** In the case of application servers supplying service requests to clusters of servers the application server guesses from the service requests delivered thus far which server is the least burdened and then pushes the next service request to that server. When using a messaging-driven beans, the JMS allows for both, the beans themselves (the service providers) pull the next message from the messaging service. This results in smoother load balancing.
- **Prioritised processing of service requests** The message queue itself may provide messages in a different order to that in which the messages arrived. It can re-order messages to achieve some form of prioritisation, based perhaps on some business rules.
- **Easier integration with legacy systems** Many legacy systems used messaging middle ware like IBM MQSeries, BEA's Tuxedo Q, Progress SonicMQ, Tibco Rendezvous or Microsoft MSMQ. These vendors typically provide JMS drivers for these systems, facilitating easy integration of EJB systems with legacy message-based systems.

### 11.2.3 When Should You Consider Avoiding MDBs?

The following scenarios, on the other hand, are typically not ideally implemented using message-driven beans:

- **When the subsequent client logic depends on a deliverable of the service request** If the client can only sensibly continue after having received a return value from the service, it does not make sense to use a MDB. In other words, of the service is naturally a *synchronous* one.
- **When performance on the request processing is important** Though clients may experience a perceived performance benefit due to not having to wait for a response from the server, request processing as such will be slower due to the service request being delivered over a messaging service. (i.e. higher latency)
- **When you need transaction control across the service request** Though the service provided by the MDB may itself be under transaction control, it cannot be part of a transaction which has elements outside the asynchronous service request. (i.e. the activities performed by a message-driven bean cannot form part of the same transaction as the activities of the component that placed the message on the queue).
- **When you need to propagate the client's security identity to the server** Security context propagation is a part of the *enterprise services layer* generated for session beans and, like transaction control, cannot be automatically conveyed across a messaging service.
- **When the client wants to process failures via exception mechanisms** Failures, like responses, would have to be sent to the client using either another message queue, or some other mechanism.
- **When you want to minimise system complexity** The client (message producer) contains more code to post a message, than what is required for simple synchronous service request (java method call). Debugging is also more complex for MDBs than when a service request is directly delivered via RMI/IOP. However, the benefits of messaging services as an enabler of asynchronous communication usually outweigh the additional complexity.

## 11.3 The Java Messaging Service (JMS)

The Java Messaging Service (JMS) is an open API which enables you to interact with any implementation of the JMS in a vendor-neutral way. Code thus remains portable across different Message-Oriented Middleware (MOM) implementations like IBM MQSeries, BEA Tuxedo/Q, Progress SonicMQ, Tibco Rendezvous, Microsoft MSMQ and others. These different vendors all supply JMS drivers which provide the coupling to their specific implementation. All Java EE application servers (such as JBoss or Glassfish) are packaged with their own JMS-compliant messaging service, but they can of course be configured to use any other compliant service.

### 11.3.1 Styles of Messages: Point-Point vs Publish-Subscribe Domains

MOMs have traditionally supported two types of messaging mechanisms, point-to-point messaging and publish-subscribe messaging.

### 11.3.1.1 The Point-to-Point Messaging Domain

Point-to-point messaging is used when you send messages which should be consumed only by a single consumer. Messages are sent to specific message queues and consumers extract messages from these queues.

---

#### Note

Multiple producers may send messages to the same queue and that multiple consumers may register with the same queue. Each message will, however, only be processed by a single consumer.

---

Point-to-point messaging is in some ways similar to making a telephone call. Multiple persons may have access to the same telephone but only a single person will answer a particular call (process a particular message).

#### 11.3.1.1.1 Characteristics of point-to-point messaging

- Each message has only one consumer.
- Receivers (message consumers) can fetch messages irrespective of whether the receiver was running when the message was sent or not.
- Receivers acknowledge successful processing of messages.

### 11.3.1.2 The Publish-Subscribe Messaging Domain

Publish subscribe messaging is used when

- you want to decouple clients from service providers or
- when the message can be processed by zero or more consumers.

In the case of publish-subscribe messaging, publishers publish messages under a topic and each message is received by all subscribers to that topic.

#### 11.3.1.2.1 Characteristics of publish-subscribe messaging

- Each message may have multiple consumers which are registered with a particular topic.
- If no consumer is registered with that topic, the message is discarded.
- Normal subscribers (message consumers) only receive messages which were sent after while they were registered with a topic and they must remain registered until they consume the message.
- In JMS subscribers may create durable subscriptions with topics which enables subscribers to receive messages which were sent while they were not active.

## 11.3.2 Message Types

The JMS specification supports the following 5 messages types, which are represented by the corresponding Java interface in the package `javax.jms`:

- **BytesMessage** This is the most primitive message which can contain, in principal, anything. However, publishers and message consumers must agree on some data format/protocol for this to be useful. Bytes messages offer services similar to that obtained when combining byte input/output streams with data input/output streams, i.e. one can read or write bytes, arrays of bytes or primitives.
- **TextMessage** Text messages are text-string based messages offering `setText()` and `getText()` services.

- **StreamMessage** is used to transmit a sequence of primitive datatypes (similar to a record containing primitive fields). The primitive data types may correspond to the column types of database table or to a file structure. It supplies an interface which is very similar to the combination of a `DataInputStream` and a `DataOutputStream` offering services like `readDouble()` and `writeInt(value)`.
- **ObjectMessage** An object message is used to send serialized Java objects. An `ObjectMessage` offers the services `getObject()` and `setObject(Object o)`.
- **MapMessage** Map messages enable publishers to send a map of key-value pairs with the keys restricted to strings, but the values may be strings, primitives or objects. Map messages supply set services like `setInt(name, value)` or `setObject(name, obj)` and query services like `getObject(name)`.

### 11.3.3 Using JMS queues and topics

#### 11.3.3.1 General Algorithm for Connecting to a Queue or Topic

When connecting to a message queue or topic, the following steps need to be performed:

1. Look up, via JNDI, a `javax.jms.ConnectionFactory`. Specifically, either a `QueueConnectionFactory` or `TopicConnectionFactory` will be looked up based on the type of messaging scenario (point-to-point or publish-subscribe).
2. Create a message sending/receiving connection via `ConnectionFactory.createConnection()`
3. Look up, via JNDI, the `javax.jms.Queue` or `javax.jms.Topic` which will be used to send/receive messages to/from.
4. Open a `javax.jms.Session` to the connection, via `Connection.createSession(...)`
5. Via the session, create a message producer (to send messages) or a message consumer (to receive messages).

For example:

```
// Look up a connection factory to send messages on a queue
QueueConnectionFactory factory =
    (QueueConnectionFactory) new InitialContext().lookup("MyQueueConnectionFactory");

// Look up our message queue
Queue queue = (Queue) new InitialContext().lookup("MyQueue");

// To send messages, open a message sending session (in this case, with authentication)
QueueConnection connection = factory.createQueueConnection("username", "password");
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

// Create a sender which will send messages on our queue
QueueSender sender = session.createSender(queue);

// Send a message
TextMessage msg = session.createTextMessage("Hello!");
sender.send(msg);
```

#### 11.3.3.2 Using JMS queues

Queues are usually used for asynchronous service requests. They enable a *point-to-point* scenario.

### 11.3.3.2.1 Connecting to A Message Queue

The code for connecting to a message queue is shown below:

```
Context context = new InitialContext();

QueueConnectionFactory queueFactory
= (QueueConnectionFactory)context.lookup
    ("ConnectionFactory");

QueueSession queueSession
= queueFactory.createQueueConnection().createQueueSession
  (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);

Queue queue = (Queue)context.lookup("queue/HiThere");
```

### 11.3.3.2.2 Developing Queue Senders

Message senders will use a block of code similar to that shown above to connect to a message queue. A `QueueSender` is then created via the `QueueSession`:

```
QueueSender sender = queueSession.createSender(queue);
```

Senders send messages to the queue by first creating and populating a message, and sending it to the appropriate queue. Below we create, populate and send a text message:

```
TextMessage message = queueSession.createTextMessage();
message.setText(messageField.getText());
sender.send(queue, message);
```

### 11.3.3.2.3 Creating Queue Receivers

Message receivers connect to a JMS implementation in the same way message publishers do. They then create a `QueueReceiver` via the `QueueSession` and attach a `MessageListener` to it. Message listeners are notified upon receipt of a new message:

```
QueueReceiver messageRecipient
= queueSession.createReceiver(messageQueue);

messageRecipient.setMessageListener(new MessageListener()
{
    public void onMessage(Message msg)
    {
        // process message
    }
});
```

### 11.3.3.3 An Example Application using Point-To-Point Messaging

We list below an application with a client, the `MessageSender` sending messages via a message queue to a `MessageRecipient`. The latter pops the received messages up on the desktop.

When you run the application, note that you can launch the sender and send a few messages before launching a recipient. The recipient will receive the messages as he comes alive.

Alternatively, if we have multiple receivers active, any one of these will receive a particular message.

Note also that the code below is portable across different JMS-compliant MOM vendors.

### 11.3.3.3.1 MessageSender.java

```
package TestJMS;

import javax.jms.*;
import javax.naming.*;
import javax.swing.*;
import java.awt.event.*;

public class MessageSender extends JFrame
{
    public static void main(String[] args)
    {
        new MessageSender().show();
    }

    public MessageSender()
    {
        createGUI();

        addSubmitListener();

        try
        {
            createMessageSender();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.exit(0);
        }
    }

    private void createGUI()
    {
        setTitle("MessageSender");
        JPanel messagePanel = new JPanel();
        messagePanel.setLayout(new java.awt.GridLayout(2,1));
        messagePanel.add(messageField);
        messagePanel.add(sendButton);
        getContentPane().add(messagePanel);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
    }

    private void addSubmitListener()
    {
        sendButton.addActionListener( new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                try
                {
                    TextMessage message
                        = queueSession.createTextMessage();
                    message.setText(messageField.getText());
                    messageSender.send(messageQueue, message);
                }
                catch (javax.jms.JMSEException e)
                {
                    JOptionPane.showMessageDialog(MessageSender.this,
                        e.getMessage(), "JMS Exception",
                        JOptionPane.ERROR_MESSAGE);
                }
            }
        });
    }
}
```

```
        JOptionPane.ERROR_MESSAGE);
    }
}
});

private void createMessageSender() throws NamingException,
                                         JMSException
{
    Context context = new InitialContext();

    QueueConnectionFactory queueFactory
        = (QueueConnectionFactory)context.lookup
            ("ConnectionFactory");

    System.out.println("Connected to Queue connection factory.");

    queueSession
        = queueFactory.createQueueConnection().createQueueSession
            (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);

    messageQueue = (Queue)context.lookup("queue/HiThere");

    messageSender
        = queueSession.createSender(messageQueue);
}

private JButton sendButton = new JButton("send message");
private JTextField messageField = new JTextField("", 40);
private QueueSender messageSender;
private QueueSession queueSession;
private Queue messageQueue;
}
```

#### 11.3.3.2 MessageRecipient.java

```
package TestJMS;

import javax.jms.*;
import javax.naming.*;
import javax.swing.*;
import java.awt.event.*;

public class MessageRecipient extends JFrame
{
    public static void main(String[] args)
    {
        new MessageRecipient().show();
    }

    public MessageRecipient()
    {
        createGUI();

        try
        {
            createMessageRecipient();
        }
```

```
        catch (Exception e)
    {
        e.printStackTrace();
        System.exit(0);
    }
}

private void createGUI()
{
    setTitle("Gold Price Subscriber");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(new java.awt.Dimension(150,150));
}

private void createMessageRecipient()
    throws NamingException, JMSEException
{
    Context context = new InitialContext();

    QueueConnectionFactory queueFactory
        = (QueueConnectionFactory)context.lookup
            ("ConnectionFactory");

    System.out.println("Connected to queue connection factory.");

    QueueConnection queueConnection
        = queueFactory.createQueueConnection();

    QueueSession queueSession
        = queueConnection.createQueueSession
            (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);

    Queue messageQueue = (Queue)context.lookup("queue/HiThere");

    QueueReceiver messageRecipient
        = queueSession.createReceiver(messageQueue);

    messageRecipient.setMessageListener(new MessageListener()
    {
        public void onMessage(Message msg)
        {
            JOptionPane.showMessageDialog
                (MessageRecipient.this, msg, "Received message",
                JOptionPane.INFORMATION_MESSAGE);
        }
    });
}

queueConnection.start();
}
}
```

#### 11.3.3.4 Using JMS Topics

Topics are usually used for distributing events or information. They enable a *publish-subscribe* scenario.

##### 11.3.3.4.1 Connecting to a topic

The code for connecting to a topic is very similar to that used to connect to a queue:

```
Context context = new InitialContext();

TopicConnectionFactory topicFactory
= (TopicConnectionFactory)context.lookup
    ("ConnectionFactory");

TopicConnection topicConnection
= topicFactory.createTopicConnection();

TopicSession topicSession
= topicConnection.createTopicSession
  (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);

Topic topic
= (Topic)context.lookup("topic/TopicName");
```

#### 11.3.3.4.2 Developing Publishers for a Topic

Publishers connect to a topic, create and populate messages and publish them with the topic:

```
TopicPublisher publisher = topicSession.createPublisher(topic);

...
TextMessage message = topicSession.createTextMessage();
message.setText("The text of the message.");
publisher.publish(topic, message);
```

#### 11.3.3.4.3 Developing Subscribers for a Topic

Developing topic subscribers is, once again, very similar to developing queue receivers. They first connect to a topic in the same way publishers do and then they create a subscriber and set a message listener for the subscriber which processes the messages:

```
TopicSubscriber subscriber = topicSession.createSubscriber(topic);

subscriber.setMessageListener(new MessageListener()
{
    public void onMessage(Message msg)
    {
        //process the message
    }
});

topicConnection.start();
```

#### 11.3.3.5 An Example Application using Publish-Subscribe Messaging

Working with topics is very similar to working with queues. The code is very close, yet the behaviour is different in some important ways:

- Multiple topic subscribers may receive the same message.
- Subscribers only receive messages which were sent while they were active.

Below we list a `GoldPricePublisher` which publishes gold price quotes to a `GoldPrice` topic and a `GoldPriceSubscriber+` which receives messages published with the `GoldPrice` topic:

### 11.3.3.5.1 GoldPricePublisher.java

```
package TestJMS;

import javax.jms.*;
import javax.naming.*;
import javax.swing.*;
import java.awt.event.*;

public class GoldPricePublisher extends JFrame
{
    public static void main(String[] args)
    {
        new GoldPricePublisher().show();
    }

    public GoldPricePublisher()
    {
        createGUI();

        addSubmitListener();

        try
        {
            createGoldPricePublisher();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.exit(0);
        }
    }

    private void createGUI()
    {
        setTitle("Gold Price Publisher");
        JPanel submitPanel = new JPanel();
        submitPanel.add(submitButton);
        submitPanel.add(goldPriceField);
        getContentPane().add(submitPanel);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
    }

    private void addSubmitListener()
    {
        submitButton.addActionListener( new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                try
                {
                    TextMessage message
                        = topicSession.createTextMessage();
                    message.setText(goldPriceField.getText());
                    goldPricePublisher.publish(goldPriceTopic, message);
                }
                catch (javax.jms.JMSException e)
                {
                    JOptionPane.showMessageDialog(GoldPricePublisher.this,
                        e.getMessage(), "JMS Exception",
                        JOptionPane.ERROR_MESSAGE);
                }
            }
        });
    }
}
```

```
        }
    }
}

private void createGoldPricePublisher() throws NamingException,
                                             JMSEException
{
    Context context = new InitialContext();

    TopicConnectionFactory topicFactory
        = (TopicConnectionFactory)context.lookup
            ("ConnectionFactory");

    System.out.println("Connected to topic connection factory.");

    topicSession
        = topicFactory.createTopicConnection().createTopicSession
            (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);

    goldPriceTopic = (Topic)context.lookup("topic/GoldPrice");

    goldPricePublisher
        = topicSession.createPublisher(goldPriceTopic);
}

private JButton submitButton = new JButton("submit");
private JTextField goldPriceField = new JTextField("", 8);
private TopicPublisher goldPricePublisher;
private TopicSession topicSession;
private Topic goldPriceTopic;
}
```

#### 11.3.3.5.2 GoldPriceSubscriber.java

```
package TestJMS;

import javax.jms.*;
import javax.naming.*;
import javax.swing.*;
import java.awt.event.*;

public class GoldPriceSubscriber extends JFrame
{
    public static void main(String[] args)
    {
        new GoldPriceSubscriber().show();
    }

    public GoldPriceSubscriber()
    {
        createGUI();

        try
        {
            createGoldPriceSubscriber();
        }
        catch (Exception e)
```

```
{  
    e.printStackTrace();  
    System.exit(0);  
}  
}  
  
private void createGUI()  
{  
    setTitle("Gold Price Subscriber");  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    setSize(new java.awt.Dimension(150,150));  
}  
  
private void createGoldPriceSubscriber()  
        throws NamingException, JMSException  
{  
    Context context = new InitialContext();  
  
    TopicConnectionFactory topicFactory  
        = (TopicConnectionFactory)context.lookup  
            ("ConnectionFactory");  
  
    System.out.println("Connected to topic connection factory.");  
  
    TopicConnection topicConnection  
        = topicFactory.createTopicConnection();  
  
    TopicSession topicSession  
        = topicConnection.createTopicSession  
            (false /*not transacted*/, Session.AUTO_ACKNOWLEDGE);  
  
    Topic goldPriceTopic  
        = (Topic)context.lookup("topic/GoldPrice");  
  
    TopicSubscriber goldPriceSubscriber  
        = topicSession.createSubscriber(goldPriceTopic);  
  
    goldPriceSubscriber.setMessageListener(  
        new MessageListener()  
    {  
        public void onMessage(Message msg)  
        {  
            JOptionPane.showMessageDialog  
                (GoldPriceSubscriber.this, msg,  
                    "Received message",  
                    JOptionPane.INFORMATION_MESSAGE);  
        }  
    });  
  
    topicConnection.start();  
}  
}
```

#### 11.3.4 The Structure of a JMS Message

Each message has the following elements:

- **Header** The header contains pre-defined JMS properties providing generic description of messages. The header is discussed in more detail below.

- **User-Defined Properties** These are name-value pairs for user-defined properties. Message consumers can select messages based on user-defined message properties.
- **Body** The body contains the content of the message which depends on the type of message.

#### 11.3.4.1 JMS Message Headers

JMS message headers contain JMS-defined properties. These message properties change as the message is in transit, i.e. the message publisher sets (implicitly or explicitly some of these) and some are set/modified by the message consumer.

Below is a list of the JMS pre-defined properties contained in the message header:

- **JMSDestination** The queue or topic for which the message is meant. This is provided by the publisher upon sending the message.
- **JMSTimeStamp** The time-instant at which the message was sent. The time-stamp is generated by the `send()` or `publish()` method.
- **JMSReplyTo** A reply-to destination (queue or topic) set by the message publisher.
- **JMSDelivered** A flag signaling whether a message has been delivered or not set by the JMS provider.
- **JMSCorrelationId** The correlation id is used to link a reply with a request.
- **JMSPriority** The priority assigned to the message.
- **JMSExpiration** The expiry date/time of the message generated by `send()` or `publish()`.
- **JMSMessageID** A unique identifier for the message generated by `send()` or `publish()`.
- **JMSDeliveryMode** The delivery modes supported by the JMS API are PERSISTENT and NON\_PERSISTENT
- **JMSType** The type of the message -- i.e. text, byte, stream, map or object message.
- **JMSRedelivered** This property is set by the JMS provider and signals whether a message has been delivered more than once to a message consumer (due to not receiving an acknowledgement).

#### 11.3.5 Durable Subscribers

If there is a single queue receiver, it receive all messages from the queue irrespective of whether the message was sent while they were active or not.

Normal topic subscribers, however, only receive messages which are sent while they are active. Durable subscribers subscribe to a topic and receive all messages sent from then onwards, even those which were sent while they were not active.

A durable subscriber to a topic are created in the following way:

```
TopicSubscriber subscriber = session.createDurableSubscriber  
                      (theTopic, "someSunscriberId");
```

#### 11.3.6 Messages Participating in Transactions

The first argument supplied when we create a queue session specifies whether the message is a transacted message or not.

```
QueueSession queueSession = queueFactory.createQueueConnection().createQueueSession (false ←  
/*not transacted*/, Session.AUTO_ACKNOWLEDGE);
```

Transacted messages create a transacted messaging session which can be committed or rolled back by calling `commit()` or `rollBack()` on the session object.

### 11.3.7 Selected message retrieval

JMS supports selecting messages on either pre-defined or user-defined message properties. For example, to select only messages with a priority of 6 or higher one can write the following code:

```
String msgSelector = "JMSPriority > 5";
QueueReceiver receiver = session.createReceiver(queue, selector);
```

The receiver will only receive messages with a priority exceeding 5.

## 11.4 Message-driven beans

Enterprise Java Beans defines message driven beans for processing asynchronous service requests received through a JMS supporting queue or topic.

### 11.4.1 Features of Message-Driven Beans

Message-driven beans differ quite significantly from session and entity beans. They are never accessed directly. They only process messages received asynchronously from a messaging service.

Features which differentiate message-driven beans from session and entity beans include:

- **Clients do not interface directly with MDBs** One does not define any of the standard interface used by clients including
  - The remote and local interfaces normally used to publish business services offered by an enterprise bean.
  - The home and local home interfaces providing clients to life-cycle services like creation and finding of enterprise beans.
- **MDBs have a single generic business service** This is the same service our queue receivers and topic subscribers needed to provide, namely the method `onMessage(Message msg)` which receives as argument a message which may be a byte, stream, text, map or object message. The MDB has to identify the message type (it could select only certain message types) and extract the information from the message and process it in some bean-specific way.
- **MDBs do not have return values and may not raise any exceptions** This is a direct implication of asynchronous messaging.

---

#### Note

Of course the ‘return’ message could resemble an exception in the form of, for example, a SOAP fault message.

---

- **MDBs are stateless** Analogous to stateless session beans, message-driven beans do not maintain conversational state across service requests (i.e. across message receipts).
- **MDBs may be durable topic subscribers** They may receive messages published on a topic even when they were not online to process them.

### 11.4.2 Developing Message-Driven Beans

Message driven beans are written as standard Java classes that implement the `javax.jms.MessageListener` interface, and are annotated with the `javax.ejb.MessageDriven` annotation. The `mappedName` annotation parameter specifies the JNDI name of the queue or topic to which the bean will be subscribed as soon as it is deployed. A simple example: could be written as such

```
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(mappedName="NameOfTopicHere")
```

```
public class MessageProcessor implements MessageListener
{
    public void onMessage(Message message)
    {
        // Extract contents of message and perform logic here
        //...
    }
}
```

#### 11.4.2.1 Developing message senders

A common scenario is to develop a session bean that receives client requests, and as part of the workflow places a message on the queue or topic. A simple bean, that receives a news article as a simple piece of text could be developed as follows:

```
import javax.annotation.Resource;
import javax.ejb.*;
import javax.jms.*;

@Stateless
@Local({NewsSubmitter.class})
@Remote({NewsSubmitter.class})
public class NewsSubmitterBean implements NewsSubmitter
{
    public void submitNews( String news ) throws Exception
    {
        // Setup connection to topic
        Session session = topicConnectionfactory.createTopicConnection()
            .createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageProducer sender = session.createProducer( topic );

        // Create message and place on topic
        TextMessage msg = session.createTextMessage( news );
        sender.send(msg);
    }

    // Use dependency injection to get a handle to the
    // messaging infrastructure (as per application
    // server configuration )

    @Resource(mappedName="TopicConnectionFactory")
    private TopicConnectionFactory topicConnectionfactory;

    @Resource(mappedName="NewsTopic")
    private Topic topic;
}
```

#### 11.4.2.2 Deploying Message-driven beans

Message-driven bean classes can simply be packaged within a standard EJB module (ejb-jar) with no special configuration required. The relevant message queues and/or topics need to have been created (via application server or messaging service configuration) before deploying the EJB module.

## 11.5 Exercises

1. Write a Message sender which sends Date objects (as an object message, not a text message) to a queue. Write a message recipient which receives that message, and displays the time it took from the point where the sender created the time stamp, to the point where the receiver was able to extract the date.

2. Write a session bean that receives news items from a client, and places them on a queue. (Each news item has a headline, a summary, and one or more categories such as Politics, Science, Sports). Write a message-driven bean that, for each sports-related item, displays the item (e.g. to standard output or using a logger)

# Chapter 12

## Singleton beans

### 12.1 Introduction

A singleton is a class which has, at most, a single access. Singletons were not traditionally supported in EJB, but as of EJB 3.1 there is explicit support for singletons.

A enterprise bean which has been declared a singleton bean will be instantiated only once by the application server and the application server will make that instance generally available (e.g. through dependency injection or JNDI lookup).

The core benefit of singletons is that one is able to share state across an entire application and across users. The singleton is generally accessible and can be injected into other enterprise beans or into other managed components.

---

**Note**

Singletons should not be over-used. They are seldom the best solution for business information or services and generally more useful for infrastructural components and access to technical resources.

---

### 12.2 Declaring singleton beans

To declare a singleton bean you annotate a normal POJO with the `@Singleton` annotation:

```
@Singleton  
public class MyPropertiesBean  
{  
    public String getProperty(String propertyName)  
    {  
        ...  
    }  
  
    private Properties properties;  
}
```

### 12.3 Using singleton beans

Singleton beans can be accessed like session beans, i.e. they can be either looked up via JNDI or injected by the application server. Normally one would prefer injection:

```
@Stateless
public class MyServiceBean
{
    public void someService()
    {
        ...
        myPropertiesBean.getProperty("msgs.helloWorld");
        ...
    }

    @EJB
    private MyPropertiesBean myPropertiesBean;
}
```

## 12.4 Startup and shutdown callbacks

Because singleton's usually provide access to application-wide resources, they are typically initialized during application startup. They can also be used to perform some other application startup initialization.

By default the EJB container will decide when to initialize the singleton instance. This could happen the first time the singleton is accessed. However, one can request that the singleton should be instantiated upon application startup using the `@Startup` annotation.

The standard bean life cycle interceptors `@PostConstruct` and `@PreDestroy` can be used to specify tasks which should be executed upon bean initialization and bean destruction. The bean destruction is done when the application is undeployed. Using the `@Startup` annotation one requests the bean instantiation to be done on application startup and in that case the tasks annotated with `PreConstruct` will also be executed upon application startup.

This is done as follows:

```
@Singleton
@Startup
public class PropertiesBean {

    @PostConstruct
    private void startup() { ... }

    @PreDestroy
    private void shutdown() { ... }

    ...
}
```

## Chapter 13

# Scheduled tasks with the Timer Service

### 13.1 Introduction

Most session beans offer services that are either invoked by clients (in the case of stateless and stateful session beans), or by the delivery of JMS messages (for message-driven beans). However, certain workflow steps may need to be performed based on scheduled, timed events (i.e. without direct user interaction).

Historically, this was accomplished by using an external timer service (such as `cron`) to invoke a program that connects to and invokes a service from an EJB. This mechanism, however, is laborious, and difficult to integrate with a Java EE solution.

As of EJB 2.1, JavaEE supported a timer service. However, the timer service was not very mature compared to scheduling frameworks like Unix cron, Quartz, ... As of EJB 3.1 the EJB timer service is a lot more feature rich and flexible. It is largely inspired by cron.

### 13.2 What can the EJB Timer Service be used for?

The timer service support scalendar based scheduling to schedule tasks to occur either once or repetitatively

- at specific particular date-times,
- in specific intervals, or
- at durations after the previous task has been completed.

The EJB timer service primarily exists to support long-running business processes. Since session beans conceptually only last for the duration of a session, the timer service extends the usefulness of EJBs in situations such as the following:

- **Time-based workflow steps** For example, imposing a fine when a rented item (library book, video, car) is not returned on time, or prompting a client to renew their support contract a month before it expires.
- **Monitoring / Polling** Periodically making sure that certain services, resources or devices are available and responding.
- **Periodical reminders** Sending a monthly reminder to users of a system to remind them to update their password for security purposes.
- **Initiating asynchronous / background tasks** Since EJBs may not create or manage their own threads, the traditional mechanism for this has always been to post a message on a message queue, to be processed by a message-driven bean. Scheduling an immediate call-back via the timer service provides a simple and light-weight alternative.

Scheduled timers are *reliable* and *robust* - they must survive application server crashes or machine restarts.

Timers may be used in both Stateless Session Beans as well as *Message-Driven Beans*.

## 13.3 Architecture of the timer service

The timer service is a container service which enables one to register session bean services to be called on timer events, i.e. one

1. either automatically (via annotations) or programmatically registers a stateless session bean service with a timer
2. after which the container will call the registered services on the corresponding timer events.

### 13.3.1 Timer services are persistent

Timer services need to support scheduling over extended periods as tasks are often recurring over extended periods or scheduled for some distant future. The EJB container automatically provides persistence for timer services.

### 13.3.2 Automatic versus programmatic timer registration

If a stateless session bean which is annotated with scheduling requests is deployed in an EJB container, the appropriate timer services are automatically created by the container.

Alternatively a bean can programmatically (e.g. in the context of realizing a user service) register some task with a timer service.

## 13.4 Specifying time instants

The EJB timer service uses a time instant specification approach which is analogous to the Unix cron syntax for scheduling jobs. One specifies the second, minute, hour, dayOfMonth, month, dayOfWeek and year and can specify relative to which time zone the time instant specification is done.

### 13.4.1 Parameters and their possible and default values

A scheduling request will refer to one or more time instants each of which is, in turn specified through a number of parameters. Each parameter can have a range of possible values and has a default value if that parameter is not specified. The parameters and their possible and default values are shown in the table below.

Parameter	Values	Default
second	0..59	0
minute	0..59	0
hour	0..23	0
dayOfMonth	1..31, -1..-31 (from end of month), or "Last"	*
month	1..12 or "Jan".."Dec"	*
dayOfWeek	0..7 or "Mon".."Fri" or expressions like "second Wed" or "last Fri"	*
year	use 4 digits	*
timeZone	as per zoneinfo DB	the default time zone for the system

Table 13.1: Time instant parameter values

### 13.4.2 Specifying time expressions

For each of the time instant parameters one may specify

- a single value, e.g. like month="7",

- a wild card to specify any, e.g. `dayOfWeek="*"`
- a comma-delimited list, e.g. `dayOfMonth="15,Last"`
- an increment specification specifying an initial value and an increment thereafter which is separated via a forward slash, e.g.
  - `minute="20/5"` which specifies after 20 minutes and every 5 minutes thereafter.
  - `hour="*/3"` which means start any hour, but then perform the task every 3 hours thereafter.

## 13.5 Automatic timer creation

If a stateful session bean has services which are annotated with the `@Schedule` annotation, timer services for those services will automatically be created. The annotated services will thus be automatically scheduled as per annotation.

For example, if you want to have a weekly sales report emailed every Friday afternoon at 16h30 could annotate the respective service as follows:

```
@Stateless
public class ReportingBean
{
    @Schedule(minute="30", hour="16", dayOfWeek="Fri")
    public void emailWeeklySalesReport()
    {
        ...
    }
}
```

You could also specify multiple scheduling requests as follows:

```
@Stateless
public class ReportingBean
{
    @Schedules({
        @Schedule(minute="30", hour="16", dayOfWeek="Fri")
        @Schedule(minute="30", hour="16", dayOfMonth="Last")
    })
    public void emailWeeklySalesReport()
    {
        ...
    }
}
```

which requests the report to be emailed every Friday as well as on the last day of the month.

## 13.6 Programmatic timer creation

At times you want to schedule tasks in response to certain business events. To this end the EJB specification supports the programmatic timer creation. This is done by asking a timer service provided by the EJB container to create an appropriate timer for a given schedule expression. Upon timeout event of the created timer, the application server will call the `@Timeout` annotated service of that stateless session bean from which the timer was created.

For example, in order to ensure that an invoice is paid on its due date, one could schedule, for each received invoice, an appropriate timer programmatically.

```
@Stateless
public class InvoiceProcessorBean
{
    public void processInvoice(Invoice invoice)
```

```
{  
    entityManager.persist(invoice);  
  
    Calendar dueDate = invoice.getDueDate();  
    int dayOfMonth = dueDate.get(Calendar.DAY_OF_MONTH);  
    int month = dueDate.get(Calendar.MONTH);  
    int year = dueDate.get(Calendar.YEAR);  
  
    ScheduleExpression payDateExpr = new ScheduleExpression().dayOfMonth(dayOfMonth).month(month).year(year);  
  
    // Now ask timer service to create persistent timer (true parameter)  
    timerService.createCalendarTimer(payDateExpr, new TimerConfig(invoice, true));  
}  
  
@Timeout  
public void payInvoice(Timer timer)  
{  
    Invoice invoice = (Invoice) timer.getInfo();  
    paymentProcessor.payInvoice(invoice);  
}  
  
@EJB  
PaymentProcessor paymentProcessor;  
  
@PersistenceContext  
private EntityManager entityManager;  
  
@Resource  
private TimerService timerService;  
}
```

# Chapter 14

## Transactions

### 14.1 Conceptual overview of transactions

Enterprise systems typically rely heavily on transaction support to ensure the integrity of their business processes.

#### 14.1.1 Introduction

So far it seems that EJBs have put us into a position where we can write server side business logic very efficiently with bean developers focusing exclusively on the definition of business logic and business information. The container supplies us with

- The ability to offer location transparent remote services to clients without having to explicitly code for RMI or CORBA.
- Automatic processing of concurrent clients.
- Resource pooling.
- Declarative security.
- Either
  - Declarative persistence entirely controlled by the container or
  - Programmatic persistence with the container ensuring the synchronization of information over different client views.

However, for robust enterprise applications we still need support for transactions. Generally this requires that developers use some transaction service (e.g. the CORBA transaction service). They would manually demarcate and commit or roll-back transactions via calls to a transaction API.

EJBs support complete declarative transactions, removing the responsibility of programming for transactions from the bean developer.

##### 14.1.1.1 Why Transactions?

Let's have a look at some problems which are addressed by transactions. In particular, consider the need for

- Either completing a set of operations as a whole or unwinding the set of operations.
- Safe concurrent access to shared information.
- Graceful recovery from a server crash or network problem.

#### 14.1.1.1.1 Atomic Operations

Sometimes a set of operations have to be completed as a whole or they have to be undone such that the state of the effected objects is the same as what it was before the set of operations was started -- i.e.. the set of operations has to be rolled back.

For example, in the case of a transfer one account is debited and another is credited. The transfer must be an atomic operation. If the debit fails the credit cannot be performed.

Conversely, if the debit was successful but the credit failed (because, say, the account has been closed) the debit has to be reversed.

#### 14.1.1.1.2 Safe Concurrent Access to Shared Information

If multiple clients access information simultaneously one has to ensure that

- Any information received represents a valid, sensible view of the business information and is not corrupted by the fact that clients are looking at information which is only partially updated.
- The information is safe with respect to updating by multiple clients. Simultaneous updating can lead to a situation where the resulting set of information is not a valid set because it contains elements of different interleaving updates from different information providers.

---

##### Note

Note furthermore that the information may be distributed across multiple databases within the organization (or even across organizations) and that the integrity of this distributed information must be protected.

---

#### 14.1.1.1.3 Graceful Failure Recovery

Consider the situation where a client withdraws funds from a auto teller. In the process of the withdrawal the network fails. The bank server may not be in a position to know whether the funds have been issued to the client or not.

After recovering from the network failure the information of whether the money has been issued to the client or not must be communicated to the server. If not, the debit of the account must be rolled back.

### 14.1.2 ACID guarantees by transaction processors

Transaction processors have to provide a number of guarantees to their clients. The core guarantees are called the ACID criteria which is short for atomicity, consistency, isolation and durability.

#### 14.1.2.1 Atomicity

Atomicity guarantees that the operations demarcated by the transaction are seen as a single unit of work which must either be completed in its entirety or, if any single operation which falls within the transaction boundaries fails, the entire set of operations have to be undone (rolled back) and the transaction is said to have failed.

#### 14.1.2.2 Consistency

Consistency guarantees that the system is left by the transaction in a valid state.

Note that during the processing of a transaction the system may be in an invalid state -- i.e.. some of the system invariants (system rules) may not be satisfied. However, before the transaction is started and after the transaction has been completed all system invariants are guaranteed to be satisfied.

---

#### 14.1.2.3 Isolation

Isolation guarantees that concurrently performing transactions do not see each others incomplete results. Isolation is achieved through locks or monitors on shared resources, i.e.. that a shared resource blocks out other threads while one thread is still busy completing a process on the shared resource.

#### 14.1.2.4 Durability

Durability guarantees that updates made by transactions persist in the persistent storage irrespective of failures which may occur -- i.e.. that the persistent storage survives failures and that committed updates remain in the database and are not affected by any failure or crash occurring after the commit.

### 14.1.3 Rolling Transaction Back

It may seem that the process of rolling back a transaction is an excessively complex process. However, database vendors have introduced the notion of provisional and permanent updates.

During the process of a transaction modifications may be made to the business data. However, databases regard these modifications as temporary until they received the go ahead that the updates may be committed to the database. The commit message is sent by the transaction manager upon having successfully completed all operations which fall within the transaction's domain.

Alternatively, if the transaction is aborted, all the temporary updates are deleted and the permanent records of the database are not affected.

#### 14.1.4 Compensating Transactions

Sometimes transactions which have been committed need to be rolled back at some later stage. The cause of this is not that a problem with any of the steps of the transaction has been encountered, but that at some higher level the decision of performing the transaction was reversed (e.g. the user changed his/her mind).

A compensating transaction is an inverse workflow which undoes a transaction by performing a sequence of reversal steps.

### 14.1.5 Transactions across distributed resources

One of the ACID properties of transactions is durability, i.e. that data which has been committed to a database are permanent -- at most they may be effectively reversed via a compensating transaction.

Durability is non-trivial for distributed transactions, i.e. if multiple datasources are involved in the same transaction. In that case you cannot complete a successful workflow by simply committing the modifications to the various datasources sequentially. Otherwise one may commit the modifications to one datasource and thereafter another datasource may fail to commit (perhaps due to a database crash or a network problem).

The problem of committing transactions which span multiple datasources is tackled via a two-phase commit.

#### 14.1.5.1 Two-Phase Commit

The first phase is a preparation phase where every element of the transaction (e.g. each database statement) is ready to commit. The transaction to be committed is written into a database log before the database acknowledges its readiness to commit. If all the resources are ready to commit the second phase of actually committing the transaction elements is launched.

Should one of the databases crash within this process, the committing of the data is performed upon restart -- the information of outstanding commits is obtained from the database log.

## 14.2 Overview

The EJB developer can choose to have the container manage transactions by setting the transaction attributes for a bean or for individual bean services or can manage the transactions programmatically.

### 14.2.1 JTS and JTA in EJB

The *Java Transaction Service* (JTS) is a Java binding of the *CORBA Transaction Service* (OTS) 1.1 specification. JTS thus provides transaction support across technologies by using the standard transaction propagation support provided by CORBA's IIOP protocol.

The EJB specification does not require the EJB server to implement JTS. It only requires the application server supports the Java Transaction API (JTA) and the Java Connector API (JCA). The latter supports the ability to propagate the transactional context across to other enterprise systems which are potentially realized in very different enterprise technologies.

### 14.2.2 Transaction managers

In the case of programmatic transaction demarcation (i.e. bean managed transactions), one interacts directly with a transaction manager. `TransactionManager` offers the following services:

- **void begin()** Create a new transaction and associate it with the current thread.
- **void commit()** Complete the transaction associated with the current thread.
- **int getStatus()** Obtain the status of the transaction associated with the current thread.
- **Transaction getTransaction()** Get the transaction object that represents the transaction context of the calling thread.
- **void resume(Transaction tobj)** Resume the transaction context association of the calling thread with the transaction represented by the supplied `Transaction` object.
- **void rollback()** Roll back the transaction associated with the current thread.
- **void setRollbackOnly()** Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.
- **void setTransactionTimeout(int seconds)** Modify the value of the timeout value that is associated with the transactions started by the current thread with the begin method.
- **Transaction suspend()** Suspend the transaction currently associated with the calling thread and return a `Transaction` object that represents the transaction context being suspended.

### 14.2.3 JTA transactions

One can also interface with the transaction itself via

- **void commit()** Complete the transaction represented by this `Transaction` object.
- **boolean delistResource(XAResource xaRes, int flag)** Delist the resource specified from the current transaction associated with the calling thread.
- **boolean enlistResource(XAResource xaRes)** Enlist the resource specified with the current transaction context of the calling thread.
- **int getStatus()** Obtain the status of the transaction associated with the current thread.
- **void registerSynchronization(Synchronization sync)** Register a synchronization object for the transaction currently associated with the calling thread.
- **void rollback()** Rollback the transaction represented by this `Transaction` object.
- **void setRollbackOnly()** Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

#### 14.2.3.1 The status of a transaction

The status of a transaction can be one of the following:

- **Status.STATUS\_ACTIVE** A transaction is associated with the target object and it is in the active state.
- **Status.STATUS\_COMMITTED** A transaction is associated with the target object and it has been committed.
- **Status.STATUS\_COMMITTING** A transaction is associated with the target object and it is in the process of committing.
- **Status.STATUS\_MARKED\_ROLLBACK** A transaction is associated with the target object and it has been marked for rollback, perhaps as a result of a `setRollbackOnly` operation.
- **Status.STATUS\_NO\_TRANSACTION** No transaction is currently associated with the target object.
- **Status.STATUS\_PREPARED** A transaction is associated with the target object and it has been prepared.
- **Status.STATUS\_PREPARING** A transaction is associated with the target object and it is in the process of preparing.
- **Status.STATUS\_ROLLEDBACK** A transaction is associated with the target object and the outcome has been determined as rollback.
- **Status.STATUS\_ROLLING\_BACK** A transaction is associated with the target object and it is in the process of rolling back.
- **Status.STATUS\_UNKNOWN** A transaction is associated with the target object but its current status cannot be determined.

#### 14.2.3.2 Marking a transaction as failed

The `setRollbackOnly()` is used quite frequently to mark a transaction as failed.

Thus, in programmatic transaction demarcation one calls `transactionManager.begin()` to start a new transaction and calls `transaction.commit()` to commit a transaction or alternatively `transaction.abort()` to abort a transaction resulting in an automatic roll-back of

- this transaction,
- all its parent transactions and
- all all other transactions hosted by the highest-level parent transaction.

#### 14.2.4 Support for multiple resources

JTA supports enlisting multiple resources into the same transaction and committing or rolling back a transaction across multiple resources. In order to decrease the likelihood of false commits across multiple resources, a two phase commit algorithm is used.

### 14.3 Declarative Transaction Demarcation

In the case of declarative transaction demarcation, the bean developer is not confronted with any transaction logic whatsoever. Once again, the bean contains only business logic. The application assembler who assembles applications from lower level beans understands the higher level workflows and the transaction support required for them.

He then specifies the required transaction support in the EJB deployment descriptors.

### 14.3.1 Transaction Attributes

Services offered by enterprise beans must have a transaction attribute which specifies the type of transaction support required by the service (or all services of the enterprise bean). The transactions attributes supported by the EJB specification are:

- BeanManaged,
- NotSupported,
- Required,
- Supports,
- Mandatory,
- RequiresNew and
- Never

The transaction attribute is specified in the `ejb-jar.xml` deployment descriptor either

- for all methods of an enterprise bean, or
- for individual methods of that bean.

---

**Note**

The latter overrides the former.

---

#### 14.3.1.1 BeanManaged

From EJB onwards only session beans are allowed to manage their own transactions. In this case the bean developer controls transactions via `begin()`, `abort()`, `commit()` and `rollback()` messages sent to the Java Transaction API (JTA).

#### 14.3.1.2 NotSupported

A bean defined with the `NotSupported` transactional attribute is not allowed to partake in any transaction. Invoking a method on a bean with this attribute has the consequence that the EJB container suspends any transaction until the method has been completed.

#### 14.3.1.3 Required

This attribute guarantees that all bean services are performed always within a transaction context. If the calling client or bean is within the scope of a transaction the requested service will be included within that transaction scope. Otherwise a new transaction is created for the service request.

In the case where a new transaction is created for a service request the transaction will be committed upon successful completion and will be rolled back if an exception is thrown and not handled within the context of the service.

#### 14.3.1.4 Supports

This attribute specifies that the bean will be included in the transaction scope if it is called from a transaction scope, but it will not create a new transaction scope if it is not called from a transaction scope.

---

#### 14.3.1.5 Manadatory

This attributes specifies that the bean services must always be called from within a transaction scope. A

```
javax.transaction.TransactionRequiredException
```

will be thrown if it is called from a client who is not operating within the context of a transaction.

#### 14.3.1.6 RequiresNew

This attribute specifies that if a bean service is called from within a transaction scope, that a new transaction is created. The current transaction is suspended until the new transaction has been committed.

If a bean service is called outside any transaction scope it simply creates a new transaction scope for that service request.

#### 14.3.1.7 Never

This attribute specifies that the bean services may not be called from any client operating within the scope of a transaction. A

```
java.rmi.RemoteException
```

is thrown if a bean service is called from a transaction scope.

### 14.3.2 Selecting a Transaction Attribute

If your method modifies information in a database you should consider using the `Required` transaction attribute which includes your service in the transaction scope of your client or, if the client does not operate within the scope of a transaction, a new transaction scope is created for that transaction.

If your business methods retrieve independent data elements from data stores you could operate within or outside transaction scope, i.e. if your client requires it you support it, but you do not require the creation of a new transaction scope. You would most probably select the `Supports` transaction attribute for your bean.

Finally, if you are using resources which do not support transaction management by an external transaction manager, then you have no choice but selected the `NotSupported` attribute for your bean methods.

### 14.3.3 Annotating transaction attributes

Transaction attributes are specified by annotating bean methods using the `TransactionAttribute` annotation:

```
@Stateful
public class ShoppingBean implements ShoppingRemote
{
    ...

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public Invoice process(Order order)
    {
        ...
    }
}
```

### 14.3.4 Transaction boundaries on method boundaries

Using container managed transactions requires that transaction boundaries fall on method boundaries. This is usually not a problem and if that is not the case, it usually means that responsibilities are not sufficiently localized and that the design could benefit from refactoring. If one still requires finer grained transaction boundaries, one needs to go to container managed transactions.

## 14.4 Transaction Isolation Levels

Enterprise bean developers can control how isolated your bean's transaction scope is from other transactions. The choice of isolation level must be weighed up against its performance cost.

### 14.4.1 Serializable

This is the most stringent isolation level which enforces that your transaction requires exclusive read and write privileges to data. Transactions can neither read nor write the same data your transaction accesses until your transaction has committed.

### 14.4.2 Repeatable Reads

This isolation level prevents your transaction to modify any data which is read by other transactions, i.e. there may be multiple concurrent reads, but no thread can be writing while other threads are reading.

Phantom reads can occur. A phantom read is the reading of a database record which was not yet present when your transaction started, but has been added while your transaction is busy processing.

### 14.4.3 Read Committed

This isolation level prevents your transaction from reading uncommitted data. As with repeatable reads, phantom reads can occur. Furthermore non-repeatable reads may occur. A non-repeatable read is one where the information may change during the same transaction, i.e. if the data is read again during the same transaction it may lead to different results.

### 14.4.4 ReadUncommitted

This is the weakest isolation level allowing your transaction to read data from databases which has not yet been committed. Reading of uncommitted data is referred to as dirty reads.

In the case of the Dirty Read Problem one transaction can perform a credit and before the transaction is committed another application reads the uncommitted ("dirty") new balance. At a later stage it may happen that the credit transaction is rolled back and the application is working with the incorrect balance.

## 14.5 Setting EJB Isolation Levels

The EJB and JTA specifications don't specify a standard API for managing the isolation level of a resource, i.e. the degree to which the access to the resource by one transaction is isolated from other transactions. The setting of the isolation level will thus be resource manager specific.

# Chapter 15

## Java EE Security

### 15.1 Java EE Security Overview

Java EE Applications typically consist of different component types, deployed into different containers (such as the Web Container and the EJB Container). These component containers provide *security services* that can provide restricted access to services in your application, based on the developer abstractly specifying

- The different *roles* played by users in the system (typically a direct mapping of the *actors* in your use-case model)
- The behaviour of service requests made by users in the different roles. This is either done simply and declaratively (allowed/denied) or may be customised to the application's needs (for example, by showing a role-customised user interface, or returning role-specific results in an operation) by programmatically writing behaviour. Often, a combination of both are implemented.
- The mechanism to interact with the user, and request security *credentials* when needed.

None of these aspects in Java EE are designed to reside with your logic components, but are typically ‘layered’ on top of application components. This ensures that, even in the context of elaborate security schemes, your logic components remain loosely coupled, portable, and simple to maintain.

Only when your Java EE Server is configured or your application is deployed, are decisions made such as the security *realm* (users database) to be used, and how user *roles* are resolved (for example, against your company’s existing LDAP-based user authentication system).

#### 15.1.1 Authentication and authorization

Authentication is done at the access/presentation layer (e.g. in the web container) whilst the business logic / services performs authorization on the business services offered.

In the authentication phase the following steps are performed:

1. The user provides some authentication credentials (e.g. username and password).
2. The authenticity of the user is verified by checking the authentication credentials and the user is assigned a known principal.
3. The security roles assigned to the principal are sourced and maintained within the session context.

The information about the authenticated principal and assigned security roles is transmitted across service boundaries and even containers with the session context

---

#### On need authentication

It is common to automatically request authentication when a service which is accessible only to authorized user roles is requested. This is supported by application servers which, in the presentation layer, will automatically throw up a login panel when the need arises.

---

### 15.1.2 Mechanisms to specify security services

Throughout the Java EE environment, there are typically two mechanisms used to specify how security is applied to your components:

- **Declarative security** Enables the specification of the security requirements for a component or service declaratively, either using *annotations* or localizing that information externally within a *deployment descriptor*. When the application is deployed, the container may use the security requirement declarations to configure security, or may override it if configured to do so.
- **Programmatic security** Coded logic in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. This is still typically *layered* on top of logic components, for example by using EJB Interceptors or Servlet filters.

### 15.1.3 Containers Shared Security

Figure 15.1: Java EE Container Security

Several of the different containers present in a Java EE platform (EJB, Web, Persistence, etc) often participate in the context of realising a single service. In this regard, security and permissions information needs to be shared, and this done automatically and transparently.

For example, when a user makes an authenticated request to the presentation layer running in the web container, and that presentation layer requests services from a secured Session Bean in the EJB container, authentication and role information is automatically propagated.

This scheme, however, relies on trust between the different containers, as the user is not re-authenticated every time the services of a different container are accessed based on his request.

## 15.2 Declarative security

When using declarative the role requirements for a service (or all services of a bean) are annotated, specifying that authenticated principals require a particular security role in order to be able to access that service.

### 15.2.1 Authorization annotations

The EJB specification provides the following set of authorization annotations:

### 15.2.2 Specifying authorization requirements

Commonly one uses the `@RolesAllowed` annotation on session beans and session bean services to specify that certain services should only be accessible by authenticated principals who have been assigned the required security roles.

#### 15.2.2.1 Specifying the authorization requirements for a service

To specify that a particular service should be available only to authenticated principals who have one of a number of security roles assigned to them, one annotated the respective service with a list of roles:

Annotation	Authorization requirement	Bean	Service
@RolesAllowed	Resource accessible by a list of roles	yes	yes
@PermitAll	Resource accessible by all roles	yes	yes
@DenyAll	Resource not accessible by any roles	yes	no
@RunAs	Assign role to principal for that component's services only	yes	no
@DeclareRoles	Declare a list of roles	yes	no

Table 15.1: Supported EJB authorization annotations

```

@Stateless
public class CourseServices
{
    @RolesAllowed({"user", "trainingAdministrator", "standardsManager"})
    public CourseDetails provideCourseDetails(CourseDetailsRequest request)
    {
        ...
    }
}

```

This specifies that the `provideCourseDetails` service is available to users who have any of the three specified security roles.

### 15.2.2.2 Default authorization requirements

By annotating a bean with a `@RolesAllowed` annotation, one specifies the default role requirements for services offered by that bean.

```

@Stateless
@RolesAllowed({"user", "trainingAdministrator", "standardsManager"})
public class CourseServices
{
    public CourseDetails provideCourseDetails(CourseDetailsRequest request)
    {
        ...
    }

    public PrerequisitesList provideCoursePrerequisites(CoursePrerequisitesRequest request)
    {
        ...
    }
}

```

This specifies that all services of the `CourseServices` bean are by default available to users who have any of the three specified security roles.

#### 15.2.2.2.1 Overriding default authorization requirements

One can override the default authorization requirements for a bean by specifying different authorization requirements for certain of the bean services. For example

```
@Stateless
@RolesAllowed({"user", "trainingAdministrator", "standardsManager"})
public class CourseServices
{
    public CourseDetails provideCourseDetails(CourseDetailsRequest request)
    {
        ...
    }

    public PrerequisitesList provideCoursePrerequisites(CoursePrerequisitesRequest request)
    {
        ...
    }

    @RolesAllowed("trainingAdministrator")
    public void updateCourseDetails(CourseDetails courseDetails)
    {
        ...
    }
}
```

specifies that the bean services are by default accessible by all 3 roles, but that the `updateCourseDetails` service is accessible only to the `trainingAdministrator` role.

### 15.2.3 Run-as

In the context of assembling higher level services from lower level services one encounters, at times, a situation where a user has the required security roles for the higher level service, but not for one of the lower level services called from the higher level services. In such cases one can temporarily assign the required security role for the lower level service to users who make use of the higher level service by annotating the bean offering the higher level service with a `@RunAs` annotation.

For example,

```
@Stateless
@RolesAllowed({"client", "salesRep"})
@RunAs("stockManagement")
public class OrderProcessorBean
{
    public OrderResult processOrder(Order order)
    {
        ...

        // The following service requires the stockManagement role:
        inventory.releaseStock(stockReleaseRequest);

        ...
    }

    @EJB
    private Inventory inventory;
}
```

ensures that the `releaseStock` service is accessible from the `processOrder` service even though users of the `processOrder` service may not have the required security role for the `releaseStock` service by temporarily allocating the `stockManagement` role to the context of the services of the `OrderProcessorBean`.

### 15.2.4 Declaring additional roles

Any referenced roles are automatically declared for the application server. The `@DeclareRoles` annotation enables one to declare some additional roles which are not used within the bean itself, but which should be made available to the application server. The `@DeclareRoles` annotation takes a list of roles, just like the `@RolesAllowed` annotation.

## 15.3 Programmatic security

For many cases declarative security is sufficient. However, for more complex security rules like that a particular role may only authorize purchases up to some amount, one may need to use programmatic authorization.

To this end the EJB session context provides two services

- `boolean sessionContext.isCallerInRole(String roleName)` which returns true if the user of the service has been assigned the specified security role, and
- `java.security.Principal sessionContext.getCallerPrincipal()` which returns the the caller principal which enables one to
  - retrieve the name of the caller principal, and
  - verify whether the principal is equal to another principal.

Upon security breaches one should throw a `java.lang.SecurityException`. For example

```
@Stateless
public class ProcurementBean
{
    public AuthorizeProcurementResult authorizeProcurement(AuthorizeProcurementRequest request)
    {
        if ((request.getTotal() > departmentalLimit)
            && (!sessionContext.isCallerInRole("organizationLevelAuthorizer")))
            throw new SecurityException("RequireOrganizationLevelAuthorization");

        ...
    }

    @Resource
    private SessionContext sessionContext;
}
```

---

#### Note

The session context is injected by the application server.

---

In exceptional situations one might even need to check whether a user is a particular authenticated principal:

```
if (sessionContext.getCallerprincipal().getName().equals("ThaboKhumalo"))
    ...
```

## Chapter 16

# The JEE Connector Architecture

## 16.1 Introduction

When developing Enterprise systems within Java based architectures like JEE (the Java Enterprise Edition) and JBI (the Java Business Integration) reference architectures, one is encouraged to use the Java EE Connector Architecture for the base connectivity. The Java EE Connector Architecture provides a standard framework for resource adapters to systems not deployed within the application server which hosts the component making use of the resource. The Java EE Connector Architecture is commonly abbreviated as JCA though this abbreviation is not actually strictly speaking correct as it is used for the Java Cryptography Extension. Using JCA connectors enables one to integrate external resources such as legacy systems or messaging providers a standard way into JEE compliant application servers such that they become an integral part of the managed application server environment.

### 16.1.1 What is the Java Connector Architecture?

The Java Connector Architecture defines a standard architecture for connecting Java based systems (and in particular JEE systems) to heterogenous Enterprise Information Systems (EISs).

Resource adapters are typically used to provide scalable, secure, transactional integration with Enterprise Information Systems. When used by application servers, it enables the application server to

- take over the management of connection resources, and to
- provide enterprise services like security and transaction management across the enterprise solution.

---

#### Note

Once a connector for an EIS has been developed, it can be deployed on any application server that conforms to the JEE specifications.

---

### 16.1.2 Example Enterprise Information (EIS) systems

Examples for EISs include

- legacy systems accessible through a proprietary protocol,
- database systems supporting either some version of SQL or some proprietary protocol,
- mainframe transaction processing (TP) systems,
- message queues,
- devices like ATMs, ...

### 16.1.3 Simplifying resource managers via the Java Connector Architecture

In many ways the Java Connector Architecture simplifies the application server/resource manager's responsibilities. resource managers/application servers need no longer know about JDBC drivers, message queues, web services end points and so on as these can all be treated generically within the connector specification.

### 16.1.4 Who uses connector resources?

Resource adaptors can be used by any Java based system. The Java Connector Architecture specification exposes a standard API for connection resource management, security and transactions which can be used by any Java based system.

Very commonly, however, connector resources are used from within application servers and on the Enterprise Services Bus (ESB) - an implementation of a Services Oriented Architecture (SOA) which may be based on the Java Business Integration (JBI) specification. This enables the application server or ESB to effectively manage the connection resources and in order to address non-functional concerns like

- scalability through effective resource management,
- security, and
- transactions.

### 16.1.5 Who provides JCA adapters?

JCA adaptors are often provided by

- vendors of the various enterprise information systems (EISs) themselves,
- by third-party vendors,
- or by the open source community.

In cases where there is no JCA adaptor available, one may need to develop a JCA adaptor.

### 16.1.6 Features of Java Connector Architecture

The Java Connector architecture offers the following features:

- JCA enables application servers to manage connection pools to proprietary and legacy systems.
- JCA supports both, outbound as well as inbound connection management.
- JCA enables one to include external resources into application server managed transactions.
- JCA supports application server authentication with EISs as well as the transmission of security credentials enabling the external EIS to enforce authorization rules.

### 16.1.7 Why use resource adapters?

Resource adapters enable one to keep non-functional aspects out of the code containing the core business logic. These include

- the logic to establish connections,
- connection pooling,
- marshalling of outgoing messages onto the EIS protocol and demarshalling of the responses,
- mapping incoming messages onto events which are processed by application components,
- transaction management across the EIS, and
- security.

## 16.2 Managed environments

More and more software is developed to be employed and executed within managed environments which provide a level of resource management to the application.

### 16.2.1 What is a managed environment?

A managed environment is an environment where the application logic cannot directly access any resources. Resources which are managed in a managed environment typically include

- the management of CPU resources via thread pooling,
- the management of memory resources via re-use of pooled objects and garbage collection, and
- the management of re-usable connection resources via connection pooling.

In a fully managed environment application developers are forbidden to directly access any resources. The application code may thus not

- spawn any threads as that would interfere with the management of CPU resources,
- open any connections to any external system as that would prevent the managed environment from managing the connection resources,
- delete any objects (this is left to the garbage collector). Furthermore, pools of entity and session objects are managed within the managed environment. Instances of these are not created by the application itself, but are requested from the managed environment.

#### 16.2.1.1 Examples of managed environments

Examples of managed environments include

- the web based presentation layer and the business logic layer of J2EE application servers, i.e. the web and EJB containers of a J2EE application server,
- the Microsoft .Net platform,
- to a lesser extend Java Runtime Environments (they only provide limited support for memory management via garbage collection),
- implementations of an enterprise services bus which typically, but not necessarily use an application server under the hood for the managed environment.

## 16.3 Design of the Java EE Connector Architecture

In this section we look at the design of the JEE Conenctor architecture. We cover

- the core contracts of the framework,
- the core components and the responsibilities assigned to each component, and at
- how the workflow is realized across these components.

### 16.3.1 The high level contracts defined for the Java Connector Architecture

Resource adapters have 3 interfaces

- The interface between the client and the resource adapter, the Common Client Interface (CCI).
- The application server communicates with the resource adapter via the service provider interface (SPI).
- The resource adapter communicates with the EIS via the EIS specific protocols/API.

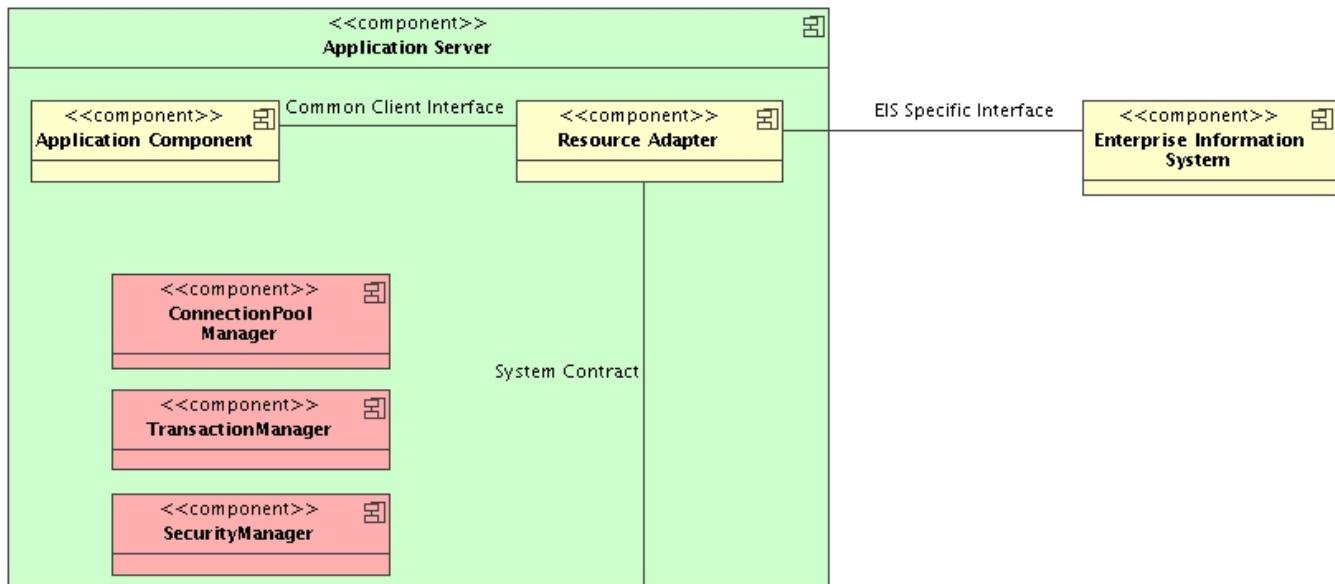


Figure 16.1: Contracts around the resource adapter

#### 16.3.1.1 The Common Client Interface (CCI)

The goal of the common client interface is to provide an EIS independent interface for identifying the service which is requested from the EIS.

#### 16.3.1.2 The Service Provider Interface (SPI)

The service provider interface enables application servers to effectively manage connection resources and connection pools as well as to apply non-functional responsibilities like security and transactions across an EIS accessed via a resource adaptor.

The SPI covers

- *life cycle management* enabling application servers to start up and bring down resource adaptors,
- *connection management* enabling application servers to effectively manage both, outbound and inbound connections,
- *security* enabling application servers to log into an EIS and to exchange security credentials with the EIS.
- *transaction management* facilitating that transactions can span across components deployed within an application server and external EISs.

### 16.3.2 Responsibility allocations across core Java Connector components

The design of the JEE Connector Architecture is based on a clean responsibility allocation across the core components. Understanding these responsibilities is core in getting comfortable with the design of Java resource adapters.

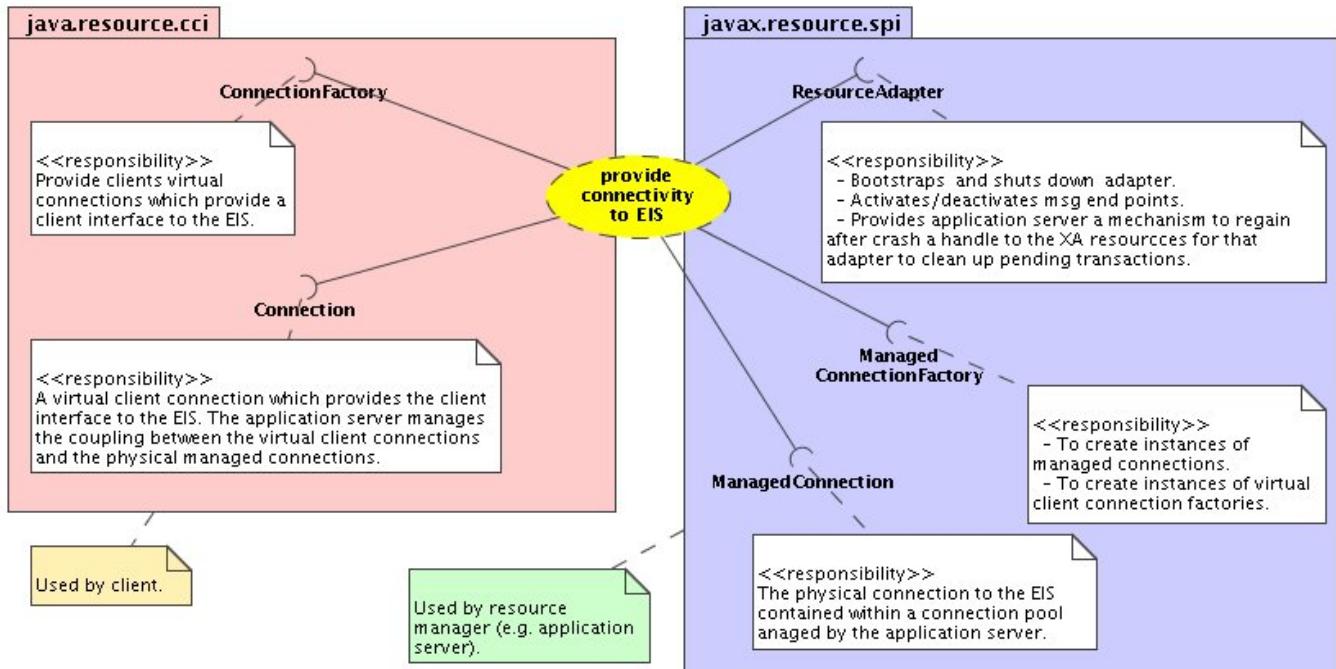


Figure 16.2: Responsibility allocations across JCA components

### 16.3.3 Resource Adapter Life Cycle Management

The managed environment (e.g. application server) will need to manage the life cycle of the resource manager. This includes

- starting a resource adapter via some bootstrapping process, and
- bringing a resource adapter down.

#### 16.3.3.1 The typical life cycle of a resource adapter

The typical life cycle of a resource adapter includes

- starting the resource adapter via a bootstrapping process,
- using the connections provided by the resource adapter, and
- bringing down a resource adapter.

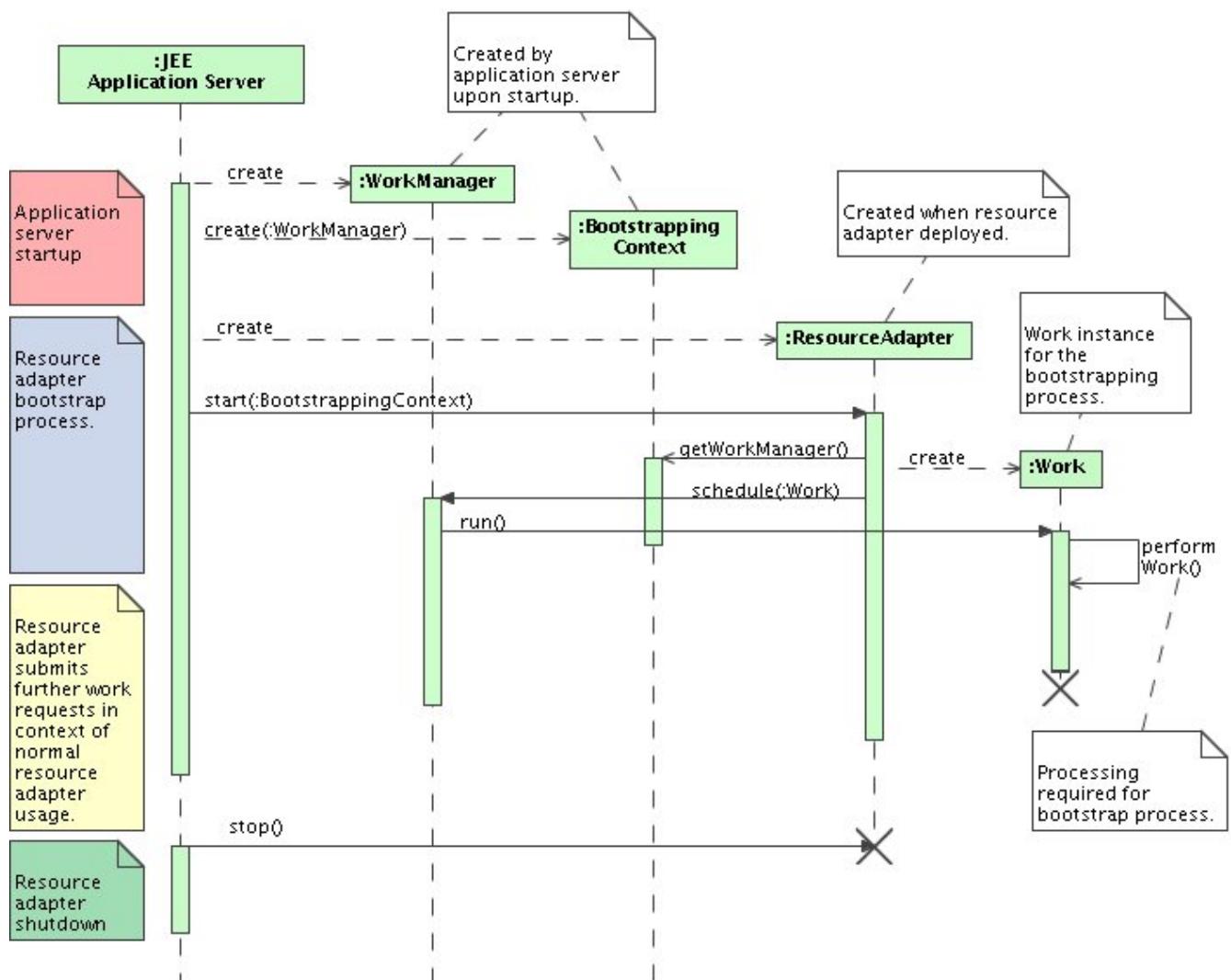


Figure 16.3: Life cycle of a resource adapter

In either of these phases the resource adapter may require processing time. Processing time could be required for

- setting up a resource adapter including setting up polling threads to poll inbound connections for information,
- work around processing requests made through connections in the context of normal use of the resource adapter, and
- work around bringing down a resource adapter.

In either case the adapter must submit any requests for significant work to the application server as not to interfere with the CPU resource management of the application server.

#### 16.3.3.2 The ResourceAdapter class

The `ResourceAdapter` class provides developers the opportunity to customize the life cycle management services of a resource adapter. Only needs to provide an implementation for this class if there are any special tasks which need to be performed when starting or stopping the resource adapter. Consequently, many resource adapter implementations do not provide their own implementation of the `ResourceAdapter`.

### 16.3.3.2.1 The bootstrap context

Before bootstrapping a resource adapter, the application server will create a work manager for the resource adapter. It is through this work manager that the resource adapter can submit work requests to the application server.

The resource adapter can obtain a reference to its work manager via the bootstrapping context and may choose to maintain a reference over its life span in order to be able to submit further work requests at a later stage.

## 16.4 Example outbound connector: ElectionConnector

In this example we access an election server from a session bean via a resource adapter providing outbound connections to the election server. The core components of the application are

1. an election server which accepts TCP/IP connections and processes service requests provided in a proprietary text protocol,
2. a resource adapter for the above election server,
3. a session bean offering the election server's services to EJB clients, and
4. an EJB client which exposes the election servers services within a swing GUI.

### 16.4.1 The election server

The election server is a simple implementation of a TCP/IP server which launches a new thread for every new connection. It reads lines of text and interprets each line as a request in the context of a simple text based protocol:

```
package za.co.solms.election.server;

import java.io.*;
import java.net.*;
import java.util.Map;
import java.util.StringTokenizer;
import java.util.TreeMap;

public class ElectionServer
{
    /**
     * A thread processing requests from a particular connection.
     * It extracts requests from a socket, forwards them to a
     * controller and send the response back via the socket.
     */
    class ConnectionHandler extends Thread
    {
        /**
         * Creates a connection handler for a provided socket and
         * a provided controller.
         */
        public ConnectionHandler
            (Socket socket, int connectionNo, Controller controller)
        {
            this.connectionNo = connectionNo;
            this.controller = controller;
            this.socket = socket;
        }

        /**
         * The run method for the thread contains the outer
         * request processing loop.
        */
    }
}
```

```
/*
public void run()
{
    try
    {
        BufferedReader in = new BufferedReader(new InputStreamReader
            (socket.getInputStream()));
        PrintStream out = new
            PrintStream(socket.getOutputStream());

        while(true)
        {
            String command = in.readLine();
            String response = controller.processCommand(command);
            out.println(response);
        }
    }
    catch ( Exception e )
    {
        System.out.println
            ("Closing connection " + connectionNo + " due to " + e);
    }
    finally
    {
        try
        {
            socket.close();
        }
        catch (Exception e)
        {}
    }
}

private Socket socket;
private int connectionNo;
private Controller controller;
}

/**
 * The controller demarshals the incoming request,
 * maps it onto the business logic layer, and
 * marshalls the response back onto the communication protocol.
 */
class Controller
{
    /**
     * Creates a controller for a provided election manager.
     */
    public Controller(ElectionManager electionManager)
    {
        this.electionManager = electionManager;
    }

    /**
     * the command processing service.
     */
    public String processCommand(String command)
    {
        StringTokenizer tokenizer = new StringTokenizer(command, "|");
        String cmd = tokenizer.nextToken();

        if (cmd.equals("addVotes"))

```

```
{  
    String party = tokenizer.nextToken();  
    electionManager.addVotes  
        (party, Integer.parseInt(tokenizer.nextToken()));  
    return "";  
}  
  
if (cmd.equals("getVotes"))  
{  
    return Integer.toString  
        (electionManager.getVotes(tokenizer.nextToken()));  
}  
  
return "Error: illegal command";  
}  
  
private ElectionManager electionManager;  
}  
  
/**  
 * The business logic class which maintains the votes  
 * for the various parties.  
 */  
class ElectionManager  
{  
    /**  
     * Adds a number of votes to a party.  
     */  
    private void addVotes(String party, int numVotes)  
    {  
        if (votes.get(party) == null)  
            votes.put(party, numVotes);  
        else  
        {  
            int currentVotes = votes.get(party);  
            votes.put(party, currentVotes + numVotes);  
        }  
    }  
  
    /**  
     * Returns the number of votes for a specified party.  
     */  
    public int getVotes(String party)  
    {  
        Integer numVotes = votes.get(party);  
  
        if (numVotes == null) return 0;  
  
        return numVotes.intValue();  
    }  
  
    private Map<String, Integer> votes  
        = new TreeMap<String, Integer>();  
}  
  
/**  
 * The main loop accepting new connections and spawning  
 * threads processing requests off these connections.  
 */  
public void run()  
{  
    int connectionNo = 1;
```

```
try
{
    ServerSocket server = new ServerSocket(serverPort);

    System.out.println
        ("Waiting for connections on port " + serverPort);

    while (true)
    {
        Socket socket = server.accept();
        System.out.println
            ("New connection no = " + connectionNo);
        new ConnectionHandler(socket, connectionNo, controller).start();
        connectionNo++;
    }
}
catch (IOException exception)
{
    System.out.println(exception);
    exception.printStackTrace();
}
}

public static void main(String[] args)
{
    new ElectionServer().run();
}

private static final int serverPort = 12345;
private ElectionManager electionManager = new ElectionManager();
private Controller controller = new Controller(electionManager);
}
```

### 16.4.2 The election connector

The election connector contains a range of interfaces and classes in order to fulfill both,

- the service provider contract.
- the client contract, and

#### 16.4.2.1 The Service Provider Interface (SPI)

The service provider interface provides the application server view onto the resource adapter. It exposes

- the resource adapter,
- physical connections managed by the application server,
- managed connection factories, and
- as well as request information objects and metadata for the resource adapter.

##### 16.4.2.1.1 The resource adapter

Our election server does not strictly require a resource adapter implementation since it does not perform any special tasks on startup and since it does not need to submit work requests to the application server.

However, for pedagogical reasons and for reasons of future extensibility we have included an implementation:

```
package za.co.solms.election.connector;

import java.util.logging.Level;
import javax.resource.spi.endpoint.MessageEndpointFactory;
import javax.resource.spi.ActivationSpec;
import javax.resource.spi.BootstrapContext;
import javax.resource.spi.ResourceAdapterInternalException;
import javax.resource.spi.ResourceAdapter;
import javax.resource.spi.work.WorkManager;
import javax.resource.ResourceException;
import javax.transaction.xa.XAResource;

import java.util.HashMap;
import java.util.logging.Logger;

/**
 * The implementation of the ElectionServer resource adaptor
 * providing life cycle services (i.e. support for bootstrapping
 * and bringing down the resource adapter) as well as
 * services for message endpoint setup.
 *
 * This class must be a JavaBean..
 *
 * @author fritz@solms.co.za
 */
public class ElectionServerAdaptor implements ResourceAdapter
{
    /**
     * Get the work manager used by the resource adapter to submit
     * work requests to the application server.
     *
     * @return the work manager used by the resource adapter.
     */
    public WorkManager getWorkManager()
    {
        return ctx.getWorkManager();
    }

    /**
     * This service is called by the application server to
     * bootstrap the resource adapter.
     *
     * The application server provides the bootstrapping context
     * which contains the work manager made available to the
     * resource adapter. The resource adapter may use the
     * work manager to submit work requests to the application server.
     */
    public void start(BootstrapContext ctx)
        throws ResourceAdapterInternalException
    {
        logger.info("Starting resource adapter.");
        this.ctx = ctx;

        /*
         * If significant further work needs to be done or
         * if polling threads need to be used to poll for
         * incoming messages, then the resource manager
         * can obtain a work manager provided
         * by the application server from the
         * bootstrapping context via
         */
    }
}
```

```
* WorkManager mgr = ctx.getWorkManager();  
*  
* and submit work requests to it.  
*/  
}  
  
/**  
 * Disassociates the work manager such that the resource  
 * adapter will no longer receive any CPU resources  
 * from the application server.  
 */  
public void stop()  
{  
    logger.info("Stopping resource adapter.");  
}  
  
/**  
 * Called by application server to activate any end points  
 * through which the resource adapter receives any incoming  
 * messages.  
 * If a resource adapter has no invoming connections  
 * (as is the case for this election adapter), this service  
 * is not used.  
 *  
 * @param endPointFactory the factory which can be used by  
 *                         the resource adapter to create new end points.  
 * @param activationSpec The specification which contains  
 *                         the configuration informaton for the messaging  
 *                         endpoint.  
 */  
public void endpointActivation(MessageEndpointFactory endpointFactory,  
    ActivationSpec activationSpec)  
throws ResourceException  
{  
    logger.info("endpointDeactivation for spec " + activationSpec);  
/*  
 * Specify here the code to activate any endpoints from which the  
 * resource adapter receives any incomning messages.  
 */  
}  
  
/**  
 * Called by application server to deactivate any end points  
 * through which the resource adapter receives any incoming  
 * messages.  
 * If a resource adapter has no invoming connections  
 * (as is the case for this election adapter), this service  
 * is not used.  
 *  
 * @param endPointFactory the factory which can be used by  
 *                         the resource adapter to create new end points.  
 * @param activationSpec The specification which contains  
 *                         the configuration informaton for the messaging endpoint  
 */  
public void endpointDeactivation(MessageEndpointFactory endpointFactory,  
    ActivationSpec activationSpec)  
{  
    logger.info("endpointDeactivation for spec " + activationSpec);  
}  
  
/**  
 * Provides the application server a handle to the XAResources
```

```
* (resources enlisted within distributed transactions).
* This method is used by application servers on crash recovery to
* clean up dangling transactions.
*/
public XAResource[] getXAResources(ActivationSpec[] specs)
        throws ResourceException
{
    return new XAResource[0];
}

private static final Logger logger
= Logger.getLogger(ElectionServerAdaptor.class.getName());

private BootstrapContext ctx;
}
```

#### 16.4.2.1.2 The managed connection

Managed connections represent physical connections which will be managed by the application server. Managed connections create the underlying physical connection and provide

- an implementation of the EIS services published through the connector,
- the association of new virtual connections to this physical connection,
- event notification around connection events (the application server and/or connection manager will register as event listeners) and
- a handle to the transaction resources associated with this physical connection.

```
package za.co.solms.election.connector;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.io.PrintWriter;
import java.util.logging.Level;
import javax.resource.spi.ManagedConnection;
import javax.resource.spi.ConnectionEventListener;
import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.spi.LocalTransaction;
import javax.resource.spi.ManagedConnectionMetaData;
import javax.resource.spi.ConnectionEvent;
import javax.resource.ResourceException;
import javax.transaction.xa.XAResource;
import javax.security.auth.Subject;

import java.util.logging.Logger;

/**
 * A ManagedConnection representing a physical connection
 * to the underlying EIS providing the application server
 * the means to
 * <ul>
```

```
* <li>create physical connections,</li>
* <li>associate physical connections
*      to the managed connection, </li>
* <li>receive state events (e.g. close events)
*      from physical connections,</li>
* <li>
*      provide access to the XAResource interface is used
*      by the transaction manager to associate and dissociate
*      a transaction with the underlying EIS resource manager
*      instance and to perform two-phase commit, and to
* </li>
* <li>
*      provide access to the LocalTransaction interface is used
*      by the application server to manage local transactions,
*      i.e. transactions which only have only enlisted local
*      resources.
* </li>
* </ul>
*
* @author Fritz Solms
*/
public class ManagedElectionServerConnection
    implements ManagedConnection
{

    /**
     * Creates new physical connection to an instance
     * of an election server.
     */
    public ManagedElectionServerConnection(Subject subject,
        ElectionServerConnectionRequestInfo connectionRequestInfo)
        throws UnknownHostException, IOException
    {
        logger.log(Level.FINEST,
            "Creating virtual connection for request info "
            + connectionRequestInfo);

        socket = new Socket
            (InetAddress.getByName
                (connectionRequestInfo.getHostName()),
            connectionRequestInfo.getPort());

        in  = new BufferedReader(new InputStreamReader
            (socket.getInputStream()));
        out = new PrintStream(socket.getOutputStream());
    }

    /**
     * Creates a new virtual connection for the underlying
     * physical connection represented
     * by the ManagedConnection instance.
     */
    public Object getConnection
        (Subject subject, ConnectionRequestInfo info)
        throws ResourceException
    {
        logger.log(Level.FINEST, ("Getting a connection for subject "
            + subject + " and info " + info));
        if( virtualConnection == null )
            virtualConnection = new ElectionServerConnectionImpl(this);
        return virtualConnection;
    }
}
```

```
/**  
 * Adding a new connection event listener.  
 */  
public void addConnectionEventListener  
        (ConnectionEventListener connectionEventListener)  
{  
    logger.log(Level.FINEST, "Adding ConnectionEventListener "  
              + connectionEventListener);  
    listeners.add(connectionEventListener);  
}  
  
/**  
 * Removing a specified connection event listener.  
 */  
public void removeConnectionEventListener  
        (ConnectionEventListener connectionEventListener)  
{  
    logger.log(Level.FINEST, "Removing ConnectionEventListener "  
              + connectionEventListener);  
    listeners.remove(connectionEventListener);  
}  
  
/**  
 * Associates a new virtual connection to this physical connection.  
 */  
public void associateConnection(Object obj) throws ResourceException  
{  
    logger.log(Level.FINEST, "Associating physical connection "  
              + obj + " to this virtual connection.");  
    try  
    {  
        virtualConnection = (ElectionServerConnectionImpl) obj;  
        virtualConnection.associateManagedConnection(this);  
    }  
    catch (ClassCastException e)  
    {  
        throw new ResourceException  
              ("Virtual connection not an ElectionServerConnectionImpl");  
    }  
}  
  
/**  
 * Requests the EIS to add a number of votes for a party.  
 *  
 * The service marshalls the request onto the EIS protocol,  
 * sends the request via the socket connection to the EIS,  
 * waits for a response, and demarshalls it.  
 */  
public void addVotes(String party, int numVotes)  
{  
    logger.log(Level.FINEST, "Adding " + numVotes + " to " + party);  
  
    String request="addVotes|" + party + "|" + numVotes;  
  
    try  
    {  
        out.println(request);  
        String response = in.readLine();  
    }  
    catch (IOException e)  
    {  
    }
```

```
        notifyOfError(e);
    }
}

/**
 * Requests the number of votes for a party from the EIS.
 *
 * The service marshalls the request onto the EIS protocol,
 * sends the request via the socket connection to the EIS,
 * waits for a response, and demarshalls it.
 */
public int getVotes(String party)
{
    logger.log(Level.FINEST, "Retrieving votes for " + party);

    String request="getVotes|" + party;

    try
    {
        out.println(request);
        String response = in.readLine();
        return Integer.parseInt(response);
    }
    catch (Exception e)
    {
        notifyOfError(e);
        return 0;
    }
}

private void notifyOfError(Exception e)
{
    logger.log(Level.ALL, e.toString());
    ConnectionEvent connectionEvent
        = new ConnectionEvent
            (this, ConnectionEvent.CONNECTION_ERROR_OCCURRED);
    connectionEvent.setConnectionHandle(virtualConnection);
    fireConnectionEvent(connectionEvent);
}

/**
 * Called by application server to force any cleanup
 * by this physical connection instance.
 */
public void cleanup() throws ResourceException
{
    logger.info("cleanup");
}

/**
 * Closes the underlying physical connection.
 */
public void destroy() throws ResourceException
{
    logger.info("Closing physical connection.");
    try
    {
        socket.close();
    } catch (Exception e) {}
}

/**
```

```
* Returns the local transaction for this
* resource manager instance (if any).
*/
public LocalTransaction getLocalTransaction()
    throws ResourceException
{
    logger.info("getLocalTransaction");
    return null;
}

/**
 * Returns the meta data for the managed connection.
 */
public ManagedConnectionMetaData getMetaData()
    throws ResourceException
{
    logger.info("getMetaData");
    return new ManagedElectionServerMetaData();
}

public XAResource getXAResource()
    throws ResourceException
{
    logger.info("getXAResource");
    return null;
}

public PrintWriter getLogWriter()
    throws ResourceException
{
    return null;
}
public void setLogWriter(PrintWriter out)
    throws ResourceException
{
}

protected void close()
{
    ConnectionEvent ce
        = new ConnectionEvent
            (this, ConnectionEvent.CONNECTION_CLOSED);
    ce.setConnectionHandle(virtualConnection);
    fireConnectionEvent(ce);
}

/**
 * Distributes connection events to all connection
 * event listeners.
 */
protected void fireConnectionEvent(ConnectionEvent evt)
{
    for(int i=listeners.size()-1; i >= 0; i--)
    {
        ConnectionEventListener listener
            = (ConnectionEventListener) listeners.get(i);
        if(evt.getId() == ConnectionEvent.CONNECTION_CLOSED)
            listener.connectionClosed(evt);
        else if(evt.getId() ==
            ConnectionEvent.CONNECTION_ERROR_OCCURRED)
            listener.connectionErrorOccurred(evt);
    }
}
```

```
}

private Socket socket;
private BufferedReader in;
private PrintStream out;

private ArrayList<ConnectionEventListener> listeners
    = new ArrayList<ConnectionEventListener>();
private ElectionServerConnectionImpl virtualConnection;

private static Logger logger
    = Logger.getLogger
        (ManagedElectionServerConnection.class.getName());
}
```

#### 16.4.2.1.3 The managed connection factory

Managed connection factories are used by the application server as factories of both ManagedConnection (i.e. the physical connections to the EIS) and of connection factory instances (the factories which are used by application components to create virtual user connections).

```
package za.co.solms.election.connector;

import java.io.Serializable;
import java.io.File;
import java.io.PrintWriter;
import java.net.UnknownHostException;
import java.util.Set;
import java.util.StringTokenizer;
import java.util.HashSet;
import java.util.Iterator;
import java.security.acl.Group;
import javax.resource.spi.ConnectionManager;
import javax.resource.spi.ManagedConnection;
import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.ResourceException;
import javax.security.auth.Subject;

import java.util.logging.Logger;

/**
 * Instances of this class are used by the application server as a
 * factory of both ManagedConnection (i.e. the physical connections
 * to the EIS) and of connection factory instances.
 * The class supports connection pooling by defining methods for
 * matching and creating connections.
 *
 * This class is implemented as a JavaBeans component enabling
 * the application server to populate bean attributes from
 * information obtained from, for example, the deployment
 * descriptor.
 *
 * @author fritz@solms.co.za
 */
public class ManagedElectionServerConnectionFactory
    implements javax.resource.spi.ManagedConnectionFactory,
               Serializable
{
    public ManagedElectionServerConnectionFactory()
```

```
{  
}  
  
/**  
 * Returns a connection factory providing virtual user connections  
 * for a provided subject  
 * and provided connection request information.  
 */  
public Object createConnectionFactory() throws ResourceException  
{  
    logger.info("createConnectionFactory");  
    throw new UnsupportedOperationException  
        ("Cannot be used in unmanaged env");  
}  
  
/**  
 * Returns a connection factory providing virtual user connections.  
 */  
public Object createConnectionFactory(ConnectionManager cm)  
    throws ResourceException  
{  
    logger.info  
        ("creating ConnectionFactory for connection manager " + cm);  
    ElectionServerConnectionRequestInfo connectionRequestInfo=null;  
    try  
    {  
        connectionRequestInfo  
            = new ElectionServerConnectionRequestInfo(hostName, port);  
    }  
    catch (Exception e)  
    {  
        logger.severe("Unable to construct URL.");  
    }  
    return new ElectionServerConnectionFactoryImpl  
  
}  
  
/**  
 * Returns a connection factory providing managed  
 * physical connections for a provided subject  
 * and provided connection request information.  
 */  
public ManagedConnection createManagedConnection  
    (Subject subject, ConnectionRequestInfo info)  
throws ResourceException  
{  
    logger.info("Creating ManagedConnection, subject=" +  
        + subject + ", info=" + info);  
    ElectionServerConnectionRequestInfo requestInfo  
        = (ElectionServerConnectionRequestInfo) info;  
    if( roles != null && roles.size() > 0 )  
    {  
        ...  
    }  
}
```

```
        validateRoles(subject);
    }
    try
    {
        logger.info
            ("Now creating managed election server implementation ...");
        return new ManagedElectionServerConnection(subject, requestInfo);
    }
    catch (Exception e)
    {
        logger.info("Caught exception " + e);
        logger.info("Throwing resource exception");
        throw new ResourceException(e);
    }
}

/**
 * Returns a matching managed physical connection for
 * a provided subject and provided connection request
 * information from the provided set of physical
 * connections.
 */
public ManagedConnection matchManagedConnections
    (Set connectionSet, Subject subject,
     ConnectionRequestInfo info)
        throws ResourceException
{
    // All instances will match
    logger.info("matchManagedConnections, connectionSet="
        + connectionSet + ", subject=" + subject + ", info=" + info);
    return (ManagedConnection) connectionSet.iterator().next();
}

public PrintWriter getLogWriter() throws ResourceException
{
    return null;
}

public void setLogWriter(PrintWriter out) throws ResourceException
{
}

public boolean equals(Object other)
{
    return super.equals(other);
}

public int hashCode()
{
    return super.hashCode();
}

/**
 * Returns the name of the device hosting the EIS.
 */
public String getHostName()
{
    return hostName;
}

/**
```

```
* Setting host name of the EIS this connector is connecting to.  
* This method is used to set the host information from the  
* deployment descriptor.  
*/  
public void setHostName(String hostName)  
{  
    this.hostName = hostName;  
}  
  
/**  
 * Returns the port through which the EIS is accessed.  
 */  
public int getPort()  
{  
    return port;  
}  
  
/**  
 * Setting port number of the EIS this connector is connecting to.  
 * This method is used to set the host information from the  
* deployment descriptor.  
*/  
public void setPort(int port)  
{  
    this.port = port;  
}  
  
public void setPort(String port)  
{  
    this.port = Integer.parseInt(port);  
}  
  
public String getRoles()  
{  
    return roles.toString();  
}  
  
public void setRoles(String roles)  
{  
    this.roles = new HashSet<String>();  
    StringTokenizer st = new StringTokenizer(roles, ",");  
    while( st.hasMoreTokens() )  
    {  
        String role = st.nextToken();  
        this.roles.add(role);  
    }  
}  
  
private void validateRoles(Subject theSubject)  
    throws ResourceException  
{  
    /*  
     Set subjectGroups = theSubject.getPrincipals(Group.class);  
     Iterator iter = subjectGroups.iterator();  
     Group roleGrp = null;  
     while (iter.hasNext())  
     {  
         Group grp = (Group) iter.next();  
         String name = grp.getName();  
         if (name.equals("Roles"))  
             roleGrp = grp;  
     }  
}
```

```
if( roleGrp == null )
    throw new ResourceException("Subject has not Roles");

boolean isValid = false;
iter = roles.iterator();
while( iter.hasNext() && isValid == false )
{
    String name = (String) iter.next();
    SimplePrincipal role = new SimplePrincipal(name);
    isValid = roleGrp.isMember(role);
}
if( isValid == false )
{
    String msg = "Authorization failure, subjectRoles='"+roleGrp
        + ", requiredRoles='"+roles;
    throw new ResourceException(msg);
} */
}

private static final long serialVersionUID = 100000;
private static Logger logger = Logger.getLogger
    (ManagedElectionServerConnectionFactory.class.getName());

private String hostName; // values initialized from deployment descriptor
private int port;
private Set<String> roles;
private transient File rootDir;
}
```

---

**Note**

This class is implemented as a JavaBeans component enabling the application server to populate bean attributes from information obtained from, for example, the deployment descriptor.

---

#### 16.4.2.1.4 Connection request info

The connection request information encapsulates the information required to establish a connection to an election server instance. In our case this includes the host name and port number.

```
package za.co.solms.election.connector;

import javax.resource.spi.ConnectionRequestInfo;

/**
 * Encapsulates the information required to establish a
 * connection to an election server instance
 *
 * @author fritz@solms.co.za
 */
public class ElectionServerConnectionRequestInfo
    implements ConnectionRequestInfo
{
    /**
     * Creates an instance encapsulating the host name
     * and port used to access the EIS.
     */
    public ElectionServerConnectionRequestInfo
        (String hostName, int port)
```

```
{  
    this.hostName = hostName;  
    this.port = port;  
}  
  
/**  
 * Returns the host name of the device hosting  
 * the election server.  
 */  
public String getHostName() {return hostName;}  
  
/**  
 * Returns the port number through which the  
 * election server is accessed.  
 */  
public int getPort() {return port;}  
  
/**  
 * Returns true if the argument object is  
 * another ElectionServerConnectionRequestInfo  
 * for the same URL and the same port.  
 */  
public boolean equals(Object arg)  
{  
    ElectionServerConnectionRequestInfo argRequest  
        = (ElectionServerConnectionRequestInfo) arg;  
  
    return (hostName.equals(argRequest.hostName)  
        && (port == argRequest.port));  
}  
  
public int hashCode()  
{  
    return hostName.hashCode() + port;  
}  
  
public String toString()  
{  
    return "RequestInfo: hostName=" + hostName  
        + ", port=" + port;  
}  
  
private String hostName;  
private int port;  
}
```

#### 16.4.2.1.5 Managed connection meta data

The meta data provides information about the EIS the managed connection is interfacing with.

```
package za.co.solms.election.connector;  
  
import javax.resource.spi.ManagedConnectionMetaData;  
import javax.resource.ResourceException;  
  
/**  
 * The ManagedConnectionMetaData interface provides  
 * information about the underlying EIS instance  
 * associated with a ManagedConnection instance.  
 * An application server uses this information to
```

```
* get runtime information about a connected EIS instance.  
*  
* @author fritz.solms.co.za  
*/  
public class ManagedElectionServerMetaData  
    implements ManagedConnectionMetaData  
{  
    public ManagedElectionServerMetaData()  
    {  
    }  
  
    public String getEISProductName()  
        throws ResourceException  
    {  
        return "Election Server";  
    }  
  
    public String getEISProductVersion()  
        throws ResourceException  
    {  
        return "Election Server Version 1.0";  
    }  
  
    public int getMaxConnections()  
        throws ResourceException  
    {  
        return 100;  
    }  
  
    public String getUserName()  
        throws ResourceException  
    {  
        return "";  
    }  
}
```

#### 16.4.2.2 The elements of the Common Client Interface (CCI)

The common client interface in the connector architecture provides the application client view onto the connector. It exposes

- virtual connections, and the
- virtual connection factories,

##### 16.4.2.2.1 Connection

The connection defined the services which the virtual user connections provide to application components:

```
package za.co.solms.election.connector;  
  
/**  
 * @author fritz@solms.co.za  
 *  
 * The interface publishing the client services available  
 * via a connection to the election server.  
 */  
public interface ElectionServerConnection  
{
```

```
 /**
 * Add a number of votes for a party.
 */
public void addVotes(String party, int numVotes);

 /**
 * Query the number of votes a particular party has.
 */
public int getVotes(String party);
}
```

#### 16.4.2.2.2 The connection implementation

The connection implementation provides an implementation of a virtual user connection. It delegates requests for EIS services to the associated managed connection managed by the application server

```
 package za.co.solms.election.connector;

import java.util.logging.Level;
import javax.naming.Name;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;

import java.util.logging.Logger;
import javax.resource.spi.ConnectionEvent;
import javax.resource.spi.ConnectionEventListener;

/**
 * This class provides an implementation of the client's
 * interface to a connection provided and managed by the
 * application server.
 * It delegates any requests for business services
 * from the EIS to the underlying physical (managed)
 * connection.
 *
 * Instances of this class are instances of a virtual
 * client connection which can be realized by different
 * underlying physical connections over time.
 *
 * It maintains a handle to the physical managed connection
 * (managed by the application server).
 *
 * When a user closes a connection, the virtual user connection
 * is disassociated from the physical managed connection
 * which may be used by another virtual connection.
 *
 * @author fritz@solms.co.za
 */
public class ElectionServerConnectionImpl
    implements ElectionServerConnection
{
    /**
     * Creates new virtual connection associated to a
     * physical (managed) connection
     */
    public ElectionServerConnectionImpl
        (ManagedElectionServerConnection managedConnection)
    {
        logger.info("Constructing new virtual connection:");
        this.managedConnection = managedConnection;
    }
}
```

```
}

/**
 * Associates a new physical (managed) connection with
 * this virtual connection.
 */
protected void associateManagedConnection
    (ManagedElectionServerConnection newManagedConnection)
{
    logger.info("Associating a new managed connection" +
        " with this virtual connection.");

    managedConnection.removeConnectionEventListener
        (connectionEventListener);

    managedConnection = newManagedConnection;

    managedConnection.addConnectionEventListener
        (connectionEventListener);
}

/**
 * Disassociated this virtual connection from the
 * physical (managed) connection.
 */
public void disassociateManagedConnection()
{
    logger.info("Disassociating the managed connection " +
        "from this virtual connection.");
    managedConnection = null;
}

/**
 * Close this virtual connection, disassociating it from
 * the physical (managed) connection.
 */
public void close()
{
    logger.log(Level.FINEST, "Closing virtual connection.");
    managedConnection.close();
}

/**
 * Add a number of votes for a particular party.
 * This virtual user connection forwards the request
 * to the physical managed connection.
 */
public void addVotes(String party, int numVotes)
{
    logger.log(Level.FINEST, "Adding " + numVotes
        + " to " + party);
    managedConnection.addVotes(party, numVotes);
}

/**
 * Retrieves the number of votes from the EIS
 * (the election server) for a particular party.
 * This virtual user connection forwards the request
 * to the physical managed connection.
 */
public int getVotes(String party)
{
```

```

        logger.info("Retrieving the number of votes for "
            + party);
        return managedConnection.getVotes(party);
    }

    private ManagedElectionServerConnection managedConnection;

    private ConnectionEventListener connectionEventListener
        = new ConnectionEventListener()
    {
        public void connectionClosed
            (ConnectionEvent connectionEvent)
        {
            logger.info("Transaction closed.");
        }
        public void connectionErrorOccurred
            (ConnectionEvent connectionEvent)
        {
            logger.info("Transaction error.");
        }
        public void localTransactionCommitted
            (ConnectionEvent connectionEvent)
        {
            logger.info("Transaction Committed.");
        }
        public void localTransactionRolledback
            (ConnectionEvent connectionEvent)
        {
            logger.info("Transaction rolled back.");
        }
        public void localTransactionStarted
            (ConnectionEvent connectionEvent)
        {
            logger.info("Transaction started.");
        }
    };

    private static final Logger logger
        = Logger.getLogger(ElectionServerConnectionImpl.class.getName());
}

```

#### 16.4.2.2.3 The connection factory

The connection factory specifies the required factory services for generating virtual user connections.

```

package za.co.solms.election.connector;

import java.io.Serializable;
import javax.resource.Referenceable;
import javax.naming.NamingException;

/**
 * Interface through which clients obtain virtual connections
 * which the application server links to
 * physical managed connections. It is this factory which clients
 * will look up via JNDI when requiring a connection.
 *
 * @author fritz@solms.co.za
 */
public interface ElectionServerConnectionFactory

```

```
        extends Referenceable, Serializable
{
    /**
     *
     * @return a virtual user connection whose coupling to the
     * underlying managed physical connections
     * is managed by the application server.
     *
     * @throws NamingException
     */
    public ElectionServerConnection getConnection()
        throws NamingException;
}
```

#### 16.4.2.2.4 The connection factory implementation

The connection factory implementation realizes the factory services for generating virtual user connections.

```
package za.co.solms.election.connector;

import javax.resource.spi.ConnectionManager;
import javax.resource.spi.ManagedConnectionFactory;
import javax.resource.ResourceException;
import javax.naming.NamingException;
import javax.naming.Reference;

import java.util.logging.Logger;

/**
 * An implementation of a factory for virtual
 * user connections to the EIS.
 *
 * @author fritz@solms.co.za
 */
public class ElectionServerConnectionFactoryImpl
    implements ElectionServerConnectionFactory
{
    /**
     * Creates an instance of a connection factory which uses
     * a provided connection manager and managed (physical)
     * connection factory in order to create connections compying
     * with a provided connection request info.
     *
     * Instances of connection facotories are created by the
     * application server which also provides the connection
     * manager.
     */
    ElectionServerConnectionFactoryImpl(ConnectionManager manager,
        ManagedConnectionFactory factory,
        ElectionServerConnectionRequestInfo requestInfo)
    {
        logger.info
            ("Creating election server connection implementation ...");
        this.manager = manager;
        this.factory = factory;
        this.requestInfo = requestInfo;
        logger.info
            ("Created election server connection factory for request info "
             + requestInfo);
    }
}
```

```
}

/**
 * Returns a virtual user connection to the EIS which
 * will be linked by the application server
 * to managed physical EIS connections.
 */
public ElectionServerConnection getConnection()
        throws NamingException
{
    logger.info("getConnection");
    ElectionServerConnection connection = null;
    try
    {
        connection = (ElectionServerConnection)
            manager.allocateConnection(factory, requestInfo);
    }
    catch(ResourceException e)
    {
        logger.info("caught resource exception");
        logger.info(e.toString());
        Object[] elemnts = e.getStackTrace();
        for (Object el:elemnts)
            logger.info(el.toString());
        throw new NamingException("Unable to get Connection: "+e);
    }
    return connection;
}

public void setReference(Reference reference)
{
    logger.info("setReference, reference="+reference);
    this.reference = reference;
}

public Reference getReference() throws NamingException
{
    logger.info("getReference");
    return reference;
}

private static final long serialVersionUID = 100000;
private transient ConnectionManager manager;
private transient ManagedConnectionFactory factory;
private transient ElectionServerConnectionRequestInfo requestInfo;
private Reference reference;

private static final Logger logger =
    Logger.getLogger(ElectionServerConnectionFactoryImpl.class.getName());
}
```

#### 16.4.2.3 The deployment descriptor for the resource adapter

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd" version="1.5">
```

```
<description>A JCA 1.5 compliant socket adaptor</description>
<display-name>
    Election Server Adaptor
</display-name>
<vendor-name>
    Solms Training, Consulting & Development
</vendor-name>
<eis-type>
    Proprietary Text/TCP/IP<
</eis-type>
<resourceadapter-version>
    1.0
</resourceadapter-version>
<resourceadapter>
    <resourceadapter-class>
        za.co.solms.election.connector.ElectionServerAdaptor
    </resourceadapter-class>
    <outbound-resourceadapter>
        <connection-definition>
            <managedconnectionfactory-class>
                za.co.solms.election.connector.ManagedElectionServerConnectionFactory
            </managedconnectionfactory-class>
            <config-property>
                <description>
                    The name of the machine hosting the election server
                </description>
                <config-property-name>hostName</config-property-name>
                <config-property-type>java.lang.String</config-property-type>
                <config-property-value>localhost</config-property-value>
            </config-property>
            <config-property>
                <description>
                    The through which the election server is accessed
                </description>
                <config-property-name>port</config-property-name>
                <config-property-type>java.lang.String</config-property-type>
                <config-property-value>12345</config-property-value>
            </config-property>
            <connectionfactory-interface>
                za.co.solms.election.connector.ElectionServerConnectionFactory
            </connectionfactory-interface>
            <connectionfactory-impl-class>
                za.co.solms.election.connector.ElectionServerConnectionFactoryImpl
            </connectionfactory-impl-class>
            <connection-interface>
                za.co.solms.election.connector.ElectionServerConnection
            </connection-interface>
            <connection-impl-class>
                za.co.solms.election.connector.ElectionServerConnectionImpl
            </connection-impl-class>
        </connection-definition>
        <transaction-support>NoTransaction</transaction-support>
        <authentication-mechanism>
            <authentication-mechanism-type>
                BasicPassword
            </authentication-mechanism-type>
            <credential-interface>
                javax.resource.spi.security.PasswordCredential
            </credential-interface>
        </authentication-mechanism>
        <reauthentication-support>true</reauthentication-support>
    </outbound-resourceadapter>
```

```
</resourceadapter>
</connector>
```

#### 16.4.2.4 The deployment descriptor for the data source defined for the resource adapter

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connection-factories PUBLIC "-//JBoss//DTD JBOSS JCA Config 1.5//EN" "http://www.↓
    jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">

<!--
    This is JBoss-specific descriptor.
    It should be placed in the "deploy" directory
    of desired server configuration.
-->
<connection-factories>
    <no-tx-connection-factory>
        <jndi-name>
            ElectionConnector
        </jndi-name>
        <rar-name>
            electionConnector.rar
        </rar-name>
        <connection-definition>
            za.co.solms.election.connector.ElectionServerConnectionFactory
        </connection-definition>
        <config-property name="RepositoryPath" type="java.lang.String">
            ..
        </config-property>
    </no-tx-connection-factory>
</connection-factories>
```

#### 16.4.2.5 The Ant build script for the resource adapter

```
<?xml version="1.0"?>

<project name="ElectionConnector" default="help" basedir=".">
    <target name="help">
        <echo message=
            "Available targets: (typically used in this order)"/>
        <echo message=
            "    "/>
        <echo message=
            "    clean - deletes all but the source files."/>
        <echo message=
            "    prepare - creates the directories for compilation and distribution."/>
        <echo message=
            "    compile - compiles the Java source files."/>
        <echo message=
            "    jar - creates an archive for distribution."/>
        <echo message=
            "    resourceAdapter.deploy - deploys the resource adapter."/>
        <echo message=
            "    connectorDataSource.deploy - deploys the datasource definition for the resource    ↓
                adapter."/>
        <echo message=
```

```
" undeploy - undeploys both, the resource adapter and its datasource defintion."/>
<echo message=" "/>
</target>

<!-- File name for target jar file for
the resource adapter -->
<property name="connector.jar"
value="electionConnector.rar"/>

<!-- File name of datasource descriptor file
for connector -->
<property name="connector.dataSource"
value="electionConnector-ds.xml"/>

<!-- Handle to environment -->
<property environment="env"/>

<!-- Directories -->
<property name="src.dir" value="src"/>
<property name="build.dir" value="build" />
<property name="metaInf.dir" value="src/META-INF"/>
<property name="shelf.dir" value="shelf" />
<property name="dist.dir" value="dist"/>
<property name="deploy.dir"
value="\${env.JBOSS_HOME}/server/default/deploy"/>

<!-- Compiler flags -->
<property name="compile.debug" value="true"/>
<property name="compile.optimize" value="false"/>
<property name="compile.deprecation" value="true"/>

<!-- Libraries path -->
<path id="shelf">
  <fileset dir="\${shelf.dir}">
    <include name="**/*.jar" />
    <include name="**/*.rar" />
  </fileset>
</path>

<!-- path used for build target -->
<path id="build.path">
  <path refid="shelf"/>
  <pathelement path="\${build.dir}"/>
</path>

<!-- Removes all generated artifacts -->
<target name="clean">
  <delete dir="\${build.dir}"/>
  <delete dir="\${dist.dir}"/>
</target>

<!-- Prepares the required directories
for the build -->
<target name="prepare">
  <mkdir dir="\${build.dir}"/>
  <mkdir dir="\${dist.dir}"/>
</target>

<!-- Compiles the Java source files -->
<target name="compile" depends="prepare">
  <javac srcdir="\${src.dir}"
  destdir="\${build.dir}"
```

```
deprecation="${compile.deprecation}"
optimize="${compile.optimize}"
debug="${compile.debug}">
<classpath>
    <path refid="build.path"/>
</classpath>
</javac>
</target>

<!-- Generates the Java archive for
the resource adapter -->
<target name="jar" depends="compile">
    <delete file="${dist.dir}/${connector.jar}" />
    <jar jarfile="${dist.dir}/${connector.jar}">
        <fileset dir="${build.dir}"
            includes="**/*.class"/>
        <fileset dir="${src.dir}"
            includes="META-INF/*.xml"/>
        <fileset dir="${src.dir}"
            includes="**/*.properties"/>
        <fileset dir="${src.dir}"
            includes="**/*.policy"/>
    </jar>
</target>

<!-- Deploys resource adapter -->
<target name="resourceAdapter.deploy">
    <copy file="${dist.dir}/${connector.jar}"
        todir="${deploy.dir}" />
</target>

<!-- Deploys connector data source -->
<target name="connectorDataSource.deploy">
    <copy file="${metaInf.dir}/${connector.dataSource}"
        todir="${deploy.dir}" />
</target>

<!-- Undeploys the datasource as well as
the resource adapter. -->
<target name="undeploy">
    <delete file="${deployDir}/${connector.dataSource}" />
    <delete file="${deployDir}/${connector.jar}" />
</target>

</project>
```

#### 16.4.2.6 Deploying the resource adapter

The deployment is simplified using a Ant build script. The script assumes

- you are deploying into a JBoss application server, and
- that the `JBOSS_HOME` environment variable has been set.

---

#### Note

Only minor modifications would be required to deploy to other application servers.

---

Running the ant script with the `jar` target via

```
ant jar
```

compiles all components and packages them up within a resource archive. The resource adapter can then be deployed via

```
ant resourceAdapter.deploy
```

and published as a datasource to the application server via

```
ant connectorDataSource.deploy
```

### 16.4.3 A client application for the resource adapter

In this section we show the implementation of a session bean which uses the resource adapter in order to make the EIS services available to EJB clients. We then develop a simple web client (JSP and servlet) providing web users access to the EIS services offered by the resource adapter.

#### 16.4.3.1 The session bean

```
package za.co.solms.election.sessionBean.impl;

import java.util.logging.Level;
import javax.ejb.Stateless;

import za.co.solms.election.sessionBean.contract.ElectionRemote;
import za.co.solms.election.sessionBean.contract.ElectionLocal;

import javax.naming.InitialContext;
import za.co.solms.election.connector.ElectionServerConnectionFactory;
import za.co.solms.election.connector.ElectionServerConnection;

/**
 * A session bean publishing the EIS services available via
 * the resource adapter.
 *
 * @author fritz@solms.co.za
 */
@Stateless
public class ElectionBean
    implements ElectionLocal, ElectionRemote
{

    /**
     * Creates a new instance of ElectionBean */
    public ElectionBean() throws javax.naming.NamingException
    {
        logger.setLevel(Level.INFO);
        connection = getConnection();
    }

    /**
     * Adds a number of votes to a party.
     */
    public void addVotes(String party, int numVotes)
    {
        logger.info("Sending addVotes request connector ...");
        connection.addVotes(party, numVotes);
    }
}
```

```
/*
 * Retrieves the number of votes for a party.
 */
public int getVotes(String party)
{
    logger.info("Retrieving votes for " + party);
    return connection.getVotes(party);
}

/**
 * Establishes a connection to the EIS as provided
 * by the resource adapter.
 */
private ElectionServerConnection getConnection()
    throws javax.naming.NamingException
{
    logger.info("looking up naming context");
    InitialContext context = new InitialContext();
    logger.info
        ("looking up election server connection factory ...");
    ElectionServerConnectionFactory connectionFactory
        = (ElectionServerConnectionFactory)context.lookup
            ("java:ElectionConnector");
    logger.info("Getting connection from factory ...");
    connection = connectionFactory.getConnection();
    logger.info("Got connection, returning it...");
    return connection;
}

private ElectionServerConnection connection;

private final java.util.logging.Logger logger
    = java.util.logging.Logger.getLogger
        (ElectionBean.class.getName());
}
```

#### 16.4.3.2 A JSP-based user interface

```
package za.co.solms.election.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.util.logging.Logger;

import za.co.solms.election.sessionBean.contract.ElectionLocal;

/**
 * A servlet processing the user events received from the
 * election client view (JSP).
 *
 * @author fritz@solms.co.za

```

```
/*
public class ElectionServlet extends HttpServlet
{

    /**
     * Creates a new instance of ElectionServlet
     */
    public ElectionServlet()
    {
    }

    /**
     * Processing HTTP Get requests.
     */
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse resp)
        throws ServletException, IOException
    {
        doit(req, resp);
    }

    /**
     * Processing HTTP Post requests.
     */
    protected void doPost(HttpServletRequest req,
                         HttpServletResponse resp)
        throws ServletException, IOException
    {
        doit(req, resp);
    }

    /**
     * Gets handle to bean and looks up bean from naming service.
     */
    public void init() throws ServletException
    {
        super.init();
        try
        {
            InitialContext ctx = new InitialContext();
            logger.info("Got handle to naming context.");
            electionConnection
                = (ElectionLocal)ctx.lookup(ElectionLocal.class.getName());

            logger.info("Looked up average bean");
        }
        catch (NamingException e)
        {
            throw new RuntimeException(e);
        }
    }

    /**
     * Demarshalls the HTTP request parameters, calls the session bean
     * and generates the HTTP response.
     *
     * @param req
     *          the request
     * @param resp
     *          the response
     * @throws ServletException
     * @throws IOException
     */
}
```

```
/*
protected void doit(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException
{
    /*
     * De-marshalling request:
     */
    String party = request.getParameter("party");
    String action = request.getParameter("action");

    /*
     * Delegating business logic to session bean:
     */
    int numVotes = -1;
    if (action.equalsIgnoreCase("getVotes"))
    {
        try
        {
            logger.info("Requesting the number of votes for " + party);
            numVotes = electionConnection.getVotes(party);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
    else if (action.equalsIgnoreCase("addVotes"))
    {
        try
        {
            try
            {
                numVotes = Integer.parseInt(request.getParameter("numVotes"));
                logger.info("Adding " + numVotes + " to " + party);
                electionConnection.addVotes(party, numVotes);
                numVotes = electionConnection.getVotes(party);
            }
            catch (Exception e) {logger.info("Invalid input");}
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    /*
     * Storing result in session context:
     */
    request.setAttribute("numVotes", new Integer(numVotes));
    request.setAttribute("party", party);

    /*
     * Delegating response view to JSP:
     */
    RequestDispatcher dispatcher
        = getServletContext().getRequestDispatcher("/ElectionClient.jsp");
    dispatcher.forward(request, response);
}

private ElectionLocal electionConnection;
```

```
    private Logger logger = Logger.getLogger(ElectionServlet.class.getName());  
}
```

#### 16.4.3.3 A servlet controller

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE web-app PUBLIC  
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
  "http://java.sun.com/dtd/web-app_2_3.dtd">  
  
<web-app>  
  <servlet>  
    <servlet-name>ElectionServlet</servlet-name>  
    <servlet-class>  
      za.co.solms.election.servlet.ElectionServlet  
    </servlet-class>  
  </servlet>  
  
  <welcome-file-list>  
    <welcome-file>ElectionClient.jsp</welcome-file>  
  </welcome-file-list>  
  
  <servlet-mapping>  
    <servlet-name>ElectionServlet</servlet-name>  
    <url-pattern>/ElectionServlet</url-pattern>  
  </servlet-mapping>  
  
</web-app>
```

#### 16.4.3.4 The web deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE web-app PUBLIC  
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
  "http://java.sun.com/dtd/web-app_2_3.dtd">  
  
<web-app>  
  <servlet>  
    <servlet-name>ElectionServlet</servlet-name>  
    <servlet-class>  
      za.co.solms.election.servlet.ElectionServlet  
    </servlet-class>  
  </servlet>  
  
  <welcome-file-list>  
    <welcome-file>ElectionClient.jsp</welcome-file>  
  </welcome-file-list>  
  
  <servlet-mapping>  
    <servlet-name>ElectionServlet</servlet-name>  
    <url-pattern>/ElectionServlet</url-pattern>  
  </servlet-mapping>  
  
</web-app>
```

#### 16.4.3.5 The application deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
  "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
  "http://java.sun.com/dtd/application_1_3.dtd">

<application>
  <display-name>Election EAR</display-name>

  <module>

    <web>
      <web-uri>election.war</web-uri>
      <context-root>/election</context-root>
    </web>
  </module>
  <module>
    <ejb>election.ejb3</ejb>
  </module>
</application>
```

#### 16.4.3.6 The Ant build script

```
<?xml version="1.0"?>

<project name="ElectionClient" default="help" basedir=".">
  <target name="help">
    <echo message=
      "Available targets: (typically used in this order)"/>
    <echo message=
      "  "/>
    <echo message=
      "  clean - deletes all but the source files."/>
    <echo message=
      "  prepare - creates the directories for compilation and distribution."/>
    <echo message=
      "  common.compile - compiles the common Java source files."/>
    <echo message=
      "  ejb.compile - compiles the EJB source files."/>
    <echo message=
      "  client.compile - compiles the Java client source files."/>
    <echo message=
      "  servlet.compile - compiles the common Java source files."/>
    <echo message=
      "  ejb.jar - creates an archive for the ejb distribution."/>
    <echo message=
      "  client.jar - creates an archive for the client distribution."/>
    <echo message=
      "  servlet.jar - creates the servlet archive (war)."/>
    <echo message=
      "  ear - creates the enterprise archive."/>
    <echo message=
      "  ejb.deploy - deploys the session bean."/>
  </target>
</project>
```

```
<echo message=
  " ear.deploy - deploys the enterprise application."/>
<echo message=
  " undeploy - undeploys the all deployed elements."/>
<echo message=
  " run - launches the client application."/>
<echo message=
  " "/>
</target>

<!-- The server application which is launched in the start target -->
<property name="client.app"
  value="za.co.solms.election.client.ElectionClient"/>

<!-- File name for target jar file for the server -->
<property name="client.jar" value="electionClient.jar"/>

<!-- File name for target jar file for the server -->
<property name="ejb.jar" value="election.ejb3"/>

<!-- File name for target jar file for the server -->
<property name="servlet.jar" value="election.war"/>

<!-- File name for target jar file for the enterprise archive -->
<property name="ear" value="election.ear"/>

<!-- Handle to environment -->
<property environment="env"/>

<!-- Directories -->
<property name="src.dir" value="src"/>
<property name="src.common.dir" value="src/common"/>
<property name="src.ejb.dir" value="src/ejb"/>
<property name="src.client.dir" value="src/client"/>
<property name="src.servlet.dir" value="src/servlet"/>
<property name="src.application.dir" value="src/application"/>
<property name="build.dir" value="build" />
<property name="build.common.dir" value="build/common" />
<property name="build.ejb.dir" value="build/ejb" />
<property name="build.client.dir" value="build/client" />
<property name="build.servlet.dir" value="build/servlet" />
<property name="shelf.dir" value="shelf" />
<property name="dist.dir" value="dist"/>
<property name="deploy.dir"
  value="${env.JBOSS_HOME}/server/default/deploy"/>

<!-- Compiler flags -->
<property name="compile.debug" value="true"/>
<property name="compile.optimize" value="false"/>
<property name="compile.deprecation" value="true"/>

<!-- Libraries path -->
<path id="shelf">
  <fileset dir="${shelf.dir}">
    <include name="**/*.jar" />
    <include name="**/*.rar" />
  </fileset>
</path>

<!-- path used for build target -->
<path id="build.path">
  <path refid="shelf"/>
```

```
<path element path="${build.common.dir}" />
</path>

<!-- Removes all generated artifacts -->
<target name="clean">
    <delete dir="${build.dir}" />
    <delete dir="${dist.dir}" />
</target>

<!-- Prepares the required directories for the build -->
<target name="prepare">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${build.common.dir}" />
    <mkdir dir="${build.ejb.dir}" />
    <mkdir dir="${build.client.dir}" />
    <mkdir dir="${build.servlet.dir}" />
    <mkdir dir="${dist.dir}" />
</target>

<!-- Compiles the common Java source -->
<target name="common.compile" depends="prepare">
    <javac srcdir="${src.common.dir}"
        destdir="${build.common.dir}"
        deprecation="${compile.deprecation}"
        optimize="${compile.optimize}"
        debug="${compile.debug}">
        <classpath>
            <path refid="shelf"/>
            <path element path="${build.common.dir}" />
        </classpath>
    </javac>
</target>

<!-- Compiles the EJB source files -->
<target name="ejb.compile" depends="common.compile">
    <javac srcdir="${src.ejb.dir}" destdir="${build.ejb.dir}"
        deprecation="${compile.deprecation}"
        optimize="${compile.optimize}"
        debug="${compile.debug}">
        <classpath>
            <path refid="shelf"/>
            <path element path="${build.common.dir}" />
            <path element path="${build.ejb.dir}" />
        </classpath>
    </javac>
</target>

<!-- Compiles the Java client source files -->
<target name="client.compile" depends="common.compile">
    <javac srcdir="${src.client.dir}" destdir="${build.client.dir}"
        deprecation="${compile.deprecation}"
        optimize="${compile.optimize}"
        debug="${compile.debug}">
        <classpath>
            <path refid="shelf"/>
            <path element path="${build.common.dir}" />
            <path element path="${build.client.dir}" />
        </classpath>
    </javac>
</target>

<!-- Compiles the Servlet source files -->
```

```
<target name="servlet.compile" depends="common.compile">
  <javac srcdir="${src.servlet.dir}" destdir="${build.servlet.dir}"
    deprecation="${compile.deprecation}"
    optimize="${compile.optimize}"
    debug="${compile.debug}">
    <classpath>
      <path refid="shelf"/>
      <pathelement path="${build.common.dir}"/>
      <pathelement path="${build.servlet.dir}"/>
    </classpath>
  </javac>
</target>

<!-- Generates the deployable Java archive for the
     session bean implementation -->
<target name="ejb.jar" depends="ejb.compile">
  <delete file="${dist.dir}/${ejb.jar}"/>
  <jar jarfile="${dist.dir}/${ejb.jar}">
    <fileset dir="${build.common.dir}"
      includes="**/*.class"/>
    <fileset dir="${build.ejb.dir}" includes="**/*.class"/>
  </jar>
</target>

<!-- Generates the Web archive containing the
     server side presentation layer -->
<target name="servlet.jar" depends="servlet.compile">
  <delete file="${dist.dir}/${servlet.jar}"/>
  <zip zipfile="${dist.dir}/${servlet.jar}">
    <zipfileset dir="${build.common.dir}"
      prefix="WEB-INF/classes">
      <include name="**/*.class"/>
    </zipfileset>
    <zipfileset dir="${build.servlet.dir}"
      prefix="WEB-INF/classes">
      <include name="**/*.class"/>
    </zipfileset>
    <zipfileset dir="${src.servlet.dir}"
      <include name="*.jsp"/>
    </zipfileset>
    <zipfileset dir="${src.servlet.dir}"
      prefix="WEB-INF">
      <include name="*.xml"/>
    </zipfileset>
  </zip>
</target>

<!-- Creates the enterprise archive containing both,
     server-side presentation and business logic layers -->
<target name="ear" depends="ejb.jar,servlet.jar">
  <delete file="${dist.dir}/${ear}"/>
  <jar jarfile="${dist.dir}/${ear}">
    <fileset dir="${dist.dir}"
      includes="${ejb.jar},${servlet.jar}"/>
    <fileset dir="${src.dir}"
      includes="META-INF/application.xml"/>
  </jar>
</target>

<!-- Generates the deployable Java archive for the
     client application -->
```

```
<target name="client.jar" depends="client.compile">
  <delete file="\${dist.dir}/\${client.jar}"/>
  <jar jarfile="\${dist.dir}/\${client.jar}">
    <fileset dir="\${build.common.dir}">
      <includes>**/*.class</includes>
    </fileset>
    <fileset dir="\${build.client.dir}">
      <includes>**/*.class</includes>
    </fileset>
    <fileset dir="\${src.client.dir}">
      <includes>**/*.properties</includes>
    </fileset>
    <fileset dir="\${src.client.dir}">
      <includes>**/*.policy</includes>
    </fileset>
  </jar>
</target>

<!-- Deploys the session bean -->
<target name="ejb.deploy">
  <copy file="\${dist.dir}/\${ejb.jar}" todir="\${deploy.dir}"/>
</target>

<!-- Deploys the servlet layer -->
<target name="servlet.deploy" depends="servlet.jar">
  <copy file="\${dist.dir}/\${servlet.jar}" todir="\${deploy.dir}"/>
</target>

<!-- Deploys the enterprise application -->
<target name="ear.deploy" depends="ear">
  <copy file="\${dist.dir}/\${ear}" todir="\${deploy.dir}"/>
</target>

<target name="undeploy">
  <delete file="\${deploy.dir}/\${ejb.jar}"/>
  <delete file="\${deploy.dir}/\${servlet.jar}"/>
  <delete file="\${deploy.dir}/\${ear}"/>
</target>

<!-- Launches the client -->
<target name="run">
  <echo message="Running the \${client.app} application: " />
  <echo message=
    "  java -classpath \${dist.dir}/\${client.jar} \${client.app}"/>
  <java fork="on" classname="\${client.app}">
    <!--<arg value="-Djava.security.manager
      -Djava.security.policy=policy.all"/><!--&gt;
    &lt;!--&lt;arg value="-Djava.security.policy=security.policy"/><!-- --&gt;
    &lt;classpath&gt;
      &lt;pathelment location="\${dist.dir}/\${client.jar}"/&gt;
      &lt;path refid="shelf"/&gt;
    &lt;/classpath&gt;
  &lt;/java&gt;
&lt;/target&gt;

&lt;/project&gt;</pre>
```

#### 16.4.3.7 Deploying the application

Running

```
ant ear
```

creates the enterprise archive containing both,

- the EJB based business logic layer, and
- the Servlet/JSP based presentation layer.

It can be deployed within JBoss via

```
ant deploy.ear
```

#### 16.4.3.8 Using the web client

The web client is accessed via the URL

```
http://hostName:port/election
```

# Chapter 17

## Index

### A

application  
    management, 2  
auditability  
    infrastructure, 2

### B

BEA Weblogic, 3

### C

Common Object Request Broker Architecture (CORBA), 5  
component  
    activation, 5  
    business logic, 2  
    de-activation, 5  
    model, 2  
    presentation layer, 2  
controller  
    business logic, 3

### D

database  
    mapping layer, 5  
    relational, 5

### E

EJB  
    Timer Service, 216  
    uses, 216

EJB-QL, *see* Enterprise Java Beans (EJB)  
Enterprise Java Beans (EJB), 5  
    query language (EJB-QL), 5

### F

framework  
    Java EE, 2

### G

Geronimo, 3  
Glassfish, 3

### I

IBM WebSphere, 3  
integration

infrastructure, 2

### J

Java Data Objects (JDO), 5  
Java EE, *see* Java Platform, Enterprise Edition (Java EE)  
Java Persistence API (JPA), 5  
Java Platform, Enterprise Edition (Java EE), 2  
    as Model View Controller, 3  
    definition of, 2  
    frameworks, 2  
    multi-tierframeworks, 2  
Java Platform, Standard Edition (Java SE), 2  
Java Server Faces (JSF), 3  
Java Server Pages (JSP), 3  
JBoss, 3  
JDO, *see* Java Data Objects (JDO)  
Jonas, 3

### L

layer  
    back-end, 3  
    domain objects, 3  
    infrastructure, 3  
    middleware, 3  
    presentation, 3  
    services, 3  
layering, 2

### M

message-driven bean, 3  
Model-View-Controller  
    Java EE as, 3

### N

network enabling, 5

### O

object  
    virtual, 5  
object-relational mapping, 5  
Oracle, 3

### P

persistence

infrastructure, 2  
support for, 5

portability, 2  
presentation layer  
web based, 3

## Q

query language  
object-oriented, 5

## R

reliability  
infrastructure, 2  
resource  
management, 2  
robustness, 2

## S

scalability, 2, 5  
security, 2  
infrastructure, 2  
support for, 5  
servlet, 3  
session bean, 3  
SQL, *see* Standard Query Language (SQL)  
Standard Query Language (SQL), 5  
system  
enterprise, 2

## T

transactions  
support for, 5

## U

UML  
relationships  
summary, 119

---