

Developing EJB Applications with Eclipse

Martin Heinemann

martin.heinemann@tudor.lu

Linuxdays.lu 2007

Februray 2007

Abstract

This paper is the accompanying documentation for the Enterprise Java Beans/Eclipse Tutorial taking place at the Linuxdays 2007 in Luxembourg. The main focus lays on the basic knowledge of the Enterprise Java Beans 3 specification and on how to develop EJB3 applications by using the JBoss Application Server, the Eclipse IDE and the JBoss Eclipse IDE Plugin. The examples in this document refer to the ones of the tutorial.

Contents

Prerequisites: Enterprise JavaBean	3
1.1 JavaBeans	3
1.2 Enterprise JavaBeans Technology	3
1.3 Music Store Example	5
1.4 Entities	5
1.5 SessionBeans	9
1.6 Message Driven Beans	11
1.7 Container	13
1.8 JBoss Application Server	18
1.9 Loading Strategies	18
1.10 CascadingTypes	19
1.11 Deployment	20
Creating EJB Applications using the JBoss Eclipse IDE	22
2.1 Eclipse	22
2.2 Getting Started	22
2.2.1 Download Eclipse SDK	22
2.2.2 Download JBoss IDE for Eclipse	24
2.2.3 Download JBoss Application Server	27
2.3 Prepare the Workspace	28
2.3.1 Create an Eclipse JBoss Configuration	28
2.3.2 Create an EJB3-Project	31
2.4 Development of an Application	31
2.5 JBoss Configuration	32
2.6 Packaging	33
2.7 Deployment	35
2.8 Run the clients	35
2.9 Information	36
References	37

Prerequisites: Enterprise JavaBean

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

The following chapters will give a fundamental introduction to Enterprise Java Beans, the core architecture and the improvements of the latest specification 3.0 compared with the 2.1 specification.

1.1 JavaBeans

A JavaBean is a software component written in the Java programming language.

”Reusable software component that can be manipulated visually in a builder tool” [Sun06a]

The complete definition of JavaBeans can be found in the JavaBeans Specification [Sun97]. Its main targets are the JavaBean conventions. They define rules in what way a valid JavaBean must be implemented. The rules are:

- The class should be serializable (able to persistently save and restore its state)
- It should have a no-argument constructor
- Its properties should be accessed using get and set methods that follow a standard naming convention
- It should contain any required event-handling methods

1.2 Enterprise JavaBeans Technology

Java Platform Enterprise Edition 5

“Java Platform, Enterprise Edition (Java EE) is a set of coordinated technologies and practices that enable solutions for developing, deploying, and

managing multi-tier, server-centric applications. Building on Java Platform, Standard Edition (Java SE), Java EE adds the capabilities that provide a complete, stable, secure, and fast Java platform for the enterprise. Java EE significantly reduces the cost and complexity of developing and deploying multi-tier solutions, resulting in services that can be rapidly deployed and easily enhanced.” [Sun06b]

The Enterprise JavaBeans specification defines a programming API with

- conventions,
- protocols,
- classes and
- interfaces

which specify how an application server should provide objects with

- persistence,
- transactions,
- concurrency control,
- remote communication,
- events using JMS (Java Messaging Service),
- naming and directory services,
- security and
- deployment of components.

The Java Platform Enterprise Edition 5 consists of the following technologies:

- Web Services Technologies:
 - Java API for XML-Based Web Services (JAX-WS) 2.0
 - Java API for XML-Based RPC (JAX-RPC) 1.1
 - Java Architecture for XML Binding (JAXB) 2.0
 - SOAP with Attachments API for Java (SAAJ)
 - Streaming API for XML
- Component Model Technologies:
 - Enterprise JavaBeans 3.0**
 - J2EE Connector Architecture 1.5
 - Java Servlet 2.5
 - JavaServer Faces 1.2
 - JavaServer Pages 2.1
 - JavaServer Pages Standard Tag Library
- Management Technologies:
 - J2EE Management
 - J2EE Application Deployment
 - Java Authorization Contract for Containers
- Other J2EE Technologies:
 - Common Annotations for the Java Platform
 - Java Transaction API (JTA)
 - JavaBeans Activation Framework (JAF) 1.1

JavaMail Web Service Metadata for the Java Platform

The list above demonstrates that Enterprise JavaBeans 3.0 (EJB3) is one of numerous technologies in the Java Platform Enterprise Edition 5 (J2EE 5).

The following sections will demonstrate the usage of Enterprise JavaBeans considering an imaginary application as example.

1.3 Music Store Example

The sample application in this document will be the online system of a music store. A visitor can go to the website of the music store, browse the Cd collection and put records into the shopping cart. He/she has the possibility to buy the Cds or remove elements from the shopping cart.

1.4 Entities

Entities (formerly called Entity Beans) are the components which contain the data used by the application. This data will be saved in a database backend which is accessed by the application server using JDBC. Java entity classes have to meet the JavaBeans requirements as described in section 1.1.

In the music store example an entity would be an music-cd. Since Entities are just Plain Old Java Objects (POJOs), a Java class with some additional metadata annotations that describe the properties, is required. “Metadata Annotations” is a language feature that was added in Java 2 Platform Standard Edition 5.0 (J2SE 5) and that is used widely throughout the EJB3 specification.

To demonstrate the different types of relationships between entities, the music-cd has several properties:

- an internal identification number
- an artist
- a price
- several songs
- a title
- a year of production

Listing 1.1 shows the POJO without annotations:

```

1 import java.io.Serializable;
2 import java.util.Collection;
3
4 import javax.persistence.*;
5
6
7 class MusicCd implements Serializable {
8
9     public Integer id;
10    public String title;

```

```

11     public Integer year;
12     public Double price;
13
14     public Artist artist;
15     public Collection<Song> songs;
16
17     public MusicCd() {
18     }
19     public MusicCd(String title) {
20         this.title = title;
21     }
22     public Integer getId() {
23         return id;
24     }
25     public void setId(Integer id) {
26         this.id = id;
27     }
28     public String getTitle() {
29         return this.title;
30     }
31     public void setTitle(String title) {
32         this.title = title;
33     }
34     public Integer getYear() {
35         return year;
36     }
37     public void setYear(Integer year) {
38         this.year = year;
39     }
40     public Double getPrice() {
41         return price;
42     }
43     public void setPrice(Double price) {
44         this.price = price;
45     }
46     public Artist getArtist() {
47         return artist;
48     }
49     public void setArtist(Artist artist) {
50         this.artist = artist;
51     }
52     public Collection<Song> getSongs() {
53         return songs;
54     }
55     public void setSongs(Collection<Song> songs) {
56         this.songs = songs;
57     }
58 }

```

Listing 1.1. MusicCd class without annotations

In the next steps the annotations are added. An annotation always belongs to the following element (e.g. class, method etc.). The first annotation tells the Application Server that this Java class is an Entity:

The `@Table` annotation states that the entity should be mapped to the database table “music_cd”. By default the entity would be mapped to the table “musiccd” but since in database design the underscore is a common separator character we

```

5 @Entity
6 @Table(name="music_cd")
7 @NamedQuery(name="findAllMusicCdByArtist", query="SELECT OBJECT(o) from
      MusicCd o WHERE o.artist = :artist")
8 class MusicCd implements Serializable {

```

Listing 1.2. MusicCd class annotations

define the table name in the annotation.

In the annotation `@NamedQuery` an EJB QL (EJB Query Language) query is defined. Multiple of these queries could be defined inside a `NamedQueries` annotation. The query language is a subset of SQL and is used to retrieve or manipulate entity data from the database. The example code retrieves all `MusicCd` entities that belong to one artist by using the named parameter `:artist`. Note that EJB QL uses Java object notation, not SQL database notation (e.g. `MusicCd` is the name of the Java class, not the name of the database table).

The next annotations will define the primary key property of the `MusicCd` class:

```

25 @Id
26 @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="MusicCdSeq")
27 @SequenceGenerator(name="MusicCdSeq", sequenceName="music_cd_seq")
28 @Column(name = "id")
29 public Integer getId() {
30     return id;
31 }

```

Listing 1.3. MusicCd primary key annotations

The `id` annotation defines the following method as accessory method to the primary key field. On the next line, the primary key is defined to be automatically generated. In the music store we want the internal id to be increased for each newly added Cd, so we use the sequence that is defined in the database scheme. The `@GeneratedValue` annotation takes two parameters, *strategy* and *generator*. The *strategy* parameter can be one of `TABLE`, `SEQUENCE`, `IDENTITY`, which are database specific strategies, or `AUTO` which will tell the persistence provider to choose the appropriate strategy. The *generator* parameter defines the object that should be used for generating the values. This is not the database sequence but a Java object representing the sequence in the application server. The database sequence itself, is defined in the `@SequenceGenerator` annotation by setting the *sequenceName* parameter to the sequence name in the database. The last annotation of Listing 1.3 is the `@Column` annotation that defines the database column that the property will be mapped to.

```

36 @Column(name="title")
37 public String getTitle() {
38     return this.title;
39 }

```

Listing 1.4. MusicCd column annotations

The `@Column` annotation is defined for each primitive property. In this case primitive properties are those which are of primitive types (int, float etc.) or their wrapper

```

51 @ManyToMany(mappedBy="songs")
52 public Collection<MusicCd> getMusicCds() {
53     return musicCds;
54 }
55 public void setMusicCds(Collection<MusicCd> musicCds) {
56     this.musicCds = musicCds;
57 }

```

Listing 1.7. Song ManyToMany relationship

classes (Integer, Float etc.). For EJB it is better to use the wrapper classes because database rows may contain NULL values but primitives cannot be declared null. In a relational database there is not only one table, there are also relations between tables.

```

60 @ManyToOne
61 @JoinColumn(name="artist")
62 public Artist getArtist() {
63     return artist;
64 }

```

Listing 1.5. MusicCd ManyToOne relationship

Listing 1.5 demonstrates a “Many to One” relation for the property artist. There can be many Cds by an artist but a Cd can have only one artist (excluding samplers from the example). The `@ManyToOne` annotation states that we have a “Many to One” relationship. We do not have to specify the table from which the foreign key is imported because the property artist is an object of the type Artist which has its own annotations. By using the `@JoinColumn` annotation we define which column should be mapped to the property — this would be “artist” by default but we declare it for reasons of readability. The foreign key will reference the primary key of the related table by default.

On the other side of the relationship — in the Artist class — we have the opposite relationship type: a “One to Many” relation.

```

55 @OneToMany(mappedBy="artist")
56 public Collection<MusicCd> getMusicCds() {
57     return musicCds;
58 }

```

Listing 1.6. Artist OneToMany relationship

The *mappedBy* parameter declares that this property is referenced by the property artist in the class MusicCd. Using Listing 1.5 and Listing 1.6 there is a bidirectional mapping: we can get the artist of a Cd and all Cds of an artist.

There is a third type of relationships called “Many to Many”. In the music store we have a third entity class called Song which has a “Many to Many” relationship to MusicCd: a CD can have any number of songs and a song can be on any number of Cds. To access the Entities we need SessionBeans which are described in the next section.

```

69 @ManyToMany
70 public Collection<Song> getSongs() {
71     return songs;
72 }
73 public void setSongs(Collection<Song> songs) {
74     this.songs = songs;
75 }

```

Listing 1.8. MusicCd ManyToMany relationship

1.5 SessionBeans

SessionBeans are used to execute the following tasks:

- accessing Entities
- processing business logic

For the first task — accessing the Entities — the Application Server provides the EntityManager Object. The EntityManager has several important methods to perform operations on the Entities:

- `contains(Object entity)`
check if the provided object is already persisted in the container
- `createNamedQuery(String name)`
prepare a previously specified EJB QL query for execution
- `createNativeQuery(String sqlString)`
create a native SQL query
- `createQuery(String ejbqlString)`
create an EJB QL query
- `find(Class<T> entityClass, Object primaryKey)`
retrieve an Entity according to the provided primary key
- `flush()`
synchronise the Java object pool with the backing database
- `merge(T entity)`
commit the changes on the entity object to the underlying database
- `persist(Object entity)`
insert the object data into the database (executes an INSERT query)
- `refresh(Object entity)`
updates the objects properties with the values from the database
- `remove(Object entity)`
drop the rows associated with the entity from the database

For example, if we have a music Cd which has the internal database id 42, we could use the following code to retrieve a MusicCD object (assuming `em` is the EntityManager):

```
MusicCD a = em.find(MusicCD.class, 42);
```

Now that we have an object which is detached from the container we can make changes to the object (e.g. change a typo in the title using the `setTitle` method)

and commit the changes to the database layer:

```
MusicCD b = em.merge(a);
```

After executing the last method call, `b` holds a copy of the object which has just been merged. The `EntityManager` can be used in the Session Beans' methods, where also the business logic is processed. This can be very simple tasks, for example calculating the average price of all Cds by one artist.

The Enterprise JavaBeans Specification defines two types of Session Beans: Stateless and Stateful SessionBeans.

Stateless SessionBeans

Stateless SessionBeans do not retain information about their state. For the software developer this means that a method called on a Stateless SessionBean will return the same value for each client calling the method (assuming the database does not change between calls). A well known use case for this kind of SessionBeans is Web Services. Imagine the Music Store had a web site where the visitor can search for Cds. This web site could also provide a Web Service, therefore remote Websites could display information about the store.

```

1 import java.util.*;
2 import javax.ejb.*;
3 import javax.persistence.*;
4
5 @Stateless
6 public class StoreBrowserBean implements StoreBrowser {
7
8     @PersistenceContext
9     EntityManager em;
10
11     public MusicCd getMusicCd(Integer id) {
12         return em.find(MusicCd.class, id);
13     }
14
15     public Collection<MusicCd> getMusicCdsByArtist(Artist artist) {
16         Collection<MusicCd> musicCds = em.createNamedQuery("
17             findAllMusicCdByArtist").setParameter("artist", artist
18             ).getResultList();
19         return musicCds;
20     }
21 }

```

Listing 1.9. Stateless SessionBean

Listing 1.9 shows an example of a Stateless SessionBean with two methods. Both methods retrieve data using the `EntityManager` which is initialised using dependency injection. Notice that there are no fields in the class. Thus no state information can be carried over method calls.

Stateful SessionBeans

The second type of SessionBeans is able to save its state. An example object that would reasonably be implemented by using a Stateful SessionBean is a shopping

session in our online music store. The indicator for the state of this shopping session would be the status of the shopping cart, which is initially empty. If the visitor tries to submit the order with an empty shopping cart, the `submitOrder` method throws an exception. By saving this state information, the Bean is able to permit access to special data only for a certain state, e.g. the customer may only finish the shopping session after putting something into the shopping cart. The Stateful Session Bean `ShoppingSessionBean` is shown in Listing 1.10.

```

1 import java.util.*;
2 import javax.ejb.*;
3 import javax.persistence.*;
4
5 @Stateful
6 public class ShoppingSessionBean implements ShoppingSession {
7
8     @PersistenceContext
9     EntityManager em;
10
11     private Collection<MusicCd> shoppingCart;
12
13     public void addMusicCd(MusicCd musicCd) {
14         if (shoppingCart == null)
15             shoppingCart = new ArrayList<MusicCd>();
16         shoppingCart.add(musicCd);
17     }
18
19     public void removeMusicCd(MusicCd musicCd) {
20         if (shoppingCart != null && shoppingCart.contains(musicCd
21             ))
22             shoppingCart.remove(musicCd);
23     }
24
25     public void submitOrder() throws EmptyShoppingCartException {
26         if (shoppingCart != null && shoppingCart.size() > 0) {
27             // submit order to queue...
28         } else throw new EmptyShoppingCartException();
29 }

```

Listing 1.10. Stateful Session Bean

The Session Bean has one method for each adding and removing Cds to the shopping cart and one method to submit the order. The `submitOrder` checks if the field `shoppingCart` which holds a `Collection` of `MusicCd` objects is empty. If not, the order will be sent to a message queue on the application server. If the `shoppingCart` does not contain any objects, the `EmptyShoppingCartException` will be thrown.

1.6 Message Driven Beans

Java Message Service

”The Java Message Service (JMS) API is a messaging standard that allows application components based on the Java 2 Platform Enterprise

Edition (J2EE) to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous” [Sun06a]

The JMS API supports two different models: the *Point-to-Point Messaging (Queue)* and the *Publish/Subscribe Messaging (Topic)*. The JMS architecture has a set of elements, those are:

- the **JMS Provider** is an implementation of the JMS interface that is capable to maintain topics and queues. The provider does not need to be implemented with Java.
- the **JMS Client** is an application that produces or/and consumes messages from/to a queue or topic.
- the **JMS Producer** is a JMS client which creates and sends messages to a queue or topic.
- the **JMS Consumer** is a client which receives messages.
- the **JMS Message** is the message or object which is transferred between JMS clients.
- the **JMS Queue** is a mechanism to receive messages and store them until they are fetched by a JMS client. The messages are delivered in the order they came in.
- the **JMS Topic** is a mechanism to receive messages and deliver them to multiple JMS clients.

In the Point-to-Point model or queueing model (Fig. 1.1), a client (producer) creates a message and sends it to a queue. The subscribed consumer receives this message and sends an acknowledgement to the queue. Important for the message queues is that each message has only one consumer and every sent message must be processed successfully.

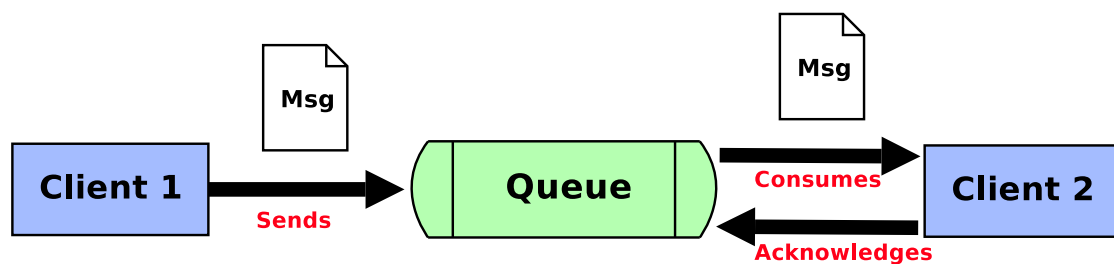


Fig. 1.1. The Point-to-Point Message Service (Queue)

The Publish/Subscribe model (Fig. 1.2) takes care of multiple consumers of a message. A producer sends a message to a topic and each consumer who is subscribed to this topic will receive this message. The consumer must be subscribed to the topic before the message was sent by the producer to receive the message.

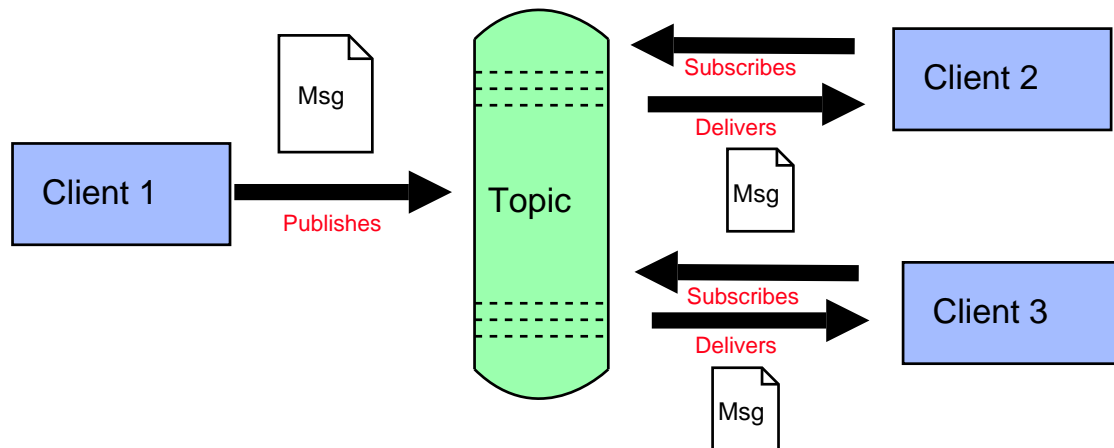


Fig. 1.2. The Publish/Subscribe Message Service (Topic)

Message Driven Beans (MDB) can act as JMS producer as well as JMS consumer. They have similar properties like stateless SessionBeans. A Message Driven Bean's instances retain no data or conversational state for a specific client. All instances of a Message Driven Bean are equivalent. The container can pool these instances to allow streams of messages to be processed concurrently. A single Message Driven Bean can process messages from multiple clients. Unlike SessionBeans a Message Driven Bean can receive asynchronous JMS messages instead of synchronous method invocations. Asynchronous messages can be sent by any J2EE component - an application client, another enterprise bean or a web component.

According to EJB 3.0, the MDB bean class is annotated with the *@MessageDriven* annotation, that specifies the message queue that is monitored by this MDB (i.e., queue/musicstore). If the queue does not exist, the EJB container automatically creates it at deploy time.

The bean class needs to implement the *MessageListener* interface, which defines only one method *onMessage()*. When a message arrives in the queue monitored by this MDB, the EJB container calls the bean class's *onMessage()* method and passes the incoming message on as the call parameter (Listing 1.11).

1.7 Container

The EJB container is a special environment in whose scope all enterprise beans are executed and managed. It acts in the same way as a Java Web Server hosts a servlet or an HTML browser hosts a Java applet. They cannot function outside of the container. The EJB container manages all aspects which are needed at runtime like concurrency, persistence, transactions, security, remote access to the beans and the pooling of resources.

```

1 @MessageDriven(activateConfig =
2 {
3   @ActivationConfigProperty(propertyName="destinationType",
4     propertyValue="javax.jms.Queue"),
5   @ActivationConfigProperty(propertyName="destination",
6     propertyValue="queue/musicstore")
7 })
8 public class OrderListenerBean implements MessageListener {
9
10  public void onMessage (Message msg) {
11    try {
12      TextMessage tmsg = (TextMessage) msg;
13      // do something
14
15    } catch (Exception e) {
16      e.printStackTrace ();
17    }
18  }
19 }
20 }

```

Listing 1.11. A Message Driven Bean

The enterprise beans are completely insulated in the EJB container from direct access by remote clients. When a client wants to connect to an enterprise bean inside, the container intercepts this call to ensure properly applied persistence, transaction and security on to the bean. All these basic and very important features are managed automatically by the container to allow the bean developer to only concentrate on implementing the business logic. Compare [JGu00].

Bean Pooling

Once deployed, the container creates a collection of instances for each enterprise bean. To reduce memory consumption the container keeps these instances in a pool. Each time a client requests an instance of a bean, the container binds an instance from the pool to the request. After the connection is reset, the bean instance is pushed back into the pool. If there are more requests than instances the container creates additional instances and binds them to the requests. During this life-cycle, the instances pass through different states (fig 1.3). An enterprise bean is in "no state" when it is deployed but the container has not created an instance. The instances in the pool that are not bound to a connection are in "pooled state". The ones bound to a client connection are in "ready state".

To instruct the enterprise bean to run some special operations before or after it changes to a special state there are annotations to mark methods for that purpose. See table 1.1 for details.

Annotation	Description
@PrePersist	The annotated method is invoked right before the entity is created in the database.
@PostPersist	The annotated method is invoked right after the entity is created in the database.
@PreRemove	The annotated method is invoked right before the entity is deleted in the database.
@PostRemove	The annotated method is invoked right after the entity is deleted in the database.
@PreUpdate	The annotated method is invoked right before the database is updated.
@PostUpdate	The annotated method is invoked immediately after the database has been updated.
@PostLoad	The annotated method is invoked right after data has been loaded from the database and associated with the entity.

Table 1.1. Life-cycle Callback Methods

As the container encapsulates the enterprise beans from remote calls, the clients are completely unaware of the bean pooling. A bean that is not in use at the moment can be pooled or evicted while its remote reference in a client is still alive. In case of a re-invoke by the client, the bean instance is reanimated and bound again to the request.

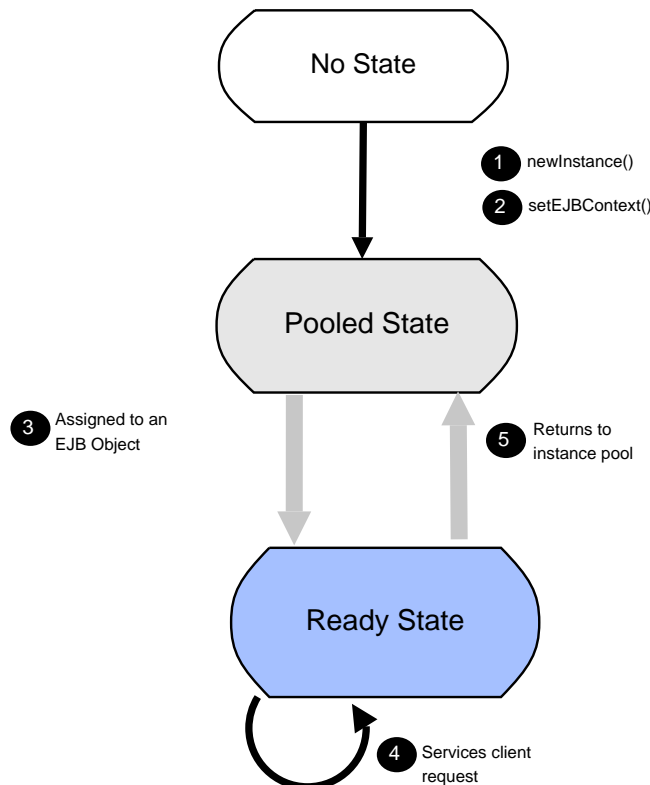


Fig. 1.3. Enterprise Bean Instance Pooling

Accessing Enterprise Beans

As mentioned before, all enterprise beans can only function inside an EJB container. Hence, there is a way to access Session and Message Driven Beans from remote. Entities cannot be accessed from remote clients. During the deployment, all remote interfaces are bound to the *Java Naming and Directory Interface (JNDI)*

```

1 try {
2     InitialContext ctx = new InitialContext();
3     ShoppingSession bean = (ShoppingSession)
4         ctx.lookup("ShoppingSessionBean/remote");
5 } catch (NamingException e) {
6     e.printStackTrace();
7 }

```

Listing 1.12. JNDI Look Up

```

1 java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
2 java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
3 java.naming.provider.url=127.0.0.1:1099

```

Listing 1.13. JNDI Properties File

to be available for remote connections. The clients can send a request to the JNDI service to retrieve an instance of the bean. Listing 1.12 shows a look up for a `SessionBean`.

The *InitialContext* is the gateway to the JNDI service. The destination of the JNDI service must be specified for this `InitialContext`, e.g. in a properties file like in Listing 1.13

The look up for the `SessionBean`'s remote interface is done by asking the JNDI service for the *ShoppingSessionBean*'s remote interface ("*ShoppingSessionBean/remote*"). This will prompt the container to get an instance of the `ShoppingSessionBean` from the instance pool and bind it to this request. The `SessionBean` is now ready for use. The *NamingException* is thrown when the JNDI service is not available.

The underlying process can be seen in Fig. 1.4. After the client has sent the request to the JNDI service it demands the adequate stubs for the `SessionBean`'s interface from the application server as it is known from *RMI*. The client executes methods on the stub that sends this "method request" to the application server's object of the `SessionBean`. The application server can intercept the execution of the request in order to do security checks, to ensure transactions and persistence. If this is passed, the `SessionBean` is allowed to execute the request and send the response to the client's stub.

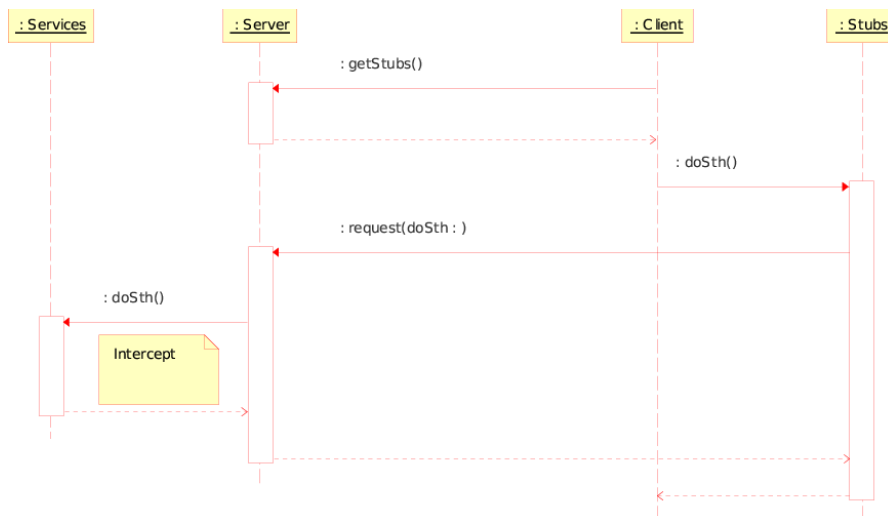


Fig. 1.4. Client - Server Communication

The usage of enterprise beans from inside the container, e.g. within other SessionBeans, is much easier. If two session beans run inside the same container, they are in the same scope. Therefore the container offers a possibility to inject one instance of a SessionBean into another at run-time. The *Dependency Injection* (Inversion of Control, IoC) design pattern means that an application component defines which type of service components it requires. The container will determine these requirements, create the instances of the service objects and inject them into the application component at run-time.

EJB 3.0 offers the possibility to achieve the injection by using annotations. The `@EJB` tag can be used to inject EJB stubs. It replaces the SessionBean look up code. The container figures out how to retrieve a stub of a bean and assigns it before it is first used (Listing 1.14). The injected reference can be used as normal object reference.

```

1 @Stateless
2 public class ShoppingSessionBean implements ShoppingSession {
3
4     @EJB
5     AnotherSessionBean secondBean;
6
7     public ShoppingSessionBean () {
8         // ....
9         secondBean.doSomething();
10    }
11 }

```

Listing 1.14. EJB Dependency Injection

1.8 JBoss Application Server

The JBoss Application Server [JBo06] is a free and open source Java 2 Enterprise Application server. It is licensed under the LGPL (GNU Lesser General Public License)[Fre99] and can be used for free. JBoss implements the full J2EE stack of services:

- EJB (Enterprise JavaBeans)
- JMS (Java Messaging Service)
- JTS/JTA (Java Transaction Service/Java Transaction API)
- Servlets and JSP (JavaServer Pages)
- JNDI (Java Naming and Directory Interface)

It also provides advanced features such as clustering, JMX, web services and IIOP (Internet Inter-ORB Protocol) integration. The first version of GeCam uses EJB 2.1 and JBoss version 4.0.1 which is certified by Sun Microsystems to work with J2EE 1.4. The new specification of Enterprise JavaBeans, 3.0, simplifies the development of enterprise applications a lot, therefore the decision was made to migrate to EJB 3.0. As the EJB 3.0 specification had reached its final state in December 2005, the used JBoss version for the current GeCam version is 4.0.4 RC1, which is a stable release candidate and requires Java 1.5.

1.9 Loading Strategies

EJB 3.0 offers two loading strategies to minimise the data that is transferred between server and clients, the lazy and eager loading. According to the music store example the class *Artist* has a collection of *MusicCD*. If it is sufficient to read only the artist's data (name, nationality, id) and not the nested collections of Songs and MusicCDs; these relations can be marked for the so called "Lazy Loading". This can be achieved by adding an option to the relationship annotation. Listing 1.15 shows the OneToMany relation of the Artist to its MusicCDs. The option *fetch = FetchType.LAZY* effects that during a database look up for an Artist its nested MusicCD objects will not be initialised. After having transferred the entity to a client the appropriate field in the entity is *null*.

```

1 @OneToMany(mappedBy="artist", fetch = FetchType.LAZY)
2 public Collection<MusicCd> getMusicCds() {
3     return musicCds;
4 }

```

Listing 1.15. Lazy Fetching Annotation

When a client needs all MusicCDs of an Artist, the nested collection must be explicitly initialised by iterating over the Artist's MusicCDs and call a getter Method. This must be done within a SessionBean. Listing 1.16 shows the procedure. After the Artist object is obtained from the database, the collection of MusicCDs is iterated and for each nested MusicCD the getter *getId()* is called. This will cause the container to fully load this object from the database.

```

1 public Artist getHeavyWeightArtist(Integer artistId) {
2     // Obtain the Artist object from the database
3     Artist artist = em.lookup(Artist.class, artistId);
4
5     // initialize the MusicCDs
6     if (artist.getMusicCds() != null) {
7         for (MusicCD cd : artist.getMusicCds())
8             cd.getId();
9     }
10
11     return artist;
12 }

```

Listing 1.16. Initialize Lazy Annotated Relations

A relation marked with the fetch type *EAGER* will be automatically initialised when the parent object is retrieved from the database. If no fetch type is specified all simple data types are eagerly loaded and BLOBS are lazily fetched.

The EJB 2.1 specification did not allow to transfer entities to the client. The developer had to fall back on self written data objects to transfer the desired data. This led to an enormous overhead to read the entities, create the data objects, transfer them to the clients and to receive data objects from the clients and deliver the data to the appropriate entities. The advantage of this mechanism is that there is only the data transferred that is actually needed. In EJB 3.0 it is now possible to transfer directly the loaded entities to the clients. In case of small entities this is the easiest way but if there are very large or complex objects to transfer it would be better to create data classes that only contain the important data.

1.10 CascadingTypes

As known from database systems there are also in EJB the so called *Cascading Types* that define the behaviour of database table relations in case of an insert, update and delete. Regarding the music store example in case of an artist being deleted it would be nice if all his songs and music Cds would also be deleted. But if a Cd is going to be deleted its songs should not be, because they can be used in other Cds as well. The cascade type is also defined as an option of the relation annotation, see Listing 1.17. Table 1.2 shows all available cascading types.

```

1 @OneToMany(mappedBy="artist", cascade = CascadeType.REMOVE)
2 public Collection<MusicCd> getMusicCds() {
3     return musicCds;
4 }

```

Listing 1.17. Cascade Type Remove

Annotation	Description
PERSIST	In case of persisting the parent entity, the nested collection is also persisted.
MERGE	If a persisted entity is stored again, the nested collection is also merged.
REMOVE	If the parent entity is deleted, all nested entities will also be deleted.
REFRESH	If an entity is refreshed by the EntityManager, the nested collection is also refreshed.
ALL	Combination of all three types (Persist, Merge, Remove, Refresh)

Table 1.2. Cascade Types

1.11 Deployment

Deployment describes the distribution and installation of software on target systems, including the configuration. That means, all components like entities and session beans are pushed into the container. It will map the available entities to the appropriate database tables and activate and publish the session beans to the JNDI service. The Java 2 EE Deployment is described in the EJB3 persistence API. This ensures the vendor independent development of enterprise beans.

Deployment takes place when a file (class file, xml file, jar file, other supported formats) is copied to the server's deploy directory — for example `/jboss-4.0.4RC1/server/default/deploy`. This is called hot deployment because the application server does not have to be restarted. When a jar file is put into this directory, the container scans the archive for an xml file called META-INF/persistence.xml which describes the data source(s) for the persistence unit(s) this archive represents. The following Listing shows the music store persistence.xml file:

```

1<?xml version="1.0" encoding="UTF-8"?>
2<persistence>
3  <persistence-unit name="musicstore">
4    <jta-data-source>java:/musicstore</jta-data-source>
5    <properties>
6      <property name="hibernate.dialect"
7        value="org.hibernate.dialect.PostgreSQLDialect"/>
8      <property name="hibernate.hbm2ddl.auto" value="none"/>
9    </properties>
10  </persistence-unit>
11</persistence>

```

Listing 1.18. persistence.xml

The GeCam persistence.xml defines a persistence unit called gecam. At line 4 the data source for the database which will contain the entity data is referenced. This must be the JNDI name which was previously declared in the jboss configuration. For this datasource two properties are defined:

- `hibernate.dialect`
This configures the SQL Dialect hibernate is using. In the example the application server is connected to a PostgreSQL server hence hibernate uses the `PostgreSQLDialect`.
- `hibernate.hbm2ddl.auto`
Hibernate is able to create the database schema (tables, sequences) as defined by the annotations. This can be useful if there is no need to access

a legacy database or if the data does not need to be persisted when the application server is shut down. There are three values for this property: `validate`, `update`, `create` and `create-drop`.

If set to `validate` the persistence provider will check if the database matches the annotations. If the schemas do not match, deployment is stopped and an exception is thrown. If set to `update` the database scheme will be changed to match the EJB3 annotations. `create` will tell the persistence provider to create the database structure (once at deployment time) and `create-drop` is used if the database should be dropped when undeploying.

Creating EJB Applications using the JBoss Eclipse IDE

2.1 Eclipse

Eclipse is an open source software framework written in Java and therefore platform-independent. It is used to create Rich-Client Applications like the Integrated Development Environment (IDE) for Java, also named *Java Development Toolkit* (JDT). But there are many other project using the Eclipse software framework as base structure (e.G. Azureus, Lotus Notes 7, ..). There are many extensions for the IDE, hence it could be used to develop PHP, Python, C++, COBOL, etc. See <http://www.eclipse.org> for plugins, extensions and projects.

2.2 Getting Started

2.2.1 Download Eclipse SDK

The first step is to download the latest Eclipse SDK release from

```
1 http://www.eclipse.org/downloads/
```

Be sure to download the **Software Development Kit (SDK)** version compiled for your operating system. This is important because Eclipse uses a windowing toolkit that uses components from the underlying operating system. The current version is **Eclipse 3.2.1**.

Unzip the archive to your preferred directory. Since Eclipse is written completely in Java there is no installer that will put it somewhere on your hard drive. Just run the **eclipse** executable to start the software.

During the startup, you will be asked to select a **workspace** (Fig. 2.1). This is the place where all your project's data will be stored. We will use a workspace located in:

```
1 /media/data/workspace
```

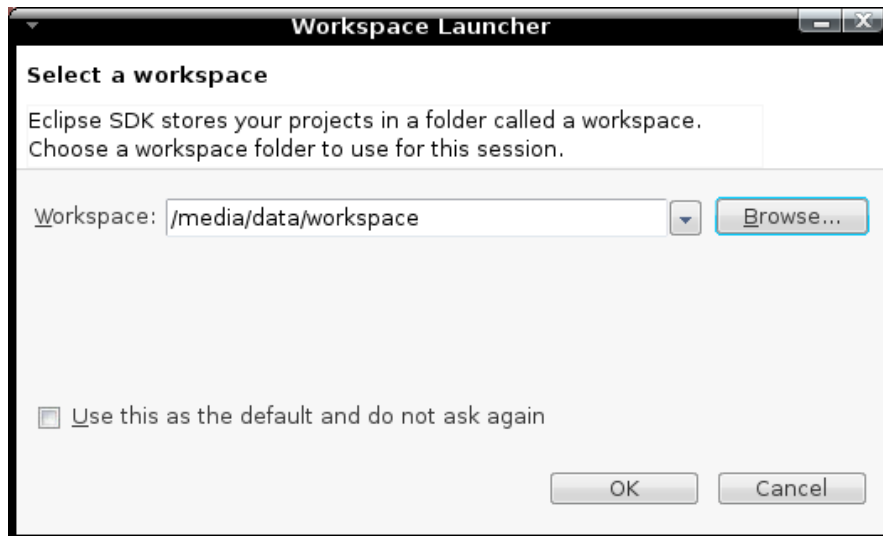


Fig. 2.1. Select a workspace for your projects

The next screen you will see is the *Welcome* screen (Fig. 2.2). It is (some kind of) a starting page for new users and offers quick access to get an overview of the Eclipse features, find out about the new things that have been implemented since the last version, can browse a large bunch of tutorials and samples. If this is your first time using the Eclipse IDE, or if you are not very experienced using it, feel free to have a look in this sections. The tutorial assumes that you are familiar with the basic functionalities.



Fig. 2.2. The Welcome page

When you get this far, its time to switch to the **workbench**. Your eclipse installation is now ready to be used.

2.2.2 Download JBoss IDE for Eclipse

Having in mind, that we want to develop an *EJB3* application, the IDE must be extended by a plugin-in that offers features, libraries and functionalities especially for that kind of application: the **JBoss IDE for Eclipse**. It is a set of Eclipse plug-ins:

- Integrated development environment via integration with Eclipse 3
- EJB 3.0 Project Wizard
- Hibernate Tools
- Aspect Oriented Programming (AOP) Tools
- JBoss jBPM Graphical Process Designer
- Wizards to ease and simplify J2EE development
- Integrated debugging, monitoring, and lifecycle control of JBoss servers
- JSP, HTML, XML, CSS, and Javascript editors with syntax highlighting
- Easy configuration and deployment of package archives

Source [<http://www.jboss.org>]

The plug-in is going to be installed by using the built-in update mechanism of eclipse. It can be found in

```
1 Help > Software Updates > Find and Install
```

(Fig. 2.3)

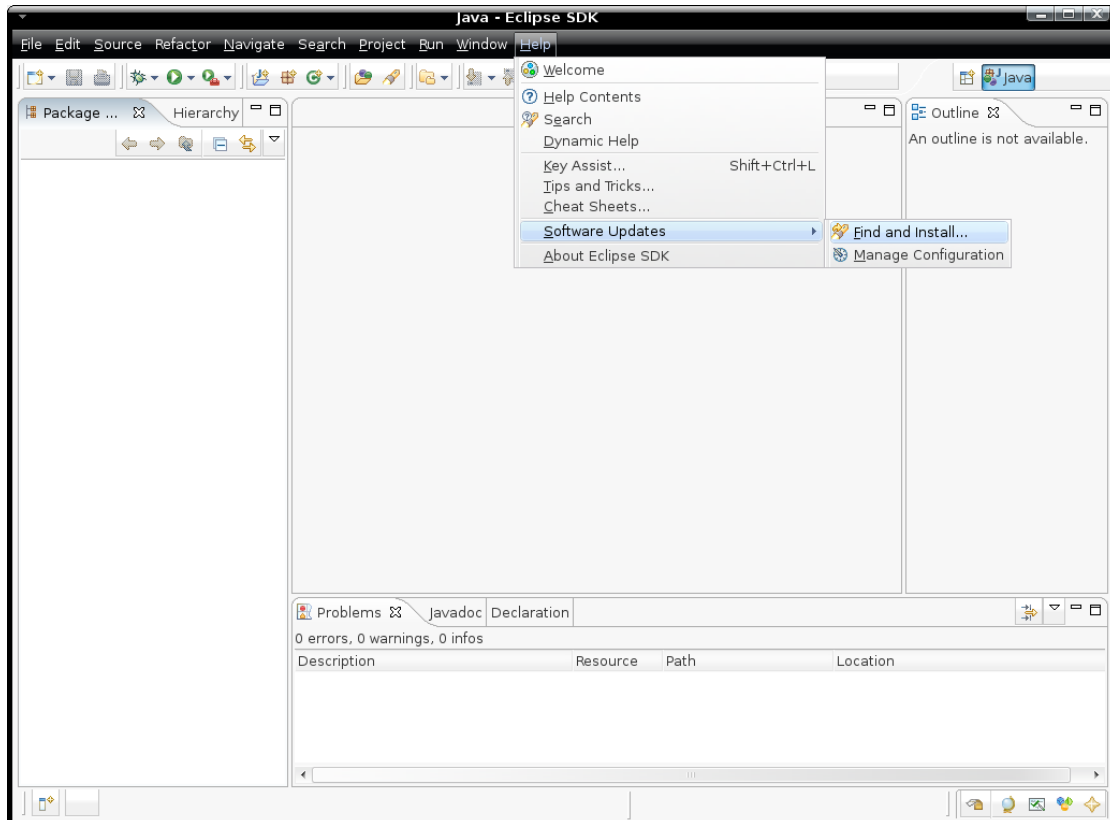


Fig. 2.3. Install Eclipse Plugins via update-site

Select "Search for new features to install" and create a "New Remote Site..". Fill the name with "JBoss Eclipse IDE" and the URL with

```
1 http://download.jboss.org/jbosside/updates/development
```

Make sure that *JBoss Eclipse IDE* and *Callisto Discovery Site* have a check next to it and press *Finish*. The latest version of the JBoss IDE is *2 beta*. Select the packages as displayed in Fig. 2.4. We need the *Eclipse Modeling Framework*, the *EMF Service Data Objects (SDO)* and, of course, the *JBoss Eclipse IDE*. The following screens can be left as they are. Just accept the license agreements.

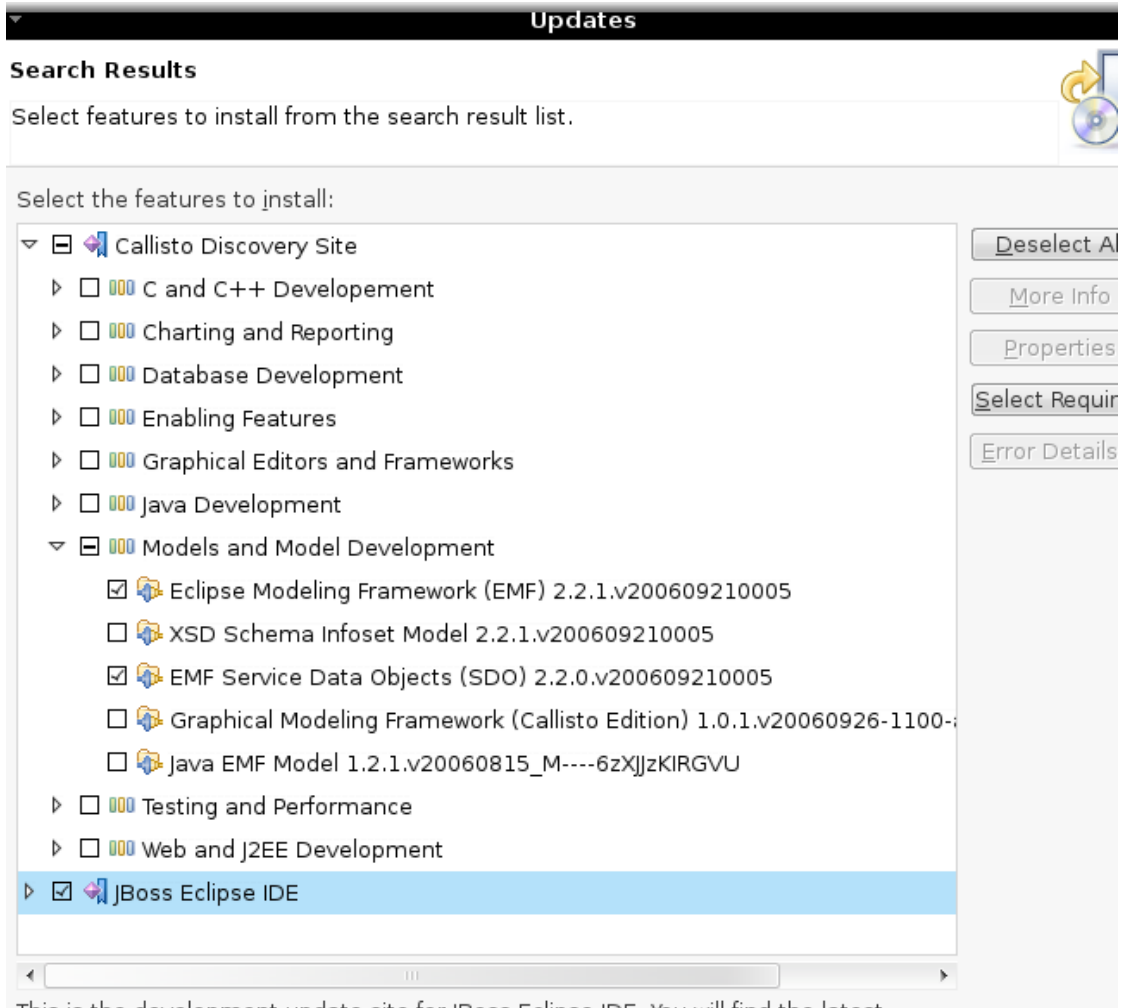


Fig. 2.4. Select Packages to Install

The plug-in is now going to be installed. The installation time depends on your internet connection. When all packages are downloaded, you will be prompted by a verification request for an "unsigned feature" that should be installed. Select *Install All* to proceed (Fig. 2.5).

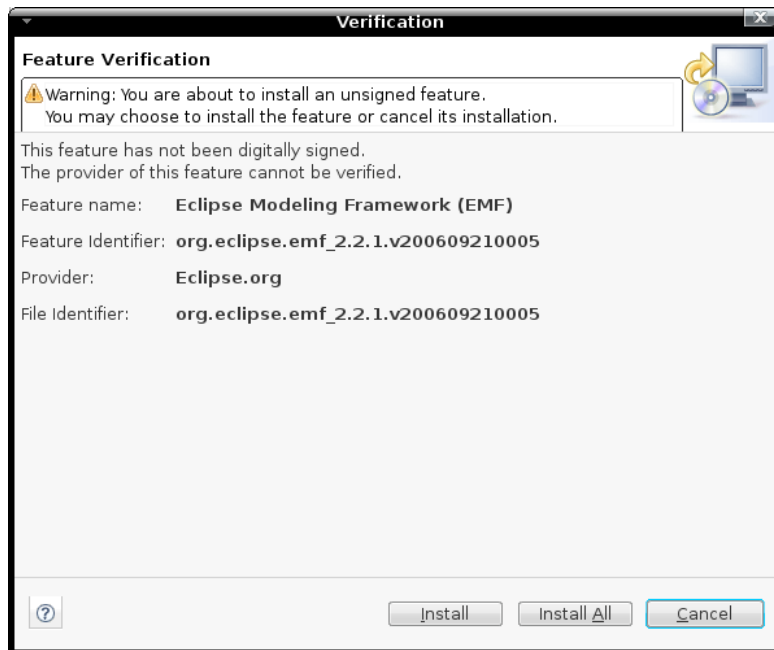


Fig. 2.5. Verify unsigned features

When this is all done, you will be asked to restart the Eclipse SDK. Just do it. When it comes up again, you will be asked to convert projects that were created using older JBoss IDE versions. As we do not have any of these now, press *Cancel*. The IDE is now prepared for EJB3 projects.

2.2.3 Download JBoss Application Server

As we now have installed the eclipse IDE and the JBoss IDE plug-in, the server should also be installed. Download the latest "Production" release of the JBoss Application Server from

1 <http://labs.jboss.com/portal/jbossas/download>

You can do it by running the webstart installer, which is definitely the easiest way. Select the directory in which JBoss should be installed. The next screen you will see is the following one (Fig. 2.6)

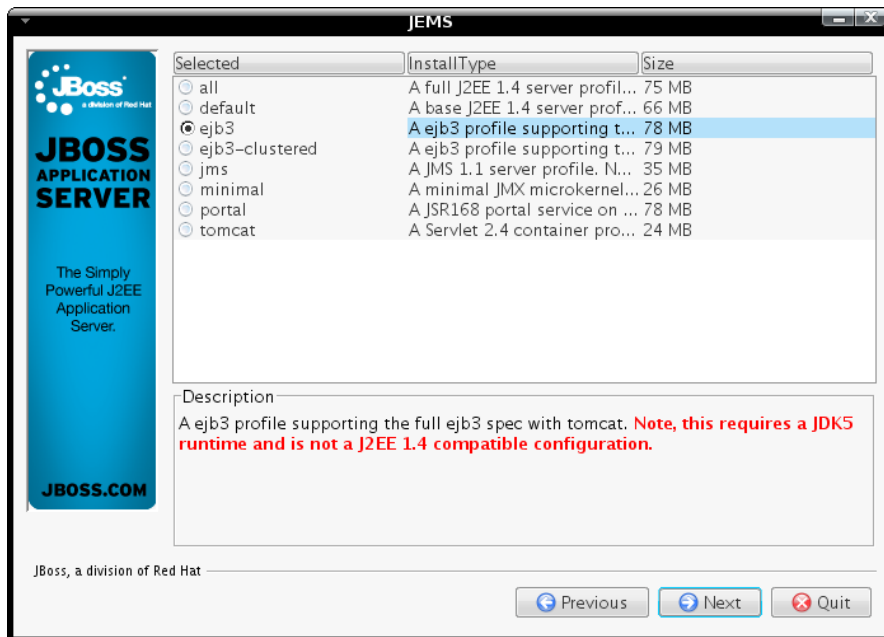


Fig. 2.6. Select the JBoss configuration

Select the "ejb3" option. Note that you must have JDK5 installed on your machine. The EJB3 specification is based on the improvements of the JDK5 and it would not work with version 1.4 or older. The following screens can be left as they are, just click "Next" and finish the installer. That's all for now for the JBoss Application Server.

2.3 Prepare the Workspace

At this point, we have all the tools installed that we need for our project. But like always, the tools have to be configured to work properly. There are two tasks left that have to be done before starting implementing anything.

- Create a JBoss configuration.
- Create an EJB3-Project

2.3.1 Create an Eclipse JBoss Configuration

Why do we need this? As it is normal to have different versions of software, the configuration for a JBoss server in Eclipse allows to choose for each project for which version of JBoss the application should be built. Which means in fact, which libraries should be used when the final packages are built. And the server can be started out of eclipse and also run in debug mode which allows to use breakpoints in the server application.

The configuration is created by using the "JBoss dropdown box" as it is shown in Fig. 2.7.

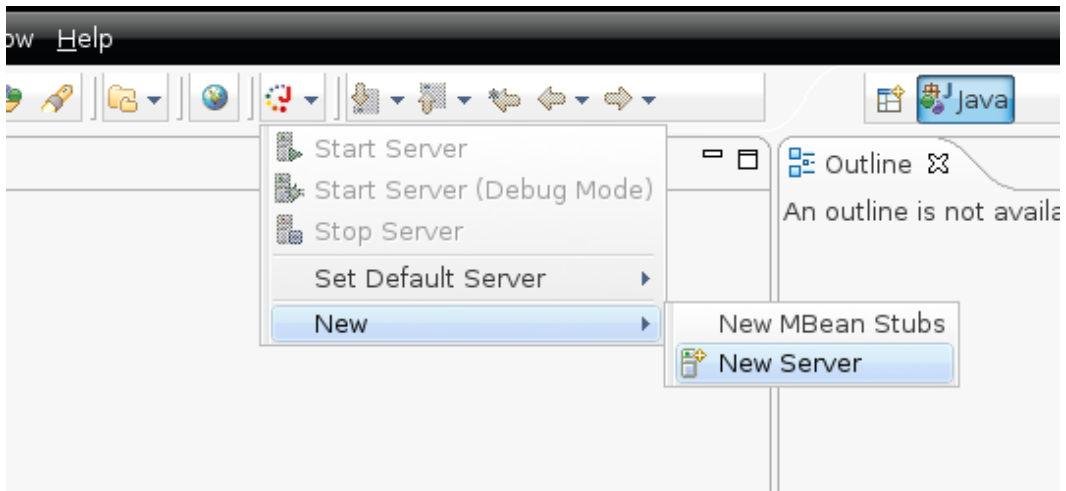


Fig. 2.7. Open the JBoss configuration screen

Select

1 JBoss Inc > JBoss AS 4.0

In the next screen you must create a JBoss Server Runtime, this is the default location of the JBoss AS. Choose a name and select the directory in which the JBoss AS has been installed.(Fig. 2.8)

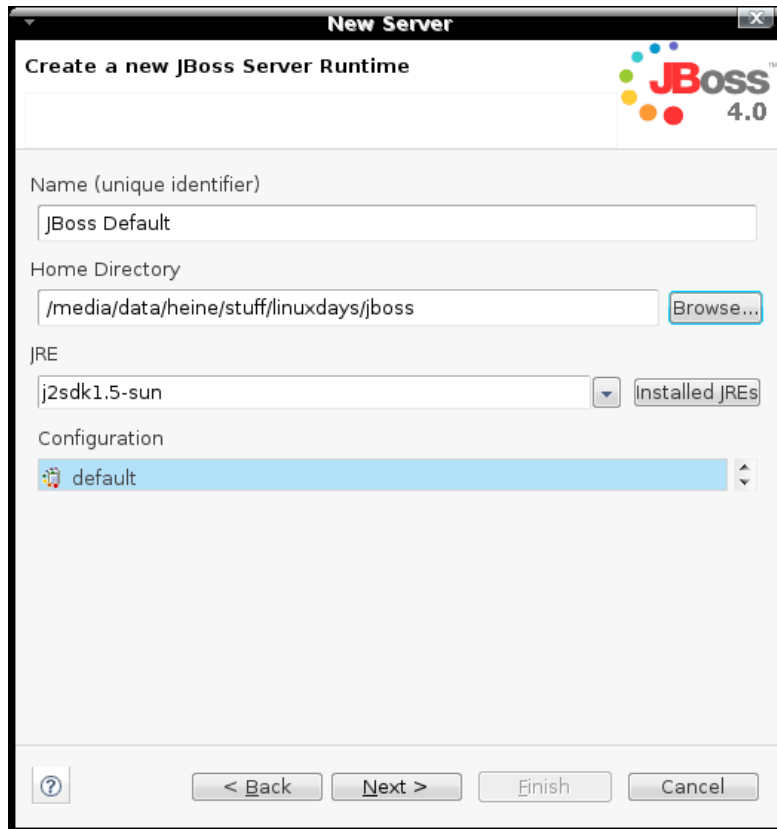


Fig. 2.8. Create a JBoss Server Runtime

The design of the JBoss architecture allows it to run multiple instances of one installation. Each instance must have its own directory and configuration files. Therefore the next step in the wizard is to create a JBoss configuration that we will use for our application. Call it "MusicStoreServer". We will use the default configuration that is shipped with JBoss. Click *Finish*.

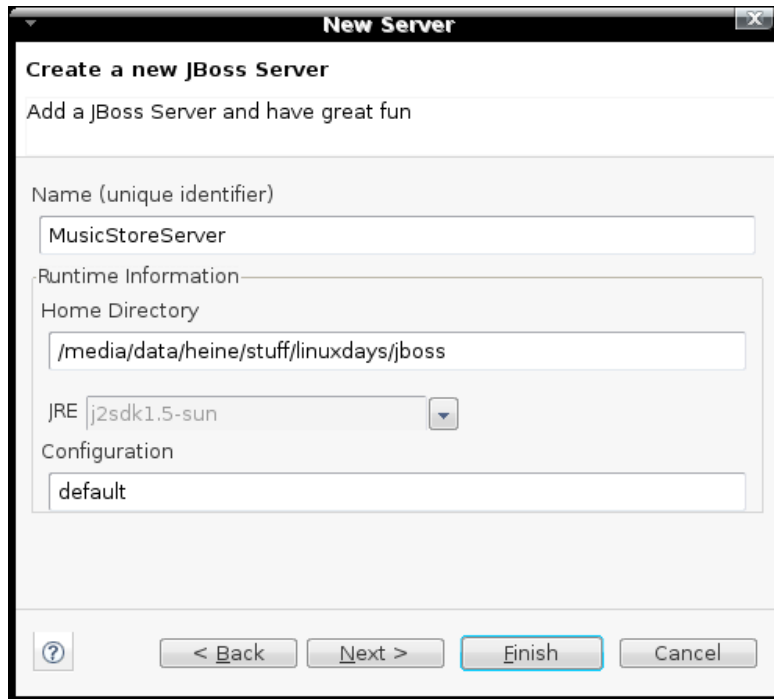


Fig. 2.9. Create a Server Configuration

The new server configuration now appears in the Package Explorer on the left side of the eclipse screen.

2.3.2 Create an EJB3-Project

Create a new Project. Select

```
1 File > New > Project
```

Select an "EJB 3.0 Project". Name it "MusicStore" and select the newly created server "MusicStoreServer". The project is displayed on top of the server configuration in the Package Explorer.

2.4 Development of an Application

Add the Java classes mentioned in the previous sections into the new project.

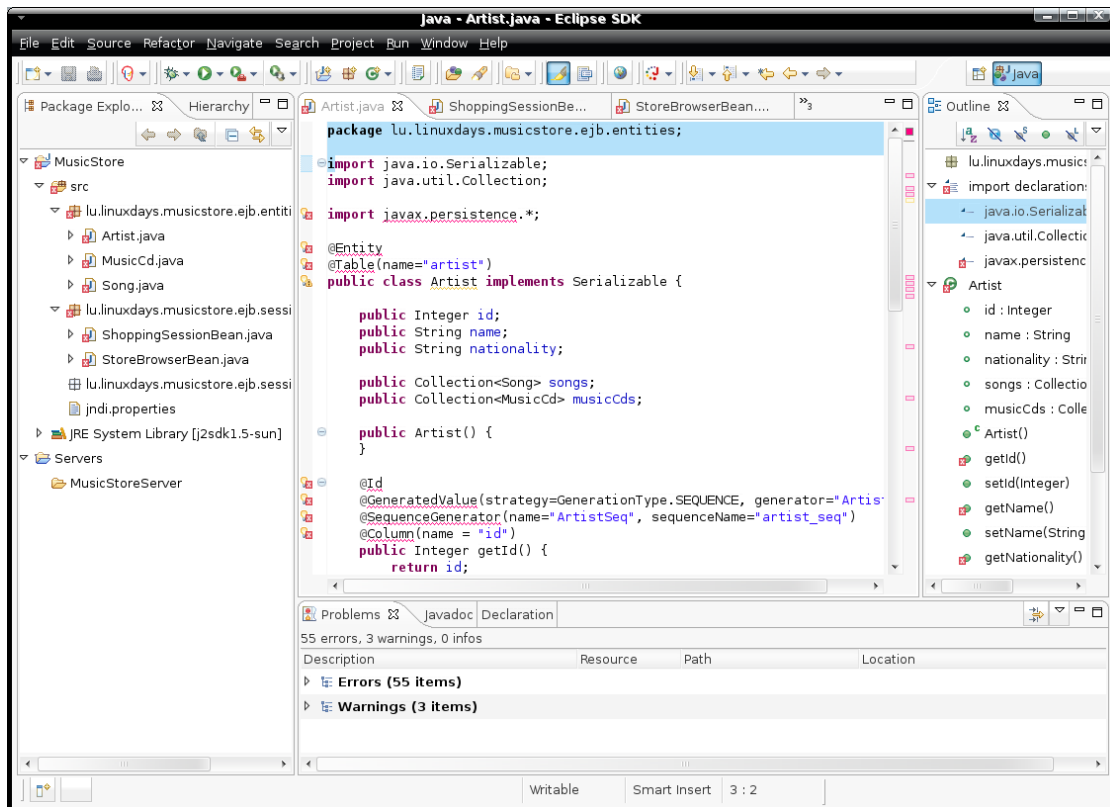


Fig. 2.10. Import the java classes into the project

As you can see, there are many errors regarding used imports that can not be resolved, e.g.

```
1 import javax.persistence.*;
```

At this point we can make use of the features from the JBoss IDE plug-in. The imports can be fixed by adding the libraries that contains these packages. They are all shipped with the JBoss AS package and we can just take the ones from the server configuration that we have already created. Do right click on the project, select "Build Path", "Add Libraries". Select "JBoss EJB3 Libraries" and choose the configured JBoss. You will see that all the import problems have been fixed.

The Session Beans implement an interface. The intercaes specify which methods of the Session Beans are accessible from the outside. Hence we must create them in the package "lu.linuxdays.musicstore.session.interfaces".

2.5 JBoss Configuration

Each application needs a datasink to store its data. In our case we will use a database. To make it as easy as possible, we will use the internal Hypersonic SQL (hsq) database of the JBoss. Its a database written in pure java that can be run in different modes like in-memory and disk-based tables and supports embedded and server modes. The connection property files for databases can be found in the

JBoss folder *jboss/server/default/deploy*. For the hsql it is *hsqldb-ds.xml*. Have a look at it and you will recognize the jndi-name "DefaultDS". A new application that is deployed on this server must specify which database should be used. This is done by using the "jndi-name". Leave this file untouched. It will work perfect for our needs. In case you want to use another database such as postgresSQL or mySQL, make shure to put the JDBC driver file into the deploy directory as well as a *xxx-ds.xml* file for the connection properties.

To ensure, that the application can access the database respectively the JBoss knows which datasource should be used, there must be a configuration file in the application. It must be named *persistence.xml* and should be placed in *src/META-INF*. The content of this file is depending on the persistence provider, that means since JBoss uses Hibernate as middleware, it is just an Hibernate-Configuration file. Therefore, we have on the one hand, the database configuration on server side in the *xxx-ds.xml* files which contains the connection properties for the database and on the other hand the configuration file for the application that includes which in the server configured database should be used and in which way. See section 1.11 for detailed information.

```

1  <persistence>
2    <persistence-unit name="musicstore">
3      <jta-data-source>java:/DefaultDS</jta-data-source>
4      <properties>
5        <property name="hibernate.hbm2ddl.auto"
6          value="create" />
7      </properties>
8    </persistence-unit>
9  </persistence>

```

Listing 2.1. persistence.xml

2.6 Packaging

Each applicaton that should be run in the JBoss must be packed into a single file. Its just a jar file that contains all compiled java classes and configuration files. To create one, right-click on the project, select "Properties" and in the new window "Packaging Configuration". Enable packaging and afterwards add a new configuration. Name it "MusicStore.jar". Now we must add all files and folders that should be packed into the jar file. You can add files and folders by right-clicking on the configuration. But, only select the files and folders from the **bin** directory!! At first the persistence.xml. Add a new file and select persistence.xml from bin/META-INF/. The prefix must contain "META-INF"! Then, add all files needed by the application server, in fact all Entities, Session Beans and their interfaces. Add a new folder and select it as displayed in Fig. 2.11

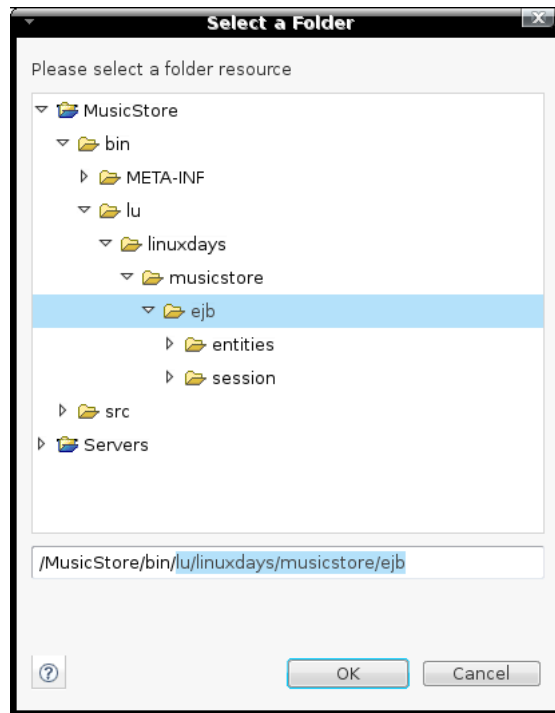


Fig. 2.11. Add folders to the packaging configuration

The "Include" field should contain the string `**/*.class`, meaning that only `.class` files should be packaged from this folder. The "Prefix" must contain the complete package path to the selected one. Otherwise the container can not find any class files. (Fig. 2.12)

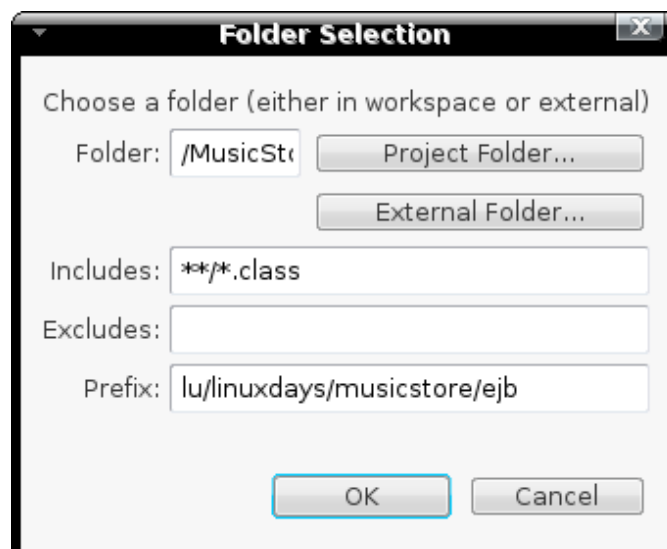


Fig. 2.12. Add package prefix for folders

To start the package generation, right-click on the project and select "Run Packaging". The "Console" view will display the log of the ongoing process. The generated jar file is placed in the project folder.

2.7 Deployment

How can Eclipse help to deploy the newly generated package? Its very simple. right-click on your package and select

```
1 Run As > Run on Server
```

Select your configured JBoss server and click *Finish*. Eclipse will automatically start the JBoss and deploys the package in the folder *JBOSS_HOME/server/default/deploy*. Thats all. There exists another possibility to deploy a package. It is normal to deploy a package sometimes a hundred times during development. It can get very annoying if each time the server needs to be restarted. The nicer way to deploy during development is to start the server by hand in the console. Go to the directory *JBOSS_HOME/bin* and type *./run.sh*. This will start the JBoss using the default configuration. Followed by this, create a softlink from the deploy folder to your project folder.

```
1 #>mkdir JBOSS_HOME/server/default/deploy/music
2 #>ln -s JBOSS_HOME/server/default/deploy/music
3 /media/data/workspace/MusicStore/deploy
```

Refresh your workspace and you will find a new folder "deploy". Go to the Packaging Configuration and edit the settings for the jar file. Point the "Destination" to the new created "deploy" folder in the project. The effect of this will be that each time a package is generated it will be copied into that folder. And as this folder is a subfolder of the deploy directory of JBoss, it will be deployed automatically.

2.8 Run the clients

To run a remote client that should access the remote interfaces of the session beans we must add some parameters to the command that starts the application. Add the following lines to the "VM arguments" of the run configuration of the client.

```
1 -Djava.security.manager
2 -Djava.security.policy=client.policy
```

The file "client.policy" must contain the following content:

```
1 grant {
2     permission java.security.AllPermission;
3 };
```

Additionally the client must know where to ask for the services. Hence the client is looking per default for a file called *jndi.properties*. It contains the url to the JNDI naming server.

```
1 java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
2 java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
3 java.naming.provider.url=localhost:1099
```

The file must be placed into the classpath of the client application. Thats all.

2.9 Information

For further information the following resources might be of interest.

- *Enterprise JavaBeans 3.0*
Bill Burke & Richard Monson-Haefel, O'Reilly, ISBN 0-596-00978-x
- <http://www.eclipse.org>
- <http://www.jboss.org>
For all JBoss Projects, also have a look at the forums for any kinds of problems.
- <http://trailblazer.demo.jboss.com/EJB3Trail/>
The official JBoss EJB3 tutorial
- http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html
Hibernate documentation for annotations.

References

- [Fre99] Free Software Foundation, Inc. Lesser Gnu Public License(LGPL), 1991 - 1999.
<http://www.gnu.org/copyleft/lesser.html>.
- [JBo06] JBoss Inc, 2002 - 2006.
<http://www.jboss.com>.
- [JGu00] jGuru: Enterprise JavaBeans Fundamentals, 2000. Sun Developer Network/ Tutorials and Code Camps
<http://java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html>.
- [Sun97] Sun Microsystems Inc. JavaBeans(TM) Specification 1.01 Final Release, 1996,1997.
- [Sun06a] Sun Microsystems Inc, 1994 - 2006. Sun Microsystems Inc.
<http://www.sun.com/>.
- [Sun06b] Sun Microsystems, Inc. Java Platform, Enterprise Edition (Java EE) 5 Technologies. <http://java.sun.com/javaee/5/javatech.html>, 2006.