

Building and Deploying Java EE Applications

Contents

Chapter 1	Introduction to JEE
Chapter 2	Introduction to JBoss and Wildfly
Chapter 3	Introduction to REST
Chapter 4	Working with REST and JAXRS
Chapter 5	The Java Persistence API (JPA)
Chapter 6	Context Dependency Injection
Chapter 7	Enterprise JavaBean Overview
Chapter 8	Java Naming and Directory Interfaces
Chapter 9	Session Beans
Chapter 10	Going Further with JPA
Chapter 11	Configuring Wildfly and JBoss DataSources
Chapter 12	Building Enterprise Applications
Chapter 13	Java EE Security
Chapter 14	Messaging and JMS
Chapter 15	Message Driven Beans

Lab Exercises

©Copyright 17/11/2015 Conygre IT Limited
All rights reserved

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written consent of the publisher.

Conygre IT Limited shall not be liable in any way in respect of any loss, damages, costs, liabilities, or expenses suffered by the user, whether directly or indirectly as a result of the content, format, presentation, or any other aspect of the materials.

Java EE

Introduction to Java Enterprise Edition

- [conygre] -
f i l l t h e g a p

Introduction to Java EE

- Java EE is all about standards
- It is about building enterprise applications which can be deployed in a variety of application servers
 - Oracle WebLogic
 - Oracle GlassFish
 - IBM WebSphere
 - JBoss
- Java EE is based upon proven Java technologies

- [conygre] -
fill the gap

Java Enterprise Edition Versions

- There have been many iterations of JEE over the years

Version	Date	Usage
1.0	1999	Very first enterprise edition. Introduced us to the world of EJBs
1.1	1999	EJB model refined. Serialized classes no longer used for primary keys
1.3	2001	Current mainstream version. Introduced entity relationships for entity beans, and message driven EJBs among other things
1.4	2003	Added support for Web services.
5	2006	Major simplification of EJBs to use annotations, JSF web framework included
6	2009	CDI introduced for dependency injection, JAX-RS for rest services
7	2013	Native JSON support, WebSockets API, new JMS API

- [conygre] -
fill the gap

The J2EE platform has also undergone a number of revisions.

1.0 was the first version that introduced EJBs to the Java developer. These were refined in J2EE 1.1 to remove the need for serialization of primary keys.

There was no J2EE 1.2, and in J2EE 1.3, a new type of EJB called message driven beans was introduced, along with support for entity relationships between entity beans.

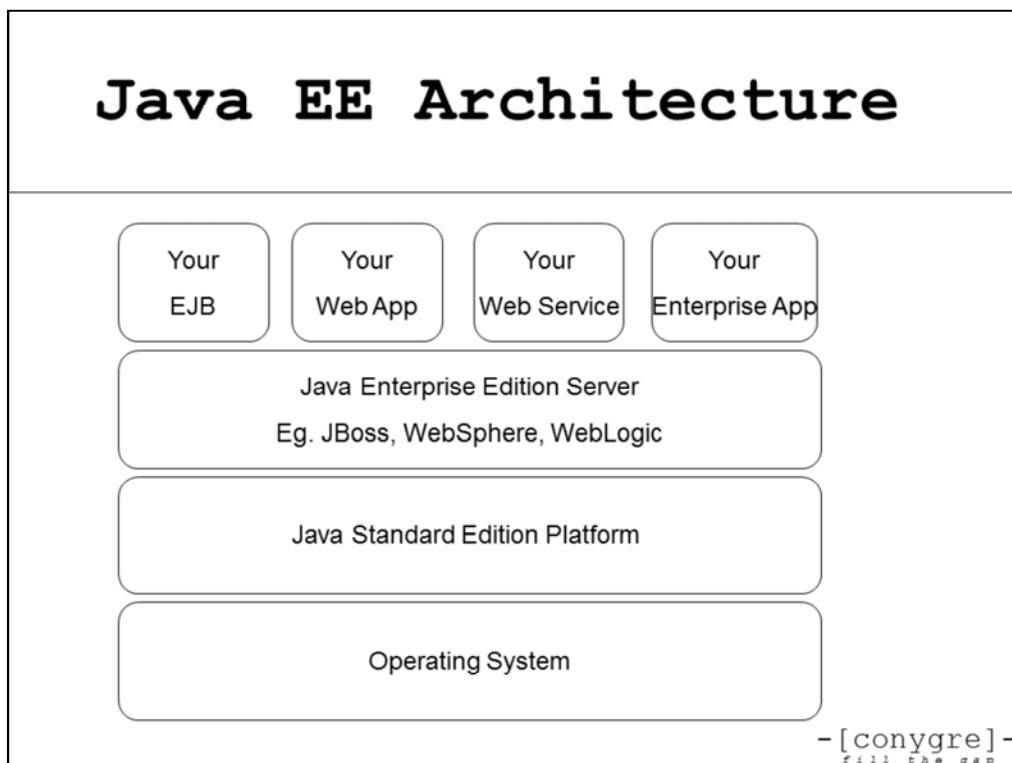
In J2EE 1.4, Web service support was integrated into the platform, in the form of JAX-RPC, SAAJ, and JAXR APIs, as well as support for stateless session beans being exposed as Web services.

Since then, as the enterprise landscape has changed with the emergence of dependency injection, the need for unit testing, REST APIs and HTML5, the platform has continued to evolve and accommodate these changes.

Java EE Java Technologies

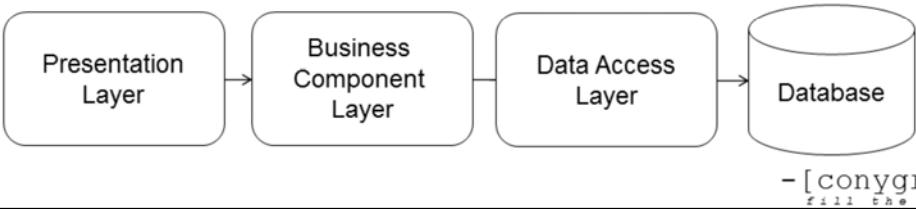
- Below is a list of just some of the Java EE technology standards
 - Enterprise JavaBeans
 - JavaServer Pages
 - Java Servlets
 - Java Messaging Services
 - Java Transaction Services
 - Java Naming and Directory Interface
 - Context Dependency Injection (CDI)
 - Java Persistence API

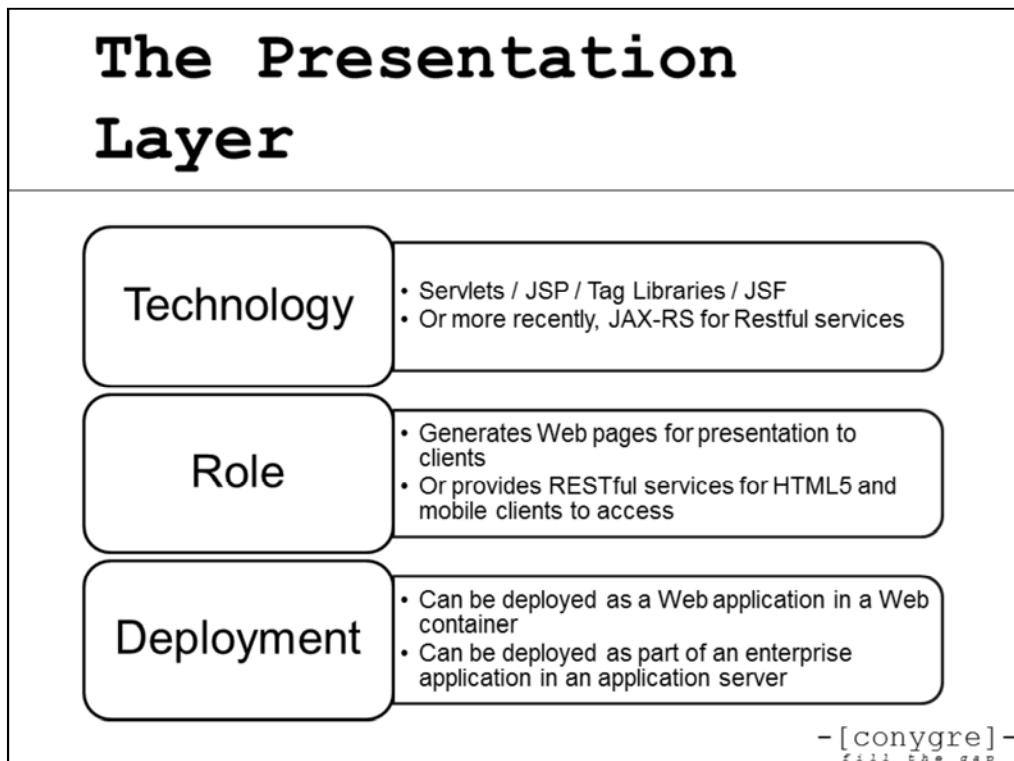
- [conygre] -
fill the gap

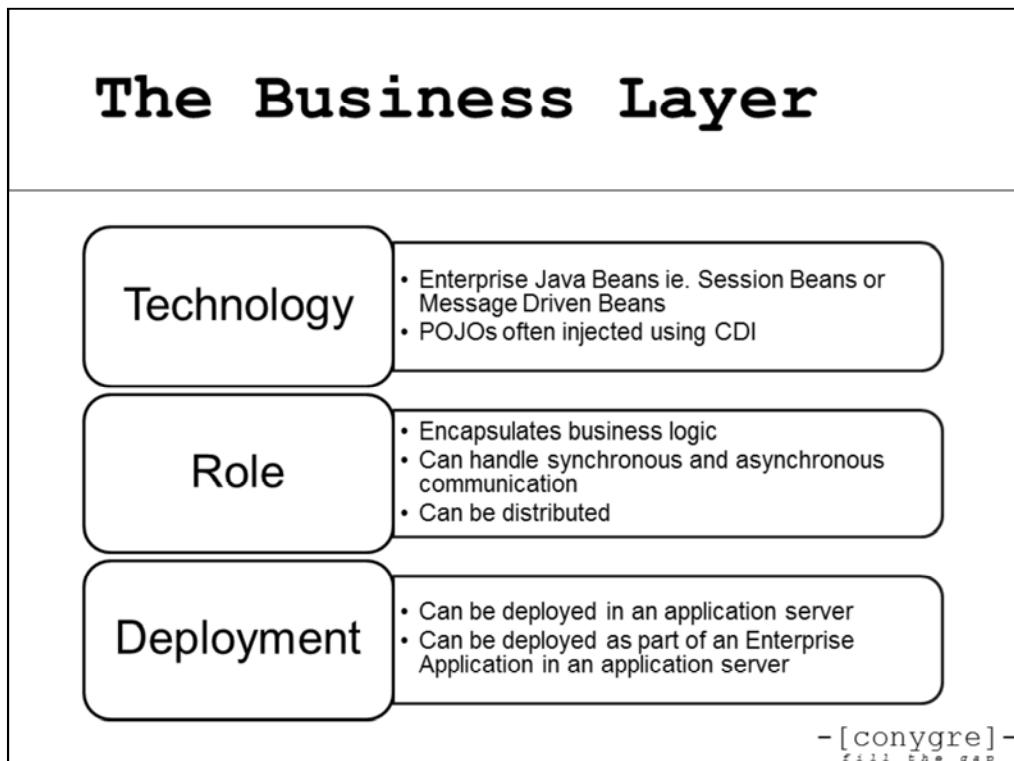


Java EE Application Layers

- Java EE Applications are typically layered
 - Presentation Layer uses Java Web technology and JavaBeans often accessed from HTML5 using REST
 - Business Component layer can use EJBs or POJOs and focuses on the business logic involved in processing that data
 - Data access layer –which interfaces to the data, typically in a database using JPA as the technology

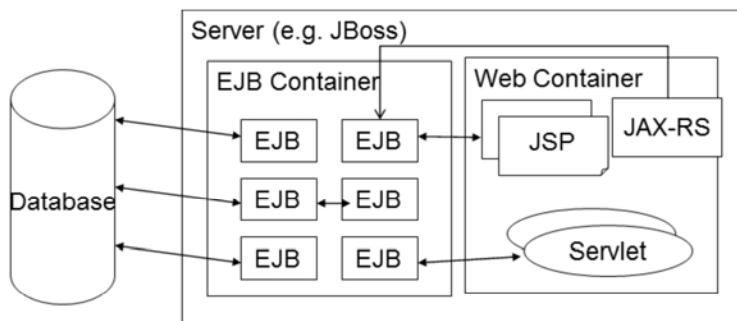






Java EE Technology

- Application servers have **containers** hosting different parts of your applications



- [conygre] -
fill the gap

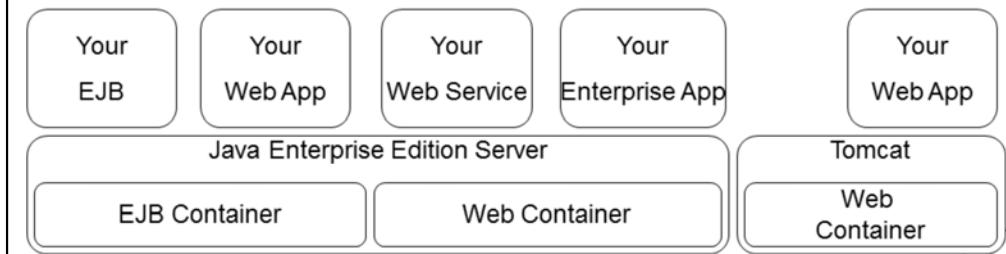
Java EE Container Role

- According to the Java EE specification, containers should provide the following runtime services
 - Transaction Management
 - Persistence Management
 - Thread management
- This frees up the application developer to focus on application specific logic

- [conygre] -
fill the gap

EJB and Web Containers

- Applications using EJBs and Web technologies use both the Web and EJB container
- Many applications do not use EJBs so only need a Web container
 - Tomcat is an example of a ‘Web container only’ server



Tomcat



- Tomcat is an example of a web container
- Tomcat is free!
- Tomcat is available from
 - <http://tomcat.apache.org/>
- Tomcat is also the reference implementation for the Servlet and JSP specifications

- [conygre] -
fill the gap

Summary

- Java EE is a collection of standards
- Java EE standards are implemented by an application server
- A Java EE Application can consist of multiple tiers
 - Web tier in a web container
 - Business tier in an EJB Container

- [conygre] -
fill the gap

Introduction to JBoss AS

Introduction to WildFly

- [conygre] -
e l l l t h e o g a p

Objectives

- Introduction to the WildFly Application Server
- Demonstrate how to deploy and configure different kinds of Java EE applications for WildFly

- [conygre] -
fill the gap

Introducing WildFly

- According to the Web site
 - “JBoss, a division of Red Hat, is the leader in enterprise-class open source middleware. JBoss Enterprise Middleware is comprised of certified, supported platform and framework distributions that are based on JBoss.org Community projects.”
- WildFly is the new name for the JBoss Application Server product

- [conygre] -
fill the gap

JBoss Products

- **JBoss Enterprise Application Platform**
 - The Java EE application server
- **JBoss Enterprise Portal Platform**
 - Portals server platform
- **MetaMatrix Enterprise Data Services Platform**
 - Data transfer and transformation platform
- **JBoss Enterprise SOA Platform**
 - A flexible, standards-based platform to integrate applications, SOA services, and business events as well as to automate business processes.

- [conygre] -
fill the gap

WildFly History

- When the application server market was in its infancy, vendors typically charged for server licenses
- The JBoss EJB container was released in 1999
- The full J2EE server came out around 2000 and was free!
 - Caused a bit of a stir!
- JBoss is now well established as one of the mainstream servers

- [conygre] -
fill the gap

WildFly Facts

- It's free to download, build, and deploy applications on
 - Alternatively there is a paid for edition that comes with support
- Standards based
- Around 25% market share

- [conygre] -
fill the gap

Installing WildFly

- The installation is very straightforward
 1. Download the ZIP for the version that you want and extract it to where you prefer
 - <http://wildfly.org/downloads/>
 2. Ensure JDK is installed and JAVA_HOME is set

- [conygre] -
fill the gap

The WildFly Folders

- The WildFly folder structure is as follows

Name
.installation
appclient
bin
docs
domain
modules
standalone
welcome-content
copyright.txt
jboss-modules.jar
LICENSE.txt
README.txt

- [conygre] -
fill the gap

The bin folder

- The bin folder contains the various batch files and shell scripts to start and stop and create new servers
- To launch the server, use
 - **standalone.bat**
 - **standalone.sh**
- It can also be set up as a service if desired

- [conygre] -
fill the gap

A Running Server

- Below is the WildFly running server console



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains a log of server startup messages from WildFly. Key log entries include:

- INFO [MSC service thread 1-4] JBAS020522: Starting JBoss Web Services - Stack CXF Server 4.2.4.Final
- INFO [MSC service thread 1-4] JBAS010404: Deploying non-JDBC-compliant driver class com.mysql.jdbc.Driver (version 5.1)
- INFO [MSC service thread 1-4] JBAS010404: Deploying non-JDBC-compliant driver class com.mysql.fabric.jdbc.FabricMySQLDriver (version 5.1)
- INFO [MSC service thread 1-4] JBAS010417: Started Driver service with driver-name = mysql-connector-java-5.1.32-bin.jar.com.mysql.fabric.jdbc.FabricMySQLDriver_5_1
- INFO [MSC service thread 1-4] JBAS010417: Started Driver service with driver-name = mysql-connector-jar_5_1_32-bin.jar
- INFO [MSC service thread 1-4] JBAS010417: Started Driver service with driver-name = mysql-connector-java-5.1.32-bin.jar
- INFO [ServerService Thread Pool -- 28] JBAS0105961: Http management interface listening on http://127.0.0.1:9991/management
- INFO [Controller Boot Thread] JBAS015951: Admin console listening on http://127.0.0.1:9991
- INFO [Controller Boot Thread] JBAS015874: WildFly 8.1.0.Final "Wenny" started in 4399ms - Started 219 of 275 services <99 services are lazy, passive or on-demand>

- [Conygre] -
fill the gap

The bin\client Folder

- The client folder contains the JAR required by client applications
- The CLI client jar is for the command line interface

Name
jboss-cli-client.jar
jboss-client.jar
README-CLI-JCONSOLE.txt
README-EJB-JMS.txt

- [conygre] -
fill the gap

The docs Folder

- Doesn't contain documentation
- It contains various XML configuration elements that you will use as templates for your configuration
 - Data source configurations
 - JMS destination configurations
- It also contains the XML Schemas for the various configuration elements that you might wish to use
 - Some useful documentation can be found in these schemas

- [conygre] -
fill the gap

Domain / Standalone

- WildFly can be configured to run in two different modes
 - Domain Mode
 - Domain mode is where multiple servers are managed together through a domain server, allowing configuration and deployments to be done separately
 - Standalone Mode
 - Standalone mode is where the servers are managed individually and all deployment and configuration is done on a server by server basis
- Note that in either mode, the servers can be clustered

- [conygre] -
fill the gap

Configuration

- For standalone mode, the configuration is largely done in
 - standalone/configuration/standalone.xml
- For domain mode, the configuration is mostly found in
 - domain/configuration/domain.xml

- [conygre] -
fill the gap

Multiple Configurations

- The standalone/configurations folder contains all the server configurations
- Each configuration has different services enabled by default
 - standalone.xml
 - Standard configuration with all commonly used JEE services
 - standalone-full.xml
 - A complete configuration with all JEE services enabled
 - standalone-full-ha.xml
 - A complete configuration that also includes clustering
- You can also create your own configurations

- [conygre] -
fill the gap

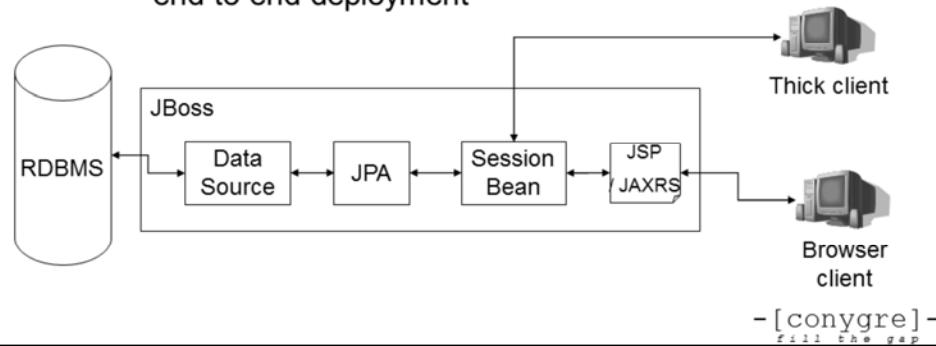
Deployments

- Deployments can be done in one of three ways
 1. Place deployable items into standalone/deployments
 2. Deploy through the command line interface (CLI)
 3. Deploy through the web console
- Items deployed through the CLI or Web console do not end up in standalone/deployments!
 - They end up in standalone/data/content

- [conygre] -
fill the gap

Deploying a WildFly Application

- Application Architecture
 - Example is not meant to emphasise best architectural practice, but rather demonstrate an end to end deployment



Deployment Demonstration

1. Build and Deploy a Web application
2. Build and deploy an EJB (EJB3)
3. Invoke an EJB from a remote client
4. Build and deploy an end to end application using JPA / Session Beans / Web app
5. Build and deploy MDBs and send messages from remote client
6. Build and deploy a Web service, and test from a Web service client

- [conygre] -
fill the gap

Summary

- Introduction to the WildFly Application Server
- Demonstrate how to deploy and configure different kinds of Java EE applications for WildFly

- [conygre] -
fill the gap

RESTful Web Services

RESTful Web Services

- [conygre] -
f i l l t h e g a p

Objectives

- Introducing REST
- The principles of REST
- Implementing a REST Web service

- [conygre] -
fill the gap

Introducing REST

- REST stands for **Representational State Transfer**
- REST is a style of architecture
 - The Web is a large set of resources
 - As a client selects a resource, they are then in a particular state
 - The client can then select a different resource, and be in a different state as they now have a different resource
- For example, drilling into eBay to find a specific item

- [conygre] -
fill the gap

Identifying Resources

- One key concept in REST is the concept of a resource
- A resource always has a simple URL
 - <http://www.conygre.com/courseList>
- At the end of the URL will be URLs for other resources allowing you to get more information (and therefore change state)

- [conygre] -
fill the gap

Resource Example

- The courseList resource could be an XML document, containing links to further information for each individual course

```
<courses xmlns="http://www.conygre.com/courses"  
        xmlns:xlink="http://www.w3.org/1999/xlink">  
    <course id="XMLOV" xlink:href="http://www.conygre.com/site/xml/xmloverview"/>  
    <course id="J2EEJBoss" xlink:href="http://www.conygre.com/site/xml/j2eeJBoss"/>  
    <course id="RUP" xlink:href="http://www.conygre.com/site/xml/rup"/>  
    <course id="XSLT" xlink:href="http://www.conygre.com/site/xml/xslt"/>  
</courses>
```

http://www.conygre.com/courseList

- [conygre] -
fill the gap

REST URLs

- Once important principle in REST is that the URLs are logical names for resources, not physical locations
- The actual location should be resolved by technology on the server
 - Such as a Java Servlet or ASP.NET routing rule

- [conygre] -
fill the gap

The REST Principles

- There is a request and a response
- Resources are all named through URLs
- Resources provide links to other resources
- There is a standard interface into the application using one of the HTTP methods

HTTP Method	Role in REST
GET	Retrieve data
POST	Edit data
PUT	Add data
DELETE	Remove data

- [conygre] -
fill the gap

REST and Web Services

- Web services that use REST tend to have the following characteristics
 - Clients send HTTP requests using the four HTTP methods
 - Simple parameters can be passed in the URL
 - Complex parameters are passed in the header as XML or JSON
- There is no need for
 - SOAP wrappers from the client requests
 - WSDL definitions of operations from the server

- [conygre] -
fill the gap

Designing a RESTful API

- APIs can be developed using REST
- For example
 - GET www.meals.co.uk/restaurants/bristol
 - Get a list of Bristol restaurants
 - GET www.meals.co.uk/restaurants/bristol/BurgerJoint
 - Get details for the Burger Joint Restaurant
 - GET www.meals.co.uk/orders/454
 - Get the details for order number 454
 - DELETE www.meals.co.uk/orders/454
 - Remove order number 454
 - PUT www.meals.co.uk/orders
 - (provide a JSON food order in the request header)
 - Create a new order

- [conygre] -
fill the gap

Security with REST

- With the URLs for RESTful APIs very simple and even guessable, it can be important to secure the API
- Approaches can include
 - Providing a user token with each request
 - Enforcing clients to provide an authorisation code signed with a client private key
 - Enforcing clients to timestamp every request
 - Servers can limit IP ranges for client access

- [conygre] -
fill the gap

Benefits of REST

- REST based services are far simpler to implement
- REST based services are easier to invoke than SOAP based services
- REST based services are ideal for simple request/response Web services

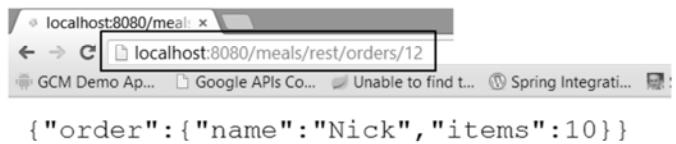
- [conygre] -
fill the gap

Example

- Note the URLs in these two examples



```
▼<order>
  <name>Nick</name>
  <items>10</items>
</order>
```



- [conygre] -
fill the gap

Summary

- Introducing REST
- The principles of REST
- Implementing a REST Web service

- [conygre] -
fill the gap

Creating RESTful Services

- [conygre] -
e l l l t h e o a p

Objectives

- JAX-RS and REST
- Setting up a Web application with Jersey
- Creating JAX-RS classes
- The JAX-RS Annotations
- An Example CRUD Class
- Building a RESTful client using JAX-RS

- [conygre] -
fill the gap

Java API for Restful Services JAX-RS

- JAX RS is the Java standard for implementing Restful services
- JAX RS is based on JSR 311
- The fundamentals are quite straightforward
 - Java methods are mapped to URLs
 - Return types are converted into JSON or XML by the implementation
- The reference implementation of JAX RS is called **Jersey**

- [conygre] -
fill the gap

What do you need to do?

1. Create a Java Web application
2. Add the Jersey libraries
3. Within web.xml configure the Jersey servlet dispatcher to handle HTTP requests
4. Within web.xml specify which packages you will be using for your RESTful classes
5. Create an annotated class to handle the REST requests

- [conygre] -
fill the gap

Creating a Web Application

- The Java Web application is a standard Web app as specified by JEE
- The libraries for Jersey can be added manually or through Maven

```
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-server</artifactId>
  <version>1.13</version>
</dependency>
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-servlet</artifactId>
  <version>1.13</version>
</dependency>
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-json</artifactId>
  <version>1.13</version>
</dependency>
```

- [Conygre] -
fill the gap

The **jersey-server** library is the core jersey library, and then the **jersey-servlet** library provides the entry point servlet for your application, and then finally **jersey-json** is needed if you are working with JSON. If you are only using XML then you will not require the JSON library.

The web.xml Entry

- The web.xml deployment descriptor requires the Jersey servlet to be configured as an endpoint for REST requests

```
<web-app ...>
<servlet>
<servlet-name>Jersey</servlet-name>
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
....
</servlet>
<servlet-mapping>
<servlet-name>Jersey</servlet-name>
<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

- The example maps all requests **/rest/*** to Jersey

- [conygre] -
fill the gap

Configuring Your Packages

- You will also need to configure the Jersey servlet with your packages containing the RESTful classes
 - This is done through a servlet **init-param**

```
<web-app ...>
<servlet>
<servlet-name>Jersey</servlet-name>
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
<init-param>
  <param-name>com.sun.jersey.config.property.packages</param-name>
  <param-value>com.conygre.rest.service</param-value>
</init-param>
</servlet>
...
</web-app>
```

- [conygre] -
fill the gap

The RESTful Class

- The RESTful class is created using annotations which control which method is triggered by which HTTP request

```
package com.conygre.rest.service;  
@Path("/hello") // path will mean URL will be /rest/hello  
public class Hello {  
    @GET // all get requests to /rest/hello end up here  
    public String sayHello() {  
        return "Hello Jersey";  
    }  
}
```

- [conygre] -
fill the gap

Read Operations

- Read operations are handled by GET requests
- For example, two overloaded operations can be used to list values or retrieve just one value based on an id

```
@GET  
public String sayHello() {  
    return "hello world";  
}  
@GET  
@Path("/{name}") // name is a parameter accessed by @PathParam  
public String sayHello(@PathParam("name") String name) {  
    return "hello " + name;  
}
```

- [conygre] -
fill the gap

The URL:

<http://localhost:8080/appname/rest/hello> would return hello world

Whereas the URL

<http://localhost:8080/appname/rest/hello/nick> would return hello nick

The Java Persistence API Annotations

JAX-RS Annotations

@PATH(some_path)	The URL pattern that triggers the methods in the class.
@POST	Method is for POST requests
@GET	Method is for GET requests
@PUT	Method is for PUT requests
@DELETE	Method is for DELETE requests
@Produces (MediaType.APPLICATION_JSON)	Specifies the media type of the returned value. Typically JSON or XML
@Consumes (MediaType.APPLICATION_XML)	Specifies the media type to be consumed. Typically JSON or XML
@PathParam	Used to inject values from the URL into a method parameter. This way you inject, for example, the ID of a resource into the method to get the correct object.

- [conygre] -
fill the gap

A CRUD Example

- A more realistic example would be a class to perform simple Create Read Update Delete (CRUD) operations
- To keep it simple, the example will use a simple List of values to complete the CRUD operations
- The example will also demonstrate the use of the various annotations

- [conygre] -
fill the gap

The Rest Class

```
public class CompactDiscCRUDService {  
    private static Map<Integer, CompactDisc> library;  
    static {  
        library = new HashMap<Integer, CompactDisc>();  
        library.put(1,new CompactDisc("Gold", 12.99, "Abba", 12,1));  
        ..  
    }  
    public Response getCompactDiscs(){ ..}  
    public CompactDisc getCompactDisc(@PathParam("id") int id) { ..}  
    public void addCompactDisc(CompactDisc disc){ ..}  
    public void updateCompactDisc(CompactDisc updatedDisc){ ..}  
    public void deleteCompactDisc(CompactDisc discToDelete){ ..}  
}
```

- [conygre] -
fill the gap

Working With Collections

- There are three ways to return a collection
 1. Return a Collection<YourType>
 2. Use the JAX-RS GenericEntity type
 3. Create your own holder class for the list
- The benefit of using a holder class particularly is that you can name the collection

- [conygre] -
fill the gap

A Read Example Using A Collection

- The original collection can be returned as is

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public Collection<CompactDisc> getCompactDiscs(){  
    return library.values();  
}
```

<http://localhost:8080/rest/compactdiscs>

```
[{"id":1,"title":"Gold","artist":"AbbaTribute","tracks":12,"price":12},..]
```

- [conygre] -
fill the gap

A Read Example Using GenericEntity

- The original collection can be wrapped in a **GenericEntity**
 - Some implementations now name the collection in the JSON based on the Java type

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public Response getCompactDiscs(){  
    GenericEntity entity = new  
        GenericEntity<Collection<CompactDisc>>(library.values()) {};  
    return Response.ok(entity).build();  
}
```

```
http://localhost:8080/rest/compactdiscs
```

```
{compactDisc: [{"id":1,"title":"Gold","artist":"AbbaTribute","tracks":12,"price":12},...]}
```

- [conygre] -
fill the gap

GenericEntity

- The **GenericEntity** class is part of JAX-RS that allows the generic typing information to be preserved to the JAX-RS runtime
- The **Response.ok()** static method returns a **ResponseBuilder** for the GenericEntity
- The ResponseBuilder can then be used to build() the response

```
Response.ok(entity).build()
```

- [conygre] -
fill the gap

Read using a Holder class

- To control the name of the collection, use a Holder class

```
@XmlRootElement  
public class CompactDiscList implements Serializable {  
  
    public Collection<CompactDisc> getDiscCollection() {  
        return discCollection;  
    }  
    public void setDiscCollection(Collection<CompactDisc> discCollection) {  
        this.discCollection = discCollection;  
    }  
    private Collection<CompactDisc> discCollection;  
}
```

```
{discCollection:  
[{"id":1,"title":"Gold","  
artist":"AbbaTribute",  
"tracks":12,"price":1  
2},...]}
```

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public CompactDiscList getCompactDiscs() {  
    CompactDiscList discs = new CompactDiscList();  
    discs.setDiscCollection(library.values());  
    return discs;  
}
```

- [conygre] -
fill the gap

@XmlRootElement

- Any type that is to be returned to a client will need to be marshalled into JSON/XML
- This is enabled by annotating the class using
 - **@XmlRootElement**

```
@XmlElement // now this type can be passed as JSON
public class CompactDiscList implements Serializable {

    public Collection<CompactDisc> getDiscCollection() {
        return discCollection;
    }
    public void setDiscCollection(Collection<CompactDisc> discCollection) {
        this.discCollection = discCollection;
    }
    private Collection<CompactDisc> discCollection;
}
```

- [conygre] -
fill the gap

Controlling the JSON

- It is also possible to control JSON entry names using **@XmlElement**

```
@XmlRootElement  
public class CompactDisc implements Serializable {  
    private int id;  
  
    @XmlElement(name="AlbumTitle")  
    private String title;  
    private String artist;  
  
    // getters and setters  
}
```

```
{  
    "id":1,  
    "AlbumTitle":"Gold",  
    "artist":"AbbaTribute"  
}
```

- [conygre] -
fill the gap

Reading a Single Item

- Reading a single item is far simpler

```
@GET  
@Path("/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public CompactDisc getCompactDisc(@PathParam("id") int id) {  
    return library.get(id);  
}
```

- Following RESTful principles, we now provide the id on the end of the URL

```
http://localhost:8080/rest/compactdiscs/1
```

- [conygre] -
fill the gap

Create Operations

- Create operations would be done as a POST method

```
@POST  
@Consumes(MediaType.APPLICATION_JSON)  
public void addCompactDisc(CompactDisc disc) {  
    library.put(disc.getId(), disc);  
}
```

- In the above example, the input type is specified as JSON

- [conygre] -
fill the gap

Update Operations

- Update operations would be done as a PUT request, it is virtually the same as the Create operation

```
@PUT  
@Consumes(MediaType.APPLICATION_JSON)  
public void updateCompactDisc(CompactDisc updatedDisc){  
    library.put(updatedDisc.getId(), updatedDisc);  
}
```

- [conygre] -
fill the gap

Delete Operations

- Delete operations would be done as a DELETE request
- The id parameter is the id of the item to be removed

```
@DELETE  
@Path("/{id}")  
public void deleteCompactDisc(@PathParam("id") int id) {  
    library.remove(id);  
}
```

- [conygre] -
fill the gap

JAX-RS Clients

- The JAX-RS API can also be used to create Java clients to any REST endpoint
- The main object you work with is an instance of **WebTarget**
- To create a WebTarget you will need the following:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8080/rest/compactdiscs");
```

- [conygre] -
fill the gap

Making GET Requests

- A GET request for a list of CompactDiscs can be created as follows

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8080/rest/compactdiscs");
// initiate the request
Response response =
    target.request(MediaType.APPLICATION_JSON_TYPE).get();
// read the CompactDiscList entity out of the response
CompactDiscList list = response.readEntity(new GenericType<CompactDiscList>() {});
```

- A request for a single disc would be very similar

- [conygre] -
fill the gap

Making POST Requests

- A POST request can be completed as follows

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8080/rest/compactdiscs");

CompactDisc disc = new CompactDisc("Another Brick in the Wall", 8.99,
                                  "Pink Floyd", 12, 6);
target.request(MediaType.APPLICATION_JSON_TYPE).post(Entity.json(disc));
```

- The Entity class has a number of static methods to facilitate the conversion of Entity classes into RESTful formats such as JSON

- [conygre] -
fill the gap

Making Put Requests

- Put requests look very similar to POST requests from the client perspective

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8080/rest/compactdiscs");

target.request(MediaType.APPLICATION_JSON_TYPE).put(Entity.json(someModifiedDiscObject));
```

- [conygre] -
fill the gap

Making Delete Requests

- Making a delete request can be done as follows

```
Client client = ClientBuilder.newClient();
WebTarget deleteTarget = client
    .target("http://localhost:8080/rest/compactdiscs/1");
deleteTarget.request().delete();
```

- [conygre] -
fill the gap

Summary

- JAX-RS and REST
- Setting up a Web application with Jersey
- Creating JAX-RS classes
- The JAX-RS Annotations
- An Example CRUD Class
- Building a RESTful client using JAX-RS

- [conygre] -
fill the gap

Introducing the Java Persistence API

- [conygre] -
e l l l t h e o a p

Objectives

- Introducing the Java Persistence API (**JPA**)
- Configuring Persistence
- Working with annotations
- An example Persistence application
- Persisting and extracting data

- [conygre] -
fill the gap

Introducing the Persistence API

- The JPA is an **object relational mapping** API that makes it easier to
 - Extract data from relational databases and put into objects
 - Take data from objects and put into a database
- The JPA can be used in both standard and enterprise edition Java
 - So you don't need an application server to use it!

- [conygre] -
fill the gap

Key benefits of JPA

- JPA allows POJOs (Plain Old Java Objects) to be mapped to Relational data
- Fine grained object models supported
- Transparent persistence
- Simple Java Persistence query language for database independent queries (JP-QL)
- Intelligent caching of data

- [conygre] -
fill the gap

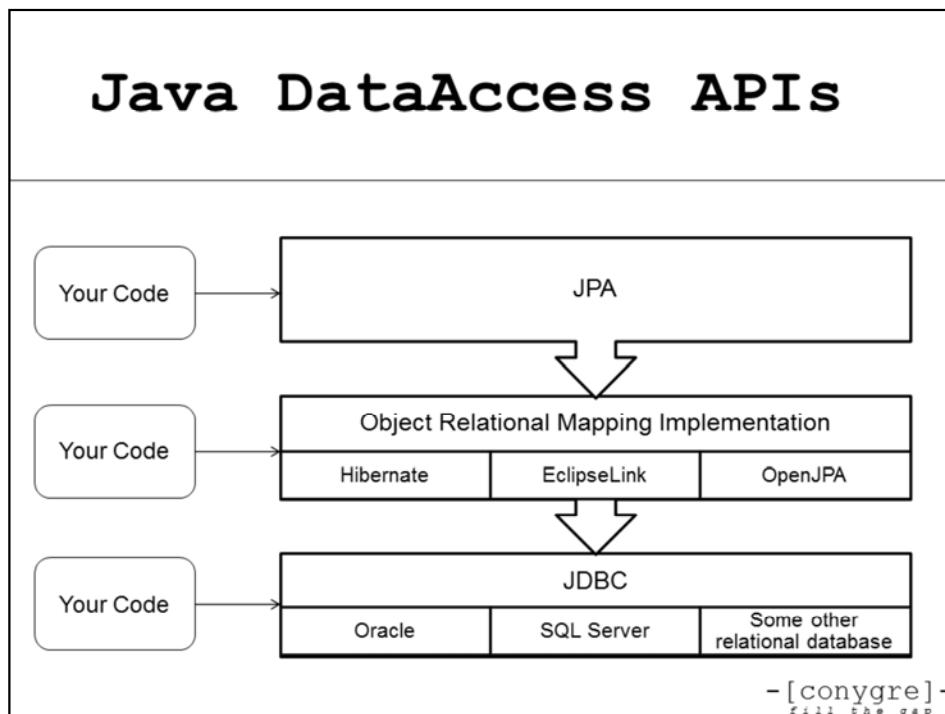
Relationship to alternative ORM Tools

- JPA is a standardisation of a number of ORM tools and APIs already available
 - JBoss Hibernate
 - EclipseLink (formerly Oracle TopLink)
 - OpenJPA

- [conygre] -
fill the gap

A growing number of JPA implementations exist. By far the most popular is Hibernate from the JBoss organisation. TopLink (now EclipseLink) from Oracle is heavily promoted by Oracle through the Netbeans platform. Then there is also OpenJPA.

The Java Persistence API Annotations



When writing applications that access databases you can use JDBC, you can use an ORM implementation such as Hibernate, now you can use the JPA API that sits on top of an ORM implementation.

JPA with Hibernate

- When using Hibernate with JPA you will need to add the following libraries to your project

Hibernate Core	This is the foundational Hibernate download. It contains the original Hibernate implementation.
Hibernate Annotations	This adds support for annotated classes being used instead of the mapping file.
Hibernate Entity Manager	The Entity Manager is a wrapper for the core that provides support for all the features of the JPA.
SLF4J	Simple Logging Façade for Java. Used as an API on top of Log4J to enable logging.

- [conygre] -
fill the gap

The libraries listed are the ones required within the project. They can be added to a project through dependencies using Maven.

Example

- The example application will use the Hibernate Core and annotations to achieve persistence
- The application will then be extended to use the Entity Manager

- [conygre] -
fill the gap

JPA Example Mapping

- A compact_discs database table will be used as follows

id	title	artist	price	tracks

```
create table compact_discs (
    id int primary key auto_increment,
    title varchar (50),
    artist varchar(30),
    tracks int,
    price double);
```

- [Conygre] -
fill the gap

The SQL above is used to create a basic table in a MySQL database. The auto_increment specifies that the primary key is to be set by the database as new rows are added.

ORM Mapped Classes

- In JPA, the mapped classes are referred to as **entities**
- An entity is a POJO with **annotations** used to specify the mapping information
- Entity objects can be used as value objects and passed through the layers of an application

- [conygre] -
fill the gap

The JPA Entity

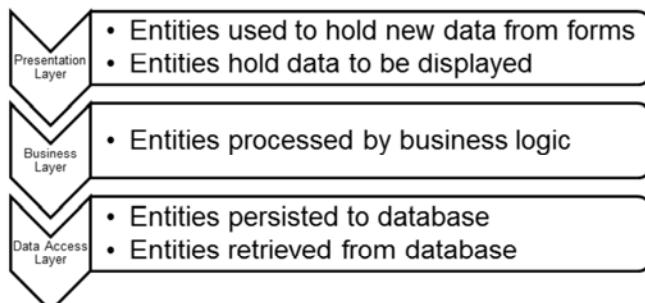
- The CompactDiscProduct bean will be used
- It is a POJO, a regular JavaBean

```
package com.conygre.cd;
public class CompactDisc implements Serializable {
    //Instance variables
    private String title;
    private double price;
    private String artist;
    private int tracks;
    private int id;
    // get and set methods and a no argument constructor
```

- [conygre] -
fill the gap

Entities used in Layers

- Entities can be used across the layers of an application



- [conygre] -
fill the gap

Entity classes can be used throughout the application layers. The data access layer is the part of the application that actually retrieves the data from the database and persists it, but the other layers can use the same instances to complete business logic or to display the data.

So with our CD example, the web interface may request the current catalog of CDs, so it requests that from the business layer, which then requests it from the data access layer. The data access layer will execute the query which will result in the CompactDisc entity objects being created. They will then be returned to the presentation layer, which will display them.

Sometimes however you will need to work with transfer objects that can be used across the layers. These are representations of the entity classes that are structured in a way to make more sense to the business or presentation layer.

The Entity Annotations

- The information mapping the bean to the database is enclosed in the bean through the use of annotations
- Annotations exist to specify characteristics such as
 - Identifying class as an entity class
 - Identifying the database table it maps to
 - Identifying which fields go with which columns
 - Identify the primary key and how it is set for new entries

- [conygre] -
fill the gap

The Entity Class

- The Entity class must be declared as an entity class using the **@Entity** annotation
- The table it maps to is defined using the **@Table** annotation

```
package com.conygre.cd;  
import java.io.Serializable;  
import javax.persistence.*;  
@Entity  
@Table(name="compact_discs")  
public class CompactDiscProduct implements Serializable { ... }
```

- [conygre] -
fill the gap

If the **@Table** annotation is not used, then the table name is assumed to be the same as the class. This will often not be the case however as tables are often named in the plural, and have underscores, such as `compact_discs`, which would be a poor candidate class name, so typically the table name must be provided.

The Entity Properties

- Properties can be defined as mapping to columns in the table using **@Column**
- When mapped in this way, properties are accessed directly using reflection

```
public class CompactDiscProduct implements Serializable {  
    @Column(name="title")  
    private String title;  
  
    public String getTitle() {  
        return title;  
    }  
  
    public void setTitle(String t) {  
        title=t;  
    }  
    ....  
}
```

- [Conygre] -
fill the gap

All the properties of the bean will then be assumed to map to columns with the same name. However, there is `@Column` annotation that can be used to identify the column that the various properties are mapped to.

The properties are accessed directly by reflection, even if they are private.

Primitives

- Primitives can be problematic when used in entity classes
 - What if the price column was nullable?
 - The value of the primitive in the entity could not be null
- To avoid this problem, it is preferred to use the wrapper classes such as Integer and Double rather than the primitive types

- [conygre] -
fill the gap

Mapping Methods

- It is also possible to map the properties to the getX methods instead of the properties

```
public class CompactDiscProduct implements Serializable {  
    private String title;  
    @Column(name="title")  
    public String getTitle() {  
        return title;  
    }  
    public void setTitle(String t) {  
        title=t;  
    }  
    ...  
}
```

- [conygre] -
fill the gap

To avoid the JPA implementation accessing properties directly and bypassing the get/set methods which may have code in them, you can annotate the get methods instead.

Primary Keys

- The primary key column must be identified using an additional **@Id** annotation
- The way the key is generated can also be specified using **@GeneratedValue**

```
@Id  
@GeneratedValue(strategy=GenerationType.AUTO)  
@Column(name="id")  
private Integer id;
```

- [conygre] -
fill the gap

The GeneratedValue annotation has the following syntax:

```
@GeneratedValue(  
    strategy=GenerationType.[TABLE|SEQUENCE|IDENTITY|AUTO],  
    generator="GEN_OR_SEQ_NAME" )
```

As you can see, there are four different GenerationTypes. If you use Sequence or Identity, then you must specify the name of the sequence or identity column for the key. The default is AUTO.

The precise behaviour of these attributes is determined by the JPA implementation.

The Persistent Unit

- The Entity classes do not contain all the information relating to the database, for example
 - The DataSource information
 - The implementation being used (such as Hibernate)
 - The type of database being used
- This additional information is configured in something called a **persistent unit**

- [conygre] -
fill the gap

Persistent Unit Configuration

- Persistent Units are configured in an XML file **persistence.xml** which is always placed in a folder called META-INF

- [conygre] -
fill the gap

persistence.xml

- The root element is persistence, and the namespace for the XML is
 - <http://java.sun.com/xml/ns/persistence>

```
<persistence 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" version="1.0">
</persistence>
```

- [conygre] -
fill the gap

persistence.xml

- persistence.xml defines
 - A name for the persistence unit
 - The implementation provider
 - The DataSource being used
 - Configured directly in the XML
 - Configured through a JNDI lookup
 - Configured as a Spring bean
 - Properties, including the SQL dialect to use

- [conygre] -
fill the gap

Complete persistence.xml

- The XML document below is a complete example of a persistence.xml document
- Note the datasource is specified through JNDI

```
<persistence>
  <persistence-unit name="compactDiscPersistentUnit">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/mySQL</jta-data-source>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

META-INF\persistence.xml

- [conygre] -
fill the gap

Defining the DataSource in XML

- The DataSource can be defined within persistence.xml through additional properties

```
<property name="hibernate.connection.driver.class" value="com.mysql.jdbc.Driver"/>
<property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/conygre"/>
<property name="hibernate.connection.username" value="root"/>
<property name="hibernate.connection.password" value="c0nygre"/>
```

- [conygre] -
fill the gap

For standalone Java applications not running in an application server, it is also possible to simply configure the datasource in the persistence.xml file. If you are using Spring, you can set the datasource as a Spring bean.

Relationships

- Relationships can also be annotated

```
public class CompactDiscProduct {  
    ...  
    @OneToMany  
    @JoinColumn(name="cd_id", referencedColumnName="id")  
    private Set<Track> trackTitles = new HashSet<Track>();  
}
```

CompactDiscProduct.java

```
public class Track {  
    ...  
    private String title;  
    private Integer id;  
}
```

Track.java

- [conygre] -
fill the gap

Relationship Types

- JPA can handle
 - One to Many (as shown on the previous page)
 - One to One
 - Many to One
 - Many to Many
- These relationships can be one way or two way
 - The CompactDisc knows about the tracks
 - Do the tracks know about the CompactDisc?

- [conygre] -
fill the gap

The modelling for all of these different options is outside the scope of this section, but there are plenty of examples online of all the various permutations.

A JPA Client Application

- The steps involved in a basic client application include
 1. Obtain a **EntityManagerFactory**
 2. Open a new **EntityManager**
 3. Obtain a **Transaction**
 4. Persist new objects / update objects / execute queries
 5. Commit the Transaction
 6. Close the EntityManager

- [conygre] -
fill the gap

Obtain an EntityManager

- First, create and configure an **EntityManagerFactory** object
- The String passed into the Persistence static method is the name of the persistent unit

```
EntityManagerFactory factory =  
Persistence.createEntityManagerFactory("compactDiscPersistentUnit");
```

- Now create an EntityManager

```
EntityManager em = factory.createEntityManager();
```

- [conygre] -
fill the gap

The EntityManagerFactory is used to create the EntityManager for you. The factory is obtained from the Persistence class. Once the factory is obtained, you can then get the EntityManager object.

The Transaction

- Once an EntityManager has been obtained, you can then obtain a transaction from the manager

```
EntityTransaction tx = em.getTransaction();
tx.begin();
// Add all your code to work with the database
tx.commit();
em.close();
```

- Once a transaction has begun, you can begin to work with the data

- [conygre] -
fill the gap

Persisting Data

- To Persist data
 - Create an instance of the object to be persisted
 - A POJO
 - Call persist() on the EntityManager

```
CompactDisc product =  
    new CompactDisc ("Bat out of Hell", 12.99, "MeatLoaf", 12);  
em.persist(product);
```

- [conygre] -
fill the gap

Persisting data in Hibernate is done by calling persist(object) on the EntityManager object. This will persist the data in the database.

Extracting Data

- To extract data you can run queries written in **Java Persistence Query Language (JPQL)**
- Queries can be created on an EntityManager object, and then executed

```
Query query = em.createQuery("from CompactDisc");
java.util.List<CompactDisc> cds = query.getResultList();
for (CompactDisc disc : cds) {
    System.out.println("The CD is " + disc.getTitle());
}
```

- [conygre] -
fill the gap

Summary

- Introducing the Java Persistence API (**JPA**)
- Configuring Persistence
- Working with annotations
- An example Persistence application
- Persisting and extracting data

- [conygre] -
fill the gap

Introduction to CDI

Objectives

- What is Context Dependency Injection (CDI)
- Why use CDI
- Creating Beans Managed by a Context
- Basic Managed Bean Example

- [conygre] -
fill the gap

What is CDI

- CDI is the JEE specification for a **Dependency Injection Container** (DI Container)
- In a nutshell, CDI is the JEE answer to Spring Dependency Injection!
- Spring introduced the concept of Dependency Injection back in the early 2000's and that capability has finally become part of JEE

- [conygre] -
fill the gap

Where did Spring come from

- **Rod Johnson** wrote a book called “Expert One-on-One J2EE Design and Development”
- This book contained some example code
- This example code became an open source project – Spring!



- [conygre] -
fill the gap

The Role of DI

- DI allows classes that depend on implementations of interfaces to have them injected

```
public interface Pet {  
    void stroke();  
}
```

```
public class Dog implements Pet {  
    @Override  
    public void stroke() {  
        System.out.println("dog wag tail");  
    }  
}
```

```
public class Person {  
    @Inject  
    private Pet pet;  
    public Pet getPet() {  
        return pet;  
    }  
    public void setPet(Pet pet) {  
        this.pet = pet;  
    }  
}
```

- [conygre] -
fill the gap

Using @Inject

- The **@Inject** annotation causes the environment to
 - Search for an implementation of the interface
 - If one is found an instance of the implementation will be injected into the object
- **@Inject** can be used on
 - Fields (for field injection)
 - Constructors (for constructor injection)
 - Methods (for parameter injection)

- [conygre] -
fill the gap

Testing CDI in Java Standard Edition

- JBoss provide a DI container called Weld that can be used in standard Java

```
// set up the Weld container
WeldContainer weld = new Weld().initialize();
// retrieve a Person object (which will have the pet injected for us)
Person p = weld.instance().select(Person.class).get();
// the pet will be there as a Dog will have been injected
p.getPet().stroke();
```

- [conygre] -
fill the gap

Configuration

- For CDI to work, you must provide a config file in the classpath META-INF/beans.xml
- The minimum for this file is as follows

```
<beans 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://jboss.org/schema/cdi/beans_1_0.xsd">
</beans>
```

- [conygre] -
fill the gap

Alternative Implementations

- What happens if there is more than one possible type that could be injected?

```
public interface Pet {  
    void stroke();  
}
```

```
public class Person {  
    @Inject  
    private Pet pet;  
    ..  
}
```

```
public class Dog implements Pet {  
    ..  
}
```

```
public class Cat implements Pet {  
    ..  
}
```

- [conygre] -
fill the gap

Using @Alternative

- Alternative options for injection can be marked using @Alternative
- If you wish, the default option can be annotated using @Default (which is the default anyway)

```
@Default  
public class Dog implements Pet{  
    ..  
}
```

```
@Alternative  
public class Cat implements Pet{  
    ..  
}
```

- [conygre] -
fill the gap

Switching to an Alternative

- To switch the application to use the alternative, add an entry in beans.xml

```
<beans 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://jboss.org/schema/cdi/beans_1_0.xsd">
  <alternatives>
    <class>com.conygre.beans.Cat</class>
  </alternatives>
</beans>
```

- Now a Cat object will be injected

- [conygre] -
fill the gap

Creating Custom Qualifiers

- You can also create your own Qualifiers to allow more control over what gets injected
- Say a Customer class wants a Broadcaster injected, but it needs to be a satellite one not terrestrial or cable

```
public class Customer {  
    @Inject  
    @Satellite  
    private Broadcaster broadcaster;
```

- [conygre] -
fill the gap

The Custom Qualifier Annotation

- The Annotation is created as follows
 - It must itself be annotated as a **Qualifier**
 - It has to be retained at runtime so it is available to the JRE

```
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({TYPE, METHOD, FIELD,  
PARAMETER})  
public @interface Satellite {  
}
```

- [conygre] -
fill the gap

Using the Custom Qualifier

- A Broadcaster implementation marked qualified as Satellite will now be injected

```
@Satellite  
public class Sky implements Broadcaster{  
    @Override  
    public void broadcast(){  
        System.out.println("Sky broadcast");  
    }  
}
```

```
public class Customer{  
    @Inject  
    @Satellite  
    private Broadcaster broadcaster;  
}
```

```
@Default  
public class BBC implements Broadcaster{  
    @Override  
    public void broadcast(){  
        System.out.println("BBC broadcast");  
    }  
}
```

- [conygre] -
fill the gap

Events and CDI

- Events can also be created and handled when using CDI
- Working with Events involves three components
 - A class that will be the Event
 - A CDI bean that will fire an event
 - A CDI bean method that can handle the event

- [conygre] -
fill the gap

The Event

- Any bean can be used as the Event itself

```
public class MyMessageEvent {  
    private String message;  
    // getter and setter for message  
}
```

- [conygre] -
fill the gap

Firing an Event

- The class firing the event requires an injected Event object that has the generic type of the class you are using as the event

```
import javax.enterprise.event.Event;
public class MyEventFirer {
    @Inject
    Event<MyMessageEvent> events;

    public void someMethodThatTriggersTheMessageEvent() {
        System.out.println("Create the event and fire it");
        MyMessageEvent message = new MyMessageEvent();
        message.setMessage("hello from the event " + new Date());
        events.fire(message);
    }
}
```

- [conygre] -
fill the gap

Handling an Event

- The object that is going to handle the fired event must then have a method annotated to receive the event object as a parameter
- There must be an instance of this class created by the container

```
public class MyMessageEventHandler {  
    public void handleTheMessage(@Observes MyMessageEvent messageEvent){  
        System.out.println("Got a MyMessageEvent: " + messageEvent.getMessage());  
    }  
}
```

- [conygre] -
fill the gap

Summary

- What is Context Dependency Injection (CDI)
- Why use CDI
- Creating Beans Managed by a Context
- Basic Managed Bean Example

- [conygre] -
fill the gap

Enterprise Java Beans Overview

- [conygre] -
- f i l l t h e o g a p

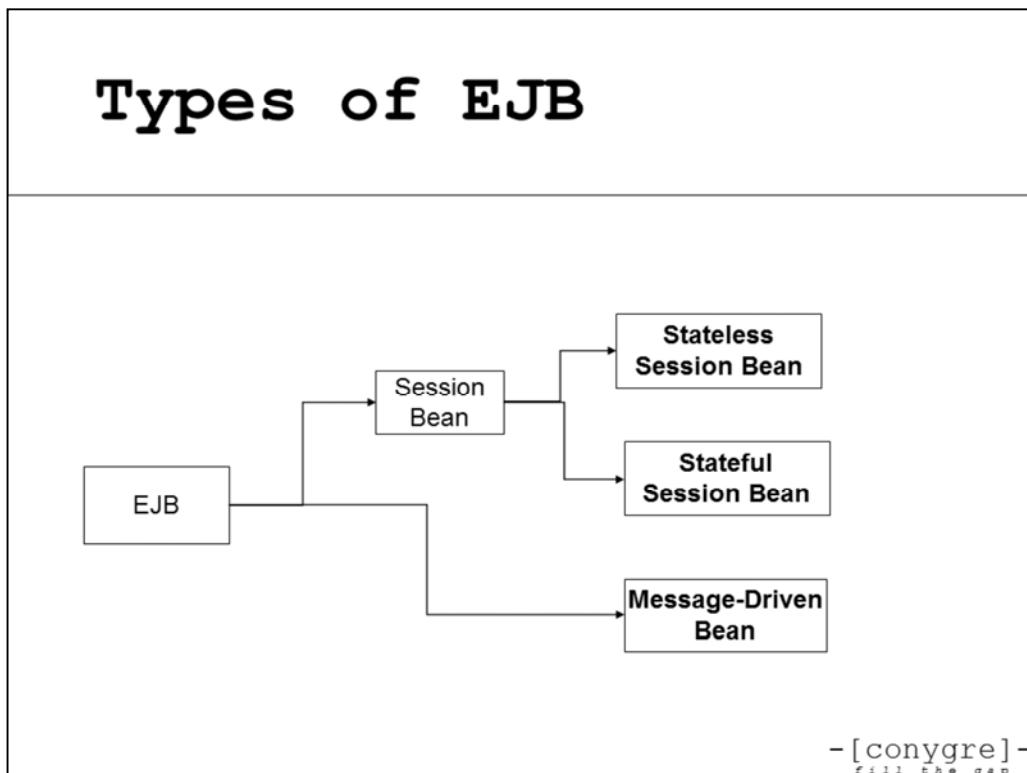
In this chapter, we will explore the concept of Enterprise Java Beans, and you will be introduced to the different kinds of beans and what they are used for. This chapter discusses EJB 3.x

Enterprise Java Beans

- EJB is a specification for java beans built specifically for multi-tier applications
- There are two types of EJB in Java EE 7

- [conygre] -
fill the gap

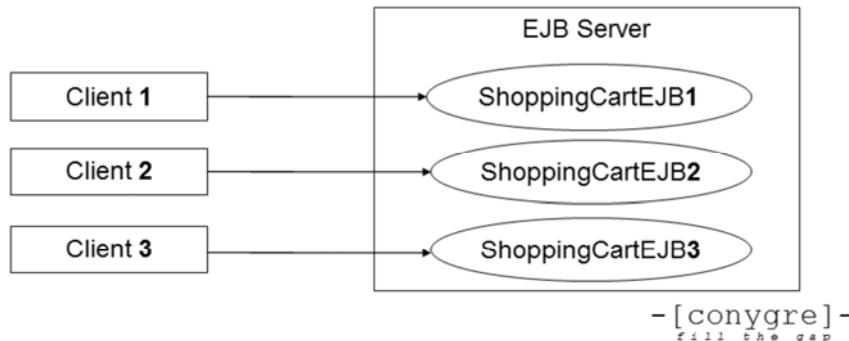
Enterprise Java Beans are very important in enterprise Java applications. In this chapter, you will explore the basic principles and architecture of EJBs.



In EJB, there are two current categories of EJB. Session beans, and message-driven beans. The entity bean, although still available for backwards compatibility, has been deprecated.

Stateful Session Beans

- Stateful Session Beans maintain client state on the server between interactions
- Used where client state must be maintained between invocations



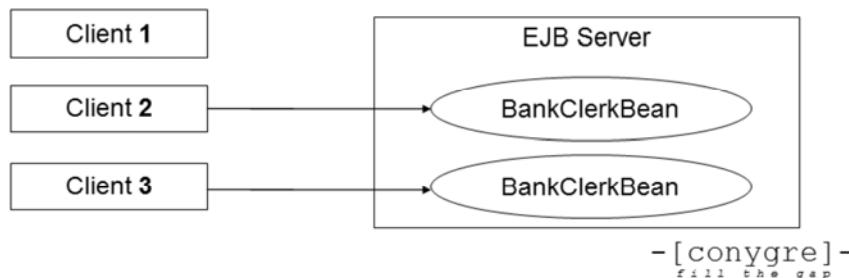
If a client needs to use some business logic or functionality that must be encapsulated by an object *that remembers them*, which means it must maintain state information about the specific client, then a stateful session bean can be used.

When a client connects for the first time, an instance is created to serve that specific client. In other words, when a client connects to the server a session bean instance is created and allocated to that client, and it will then maintain any session data for that client.

Since this could have significant memory implications for the server if there are many clients, these objects can be **passivated**, which means streamed to disk to conserve memory. They when a client returns, they can then be **activated** (recreated from the data on the disk) if they are needed subsequently.

Stateless Session Beans

- Stateless session beans do not persist any session information
 - They can be viewed as '**pure business logic**'
- These beans are pooled, and each invocation could be allocated to any available bean



Stateless session beans do not persist any session data. They could be viewed almost as a library of functions, with no data which is being preserved across sessions. They have no 'conversational state'. If you were 'talking' to a stateless session bean, it would never remember anything you said to it.

You may be wondering why this differentiation exists between stateless and stateful session beans. The reason is all down to efficiency. A stateless session bean is far less memory intensive. In fact, your EJB container will have a pool of these pre-created ready for use by clients, and when a request comes in, any appropriate stateless session bean instance from the pool can be used. Also, new instances are not constantly being created and garbage collected, and again, this improves efficiency.

These are commonly used in web applications as a façade for data access, which will be discussed next later in the course.

Message Driven Beans

- Since EJB 2.0, part of J2EE1.3, a new type of bean the **Message Driven Bean (MDB)** has been available
- A message driven bean allows asynchronous messages to be passed into an EJB container
- MDBs have one important method called **onMessage**
- onMessage is invoked every time a message is received

- [conygre] -
fill the gap

The Message Driven Bean was added to the EJB spec in version 2.0. It provides the ability for asynchronous messages to be passed to the EJB container. The most important method for a MDB is the onMessage() method. This method is invoked when a message for that EJB is received by the container.

Entity Beans – DEPRECATED !!

- Entity beans represent **business data**
 - Eg. An entity bean might represent a row in a database table
- Note that when referring to entity beans, you are referring to the **actual data**
 - Eg. If I create an entity bean, I add a row in the table, if I remove an entity bean, I remove a row from the table

- [conygre] -
fill the gap

The two kinds of session beans we talked of earlier can be regarded as the business logic of your application. The entity beans on the other hand are the business data.

Session beans – business logic

Entity beans – business data

Entity beans could represent for example, a row from a database. One important distinction to make here is in terminology. If I use the expression, ‘create an entity bean’, I am actually talking about inserting a new row in the table.

The detail of how to manage the database modifications and locking etc. (the JDBC code) can either be put into the entity bean itself, or the EJB container can be set to manage all of that for you. This is where the two different types of entity bean come in.

Bean and Container Managed Persistence

- The two types of Entity beans are
 - **Container Managed Persistence (CMP)** where the container creates the JDBC persistence code
 - **Bean Managed Persistence (BMP)** where the developer creates the JDBC code within the EJB

- [conygre] -
fill the gap

With bean managed persistence (BMP), the JDBC code is all the responsibility of the entity bean developer, in other words, the implementation class must contain all the relevant JDBC code.

With container managed persistence (CMP), the JDBC code is the responsibility of the container within which the beans are residing.

When using container managed persistence, the container stores the data from the database into the instance variables of the bean. The mapping between the EJB and the instance variables is defined within an XML file called a deployment descriptor.

Parts of an EJB

1. EJB implementation class
2. An optional component interface defining business methods available to clients, either remotely or locally
3. An optional deployment descriptor written in XML configuring the EJB to run on the server

- [conygre] -
fill the gap

Now that you have seen the different types of EJB, you will now see what an EJB looks like. An Enterprise Java Bean is more than a simple JavaBean like those seen earlier. Each EJB has at least one interface, one for local access to the EJB and / or one for remote access.

Note that MDBs do not have any need for an interface, they just have an implementation class.

Implementation Class

- This class contains all the implementation of the business logic for the EJB
- Since JEE 6 you can have EJBs that are just the implementation class and nothing more
- Simply annotate the class as @Stateless and job done!

```
@Stateless  
public class HelloBean {  
    public String sayHello() {  
        return "hello from EJB";  
    }  
}
```

- [conygre] -
fill the gap

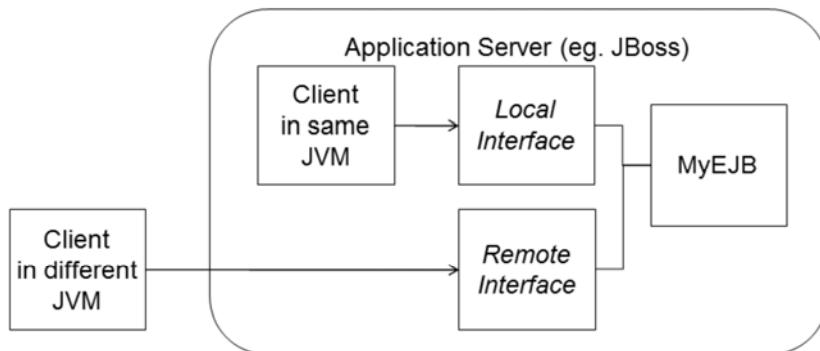
Interfaces

- Almost always you will define an interface for your EJB
- These interfaces are either local or remote
 - **Remote** component interface
 - Allowing clients access to the EJB over the RMI/IOP protocol
 - **Local** component interface
 - Allowing clients to access the EJB within the same JVM

- [conygre] -
fill the gap

EJB Invocation

- Local and Remote interfaces



- [conygre] -
fill the gap

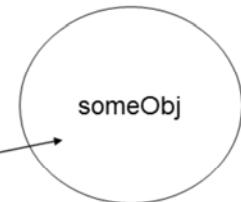
When an EJB is accessed by a client, it has to be through an interface. There are two interfaces that the developer can create. There is a local interface for clients in the same VM, and a remote interface that allows clients in different VMs to access the bean using the RMI/IOP protocol.

Invocation Semantics

- When using local interfaces, Java method calls operate as normal ie. object references are passed by value
- When using remote interfaces, the actual objects *are passed by value* – not simply a reference

```
someEJB.someEJBMethod(objReference);
```

Copy of object on the end of objReference
passed to someObj



In Java, when an object reference is a parameter in a method call, the value of the reference itself is passed to the method. This is how local EJB method invocations will work. This results in any changes being made to the object being visible to the method and the caller of the method.

However, with remote interfaces, any method calls taking object references as parameters receive copies of the object rather than references to the original object. Therefore any changes made on the object by the EJB will not be visible to the caller of the method. You therefore need to design with this in mind.

In addition, when using remote interfaces, all parameters and return types must be **Serializable** in order for them to be passed over the wire.

There is one exception to the behaviour of passing the value of objects themselves when using remote interfaces, and that is where the object being passed is itself a Remote object created in Java using RMI technology. Remote objects are capable of being passed by reference. In practice however, this is rarely necessary in EJB applications.

Single Thread Access

- Developer can assume that business logic
 - Will only be accessed by one thread at a time
- Developer can avoid issues associated with
 - Deadlock
 - Synchronisation
 - Access to shared resources

- [conygre] -
fill the gap

EJB's are always accessed by a single thread at a time. The EJB specification removes these issues from the developer creating EJB classes. These issues are now down to the container of the EJB rather than the EJB itself.

Deployment Descriptor

- Deployment information for EJBs is done in one of two ways
 - Using an XML **Deployment Descriptor** to contain configuration data
 - ejb-jar.xml
 - Using special annotations in the code itself to contain deployment information
 - Available since EJB 3.0

- [conygre] -
fill the gap

For the application to use the EJB, it needs to know where the classes are, what the remote methods are, what type of EJB it is etc. This is done either by way of an XML file called the deployment descriptor, or through the use of annotations in the Java code itself. Annotations are a new way of doing this, and are only available in EJB 3.0.

The filename for this deployment descriptor is ejb-jar.xml. Deployment tools will create this file for you.

Packaging

- The bean and its associated files can then be placed into a **JAR** file, ready for deployment into an application server
- The jar file needs to be set out as follows
 - The deployment descriptor goes into a folder called META-INF
 - The Java classes for the EJB go into folders based on their package location from the root of the JAR file

- [conygre] -
fill the gap

There must be a folder within the JAR called META-INF, and this folder contains the XML deployment descriptor for the bean. As we shall see later, this file specifies to the application server how this particular bean is to be deployed and used. The Java classes must then be within the folders as specified by the package location. See the next slide for an example. This format is the same for all the different types of EJB.

If the EJB needs any other specific resources, such as helper classes for example, then these would also be place within this JAR file.

Packaging Example

- If we had an EJB called
com.conygre.MyPerfectEJB
- In the JAR file, there would be the following

```
MyPerfectEjb.jar
/META-INF
/META-INF/ejb-jar.xml
/com/conygre/
/com/conygre/MyPerfectEJB.class // implementation class
/com/conygre/MyPerfectLocal.class // local interface
```

- [conygre] -
fill the gap

Summary

- There are two kinds of EJB
 - Session beans
 - Message Driven Beans
- EJBs comprise of
 - An implementation class
 - A component interface which can be
 - Local
 - Remote
 - Deployment information
 - In an XML file
 - In the class in the form of annotations

- [conygre] -
fill the gap

JNDI

- [conygre] -
r i l l t h a g a p

Objectives

- Naming Services
- Java Naming and Directory Interfaces JNDI
- JNDI Example

- [conygre] -
fill the gap

Naming Services

- All distributed object architectures use a naming service
 - RMI has a naming service
 - CORBA has a naming service
- Naming services provide clients with a mechanism for locating objects
- Naming services provide two services
 - **Object Binding**
 - **Lookup services**

- [conygre] -
fill the gap

Naming services are integral to all distributed architectures. They enable client objects to locate and make method calls on remote objects. RMI (Remote Method Invocation) and CORBA (Common Object Request Brokerage Architecture) both have a naming service. A naming service essentially has two functions. It firstly enables local references on the client to refer to remote objects, and secondly, it enables clients to look up objects that are accessible remotely.

Essentially, a naming service will enable a client to look up a remote object within some distributed service, and obtain a reference to this remote object.

JNDI Naming Service

- Java Naming and Directory Interface is the naming service used in JEE
- The **JNDIContext** is the main JNDI interface used to
 - Register and look up distributed objects
- EJBs use the JNDIContext to locate resources

- [conygre] -
fill the gap

Java Naming and Directory Interface (JNDI) is a naming service, and it is vital to EJB applications because EJBs only communicate via remote references, even when they are residing in the same server. For one EJB to access another, it has to go via a remote reference.

JNDI will support nearly all directory and naming services. The fact it is a directory and naming service means that it will divide up all the remotely accessible objects into a hierarchical folder structure.

Metadata can also be created for these remote objects, enabling sophisticated searches to be carried out on the directory structure.

Many vendors have implemented directory services, so an EJB developer can implement the service best suited to their needs.

Our EJBs will use the main interface in JNDI, which is **JNDIContext**, to locate one another. On the next slide we can see an example of this.

Viewing the JNDI Tree

- Application servers provide browser interfaces to enable you to see what is registered in the JNDI tree



Dependency Injection

- In Java EE 5.0, dependency injection has been introduced
- Dependency injection allows you to refer to required resources in the code through the **EJB** annotation

```
public class ServletEJB3 extends HttpServlet {  
    @EJB  
    private CompactDiscFacadeLocal ejbRef;  
    ...  
}
```

- [conygre] -
fill the gap

Dependency injection makes the use of one component by another much easier than it used to be in J2EE. There are no deployment descriptor entries, but just annotations. In the above example, a Java Servlet has a reference to an EJB declared, and using the EJB annotation, this EJB is now available to the methods in the servlet.

This can also be done from other EJBs, or even CDI based POJOs

DI with JNDI Name

- When injecting a DataSource you can specify the JNDI name
- This is done using the **@Resource** annotation

```
@Resource(mappedName = "jdbc/MySQL")  
private DataSource conygreDataSource;
```

- [conygre] -
fill the gap

Programmatic Lookups

- Look ups can be done programmatically if required
- Remote EJB references involves a cast using `PortableRemoteObject.narrow()`
 - This is a requirement of the underlying RMI/IIOP protocol

```
public void findAnEJB() {
    try {
        javax.naming.Context context = new InitialContext();
        Object obj = context.lookup("java:comp/env/ejb/MathBean");

        MathInterface bean = (MathInterface)
            javax.rmi.PortableRemoteObject.narrow(obj, MathInterface.class);
    }
    catch (javax.naming.NamingException e) {}
}
```

- [conygre] -
fill the gap

Remember, the purpose of this code here is to enable the session bean to obtain a reference to the entity bean that we created called Product. Firstly, we need to obtain a reference to the initial context which represents the starting context for performing naming operations. Once we have our context, we can look up our remote object reference using the lookup method. Note the name of the remote object. When looking objects up, we must always catch the checked exception, `javax.naming.NamingException`.

The casting may appear a little weird in our example above. You may be left wondering why we cannot simply cast our Object reference `obj` straight to a `MathInterface`. The reason for this is that EJB remote access is over RMI/IIOP, not simply RMI. This protocol supports multiple languages, and you can no longer assume that everything is developed in Java that is registered with our naming service. Some languages that could have been used to define an object may not support casting or polymorphism, like COBOL, so therefore IIOP does not support the ability for a stub to have multiple interfaces. To explicitly cast to a different interface type therefore, we have to use the **PortableRemoteObject** method, `narrow()`. This method can be used to complete our cast.

JNDI Local Lookup Example

- Local lookups do not have the PortableRemoteObject.narrow() method
- Below is an example of looking up a message queue – you will often see code like this in messaging applications

```
public void findAnEJB() {  
    try {  
        javax.naming.Context context = new InitialContext();  
        Object obj = context.lookup("queue/testQueue");  
        Queue queue = (Queue) obj;  
    }  
    catch (javax.naming.NamingException e) {}  
}
```

- [conygre] -
fill the gap

Summary

- Naming Services
- Java Naming and Directory Interfaces JNDI
- Dependency injections
- Remote Lookups
- Local Lookups

- [conygre] -
fill the gap

Session Beans

Session Beans

- [conygre] -
e l l l t h e o g a p

In this chapter you will learn how to create and deploy EJB3 Session beans.

Objectives

- The Session bean interfaces
- The Session bean implementation
- Session bean annotations
- Session bean deployment descriptor
- Deployment structure

- [conygre] -
fill the gap

Session Beans

- Session beans are the business logic
- The artefacts that make up a session bean are the
 - Remote Component Interface and / or
 - Local Component Interface
 - Implementation class
 - Optional Deployment Descriptor (XML)

- [conygre] -
fill the gap

Session beans contain the business logic. You will now define a basic session bean to manipulate some data.

The parts of a session bean are the implementation class that contains the actual functionality, and one or two interfaces: The remote or local component interfaces. You can then optionally create a deployment descriptor if you choose not to put the deployment information in the implementation class in the form of annotations.

Math Example Session Bean

- We will define a **Math** stateless session bean
- The Component Interface
 - Defines the business methods of the bean
 - Provides an annotation for whether it is for remote or local access (local is the default)

```
package com.conygre.cd;
import javax.ejb.Remote;
@Remote
public interface MathInterface {
    public int add(int a, int b);
    public double average(int a, int b);
}
```

- [conygre] -
fill the gap

The component interface contains the methods which encapsulate the functionality of the EJB. In this case, there are two methods `add()` and `average()`, which will provide some basic mathematical functions. Beans are locally accessed by default. For remotely accessible beans you must use the `@Remote` annotation. This can be placed either in the interface (as shown above) or in the bean class. For remote interfaces, if the methods returned any objects, then the class definitions for those objects would have to implement `Serializable` in order to be streamed over RMI/IOP.

MathBean Implementation

- The implementation class contains the functionality
- Annotations are used to specify whether it is
 - Stateful or Stateless
 - Remotely or Locally accessible

```
package com.conygre.cd;
import javax.ejb.*;
@Remote
@Stateless
public class MathBean implements MathInterface {
    public int add(int a, int b) {
        return a + b;
    }
    public double average(int a, int b) {
        return (a+b)/2;
    }
}
```

- [conygre] -
fill the gap

Our implementation class must implement the **MathInterface**, and must implement the methods found within it. Annotations are used to specify whether it is stateful or stateless. Annotations are also used to specify if it is accessible remotely or locally. The annotations regarding Remote or Local can be placed in the interface if preferred.

SessionContext

- Session beans have access to a SessionContext which can be set as a variable using an annotation
- SessionContext provides access to
 - Security information
 - Transactional information

```
@Resource private SessionContext context;  
..  
context.isCallerInRole("guest");
```

- [conygre] -
fill the gap

SessionContext provides the bean with a runtime context in which it exists. Using it you can access both security and transactional information from the container. The example code above demonstrates how you can find out if the caller is in a particular role. Additional methods can be found in the SessionContext API documentation.

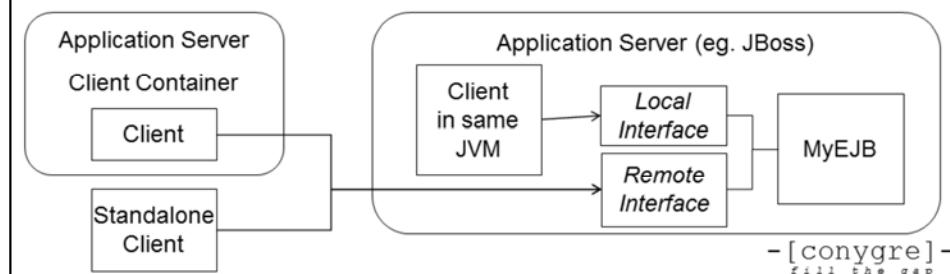
Deploying the Math Bean to WildFly

- Once the code is compiled, it can be placed in a jar file and the jar copied to
 - <WILDFLY_HOME>\standalone\deployments

- [conygre] -
fill the gap

EJB Clients

- EJB clients can either be
 - Stand alone Java applications
 - Client code running in the EJB container
 - Local interface access only
 - Clients running in a client container



Client Containers

- Client containers are provided by application server vendors
- Client containers provide access to container services such as
 - Authentication
 - Authorisation
 - Dependency injection

- [conygre] -
fill the gap

Client containers provide a number of benefits when creating clients, the most important of which is that they provide access to the container services running in the container hosting the EJB. Client containers are provided by all application server vendors.

Standalone Remote Clients

- To access an application server remotely, properties are required to specify the location of the JNDI server
- It is most convenient to set these properties through a standard properties file

jndi.properties

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

- [conygre] -
fill the gap

For a remote standalone Java program to access your server, it needs access to the JNDI tree to lookup your EJB. The location of this server, and the JBoss specific information is most easily set in a properties file.
This file must reside in the classpath for the client application.

The Remote Client Code

- The client uses JNDI to locate the EJB

```
InitialContext context = new InitialContext();
Object obj = context.lookup("MathBean/remote");
MathInterface remote = (MathInterface) (PortableRemoteObject.narrow(obj,
    MathInterface.class));
System.out.println("two plus two is " + remote.add(2,2));
```

- Notice that the lookup returns a reference of the remote component interface type

- [conygre] -
fill the gap

The client code is all that is required to invoke the EJB from outside of the container. You would also need to catch the javax.naming.NamingException if the lookup failed.

The lookup name of the bean is the same as we saw on the JNDI tree earlier.
To run the client, the <JBOSS_HOME>\client jars must be in the classpath.

Local Clients

- Local clients can take advantage of dependency injection – much easier!

```
@EJB  
MathLocalInterface local;
```

- [conygre] -
fill the gap

Session Beans

Session Bean Lifecycle

- Session beans have two states
 - Ready or Passive
- Callback methods are called as the bean enters the different states

```
graph TD; A[Does not Exist] -- "postConstruct callback" --> B[Ready]; B -- "prePassivate callback" --> C[Passive]; C -- "postActivate callback" --> B; B -- "remove() method" --> D[ ]; D -- "preDestroy callback" --> A; A -- "timeout" --> E[ ];
```

The diagram illustrates the lifecycle of a session bean through three states: Does not Exist, Ready, and Passive. Transitions are triggered by various callbacks:

- From Does not Exist to Ready is triggered by postConstruct callback.
- From Ready to Passive is triggered by prePassivate callback.
- From Passive back to Ready is triggered by postActivate callback.
- From Ready to Does not Exist is triggered by remove() method.
- From Does not Exist back to Ready is triggered by preDestroy callback.
- A timeout transition leads from Does not Exist directly to a final state (represented by an empty box).

- [conygre] -
fill the gap

Session beans have two states:

1. Ready – the bean is available in memory to receive requests from clients.
2. Passivated – the bean has had its state saved onto disk to conserve memory.

The state that the bean is in at any one time is managed by the container. There are callback methods that you can add to the code to complete tasks during any one of the state changes. Callback methods are indicated with the use of annotations.

Session Bean Callbacks

- The following callback handlers can be added to a session bean
 - PostConstruct
 - PreDestroy
 - PrePassivate
 - PostActivate

```
@PrePassivate
public void backupDataToDatabase() {
    // do something with the data before the bean is passivated
}
```

- [conygre] -
fill the gap

Session Beans as Web Services

- Session beans can also be deployed as Web services through the use of additional annotations
 - The Session bean implementation class uses
 - `@javax.jws.WebService`
 - The Remote interface uses
 - `@javax.jws.WebService`
 - `@javax.jws.soap.SOAPBinding`

- [conygre] -
fill the gap

Implementation Class

- For Web services, the implementation class uses the following annotations

```
@Remote(MathInterface.class) // specify the remote interface component interface  
@Stateless // has to be a stateless bean  
@WebService(endpointInterface = "com.conygre.cd.MathInterface")  
// the endpoint interface is used to specify the WSDL interface  
public class MathBean implements MathInterface {  
    ..  
}
```

- [conygre] -
fill the gap

Remote Interface

- The remote interface specifies additional annotations

```
@WebService // specify it is a Web service  
@SOAPBinding(style = Style.RPC) // specify the encoding style to be used  
public interface MathInterface {  
    ...  
}
```

- [conygre] -
fill the gap

The style can be DOCUMENT or RPC.

Singleton EJBs

- Since EJB 3.0 you can also deploy Session beans that are guaranteed to be singletons

```
@Singleton  
@Local  
  
public class NumberFountainImpl implements NumberFountain  
{  
    private int nextNumber = 0;  
    public int getNextNumber() {  
        return(nextNumber++);  
    }  
}
```

- [conygre] -
fill the gap

Singleton beans are managed as a single instance by the container. They have an interface and can be local or remote as any other EJB but they are not pooled. There is just one instance. Bear in mind however that singleton beans will not be genuine singletons in a clustered environment as each server would end up with a single instance of the bean.

No Interface EJBs

- You can also create a locally accessible EJB that provides no interface
- Optionally you can annotate them as `@LocalBean`
 - If you miss the annotation and have no interfaces, it will be assumed to be a no interface EJB

```
@Stateless  
@LocalBean  
public class MyEJB {  
    public void someLocalMethod() {  
        // do something  
    }  
}
```

- [conygre] -
fill the gap

You can even create an EJB without any interfaces. If you do this, it is only available locally as there is no remote interface, and clients have to know the classname, so it breaks the ‘program to an interface not a class’ best practice, but the specification does allow this.

Deployment

- The Session bean is deployed as a jar file
- The URL for the WSDL file is shown in the WildFly console
 - `http://localhost:8080/MathBeanService/MathBean?wsdl`

- [conygre] -
fill the gap

Using a Deployment Descriptor

- The deployment descriptor is optional since EJB3.0
 - Deployment information can be specified through the use of annotations
- If you do use a deployment descriptor, then it must be
 - Called **ejb-jar.xml**
 - Placed in a directory called **META-INF** in the jar file containing the EJB classes

- [conygre] -
fill the gap

The Deployment Descriptor

- The definition lies between **<ejb-jar>** ... **</ejb-jar>** elements
- The XML must adhere to the XML Schema located here
 - http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd

```
<ejb-jar version="3.0"
  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/ejb-jar_3_0.xsd">
```

```
</ejb-jar>
```

- [Conygre] -
fill the gap

The XML file must follow the rules as specified in what is called the **XML Schema** for EJB3.0. This is a file that specifies the rules and structure that the XML document must follow. This is vital, as it provides portability across vendor implementations of EJB. By providing a Schema, Sun have forced application server vendors to stick to a format for the deployment descriptor.

The actual deployment description detail is defined between opening and closing **ejb-jar** elements, as shown on the slide.

<enterprise-beans> element

- Then we define the <enterprise-beans> element

- Within these tags we define various properties of the bean

```
<ejb-jar>
<enterprise-beans>
<session>
  <description>
    This Product enterprise entity bean
    represents a product in an online store.
  </description>
  <ejb-name>MathBean</ejb-name>
  <business-remote>com.conygre.MathInterface</business-
remote>
  <ejb-class>com.conygre.MathBeanProductBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
```

- [conygre] -
fill the gap

Within the <ejb-jar> element, we specify an <enterprise-beans> section, where we define our beans. More than one EJB can be defined within a deployment descriptor.

Each bean is defined as follows.

<session> - this is a session bean.

<description> - a description of the bean

<ejb-name> - the name of the EJB

<business-remote> - the remote interface class

<ejb-class> - the implementation class

<session-type> - either stateful or stateless

<transaction-type> - either container or bean. If container, then the container will manage the transactions.

<assembly-descriptor> element

- The <assembly-descriptor> element contains
 - security roles
 - transactional attributes

```
<assembly-descriptor>
  <security-role>
    <description>This role represents those allowed full access to the product bean.
    </description>
    <role-name>everyone</role-name>
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
      <ejb-name>ProductBean</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
... continued
```

- [conygre] -
fill the gap

The further section after the enterprise-beans have been defined, is the <assembly-descriptor>. This defines the security and transactional detail for our beans.

Within the <assembly-descriptor> element, we define the security roles first of all. In our example, we are defining one role called **everyone**.

We can then specify the <method-permission>. Here we state that all methods have the same permissions, ie. Access is for the **everyone** security role. The use of the * in the <method-name> element denotes all methods.

<assembly-descriptor> (2)

- The last part of the assembly-descriptor defines the transactional state of the EJB
- Here we have stated that all methods require a transaction

```
<container-transaction>
  <method>
    <ejb-name>ProductBean</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

- [conygre] -
fill the gap

Transactions occur in a number of what are termed **demarcation modes**. We specify the demarcation mode in this final section of the deployment descriptor. Here we are stating that all our methods require a transactional context in which to be executed. There are several other transactional modes that can be adopted. These demarcation modes are identical to the transaction attributes discussed earlier in the chapter.

Summary

- The Session bean interfaces
- The Session bean implementation
- Session bean annotations
- Session bean deployment descriptor
- Deployment structure

- [conygre] -
fill the gap

Going Further with JPA

Objectives

- The role of the Entity Manager in EJBs
- Entity manager methods
- Using an EntityManager from a session bean
- Working with Queries
- The JP-QL
- Managing Transactions
- The entity object lifecycle

- [conygre] -
fill the gap

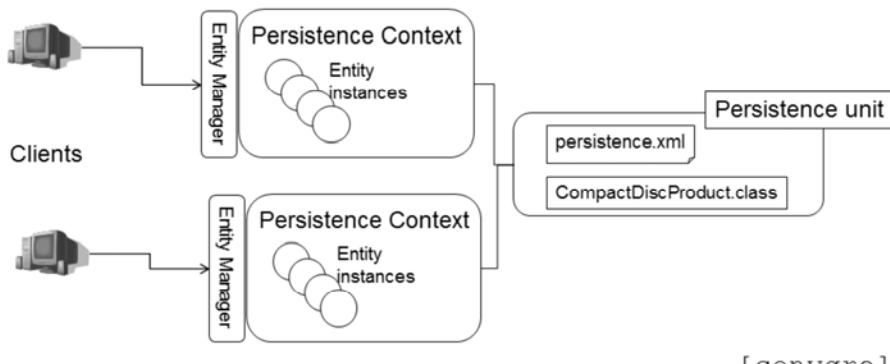
Architecture

- You must understand the following concepts
 - The **Persistence Unit** is the entity class along with the persistence.xml file
 - The **Persistence Context**, which is a set of entity instances based upon the persistence unit
 - The **Entity Manager** which provides access to a persistence context

- [conygre] -
fill the gap

The Persistence Context

- You can think of the persistence context as an instance of the persistence unit



The Role of the Entity Manager

- The **EntityManager** is a class provided by the Persistence API
 - The role of the entity manager is to allow a user to add / remove / update and basically work with a set of entity instances (a persistence context)
 - The entity manager is a direct equivalent to the Hibernate Core **Session** object

- [conygre] -
fill the gap

EntityManager Methods	
Method	Purpose
<T> find (Class<T> entityClass, Object primaryKey)	A find by primary key function.
void persist (Object entity)	Save an entity into the database.
void remove (Object entity)	Remove an entity from the database.
Query createNamedQuery (String name)	Creates a query using JP QL.
Query createNativeQuery (String sql)	Creates a native SQL query.
void refresh (Object entity)	Update the entity object with the latest data from the database.
void merge (Object entity)	Merge the state of the entity into the persistence context.
void clear()	Detach all entities from the database.

- [conygre] -
fill the gap

Using the Entity Manager

- In this example, you will see how to use an entity manager from a session bean
- The entity manager can be added using a **dependency injection**

```
@Local  
@Stateless  
public class CompactDiscFacade implements CompactDiscFacadeLocal {  
    @PersistenceContext  
    private EntityManager em;  
    ...  
}
```

- [conygre] -
fill the gap

Using the Entity Manager

- In this example, the entity manager is being used to create a new entry in the database

```
// add a new CD to the database
public void addCompactDisc(CompactDiscProduct disc) {
    em.persist(disc);
}
```

- [conygre] -
fill the gap

Running Queries

- Queries can be
 - Native SQL
 - Written using the Java Persistence Query Language (**JP-QL**)

```
// return all the CDs
public List<CompactDiscProduct> getCompactDiscs() {
    Query query = em.createQuery("from CompactDiscProduct");
    List results = query.getResultList();
    return results;
}
```

- The Query interface defines a number of methods

- [conygre] -
fill the gap

The Query Interface

- The table below shows some of the key methods of the Query interface

Method	Purpose
List getResultList()	Returns the results of the query as a List of entity objects.
int executeUpdate()	Executes an update statement, returning the number of affected rows.
Object getSingleResult()	Returns a single entity based on a query returning one result.
Query setParameter(String name, Object value)	Sets a named parameter on a query.

- [conygre] -
fill the gap

The JP-QL

- The JP-QL is very similar to SQL

```
SELECT e FROM CompactDiscProduct e  
SELECT e FROM CompactDiscProduct e WHERE e.artist = 'Abba'
```

- You can also do named queries which are cached

```
@NamedQuery(name="findAllCDs", query="SELECT e FROM CompactDiscProduct e")  
Query query = em.createNamedQuery("findAllCDs");
```

- [conygre] -
fill the gap

Parameterised Queries

- Queries can also have parameters

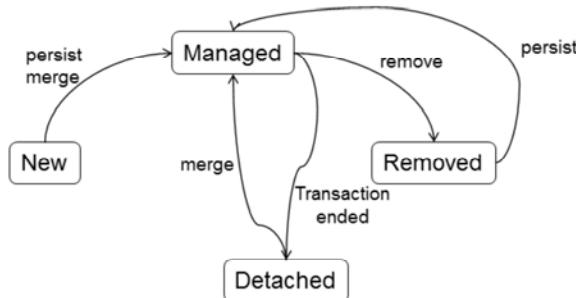
```
@NamedQuery(name="findCDByArtist",
query="SELECT e FROM CompactDiscProduct e WHERE e.Artist =
:artist")
```

```
Query query = em.createNamedQuery("findCDByArtist");
query.setParameter("artist", "Abba");
List result = query.getResultList();
```

- [conygre] -
fill the gap

Entity Object Lifecycle

- Entity objects are managed by the EntityManager, but what states can they be in?



- [conygre] -
fill the gap

Transactions

- Methods can also be part of a transaction
- The **TransactionAttribute** annotations specifies the behaviour

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public int add(int a, int b) {
    return a + b;
}
```

- [conygre] -
fill the gap

Any method can be part of a transaction. The container can be used to manage the transactions (it is by default), and if so, you can specify how the container is to treat the method in regard to transactions. In the above example, the annotation specifies that a transaction is required. There are in fact six possible behaviours that can be specified, and they are described on the next page.

Transaction Attributes	
Attribute	Behaviour
Required	A transaction is required. If one exists, the call will become part of that transaction. If no transaction exists, a new one will be created for that call.
RequiresNew	A new transaction will be started for this method call. If one exists already, it will be suspended.
NotSupported	A transaction will not be used. If one is in progress, it will be suspended for this method call.
Supports	If a transaction is running, then this method call will become part of it, if there is no transaction, then the call will not be part of a transaction.
Mandatory	Exception thrown if no transaction is present.
Never	Exception thrown if a transaction is present.

-[conygre]-
fill the gap

For business methods, the most commonly used attributes are **Required**, and **RequiresNew**.

For logging and debugging methods, the most commonly used attribute is **NotSupported**.

Supports, **Mandatory**, and **Never** are rarely used in practice.

Rolling Back Transactions

- Transactions will role back if
 - *any checked exception* thrown
 - you call `context.setRollbackOnly()`
- Transactions will not role back if
 - A runtime exception is thrown
- Checked exceptions annotated as shown do not role back a transaction
 - `@ApplicationException(rollback=false)`
 - You can use this for your own exceptions

- [conygre] -
fill the gap

Setting Rollback Only

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public int add(int a, int b) {
    ...
    catch (SomeRuntimeException e) {
        // this will not automatically rollback the tx so I will do it
        context.setRollbackOnly();
    }
}
```

- [conygre] -
fill the gap

ApplicationException Example

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public int add(int a, int b) {
    ...
    catch (MyCheckedException e) {
        // problem solved, transaction will not roll back
    }
}
```

```
@ApplicationException(rollback=false)
public class MyCheckedException extends Exception {
    ...
}
```

- [conygre] -
fill the gap

Summary

- The role of the Entity Manager in EJBs
- Entity manager methods
- Using an EntityManager from a session bean
- Working with Queries
- The JP-QL
- Managing Transactions
- The entity object lifecycle

- [conygre] -
fill the gap

Configuring a DataSource in WildFly (aka JBoss)

- [conygre] -
e l l l i h e o g a p

Objectives

- Configuring a persistence unit
- Configuring a WildFly Data Source

- [conygre] -
fill the gap

Creating a Persistence Unit

- As you have seen, a JPA application requires a persistence unit
- When deploying an application to an application server, the persistence unit can use a data source configured on the server itself

- [conygre] -
fill the gap

Complete persistence.xml

- The XML document below is a complete example of a persistence.xml document
- Note the **jta-data-source** which is the connection to be used for the database

```
<persistence>
<persistence-unit name="conygre">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<jta-data-source>java:/jdbc/mySQL</jta-data-source>
<properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
</properties>
</persistence-unit>
</persistence>
```

- [conygre] -
fill the gap

Wildfly DataSources

- DataSources can be configured on Wildfly by following these steps
 1. Copy the database driver into <WILDFLY_INSTALL>\standalone\deployments
The driver will then self register with Jboss
 2. Using the admin interface set up your datasource
(see the steps on the next page)

- [conygre] -
fill the gap

Setting up the DataSource

1. Login to the server admin console at
<http://localhost:9990>
2. Click on the **Configuration** tab and then click **Datasources** and click **Add**

JDBC Datasources

JDBC datasource configurations.

Available Datasources

Name	JNDI	Enabled?
ExampleDS	java:jboss/datasources/ExampleDS	<input checked="" type="checkbox"/>

« < 1-1 of 1 > »

- [conygre] -
fill the gap



Specify the JNDI Name

3. Set the name and JNDI name

- The JNDI name is used by applications wishing to use the connection

The screenshot shows a 'Create Datasource' dialog box. At the top, it says 'Create Datasource' and 'Step 1/3: Datasource Attributes'. There is a 'Need Help?' link. Below that, there are two input fields: 'Name:' and 'JNDI Name:', each with a corresponding text input box. At the bottom right of the dialog, there is a watermark-like text: '- [conygre] - fill the gap'.

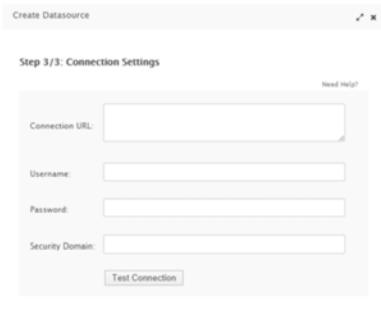
Choose the driver JAR

- If your database driver is a more recent driver, it will come up as a detected driver
 - If not, you will need to specify the details manually

The screenshot shows a 'Create Datasource' dialog box with the title 'Step 2/3: JDBC Driver'. It asks to 'Select one of the deployed JDBC driver'. There are two tabs: 'Detected Driver' (which is selected) and 'Specify Driver'. The 'Detected Driver' tab shows a single entry: 'Name: MySQLDriver'. The 'Specify Driver' tab has four input fields: 'Name:' (MySQLDriver), 'Driver Class:' (com.mysql.jdbc.Driver), 'Major Version:' (empty), and 'Minor Version:' (empty). At the bottom right of the dialog is a watermark: '- [conygre] - fill the gap'.

Configure the Connection

1. Set the URL, Username and Password
2. Click **Test Connection** to check it



The screenshot shows a 'Create Datasource' dialog box with the title 'Step 3/3: Connection Settings'. It contains four input fields: 'Connection URL', 'Username', 'Password', and 'Security Domain', each with a corresponding text input box. Below these fields is a 'Test Connection' button. At the bottom of the dialog are 'Cancel' and 'Done' buttons. A watermark '- [conygre] - fill the gap' is visible at the bottom right of the dialog.

Enable the Connection

- Once configured successfully, enable the connection
- Note the JNDI name which is used in persistence.xml

JDBC Datasources

JDBC datasource configurations.

Available Datasources

Name	JNDI	Enabled?
ExampleDS	java:jboss/datasources/ExampleDS	✓
MySQL	java:/jdbc/MySQL	✗

Add Remove Enable

Selection

« < 1-2 of 2 > »

- [conygre] -
fill the gap

Summary

- Configuring a persistence unit
- Configuring a WildFly Data Source

- [conygre] -
fill the gap

Deploying Enterprise Applications

In this chapter we will focus on how Web applications can be configured to interact with EJBs.

Objectives

- JEE Deployment process
- Enterprise Applications - EARs
- Working with application.xml

- [conygre] -
fill the gap

What is in J2EE Application?

- JEE applications consist of **modules**
- A module is either a
 - Web application module
 - An EJB module
 - A client module
- The JEE application needs to declare the modules found within the application
 - This is done in a **META-INF/application.xml** file
 - another deployment descriptor

- [conygre] -
fill the gap

A J2EE application can be defined in terms of a number of **modules**. A module is either

1. An EJB module consisting of one or more EJBs and the deployment descriptor for them
2. A web application module consisting of the JSPs, tags, and servlets along with the web.xml file
3. A client module which would consist of code to make up an J2EE client (not discussed in this course)

The modules that make up a complete application need to be declared for the container, and this is done within yet another configuration file, the application configuration file – **application.xml**. This lives in a root level folder with the name **META-INF**.

Application Deployment Descriptor

- **application.xml** describes the application for the container with the following elements

```
<application>
<display-name>My Application</display-name>
<module>
<web>
<web-uri> web-app</web-uri>
<context-root>
simpleEJB
</context-root>
</web>
</module>
<module>
<ejb> ejbs </ejb>
</module>
</application>
```

- [conygre] -
fill the gap

Here is an example of application.xml without the doctype. If you notice, it is specifying the folder locations for the EJBs and the JSP pages. These are defined within **module** elements.

In the above example, the EJBs and JSPs are not in JAR and WAR files, but are still in their respective directory structures. This is not compulsory, you could create the application.xml file and point it to JAR and WAR files instead. See below.

```
..... <web>
<web-uri>webapp.war</web-uri>
<context-root>
simpleEJB
</context-root>
</web>
<ejb>ejbs.jar</ejb> .....
```

Dynamo supports either mechanism. In Dynamo development however, it can be easier to have the EJB and web application files unextracted in the appropriate folders, as this makes debugging a great deal easier.

The **context-root** identifies the URL which will locate the application in the server environment. So in our case, the application would be accessed using <http://localhost:8840/simpleEJB>

Deployment Descriptor Elements

- **<application>** - the entire definition exists between these elements
- **<module>** - these elements define modules either web, ejb or client modules
 - EJB modules require the ejb root directory or JAR file to be specified
 - Web modules require the jsp root or WAR file to be specified

- [conygre] -
fill the gap

The definition of the application.xml file is fairly self explanatory. Within the **application** element, specify each module one at a time within **module** elements. These module elements then contain either **web** or **ejb** elements where the location of the module is defined.

Directory Structures

- The application structure would be
 - Within the folder ejbs would be all the ejb classes
 - Within the folder web-app will be all the jsp files, and then in other sub folders, the tag libraries, servlets, and Java beans
- These folder names are defined in `application.xml`

```
/META-INF/application.xml  
/ejbs  
/ejbs/META-INF/ejb-jar.xml  
/web-app  
/web-app/WEB-INF/web.xml
```

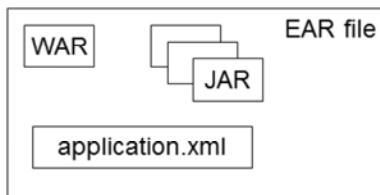
- [conygre] -
fill the gap

Once all the EJBs, JSPs and the application deployment descriptor have been defined, all the files can be placed within a folder structure like that defined in the slide. The ejb folder contains the ejbs, and this is specified in `application.xml` as you can see below. The web files are defined within web-app, and this too is specified in the `application.xml` file. These names are not obligatory, they need to be whatever you specified in `application.xml`.

```
<application>  
  <display-name>My Application</display-name>  
  <module>  
    <web>  
      <web-uri> web-app</web-uri> <!-- folder name defined here -->  
      <context-root>  
        productExample  
      </context-root>  
    </web>  
  </module>  
  <module>  
    <ejb> ejbs </ejb> <!-- folder name defined here -->  
  </module>  
</application>
```

What the Application Server needs

- Application servers need a file called an Enterprise Application Archive – **EAR**
- This archive contains the following
 - EJB JAR archives
 - JSP WAR archives
 - application.xml deployment descriptor



- [conygre] -
fill the gap

This folder structure can then be packaged up into a J2EE application archive or EAR. Various tools can be used to create this file. This EAR file can then be deployed in different EJB server environments.

DI with EARs

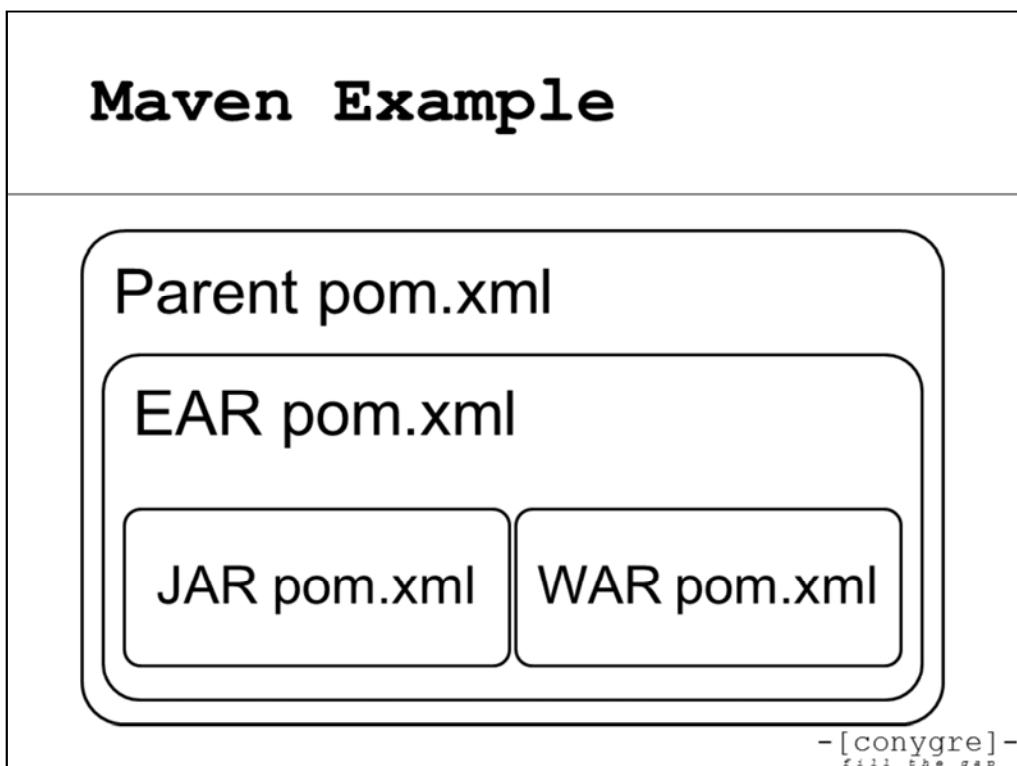
- EJBs in the JAR can be injected into components within the WAR
 - Eg. RESTful classes in the WAR file can have EJB classes from the JAR injected
- You will need to ensure that the war file has a beans.xml within the classes\META-INF folder

- [conygre] -
fill the gap

Building with Maven

- To build an EAR file using Maven, create the following Maven projects
 1. An EJB project hosting the EJB components
 2. A WAR project with a dependency on the EJB project
 3. An EAR project using the Maven EAR plugin
 - This will reference the WAR and EJB projects
 4. Create a parent project that references the other three
 - This facilitates a single build that actually builds the other three in the right order

- [conygre] -
fill the gap



The Parent Pom

- Below is an example of a parent 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>com.conygre.training.ear</groupId>
  <artifactId>CompactDiscEnterpriseApp</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>CompactDiscEnterpriseApp</name>
  <modules>
    <module>CompactDiscEnterpriseApp-ear</module>
    <module>CompactDiscEnterpriseApp-web</module>
    <module>CompactDiscEnterpriseApp-ejb</module>
  </modules>
</project>
```

- [conygre] -
fill the gap

The packaging is pom because it will not be built into a JAR, WAR, or EAR. It is simply a project that contains other projects.

The modules listed below are the names of child projects in the folder that contains this pom file.

So in the folder structure would be:

Example\pom.xml (the file shown)

Example\CompactDiscEnterpriseApp-ear\pom.xml

Example\CompactDiscEnterpriseApp-ejb\pom.xml

Example\CompactDiscEnterpriseApp-web\pom.xml

The EAR file POM

- The EAR project contains no code and the packaging is set to ear

```
<project..>
<parent>
    <artifactId>CompactDiscEnterpriseApp</artifactId>
    <groupId>com.conygre.training.ear</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
<groupId>com.conygre.training.ear</groupId>
<artifactId>CompactDiscEnterpriseApp-ear</artifactId>
<packaging>ear</packaging>
<version>1.0-SNAPSHOT</version>
<name>CompactDiscEnterpriseApp-ear</name>
...
</project>
```

- [conygre] -
fill the gap

The EAR plugin

- The EAR pom also needs to reference the plugin that builds ear files

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-ear-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <version>6</version>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- [conygre] -
fill the gap

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-ear-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <version>6</version>
      </configuration>
    </plugin>
  </plugins>
```

The EAR Dependencies

- The WAR and JAR projects are added as dependencies which triggers them to become part of the EAR

```
<dependencies>
  <dependency>
    <groupId>com.conygre.training.ear</groupId>
    <artifactId>CompactDiscEnterpriseApp-ejb</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>ejb</type>
  </dependency>
  <dependency>
    <groupId>com.conygre.training.ear</groupId>
    <artifactId>CompactDiscEnterpriseApp-web</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>war</type>
  </dependency>
</dependencies>
```

- [conygre] -
fill the gap

The EJB Project

- The EJB project is then a normal EJB project

```
<project ... >
  <parent>
    <artifactId>CompactDiscEnterpriseApp</artifactId>
    <groupId>com.conygre.training.ear</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.conygre.training.ear</groupId>
  <artifactId>CompactDiscEnterpriseApp-ejb</artifactId>
  <packaging>ejb</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>CompactDiscEnterpriseApp-ejb</name>
  ...
</project>
```

- [conygre] -
fill the gap

The WAR Project

- The WAR project is then a normal WAR Project

```
<project..>
<parent>
    <artifactId>CompactDiscEnterpriseApp</artifactId>
    <groupId>com.conygre.training.ear</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
<groupId>com.conygre.training.ear</groupId>
<artifactId>CompactDiscEnterpriseApp-web</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
<name>CompactDiscEnterpriseApp-web</name>
```

- [conygre] -
fill the gap

WAR Dependencies

- The WAR project will require a dependency on the EJB project because it will have EJB components injected
 - The scope is **provided** since the classes will end up in the final EAR file so do not need to be in the WEB-INF\lib directory

```
<dependency>
  <groupId>com.conygre.training.ear</groupId>
  <artifactId>CompactDiscEnterpriseApp-ejb</artifactId>
  <scope>provided</scope>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

- [conygre] -
fill the gap

Building the EAR

- In the root folder containing the parent pom, execute a mvn clean install and it will build all of the projects in order, resulting in an ear file being created in the EAR project

```
[INFO] Reactor Summary:  
[INFO] [INFO] CompactDiscEnterpriseApp ..... SUCCESS [ 0.391 s]  
[INFO] CompactDiscEnterpriseApp-ejb ..... SUCCESS [ 1.455 s]  
[INFO] CompactDiscEnterpriseApp-web ..... SUCCESS [ 0.924 s]  
[INFO] CompactDiscEnterpriseApp-ear ..... SUCCESS [ 0.346 s]  
[INFO] BUILD SUCCESS  
[INFO] [INFO] Total time: 3.241 s  
[INFO] Finished at: 2014-10-27T18:06:41+00:00  
[INFO] Final Memory: 17M/244M
```

- [conygre] -
fill the gap

Summary

- JEE Deployment process
- Enterprise Applications - EARs
- Working with application.xml

- [conygre] -
fill the gap

Security in Java EE

Objectives

- Security Requirements
- J2EE Security model
- Web Authentication
- Web Programmatic Security
- EJB Declarative Security
- EJB Annotations
- EJB Programmatic Security

- [conygre] -
fill the gap

This chapter will introduce the security requirements that a Java EE application can have, and then it will look at how the Java EE platform allows the issues to be addressed.

Security Requirements

- Authentication
- Access control
- Data Integrity
- Confidentiality

- [conygre] -
fill the gap

Security issues are many and complex. However, for most applications, security can be broken down into four areas:

1. **Authentication** – The means by which communicating entities prove to one another that they are acting on behalf of specific identities authorized for access.
2. **Access control** – The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.
3. **Data Integrity** – The means used to prove that a third party has not modified information while in transit.
4. **Confidentiality** – The means used to ensure that information is made available only to users who are authorized to access it.

The first two are specifically addressed in J2EE for Web applications. The second two, although can be handled through a J2EE container, are normally handled by the Web server using Transport Layer Security (TLS), formally known as Secure Sockets Layer (SSL). SSL and TLS are beyond the scope of this course.

Roles and Principals

- Java EE handles security through the use of **Roles and Principals**
 - Role is the *type* of user (like an actor in a use case)
 - Principal is the *actual* user
 - Eg. "Sarah" is the principal, and "Platinum Member" is the role
- Information on users and roles can come from
 - Lightweight Directory Access Protocol (LDAP)
 - Active Directory
 - A Database

- [conygre] -
fill the gap

In J2EE, users are managed using roles and principles. Roles are the type of user. An example would be the Administrator role on a Windows machine for example. The Principal is the actual user. So, on Windows, the Principal would be an actual account that has Administrator privileges.

Where the user information and role information comes from is not specified by J2EE. This means that you can use existing account information such as that found in Windows Active Directory, or data accessed through interfaces such as LDAP. The information could also come from a database.

Declarative or Programmatic Security

- Java EE allows security to be defined in either
 - Deployment Descriptors and Annotations
 - **Declarative** security
 - Application logic
 - **Programmatic** security

- [conygre] -
fill the gap

To work with security in a Java EE application, you can either define access rights declaratively within deployment descriptors, or you can do it programmatically using an API.

Web Application Authentication

- HTTP Basic Authentication
- HTTP Digest Authentication
- Forms Based Authentication
- HTTPS Client Authentication

- [conygre] -
fill the gap

In terms of authenticating users into a Web application, you have four options.

HTTP Basic Authentication

Basic authentication involves the sending of clear text username and passwords to authenticate the user.

HTTP Digest Authentication

A hash of the username and password are sent over the network instead of the actual username and password. Note that J2EE containers do not have to implement this form of authentication. Also, not all browsers support this mechanism.

Forms Based Authentication

Forms based authentication involves the user submitting an HTML form that is defined using specific form field names. This negates the need for a login dialog to appear in the browser as it would in basic and digest authentication.

HTTPS Client Authentication

Using HTTPS (which is secure HTTP over SSL/TLS), you can use form based authentication but send it over HTTPS which secures the username and password.

Defining Authentication

- Authentication mechanism is defined in web.xml

```
<login-config>
  <auth-method>BASIC | DIGEST | FORM | CLIENT-CERT</auth-method>
  <realm-name>Realm Name</realm-name>
</login-config>
```

- [conygre] -
fill the gap

The authentication mechanism that you propose to use within your Web application is configured in web.xml using the login-config element. This defines the authentication mechanism, and then the realm-name.

A realm is a term used in J2EE for the mechanism used to authenticate a group of users. So, for example, basic authentication is known as the HTTP realm.

Defining Protected Content

- Protected content is also defined in web.xml as a **<security-constraint>**

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-
resource-name>
    <url-pattern>/protected/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>guest</role-name>
    <role-name>customer</role-name>
  </auth-constraint>
</security-constraint>
```

- [conygre] -
fill the gap

For Tomcat, basic roles can be defined in <catalina_home>\conf\tomcat-users.xml.

The default content of the file is:

```
<tomcat-users>
  <user name="tomcat" password="tomcat" roles="tomcat"/>
  <user name="role1" password="tomcat" roles="role1"/>
  <user name="both" password="tomcat" roles="tomcat,role1"/>
</tomcat-users>
```

You can add additional users in this document. For example,

```
<user name="zach" password="password" roles="customer"/>
<user name="emily" password="password" roles="guest"/>
```

Configuring Forms Authentication

- Forms based authentication requires the following in web.xml

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/loginFailed.html</form-error-page>
  </form-login-config>
</login-config>
```

- The login form fields must follow a standard naming convention ... see next page

- [conygre] -
fill the gap

The example shown demonstrates how form based authentication would be configured in web.xml for a Web application.

Forms Based Authentication

- For forms based authentication, you create a JSP containing a form as follows

```
<form method="POST" action="j_security_check">
Username: <input type="text" name="j_username"><br>
Password: <input type="password"
name="j_password"><br>
<input type="submit" value="Continue">
</form>
```

- The form action and field names must match as shown

- [conygre] -
fill the gap

This form must appear with the field names and types as shown above. Clearly, the formatting of the HTML can change so that the login page fits in with the rest of a Web site.

Programmatic Security

- To check users credentials, there are methods in the HttpServletRequest class
 - boolean isUserInRole(String roleName)
 - Principal getUserPrincipal()
 - String getRemoteUser()
 - String getAuthType()
 - boolean isSecure()

- [Conygre] -
fill the gap

For programmatic security, methods exist within the HttpServletRequest class to allow you to interact with user information. An example is shown on the next page.

Programmatic Security Example

- Within a JSP, you can use the API as shown

```
<h1>Welcome to the protected area</h1>
<% if (request.isUserInRole("guest")) { %>
<jsp:include page="fragmentForGuests.html"/>
<%
}
else if (request.isUserInRole("customer")) { %>
<jsp:include page="fragmentForCustomers.html"/>
<% } %>
```

- [conygre] -
fill the gap

EJB Security in ejb-jar.xml

- Declarative EJB security can be configured in ejb-jar.xml

```
<security-role>
  <description>
    This role represents everyone
    allowed access
    to the MathBean.
  </description>
  <role-name>vip-user</role-name>
</security-role>
```

```
<method-permission>
  <role-name>vip-user</role-name>
  <method>
    <ejb-name>MathBean</ejb-name>
    <method-name>add</method-name>
  </method>
</method-permission>
```

- [Conygre] -
fill the gap

Roles that are to be used by the EJBs are declared using the **<security-role>** element. Roles can then be applied on a method by method level for each EJB using **<method-permission>** elements.

These constructs form part of the **<assembly-descriptor>** element.

```
<assembly-descriptor>
  <security-role>
    <description>
      This role represents everyone who is allowed full access
      to the bean.
    </description>
    <role-name>vip-user</role-name>
  </security-role>
  <method-permission>
    <role-name>vip-user</role-name>
    <method>
      <ejb-name>MathBean</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
  <container-transaction>
    ...
  </container-transaction>
```

</assembly-descriptor>

EJB Security with Annotations

- The role information can also be declared in the EJB code using annotations

```
@DeclareRoles(value={"vip-user"})
public class MathBean implements MathInterface {

    @RolesAllowed (value={"vip-user"})
    public int add (int a, int b) {
        return a + b;
    }
}
```

- [conygre] -
fill the gap

EJB Programmatic Security

- EJB Programmatic security is similar to Web programmatic security
- Methods are available through the **SessionContext**
 - boolean isCallerInRole(String roleName)
 - Principal getCallerPrincipal()
 - String getRemoteCaller()

- [conygre] -
fill the gap

Summary

- Security Requirements
- J2EE Security model
- Web Authentication
- Web Programmatic Security
- EJB Declarative Security
- EJB Annotations
- EJB Programmatic Security

- [conygre] -
fill the gap

Java Messaging Services

- [conygre] -
r i l l t h e g a p

In this chapter we will introduce the Java Messaging Service. Messaging is an inherent part of JEE, and in this section we shall see the importance of messaging, and how messaging can be implemented as part of a JEE application.

Objectives

- Define messaging
- When to use messaging
- Types of messaging
- How to create and consume messages
- JMS 2.0 Differences
- Messaging Implementations

- [conygre] -
fill the gap

In this chapter we will define messaging, we shall see when it would be appropriate to use messaging. We shall also see the two core types of messaging, and how messages can be created, sent, and consumed. We shall finally see some of the various implementation of Java Messaging.

Part 1

Sending and Receiving Messages

- [conygre] -
r i l l t h e g a p

Firstly, we shall see how to send and receive messages. In part two, we shall look at the message itself in more detail.

Introduction

- Messaging provides a mechanism for the exchanging of data in a distributed and disparate enterprise environment
- The Java Messaging Service (**JMS**) is a specification within JEE which is then implemented by a third party
- The classes are located within the **javax.jms** package

- [conygre] -
fill the gap

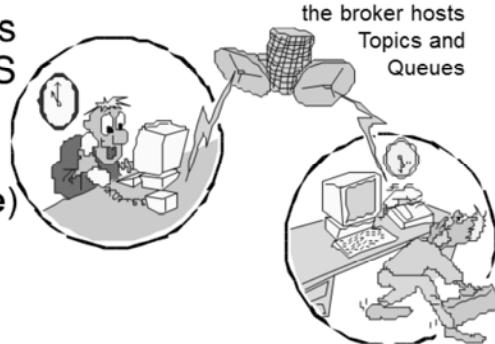
If you think about a desk top application written in Java, there will be a user interface. This user interface will have been written perhaps using the Swing APIs. There might for example be a button, and when the button is pressed, a dialog box might pop up. This kind of thing is achieved using the Java event model. This involves a class acting as a listener for events from the button, and when the button is pressed, the listener has a method called which will pop up the dialog box.

This Java event model is ideal for applications of this nature. But what about communication across virtual machines? What if the event was to be passed to an object in a different virtual machine? This would involve some kind of look up, and then a remote method call. The application may then have to wait for the remote method to return. What if the button does not know who needs to know about the event? What if the object that needs to know about the event does not happen to exist when the button is pressed? It may live on a different machine that happens to be offline when the button press occurs.

For disparate systems, the event model is not sufficient, and messaging is used instead. A message could be sent, and it is up to the receiver to collect that message when it is ready to receive it. Over the next few pages, we will expand on this idea of messaging.

Messaging Models

- There are two models for messaging in JMS
 - Publish / Subscribe (**Topic**)
 - Point to Point (**Queue**)



- [conygre] -
fill the gap

There are two models for messaging in JMS. One is where a message is sent to be received by one other object, like a letter would be sent and can only be opened by one person. This is called **point to point**. The other model is called **publish/subscribe**, where a message can be published like a notice on a notice board, and anyone else can subscribe, i.e. look at the message.

Messages are always managed by a broker. The broker will receive messages and the broker is where the messages are retrieved from by their intended recipients. The implementation is provided by your JMS provider.

The Broker

- Messages are always sent via a broker
- The broker is provided as part of the JMS implementation that you are using
- The Message Broker provides
 - Message Delivery
 - Message Storage
 - Connectivity



- [conygre] -
fill the gap

A client will need a connection to a topic or queue to enable it to send or receive messages. We will look at how this is done as we progress through the chapter. Since Topic and Queue are only interfaces, your JMS implementation will provide implementations of these for you. The broker is responsible for this connectivity, the delivery and the storage of the messages.

Persistence and Durability

- **Persistence**

- JMS brokers can support persistent or non-persistent messages
- A persistent message is stored in a database by the JMS implementation in case of failure so the message is not lost

- **Durability**

- A durable subscription is used when you wish to preserve messages published on a topic while the subscriber is not active

- [conygre] -
fill the gap

Brokers can be configured to support persistence. This means that the messages when sent are stored in a relational database until they reach the client. If the broker goes down for any reason, the messages have not been lost, and can still be retrieved by clients once the broker comes back online. If a broker does not support persistence, messages are effectively stored in memory, and if the broker fails, the messages will be lost.

Brokers can also be configured to be durable. This is most commonly done with Topics. Durability is where the messages are maintained by the Topic even when the subscribers are not active at the time that the message arrives into the Topic.

The configuration of both durable and persistent brokers will be implementation specific, and it normally involves entries in XML configuration files.

Note that Jboss and Wildfly use the term durable regarding queues to describe queues that do not persist messages after a server restart which in JEE terminology could be more accurately described as non-persistent.

Obtaining the Context on JBoss

- To obtain the context on Jboss, you will need to configure some properties

```
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory"); // Java class of context factory
env.put(Context.PROVIDER_URL, "http-remoting://localhost:8080"); // WILDFLY URL
env.put(Context.PROVIDER_URL, "remote://localhost:4447"); // JBOSS 7 URL
env.put(Context.SECURITY_PRINCIPAL, "sarah"); // username
env.put(Context.SECURITY_CREDENTIALS, "sarahsarah"); // password
```

- Note that WildFly and Jboss 7 use different remoting protocols and ports – pick the right one!

- [conygre] -
fill the gap

A set of environment properties are required for the JNDI lookup to work successfully. These properties can be set in a Properties object or a HashTable object. Once you have created the properties object, you can then use it when you create the InitialContext.

```
InitialContext context = new InitialContext(env);
```

Obtaining a ConnectionFactory

- Locate a **TopicConnectionFactory** instance or **QueueConnectionFactory** instance depending upon the type of messaging
- These implementations provide connections for Topics or Queues

```
// obtain a reference to the JNDI context
Context context = new InitialContext(env); // pass in the properties object
// look up a factory to provide a connection to a broker for us
TopicConnectionFactory factory = (TopicConnectionFactory)
    context.lookup("somePath/SomeTopicConnectionFactory");
```

- [conygre] -
fill the gap

To obtain a connection to a message broker, whether it be a topic or a queue, you will need a reference to a connection factory. From this factory, you will be able to obtain a connection. The factory type will either be a TopicConnectionFactory, or a QueueConnectionFactory. In the above example, JNDI is being used to obtain the reference to the factory. This is not the only way to do it, in the example below, using the Java Message Queue 1.1 implementation, we can do the following to obtain a factory.

QueueConnectionFactory factory = new QueueConnectionFactory();
or alternatively

QueueConnectionFactory factory = new QueueConnectionFactory(args);

Where args can be an array of String properties which can be set to define aspects of the factory. For example to define the port number and hostname.

Obtaining a Connection

- Obtain a **Connection** to the topic or queue
- For Topics use **TopicConnection**

```
// create a connection for a Topic  
TopicConnection connection =  
    factory.createTopicConnection(username, password);
```

- For Queues use **QueueConnection**

```
// create a connection for a Queue  
QueueConnection connection =  
    factory.createQueueConnection(username, password);
```

- The username and password is required for access to the JMS connection

- [conygre] -
fill the gap

Once the factory reference has been obtained, a connection to the broker can be created. This is done using one of two methods depending upon whether the factory is for Queue or Topic connections.

```
QueueConnection connection = factory.createQueueConnection();
```

```
TopicConnection connection = factory.createTopicConnection();
```

The return type is either a QueueConnection or a TopicConnection. These are sub interfaces of the Connection interface.

Messaging Sessions

- To send and receive messages, a session is required
- The session is a thread context in which you can send or receive messages

```
// create a TopicSession
TopicSession session = connection.createTopicSession
    (false, Session.DUPS_OK_ACKNOWLEDGE );
```

- The method **createTopicSession()** parameters
 - boolean - whether the session is transacted
 - int – a constant to denote what kind of acknowledgement is required, there are three possibilities which are covered later

- [conygre] -
fill the gap

To create or receive messages via a connection, you must have a session. The session is used for both the sending and the receiving of messages. The session acts as a thread context in which messages can be sent and received. There are a number of session interfaces which all extend the interface **javax.jms.Session**.

QueueSession – sessions for Queue brokers

TopicSession – sessions for Topic brokers

Sessions can also be transactional. If transactional support is wanted, you will use one of either

XAResourceSession

XATopicSession

These interfaces both extend XASession.

The sessions are created based upon connections. The method used in the above slide takes in two arguments. The first determines whether the session is transacted or not. The second is the acknowledgement mode. The value of which can be one of either:

Session.AUTO_ACKNOWLEDGE

Session.CLIENT_ACKNOWLEDGE

Session.DUPS_OK_ACKNOWLEDGE

Acknowledgement Modes

- **AUTO_ACKNOWLEDGE**
 - Session acknowledges a client's receipt of a message when the session has successfully received a message
- **CLIENT_ACKNOWLEDGE**
 - Client acknowledges a consumed message by calling the **Message.acknowledge()** method
 - Acknowledging a message acknowledges all messages that the session has consumed
- **DUPS_OK_ACKNOWLEDGE**
 - Session lazily acknowledges the delivery of messages
 - This is likely to result in the delivery of some duplicate messages if the JMS provider fails
 - Use of this mode can reduce session overhead by minimizing the work the session does to prevent duplicates

- [conygre] -
fill the gap

The three acknowledgement modes are listed and described above.

Obtaining the Topic or Queue

- Once a connection has been obtained, a Topic or Queue can now be located
- Locating the broker is typically done using JNDI

```
// locate the Topic (or Queue) itself  
Topic topic = (Topic) context.lookup("someLocation/someTopic");
```

- [conygre] -
fill the gap

Once a connection has been created, a reference to the broker can be obtained. In this example, again using JNDI. The JNDI name of the topic or queue will be dependent upon the configuration of your environment.

So for example, when using Java Message Queue 1.1, a Queue could be created as follows:

```
session = connection.createQueueSession(false,  
QueueSession.CLIENT_ACKNOWLEDGE );  
Queue queue = session.createQueue ("myNameForQueue");
```

This queue would then be accessible from another client, as long as they used the same name used here.

Sending or Receiving a Message

- To send or receive messages, a message producer or consumer is needed
- For Point to Point you need either
 - **QueueSender** or **QueueReceiver**
- For Publish/Subscribe you need either
 - **TopicPublisher** or **TopicSubscriber**

- [conygre]
fill the gap -

Now we have the Connection from the ConnectionFactory, and a Session, and a Topic or Queue, we can finally send or receive our messages. To send and receive, there are four key interfaces as can be seen in the above slide. Over the next few pages, we shall see how these can be used.

Obtaining Subscriber or Publisher

- For Publish/Subscribe systems with a Topic
 - a **TopicPublisher** or **TopicSubscriber** would need to be created

```
// obtaining a subscriber  
TopicSubscriber subscriber = session.createSubscriber (topic);  
// obtaining a publisher  
TopicPublisher publisher = session.createPublisher (topic);
```

- [conygre] -
fill the gap

In the publish/subscribe model, we have the TopicSubscriber for receiving messages, and the TopicPublisher for sending messages. Both are created on the session, and both involve a method that takes in a reference to the topic itself.

Obtaining Sender or Receiver

- For Point to Point with a Queue
 - A **QueueSender** or **QueueReceiver** would be required

```
// obtaining a sender  
QueueSender sender = session.createSender (queue);  
// obtaining a receiver  
QueueReceiver receiver = session.createReceiver (queue);
```

- [conygre] -
fill the gap

For a Queue, a QueueSender or QueueReceiver is required, and as with the topic, is created based upon a session with a reference to a specific queue.

Complete Example (1)

- Subscribing to a Topic example

```
// obtain a reference to the JNDI context
Context context = new InitialContext();
// look up a factory to provide a connection to a Topic for us
TopicConnectionFactory factory = (TopicConnectionFactory)
    context.lookup("somePath/SomeTopicConnectionFactory");
// locate the Topic
Topic topic = (Topic) context.lookup("someLocation/someTopic");
// create a connection for the Topic
TopicConnection connection = factory.createTopicConnection();
// create a TopicSession
TopicSession session = connection.createTopicSession
    (false, Session.DUPS_OK_ACKNOWLEDGE);

// subscribe to a topic
TopicSubscriber subscriber = session.createSubscriber(topic);
```

- [conygre] -
fill the gap

Above is complete example of the steps required to obtain a subscriber to a topic. Note the sequence of events.

1. Obtain the JNDI context.
2. Locate the TopicConnectionFactory.
3. Locate the Topic.
4. Obtain a connection.
5. Create a session.
6. Create a TopicSubscriber.

Complete Example (2)

- Sending to a Queue Example

```
// obtain a reference to the JNDI context
Context context = new InitialContext();
// look up a factory to provide a connection to a Queue for us
QueueConnectionFactory factory = (QueueConnectionFactory)
    context.lookup("somePath/SomeQueueConnectionFactory");
// locate the Queue
Queue queue = (Queue) context.lookup("someLocation/someQueue");
// create a connection for the Queue
QueueConnection connection = factory.createQueueConnection();
// create a QueueSession
QueueSession session = connection.createQueueSession
    (false, Session.DUPS_OK_ACKNOWLEDGE);

// obtain a sender
QueueSender sender = session.createSender(queue);
```

- [conygre] -
fill the gap

Here is another complete example, this time, creating a sender for sending messages to a queue. Note the sequence of events that need to happen – similar to the previous example.

1. Obtain the JNDI context.
2. Locate the QueueConnectionFactory.
3. Locate the Queue.
4. Obtain a connection.
5. Create a session.
6. Create a QueueSender.

Sending / Publishing a Message

- Once you have either a Publisher or a Sender, a message can be dispatched

```
// create a message (more on this later)
TextMessage message = session.createTextMessage("Hello");
// to send messages to a Queue
sender.send (message);
// to publish messages to a Topic
publisher.publish (message);
```

- [conygre] -
fill the gap

To send or publish a message, call a method on the broker. So if it is a Topic, call the publish() method, passing a reference to the message itself. If it is a Queue, then use the send() method. We will be discussing the anatomy of a message later in the course. Suffice to say for now that there are a number of different message types, and TextMessage is one of them, which is a Message type specifically for sending text based messages. These are commonly used for XML messages for example.

Receiving / Subscribing

- For a message to be received, a **MessageListener** needs to be defined
- A MessageListener is an interface containing the method
 - **public void onMessage (Message message)**
- This method is invoked when a message is received

- [conygre] -
fill the gap

For messages to be received by a subscriber or receiver, a MessageListener must be defined. If you are familiar with the Java 1.1 event model, then this will be familiar to you. A class is defined that must implement the MessageListener interface. This interface contains one method:

```
public void onMessage(Message message)
```

This method is invoked whenever a message is received. As well as defining the listener, you must also register the listener with the QueueReceiver or TopicSubscriber that you created earlier. An example is shown on the next page.

MessageListener Example

- Here is an example MessageListener

```
public class MessageListenerImpl implements MessageListener{
    public void onMessage(Message message){
        try {
            System.out.println(message.toString());
        } catch (JMSEException e) { System.out.println("Oops"); }
    }
}
```

- To register our listener with a subscriber or receiver, use the following methods

```
// point to point
receiver.setMessageListener(new MessageListenerImpl());
// publish / subscribe
subscriber.setMessageListener(new MessageListenerImpl());
```

- [conygre] -
fill the gap

Here we see an example of a MessageListener implementation, and the registration of the listener with a QueueReceiver or TopicSubscriber. This is done using the following method:

`setMessageListener(MessageListener listener)`

There is now one more thing to do before messages can be received. That is to call `start()` on the connection. See the next page for an example.

It is worth noting that using a MessageListener is asynchronous. For synchronous messaging, you can call

Message receive()

on your QueueReceiver or TopicSubscriber. This method will block until a message is received. There is an overloaded `receive(long timeout)` method which will timeout after the specified time.

Here is an example.

`// receiver is a QueueReceiver, or could alternatively be a TopicSubscriber`

Message receivedMessage = receiver.receive();

Starting to get Messages

- Lastly, to cause our connection to listen for new messages, you must call the **start()** method on the connection
 - this is the same for either messaging mechanism

```
// set the listener
receiver.setMessageListener(new MessageListenerImpl());
// now ensure that your messages will be received on the connection
connection.start();
```

- [conygre] -
fill the gap

The start() method will result in messages now being received asynchronously from the topic or queue. This method will throw a JMSException if there is some reason why it cannot start.

Summary of Steps

1. Locate a Topic / Queue Connection factory
2. Obtain a Connection from the factory
3. Locate a Topic / Queue
4. Create a session for the Connection
5. For sending, create a Sender or Publisher
6. For receiving
 1. Create a Receiver or Subscriber
 2. Define a MessageListener, registering it with receiver/subscriber
 3. Call start() on the connection to listen for messages

- [conygre] -
fill the gap

JMS 2.0 Simplifications

- From JEE 7 a simplified messaging API has been available
- You are still required to locate the destination (topic or queue) and the Connection Factory
- A new class called the **JMSContext** is now available to simplify the session, the connection and to allow sending

- [conygre] -
fill the gap

Using JMSContext to Send

```
Context context = new InitialContext();
QueueConnectionFactory factory = (QueueConnectionFactory)
    context.lookup("somePath/SomeQueueConnectionFactory");
Queue queue = (Queue) context.lookup("someLocation/someQueue");
QueueConnection connection = factory.createQueueConnection("sarah", "sarahsarah");
QueueSession session = connection.createQueueSession
    (false, Session.DUPS_OK_ACKNOWLEDGE );
QueueSender sender= session.createSender(queue);
TextMessage message = session.createTextMessage("Hello");
sender.send(message);
```

JMS 1.0

```
Context context = new InitialContext();
QueueConnectionFactory factory = (QueueConnectionFactory)
    context.lookup("somePath/SomeQueueConnectionFactory");
Queue queue = (Queue) context.lookup("someLocation/someQueue");
JMSContext context = factory.createContext("sarah", "sarahsarah",
    JMSContext.DUPS_OK_ACKNOWLEDGE);
context.createProducer().send(queue, "Hello from the producer");
```

JMS 2.0

- [conygre] -
fill the gap

Using the JMSContext wraps up creating a connection and setting up a session and then creating a message. Notice that you can just send text directly and you no longer need to create a text message object first.

The JMSContext gives you access to a JMSProducer object which can be used to send on messages

As you can see from the example above, sending using JMS 2.0 is much more straightforward than it is with JMS 1.0.

Using JMSContext to Simplify Receiving

- JMSContext also simplifies the setting up of a MessageListener
- To do this you use a **JMSConsumer**

```
JMSContext context = factory.createContext("sarah", "sarahrabbit");
JMSConsumer consumer = context.createConsumer(queue);
consumer.setMessageListener(new MyListener());
```

- [conygre] -
fill the gap

Part 2

The Message Itself

- [conygre] -
r i l l t h e g a p

We will give some attention to the message itself. In the remaining few pages we will explore the different kinds of messages that can be sent using JMS.

The Message

- The messages itself specifies aspects of the message including
 - **Header fields**
 - Specifying routing information
 - And optional **Properties**
 - A list of name value pairs used by applications to filter and decide on what actions to take with different messages
 - **Message Body**
 - The message itself

- [conygre] -
fill the gap

Regardless of the specific message type, header information is always included in the message. This header contains various pieces of standard information about the message, and also, optionally, additional user set properties.

The other key part of the message is known as the message body. This is the actual message that is being sent.

Message Headers

- There are a number of different message headers set by the client, sender, or broker, including
 - **JMSDeliveryMode**
 - **JMSDestination** – destination of the message
 - **JMSExpiration** – expiration time
 - **JMSPriority** – priority value between 0 and 9
 - **JMSReplyTo** – a reply address
 - **JMSTimeStamp** – time the message arrived at provider
- All of the above are set by the send method

- [conygre] -
fill the gap

Above are listed just some of the headers that are incorporated into a JMS message.

JMSDeliveryMode – Messages are either Persistent or Non-persistent. This refers to whether the message is to be persisted or not.

JMSDestination – the destination of the message.

JMSExpiration – The time the message is set to expire. If zero, the message does not expire. Expired messages should not be received by clients.

Although this is not guaranteed. This is stored as a long.

JMSPriority – the priority of the message. 0 is the lowest, and 9 is the highest. Typically, 0-4 is regarded as normal, and over 4 is regarded as a high priority.

JMSReplyTo – the reply to destination for this message.

JMSTimeStamp – the time a message was handed off to a broker to be sent. This is stored as a long.

These values are accessible using get/set methods in the Message class.

Message Properties

- As well as the data which is the message, you may also wish to send additional **properties**
- Properties are effectively optional header fields, some are standard, but you can also define your own
 - **Application** specific properties
 - **Provider** specific properties
- These are set *prior* to sending a message

- [conygre]
fill the gap -

In addition to the headers, messages can also have a number of properties set. These are in the form of name value pairs. There are a number of standard properties, as well the option for you or the JMS provider to add additional properties. A number of methods exist within the Message interface to enable access to these properties, such as:

Enumeration getPropertyNames()

void setXXXProperty(String name, XXX value)

XXX getXXXProperty(String name)

Where XXX is the type of property being set, e.g. boolean, int, float, Object etc.

Message Interface

- A message is an implementation of the **javax.jms.Message** interface
- This interface contains methods for getting and and, where appropriate, setting properties and headers
- There are a number of sub-interfaces for the different message types

- [conygre] -
fill the gap

The javax.jms.Message interface is the super-interface that all the specific message types then extend. These are also interfaces. The methods discussed on previous pages about headers and properties are all to be found in this Message interface.

Message Types

- A Message can be one of the following types
 - BytesMessage – a stream of uninterpreted bytes
 - MapMessage – a set of name / value pairs
 - ObjectMessage – a serialized Java object
 - StreamMessage – a stream of Java primitives
 - TextMessage – a String message
- This chapter focuses on TextMessage

- [conygre]
fill the gap -

The specific message sub-types are listed above. Here we will be looking at the TextMessage, but many of the principles we see for the TextMessage are also true for the other message types.

Creating a Message

- Messages are created using the **create[Type]Message()** method in QueueSession
- For a TextMessage set the text of the message using **setText()**
- Send the message using the **send()** method of the QueueSender

```
// create a message
TextMessage message =
    session.createTextMessage();
// set the text
message.setText("content");
// set a property (optional)
message.setIntProperty("myAge", 29);
// send the message
sender.send(message);
```

- [conygre] -
fill the gap

To create a message, there are **createXXXMessage()** methods for each of the specific types. The **create()** methods are found in the Session interface, so messages are always created on a specific session. An example of how to create a TextMessage is shown on the slide.

To create an ObjectMessage, I would have the following code:

```
// create a message
ObjectMessage message = session.createObjectMessage();

// set the object
message.setObject(someObjectReference); // must be Serializable
// send the message
sender.send(message);
```

Messaging Implementations

- All application servers come with a messaging implementation
 - They have to as it is part of Java EE
 - For example, JBoss 7+ and Wildfly incorporate Hornet MQ
- Various alternative implementations exist
 - Apache Active MQ (integrates with .NET)
 - MQ Series (IBM)
 - Open JMS

- [conygre] -
fill the gap

All Java EE compliant application servers will have an implementation of JMS built into them. Specific messaging products also exist, such as MQ Series from IBM, Apache Active MQ.

Monitoring Messages

- Hermes JMS is an open source product that allows you to monitor your JMS destinations
 - *“HermesJMS is an extensible console that helps you interact with JMS providers making it simple to publish and edit messages, browse or search queues and topics, copy messages around and delete them.”* – www.hermesjms.com



- [conygre] -
fill the gap

Most JEE servers have very limited capabilities when it comes to monitoring your topics and queues in real time. Hermes JMS is a console tool that allows you to view / edit / send / delete messages on topics and queues in a live environment.

Summary

- JMS
 - supports point to point and publish subscribe
 - is a set of interfaces implemented by a vendor
 - Since JEE7 has a simplified API
- Key interfaces include
 - Topic / Queue
 - TopicConnection / QueueConnection
 - TopicSession / QueueSession
 - QueueSender / QueueReceiver
 - TopicPublisher / TopicSubscriber
 - MessageListener
 - Message
 - JMSContext / JMSProducer / JMSConsumer (JMS 2.0 only)

- [conygre] -
fill the gap

Message Driven Beans

- [conygre] -
r i l l t h e g a p

Objectives

- Defining a Message Driven Bean (MDB)
- Creating an MDB
- Deploying an MDB

- [conygre] -
fill the gap

Defining a MDB

- MDBs enable messages to be sent asynchronously into a J2EE application
- They are the simplest type of EJB with only one class, which is an implementation class

- [conygre] -
fill the gap

Creating an MDB

- An MDB implements
 - javax.jms.MessageListener

```
package com.conygre.message;
import javax.jms.*;
import javax.ejb.*;
public class MyMessageBean implements MessageListener{
    public void onMessage(Message message) { /* process message here */ }
}
```

- Use annotations to set the properties such as
 - Destination type
 - Destination JNDI name
 - Acknowledgement mode

- [conygre] -
fill the gap

MDB Annotations

- Below is a code example with annotations

```
package com.conygre.message;
import javax.jms.*;
import javax.ejb.*;

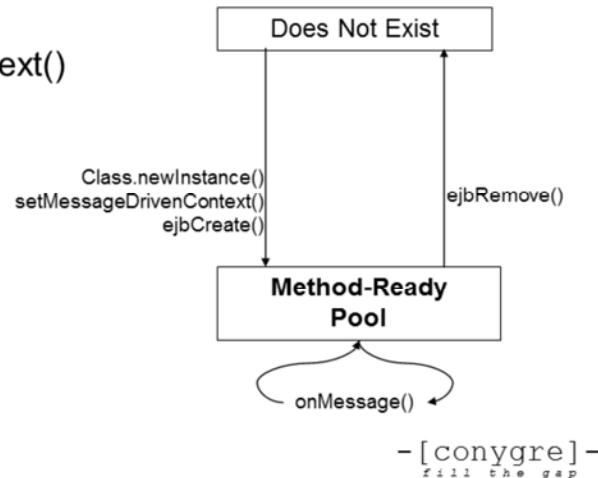
@MessageDriven(activationConfig= {
    @ActivationConfigProperty(propertyName="acknowledgeMode",
    propertyValue="Auto-acknowledge"),
    @ActivationConfigProperty(propertyName="destinationType",
    propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
    propertyValue="queue/testQueue")})

public class MyMessageBean implements MessageListener {
    ...}
```

- [conygre] -
fill the gap

MDB Lifecycle

- The four methods are called as shown
 - setMessageContext()
 - ejbCreate()
 - ejbRemove()
 - onMessage()



Configuration using ejb-jar.xml

- An alternative to annotations, is to configure the bean within ejb-jar.xml

```
<message-driven>
<ejb-name>MyMessageBean</ejb-name>
<ejb-class>com.conygre.MyMessageBean</ejb-class>
<transaction-type>Container</transaction-type>
<acknowledge-mode>Dups-ok-acknowledge</acknowledge-mode>
<message-driven-destination>
    <destination-type>javax.jms.Topic</destination-type>
    <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
<resource-ref>
    <res-ref-name>jms/TCF</res-ref-name>
    <res-type>javax.jms.TopicConnectionFactory</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
</message-driven>
```

- [conygre] -
fill the gap

Deployment

- The deployment of a message driven bean is the same as that for a session bean
 1. Add the bean to a jar file
 2. Copy the jar to
<JBOSS_HOME>\standalone\deployments

- [conygre] -
fill the gap



Summary

- Defining a Message Driven Bean (MDB)
- Creating an MDB
- Deploying an MDB

- [conygre] -
fill the gap

Building and Deploying Java EE Applications

Lab Instructions

Building and Deploying Java EE Applications	1
Lab Instructions	1
Introduction to your computer	3
Software.....	3
LabDirectories.....	3
InstallationFolders.....	3
Chapter 2 Introduction to JBoss / Wildfly	4
Aims.....	4
Create a Management User and Explore the Admin Console	4
Addendum for those new to Maven: Working with Maven	5
Aims.....	5
Create the Project.....	5
Create a simple Java HelloWorld	5
Build the Project using Maven.....	5
Reviewing the pom.xml File.....	6
Chapter 4: Creating a RESTful Service using JAX-RS.....	7
Aims.....	7
Part 1 Create the Project	7
Part 2 Configuring the REST Servlet.....	8
Part 3 Create a Basic Entity and Service	9
Part 4: Creating a Java Based RESTful Client.....	10
Chapter 5: Getting Started with JPA and Hibernate	12
Aims.....	12
Create a Hibernate Project.....	12
Creating the Database Tables (CHECK WITH INSTRUCTOR IF YOU NEED TO DO THIS)	12
Create a Mapped Entity Class.....	13
Creating the persistence.xml file.....	14
Creating a Test Application.....	14
Incorporating Relationships.....	15
Aims.....	15
Create the Annotated Track Entity	15
Update the CompactDisc Entity.....	15
Inserting a new Album.....	16
Queries.....	17
Detached Objects.....	18
Aims.....	18
Modifying Object Properties.....	18
Chapter 6: Getting Started with CDI	19
Aims.....	19

Create a CDI Project.....	19
Create some Beans.....	20
Create a Test Class.....	21
Real World Wiring.....	22
Aims.....	22
Part 1: Create a Project	22
Design the Interface.....	22
Create aFake.....	23
Create the Service Layer.....	23
Create a Working DAO	24
Chapter 9: Working with Session Beans	25
Aims.....	25
Part 1 Creating a Book Retriever bean	25
Part 2 Deploying the EJB to JBoss	26
Part 3 Testing your EJB with Arquillian.....	27
Chapter 10: Combining JAXRS, CDI, EJB, and JPA.....	29
Create theDataSource.....	29
Create the Project.....	29
Modify the DAO Class	31
Create theEJB.....	32
Create the REST API.....	32
Optional: Test the EJB using Arquillian.....	33
Chapter 14 Java Messaging Service (JMS 1.0 and JBoss 7)	34
Aims.....	34
Part 1 Create an Application User	34
Part 2 Creating a Message Producer	34
Part 3 Creating a Message Consumer.....	35
Part 4 Running your code	35
Part 5 Optional Using a Topic	36
Chapter 14 Java Messaging Service (JMS 2.0 and WildFly 8)	37
Aims.....	37
Part 1 Create an Application User and a Queue	37
Part 2 Creating a Message Producer	38
Part 3 Creating a Message Consumer.....	38
Part 4 Running your code	38
Part 5 Optional Using a Topic	39
Chapter 15 Message Driven Beans.....	40
Aims.....	40
Part 1 Creating the Message Driven Bean	40
Part 2 Test your client and MDB	40

Introduction to your computer

Software

The following software must be installed on your machine.

- Java Developers Kit Standard Edition 1.6.x or higher
- Wildfly
- MySQL 5.x
- Eclipse or IntelliJ
- Maven
- Tomcat 7 or higher

Lab Directories

During the workshop, you will be building a Java 2 Enterprise application comprising of Java Server Pages, and Enterprise Java beans. Folders will be created for these files as you work through the labs.

There are three folders that you will need to take note of:

C:\conygreJEE\labs – contains all the exercise files
C:\conygreJEE\solutions – contains solutions to the exercises
C:\conygreJEE\demos – contains demonstration code

Installation Folders

The installation location of the applications that you will be working with during this course are referred to as <XXX_HOME>, where XXX is the name of the application. For example, the following names will be referred to during the course:

- <WILDFLY_HOME>
- <MAVEN_HOME>

Each of these will be located in the folder structure like c:\Java\<product_name_version>.

Chapter 2 Introduction to JBoss / Wildfly

Aims

In this brief exercise, you will set up an admin user and application user for the server and review the admin console for the server. JBoss 7 and Wildfly 8 are very similar to each other so regardless of which server you are using you will see more or less the same things.

Create a Management User and Explore the Admin Console

1. To launch the server, from Windows Explorer, double click on <WILDFLY_HOME>\bin\standalone.bat.
2. Once the server has launched, using a Web browser, visit <http://localhost:8080> and you will see the welcome screen. You cannot do much yet however as you have not set up an admin user.
3. To create an admin user, from Windows Explorer, double click on <WILDFLY_HOME>\bin\add-user.bat.
4. When asked what type of user, select management user.
5. When prompted about the realm, leave the option blank and hit enter.
6. When prompted for a username, enter a name (don't forget it!)
7. When prompted for a password, enter a password (JBoss will warn you if you use the username). Do not forget this as you will require it later.
8. Re-enter the password when prompted and then when asked to confirm, confirm your choices.
9. Now revisit the browser, and click the link to the admin console. You will be prompted for your username and password which you set up in the previous steps.
10. Finally, explore the admin console and see the kinds of options that are there. Later in the course you will use this console to set up a datasource, and also to set up and view your messaging settings.

Addendum for those new to Maven: Working with Maven

Aims

In this first exercise you will create a project in Eclipse using the Maven Eclipse plugin called Maven Integration for Eclipse.

You may wish to use an alternative development environment such as IntelliJ. You may do that throughout the training, but check with your instructor to ensure that they will be able to support you and that you may install it on the system you are working with.

Create the Project

1. First make sure that your Eclipse environment is set to use an external Maven installation. In Eclipse, click **Window / Preferences / Maven / Installations**. If the Embedded option is selected, click **Add**, and then browse to the installation folder C:\Java\apache-maven-x.x.x and then ensure that this external Maven installation is selected.
2. Launch **Eclipse** and at the **Workspace Launcher**, click **Browse**, and select the Workspace c:\Conygre\CDI\labs\workspace. Click **OK**.
3. If the Welcome screen appears, close it.
4. In the **Project Explorer**, right click and then click **New** and then click **Other**.
5. At the **New** dialog, expand **Maven** and then select **Maven Project**. Click **Next**.
6. At the New Maven Project dialog, select Create a simple project and click Next.
7. At the **New Maven Project** dialog, enter the following information:

Group ID	com.conygre.training
Artifact ID	Basicmaven
Name	MyFirstMaven

8. Click **Next**.
9. At Select Additional Dependencies, click **Finish**.

Create a simple Java HelloWorld

1. In the **Project Explorer**, expand the Project and you will see a set of folders. If you are used to Eclipse you will notice that a Maven based project has a different folder structure.
2. Right click on the **src/main/java** folder, and click **New**, and then click **Class**.
3. Use the simple class Wizard to create a new Java class called **HelloWorld** in a package called com.conygre.maven, with a main method.
4. Edit the code to print out the text “*HelloWorld*”.
5. Right click in the code, select **Run As**, and then click **Java Application**.
6. The output will appear below in the console as normal.

Build the Project using Maven

You will now build and run the project using Maven.

1. In the **Project Explorer**, Right click on the **basicmaven** project and click **Run As**, and then click **Maven package**.
2. Watch the Console as Maven builds and then packages up your project.
3. When it is finished, in the **Project Explorer**, expand the **target** folder and you will see a new jar file has been created called **basicmaven-0.0.1-SNAPSHOT.jar** which is the packaged application.

Reviewing the pom.xml File

1. In the Project Explorer, double click on the **pom.xml** file found in the root of your project.
2. The POM file then opens in the POM editor, you can see that from here you can edit the various sections of the XML file in a visual way. We will be using this editor later in the course. Review the various sections.
3. Close all open files in Eclipse.

Chapter 4: Creating a RESTful Service using JAX-RS

Aims

In this exercise, you will create a basic RESTful CRUD project that will provide create / read / update / delete functions.

Note that the Maven dependencies are slightly different for Tomcat or JBoss/Wildfly as are the web.xml entries, so you must ensure that you use the correct entries depending upon your server. This is because Tomcat does not ship with any JAXRS implementation, but JBoss does, and if you add libraries to your application and then deploy to JBoss it will fail due to conflicts.

Part 1 Create the Project

1. In the **Project Explorer**, right click and then click **New** and then click **Other**.
2. At the **New** dialog, expand **Maven** and then select **Maven Project**. Click **Next**.
3. At the New Maven Project dialog, select Create a simple project and click Next.
4. At the **New Maven Project** dialog, enter the following information:

Group ID	com.conygre.training.jee.jaxrs
Artifact ID	CompactDiscCRUDService
Packaging	War

5. Click **Finish**.
6. *FOR TOMCAT* add the dependencies to the pom.xml as shown below:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.13</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>1.13</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.13</version>
</dependency>
```

For JBoss/Wildfly, add the following dependency only:

```
<dependency>
    <groupId>org.jboss.spec</groupId>
    <artifactId>jboss-javaee-6.0</artifactId>
    <version>1.0.0.Final</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

Part 2 Configuring the REST Servlet

1. Using the Project Explorer, navigate to src/main/webapp/WEB-INF (if the folder structure is not present, then create it).
2. Within WEB-INF, create a blank web.xml configuration file (there is no wizard for this file, so you have to manually create it as a blank file).
3. *FOR TOMCAT*, Add the following to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
  app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>REST App</display-name>
  <servlet>
    <servlet-name>RESTServlet</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>com.conygre.jee.jaxrs</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>RESTServlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

For JBoss/WildFly, add the following to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" 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">
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

This will now have set your web application up so that all requests to your application that are /rest/* will trigger the REST servlet which will then trigger your service classes.

The rest of the exercise is the same for both JBoss and Tomcat.

Part 3 Create a Basic Entity and Service

1. In the src/main/java folder, create a basic version of the CompactDisc class in a suitable package with a few basic properties including an identifier. Implement Serializable and mark it with the @XmlRootElement annotation.
2. Create a new holding class for your list. The class will have one property which is a Collection of CompactDiscs. It will also have a get/set method for the Collection. Ensure that the holding class is also annotated @XmlRootElement.

3. In the src/main/java folder, create a class called CompactDiscCRUDService. If you are deploying to Tomcat, the package MUST MATCH the package name used in web.xml in Part 2. If you are using JBoss/Wildfly, the package name can be anything of your choosing.
4. Annotate the class appropriately with a suitable path of your choice using the @Path annotation.
5. Add a static property which is a Map of CompactDiscs. Put some objects into it using the id field as the key.
6. Initially provide a method to get all the CDs as a list. The return type will be your holding class. Don't forget to annotate with @GET. Ensure that the MIME type is set to JSON for now.
7. Before we go any further, you will test that this method works. If it does, then all your configuration is correct and you will be able to proceed confidently to the next steps. So right click on the project, click **Run As**, and the click **Maven Install**. This will build and package up the project.
8. Browse to the target directory and you will see a war file. Using Windows Explorer, copy this file to <CATALINA_HOME>\webapps or <WILDFLY_HOME>/standalone/deployments.
9. Launch Tomcat by using Windows Explorer, navigate to <CATALINA_HOME>\bin and then double-click on startup.bat. Alternatively launch JBoss by running <WILDFLY_HOME>\bin\standalone.bat.
10. Check for any errors in the launch console before going any further.
11. Now launch a browser, and navigate to
<http://localhost:8080/<NAMEOFWAR>/rest/<PATH USED IN SERVICE>>
12. If you have done the exercise correctly, you will see a JSON list of compact discs. If it has failed, then the errors in the console window may help you.
13. Once this has worked correctly, continue to add methods to your service to allow the retrieval of individual CDs, updates, creates, and deletes.
14. Build and deploy this final version of the application. You will now test the new methods using a client application.

Part 4: Creating a Java Based RESTful Client

1. Create a new Maven project called CompactDiscRESTClient and ensure that it has the following dependencies:

```
<dependencies>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>javax</groupId>
```

```
<artifactId>javaee-api</artifactId>
<version>7.0</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.2</version>
</dependency>
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.json</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.0</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-moxy</artifactId>
    <version>2.2</version>
</dependency>
</dependencies>
```

2. Create a JUnit test and add an empty @Test method for each of your REST methods.
3. Mark each method with @Ignore for now. You will implement them one by one and then remove the @Ignore as you complete them.
4. Begin with the GET method for the list of CompactDiscs. Using the notes as a guide, implement a suitable test method to ensure that the Get request works correctly.
Remember, you already know the service works because you tested it in the browser!
5. Once that one is complete, work your way through the other methods, using the notes as a guide.

Chapter 5: Getting Started with JPA and Hibernate

Aims

In this exercise, you will create a bean mapped to a MySQL database provided for you that contains some data about Compact Discs (remember them!). The basic Schema consists of a simple one to many relationship between a compact_discs table and a tracks table.

Create a Hibernate Project

1. In the **Project Explorer**, right click and then click **New** and then click **Other**.
2. At the **New** dialog, expand **Maven** and then select **Maven Project**. Click **Next**.
3. At the New Maven Project dialog, select Create a simple project, and click **Next**.
4. At the **New Maven Project** dialog, enter the following information:

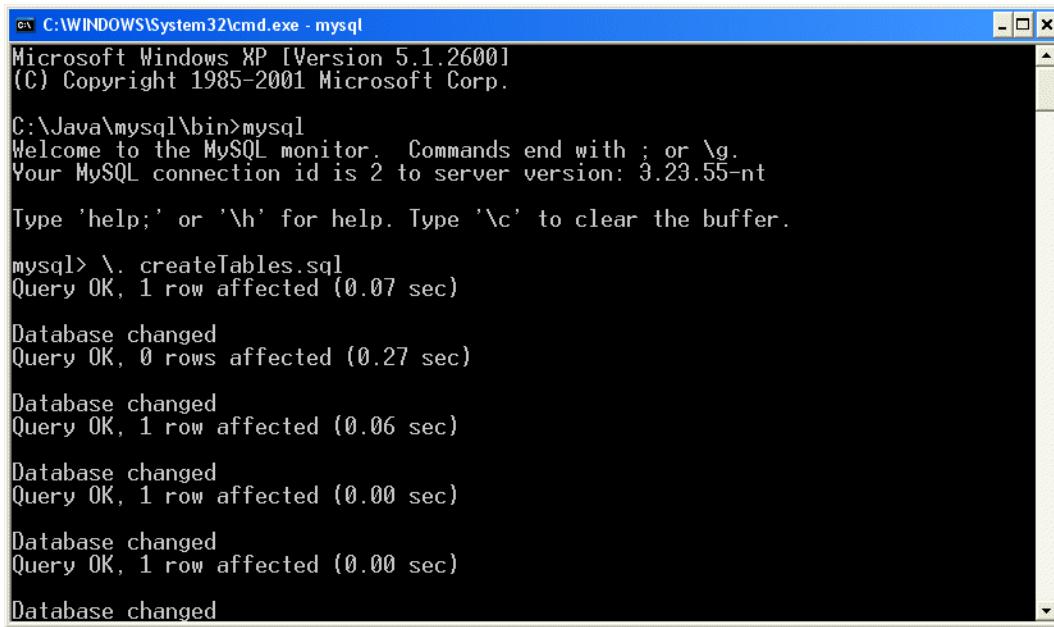
Group ID	com.conygre.training
Artifact ID	Basichibernate
Name	BasicHibernate

5. At the **Select Additional Dependencies** dialog, we can now really begin to see the benefits of using Maven. Click **Add**, and then type “mysql”. Locate **mysql mysql-connector-java**, expand and select version 5.1.27 and then click **Add**.
6. Click **Add**, and then type “hibernate”. Locate **org.hibernate hibernate-entitymanager**. Expand and select version 4.1.4.Final, and then click **Add**.
7. Click **Add**, and then type “hibernate”. Locate **org.hibernate hibernate-validator**. Expand and select version 4.3.0.Final and then click **Add**.

Creating the Database Tables (CHECK WITH INSTRUCTOR IF YOU NEED TO DO THIS)

Note that you may not need to complete this step as depending upon network capability, we may have provided a database in the cloud.

1. MySQL should be installed and running on your machine. The script for creating the Compact Disc tables can be found in <LAB_HOME>\labs\mysql\createTables.sql, so launch a command prompt in this folder.
2. Type mysql -u root -p and press enter.
3. Enter the database password, which if you are using the course, will be c0nygre.
4. To run the script, enter ‘.\ createTables.sql’. See Figure 1.
5. To confirm the tables have been created, in the console, type select * from compact_discs; to see the listing, and then select * from tracks; to see the second listing.



```
C:\WINDOWS\System32\cmd.exe - mysql
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Java\mysql\bin>mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.55-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> \. createTables.sql
Query OK, 1 row affected (0.07 sec)

Database changed
Query OK, 0 rows affected (0.27 sec)

Database changed
Query OK, 1 row affected (0.06 sec)

Database changed
Query OK, 1 row affected (0.00 sec)

Database changed
Query OK, 1 row affected (0.00 sec)

Database changed
```

Figure 1 Running MySQL

Create a Mapped Entity Class

1. Return to Eclipse, and in the **Project Explorer** right click on the **src/main/java** folder and click **New** and then click **Class**.
2. Create the class with the name **CompactDisc** in the package **com.conygre.training.entities**.
3. Add the following properties along with the get/set methods
(NOTE: Do not use Primitive types as they cannot be null and the table values might be null for some of these columns).

Id	Integer
Title	String
artist	String
price	Double

4. Add the annotation to specify that the class is an entity, and then the annotation specifying the table that you are mapping it to.
5. Ensure that the class now implements the **Serializable** interface.
6. Add annotations to each of the properties specifying which column in the database it maps to.
7. Add the necessary annotations to enable the id column to be identified and to be auto-generated from the database.

Creating the persistence.xml file

The persistence.xml file needs to be placed into the META-INF directory of src/main/resources. A partly completed file has been provided for you.

1. Using Windows Explorer locate <LAB_HOME>\labs\hibernate\persistence.xml, and drag the file into your Eclipse project explorer **src/main/resources/META-INF** folder.
2. Set the name of the persistent unit to be conygrePersistenceUnit.
3. Set up the database connection details as specified in the comments in the file.

Creating a Test Application

1. In your **src/main/java** folder, create a new Java class called TestCompactDiscs.
2. Within a main method, using the notes as a guide, write suitable code to retrieve a compact disc by its ID. Specifically, your code will need to:
 - a. Create an EntityManagerFactory
 - b. Create an EntityManager
 - c. Retrieve and start a transaction
 - d. Look up a CompactDisc
 - e. Commit the transaction and close EntityManager.
3. Test your application.

When you run it, the output will be some warnings about Log4J and then (hopefully!) you're working output. Log4J significantly makes debugging Hibernate applications a great deal easier, so you will now add a Log4J configuration file into the application.

4. Using Windows Explorer, copy <LAB_HOME>\log4j\log4j.properties into your **src\main\java** folder and rerun the application. You will now see far more detailed output, and if your application didn't work, you will have a much clearer idea of why when you read the logging messages.
5. If you have time, experiment by adding your own new albums.

Incorporating Relationships

Aims

You will now create a class for the track table, and add in the annotations for the relationship between the compact disc and the track.

Create the Annotated Track Entity

1. Return to Eclipse, and in the **Project Explorer** right click on the **src/main/java** folder and click **New** and then click **Class**.
2. Create the class with the name **Track** in the package **com.conygre.training.entities**.
3. Add the following properties along with the get/set methods.

Id	Integer
Title	String
Disc	CompactDisc

4. Add the annotation to specify that the class is an entity, and then the annotation specifying the table that you are mapping it to.
5. Ensure that the class now implements the **Serializable** interface.
6. Add annotations to each of the properties specifying which column in the database it maps to.

For the disc property it needs to know which CompactDisc the track is on.

```
@JoinColumn (name="cd_id", referencedColumnName="id", nullable = false)
@ManyToOne
private CompactDisc disc;
```

Update the CompactDisc Entity

1. Now open the **CompactDisc** class and add a new property of type **List<Track>** called **trackTitles** complete with get/set methods.
2. Annotate the List as follows:

```
@OneToMany(mappedBy="disc")
private List<Track> trackTitles = new ArrayList<Track>();
```

3. Add a new method to the CompactDisc class to allow users to add a track. In this method, you must also set the Track to know about the CompactDisc that it is on, so you can do this with:

```
public void addTrack(Track t) {
    t.setDisc(this);
```

```
    trackTitles.add(t);  
}
```

4. Open the **TestCompactDiscs** class and update the code to retrieve CD number 16 (which has tracks in the database), and then retrieve the track information and display it. A suitable query is shown below:

```
Query allSpiceGirlTracks = em.createQuery("select t.title from Track t where t.disc.id = 16");
```

5. Run the application to see it working.

Inserting a new Album

1. Create a new CD object and set its properties. Add some Track objects to the CD.
2. Persist the CD. Do the tracks go into the database? (you can check using the MySQL command line tool). If not, why not.
3. In the CompactDisc class, update the Annotation on the Track collection to cascade. Try adding an album again with tracks. Does it persist the tracks this time?

```
@OneToOne(mappedBy="disc", cascade={CascadeType.MERGE, CascadeType.PERSIST})  
private List<Track> trackTitles = new ArrayList<Track>();
```

Queries

In this exercise you will be creating some JPL queries.

1. Open the **TestCompactDiscs** class and add code to do the following:
 - a. Retrieve all CDs where the artist name starts with the letter ‘S’.
 - b. Retrieve the number of CDs.
 - c. Retrieve all CDs in a list in alphabetic order by title.
 - d. Retrieve all the tracks by the Spice Girls.
 - e. All tracks names and the album name for album with id of 16.
 - f. Create a parameterized query that will allow you to specify a title and the matching CDs will be returned.
2. Test your examples, and review the generated SQL that is output by Log4J.

Detached Objects

Aims

In this exercise, you will see for yourself how the lifecycle of our objects can affect how your applications behave.

Modifying Object Properties

1. In your existing project, create a new Java class with a main method called **LifecycleTest**.
2. Add the code required to begin a new transaction from an entity manager using the persistence context created earlier.
3. Add a few blank lines and then add the code to commit the transaction and close the entity manager.
4. In the gap between the blank lines, retrieve CD number 12 and then change the title to something else simply using the setTitle method on the CompactDisc.
5. Run the code and then query the database using the MySQL client to see if it has changed.

You will see from the console that when you ran the code an update was run against the database and the value in the table has changed. This was because the object was in a **managed** state.

6. Now move the call to setTitle to below the closing of the entity manager. This will not now update the database. You can try it if you like to see it for yourself. Run the code, and the database will not change. After the entity manager is closed the object becomes **detached**.
7. After the code closing the entity manager, open a new entity manager and get a new transaction.
8. Now merge your modified CompactDisc object and close the new entity manager and transaction. See if the database has changed. You will now see that it has. This is because a **detached** entity has become **managed** again.

Chapter 6: Getting Started with CDI

Aims

In this exercise, you will create a basic CDI application that uses a simple set of configured beans.

Create a CDI Project

1. In the **Project Explorer**, right click and then click **New** and then click **Other**.
2. At the **New** dialog, expand **Maven** and then select **Maven Project**. Click **Next**.
3. At the New Maven Project dialog, select Create a simple project, and click **Next**.
4. At the **New Maven Project** dialog, enter the following information:

Group ID	com.conygre.training
Artifact ID	Basiccdi
Name	BasicCDI

5. Click **Finish**.
6. Right click on the project, click **Maven** and then click **Add Dependency**. Type org.jboss.weld into the Enter group id... field, and then select expand the **org.jboss.weld weld-core** search result. Select version **2.2.4.Final** and click **OK**.
7. Repeat the previous step, and this time select **org.jboss.weld.se weld-se**. This time select version **2.2.4.Final**.

This will automatically download and add all the required jar files to your project. The resulting pom.xml is shown over the page:

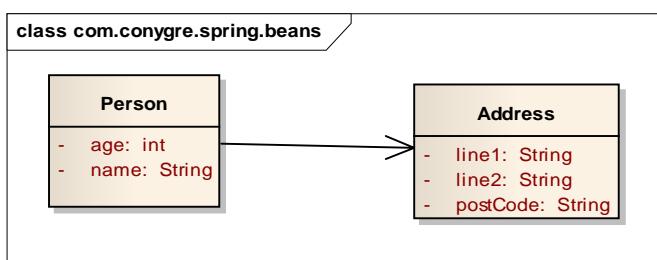
```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.conygre.training</groupId>
  <artifactId>basiccdi</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>BasicCDI</name>
  <dependencies>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core</artifactId>
      <version>2.2.4.Final</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld.se</groupId>
      <artifactId>weld-se</artifactId>
      <version>2.2.4.Final</version>
    </dependency>
  </dependencies>
</project>

```

Create some Beans

1. Use your IDE to create two simple Java classes called com.conygre.beans.Person and com.conygre.beans.Address that follows the simple UML diagram below:



2. Add getters and setters to each property using the Eclipse shortcuts.
3. Right click in the **src/main/resources** folder, click **New**, then click **Other**, and in the **New** wizard, expand **CDI**, and then select **File beans.xml**. If your IDE does not have a wizard to create beans.xml, copy the empty starter file from <LAB_HOME>\cdi.
4. Note the precompleted name "*META-INF/beans.xml*". CDI applications will not work without this, even if you do not require any entries. Click **Finish**.
5. We will not use the XML file initially, but you have now seen the XML file that can be used to wire up your beans.

6. Open the Person class, and annotate the private address property with the `@Inject` annotation.
7. Open Address.java, and provide some default values for the 3 fields line1, line2, and postCode.

That is enough to allow a basic CDI application to be created that will use your beans.

Create a Test Class

1. Within the java/main/java folder create a new Java class called PersonTest with a main method.
2. Since we are outside of a JEE container, we will need to create the CDI container programmatically. Add the following lines of code to your main method:

```
WeldContainer weld = new Weld().initialize();
Person p = weld.instance().select(Person.class).get();
```

Weld is the name of the JBoss implementation of CDI. Here we are using the initialize() method to get an instance of the Weld container. We are then using the instance() method to retrieve an instance of the class specified from the container.

3. To find out if the Address has been injected successfully into the Person, use the p reference to obtain the first line of the persons address and print to the console.
4. Run the PersonTest class and you will see the address printed to the console.

Real World Wiring

Aims

This exercise is designed to be a bit more substantial than the previous exercises. In it you will create a DAO interface, a DAO mock, and a client class. All will be configured using CDI.

The exercise will bring together both CDI and Data Access Objects written using the JPA.

The instructions for this exercise are not so detailed as previous exercises everything you are required to do has been covered in previous chapters.

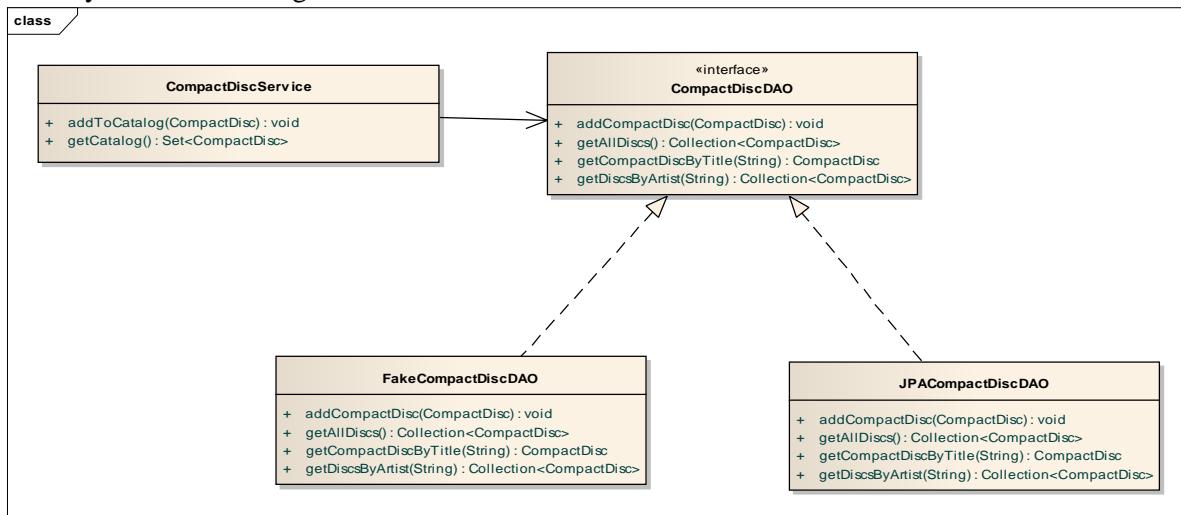
Part 1: Create a Project

1. Create a New Maven Eclipse project called **CompactDiscDAO** that will support both CDI beans and also Hibernate with the JPA.

Hint: Quick way is to copy the dependencies from the pom.xml files in your Hibernate and CDI projects into the pom.xml of the new project. The pom.xml file for this new project will have the dependencies from the CDI and Hibernate projects combined.

2. Set the project up so that it has your **CompactDisc** and **Track** entities along with the **persistence unit**. For now, simply copy and paste the files across using the Project Explorer.
 3. Create a blank beans.xml file as you did in the previous exercise.

Below is a UML model of the classes and interfaces that you will be creating in this exercise which you can use as a guide:



Design the Interface

The first part will involve designing a suitable DAO interface for the CompactDisc entity. You will then create a mock object that fakes reads, inserts, and updates. The mock object will then be wired up as a CDI bean.

4. Create a Java interface as shown below in the package com.conygre.training.dao.

```
public interface CompactDiscDAO {  
  
    void addCompactDisc(CompactDisc disc);  
    CompactDisc getCompactDiscByTitle(String title);  
    Collection<CompactDisc> getDiscsByArtist(String artist);  
    Collection<CompactDisc> getAllDiscs();  
}
```

Create a Fake

You will now create a fake implementation of the DAO interface that simply works with dummy data provided from a CDI beans.xml.

5. Create a new Java class called com.conygre.training.dao.fake.FakeCompactDiscDAO that implements the interface from the previous step.
6. Add a Set<CompactDisc> property called library, and then add some simple logic to implement the methods found in the interface that work against the set.
7. Finally, put a few instances of your CompactDisc class into the set representing some of your favorite albums.

Create the Service Layer

You will now create a sample service layer which will add CDs to the catalog and also allow users to search the catalog by title or artist. The client will be written so that it can work with the Mock or the real DAO implementation.

8. Create a new class called com.conygre.training.services.CompactDiscService.
9. Add a property for the DAO which will be of type CompactDiscDAO. You will need a set method so that CDI can inject it. You can mark it using @Inject so the CDI container injects an appropriate object.
10. Add the following methods:

```
addCDToCatalog(CompactDisc cd)  
getCatalog()
```

11. In the **addCDToCatalog** method, check the CD is not already there, and then if it isn't add it to the catalog.

Hint: You could use your IDE to generate the equals() and hashCode() methods in the CompactDisc class to help you to check if they are there.

12. In the **getCatalog** method, retrieve all the CDs and then return them.
13. Finally, test your service bean either in a JUnit test or a basic main method.

Create a Working DAO

You will now create a working DAO to replace the Fake DAO.

14. Create a new class called com.conygre.training.dao.jpa.JPACompactDiscDAO that implements the **CompactDiscDAO** interface.
15. Annotate it with the @Alternative annotation.
16. Within the methods add the necessary code to add a compact disc and then to search by title.

We will now need to replace the FakeCompactDiscDAO with the JPACompactDiscDAO. The problem you have currently is that there are now two classes that implement the CompactDiscDAO interface. A mechanism is required so that the CDI implementation can identify which one to use.

17. Open the beans.xml file and add the following entry:

```
<alternatives>
    <class>com.conygre.training.dao.jpa.JPACompactDiscDAO</class>
</alternatives>
```

18. Now rerun your service test class and it should work but this time the real DAO is replacing the fake DAO. Do you see how easy it is using CDI to switch from one implementation to another?

Chapter 9: Working with Session Beans

Aims

In this exercise you will create a basic session bean using annotations to configure the deployment information.

Part 1 Creating a Book Retriever bean

You will create a session bean acts as a book library. You give the library bean a book ID and it will return to you a book object.

1. Using Eclipse, create a new Maven Project.
2. Set the Group ID to be com.conygre.training.ejb.
3. Set the Artifact ID and name to both be BookRetrieverEJB.
4. Set the packaging type to be ejb (this will not be an option in the drop down, so you will need to type it).
5. Add the following dependency to the Maven project either using the Eclipse **Add Dependency** option, or by manually editing the pom file.

```
<dependency>
    <groupId>org.jboss.spec</groupId>
    <artifactId>jboss-javaee-6.0</artifactId>
    <version>1.0.0.Final</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

6. Add the following build entry immediately after your closing dependencies element. This forces Maven to compile with Java 6 and use EJB3.1 Without it you will get Maven errors as it will try and use Java 5 and EJB 2.x.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
                <showDeprecation>true</showDeprecation>
                <showWarnings>true</showWarnings>
                <fork>true</fork>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-ejb-plugin</artifactId>
```

```
<inherited>true</inherited>
<configuration>
  <ejbVersion>3.1</ejbVersion>
</configuration>
</plugin>
</plugins>
</build>
```

7. Right click on the project, and then click **Properties**.
8. In the properties dialog, select the **Facets** option.
9. Check that the EJB Module is set to version 3.1. If not, change it.
10. Check that the Java version is set to Java 6. If not, change it.
11. Click **Apply**.
12. Expand the **src\main\resources\META-INF** folder and remove any ejb-jar.xml that may have been created.
13. Using Windows Explorer, drag the following files directly into your Eclipse projects **src\main\java** folder:
<LAB_HOME>\labs\sessionbeanEJB3\src\Book.java
<LAB_HOME>\labs\sessionbeanEJB3\src \BookRetriever.java.
14. Modify the package names using Eclipse, and then review the code that has been provided. The Book class represents the data returned by the EJB, and the BookRetrieverEJB will become the EJB.
15. In the Book class, change it to implement the Serializable interface so that it can be used by remote clients.
16. In the BookRetrieverEJB class, add appropriate annotations to specify that this bean is a stateless session bean.
17. For EJB 3.1 the bean is actually complete already. There is nothing else required. However ,you will not create the optional interface.
18. Create a new Java interface called BookRetriever.java in the same package as the EJB.
19. Add an abstract method for the getBookByID method, ensuring that it has the same signature as the one in the BookRetrieverEJB class.
20. Annotate the interface specifying that it is a remote interface.
21. In the BookRetrieverEJB class, specify that it implements the BookRetriever interface.

Part 2 Deploying the EJB to JBoss

1. To run on the JBoss server, right click on the project and click **Run As** and then click **Run on Server**.
2. Select the JBoss 7 server as the choice and browse to the installation of JBoss 7 so Eclipse knows where it is installed.
3. Once the bean is deployed, launch your preferred browser and visit <http://localhost:9990>
4. Log in with admin/admin.
5. At the Web page, click on the Deployments link and verify that your EJB project has successfully deployed.

Part 3 Testing your EJB with Arquillian

You will now test that the EJB works as expected using the Arquillian In-container testing framework.

Adding Arquillian to the Project

1. Open the POM file and add the following dependencies:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.10</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.jboss.arquillian.junit</groupId>
<artifactId>arquillian-junit-container</artifactId>
<version>1.0.3.Final</version>
<scope>test</scope>
</dependency>
```

2. To create a test class, right click on the src\main\test folder and click **New** and then click **Other** and then expand **Java / JUnit** and then click **JUnit Test Case**.
3. At the **New JUnit Test Case** dialog, set the version to be 4, the package to be com.conygre.ejb.test, and the name to be BookRetrieverTest. Click **Finish**.
4. At the top of the JUnit test case class, add the @RunWith(Arquillian.class) annotation, importing the class and annotation.
5. Declare a reference to a BookRetriever and annotate it with @EJB which will allow it to be injected by Arquillian.

To get the EJB deployed, Arquillian dynamically creates a JAR containing the EJB and deploys it to the server. The way it does that is via a static method annotated with @Deployment. This method uses the ShrinkWrap API to dynamically create a JAR.

6. Add a static method to your test class that contains the following:

```
@Deployment
public static JavaArchive createDeployment() {
    return ShrinkWrap.create(JavaArchive.class, "test.jar")
        .addClasses(BookRetrieverEJB.class, Book.class, BookRetriever.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
}
```

7. Add a unit test method that will check that when the bean is invoked retrieving the album with the index number of 2, the return value is “Linux Desk Reference”.

If you look at this code you will see that it is putting together a JAR file with the specified classes and an empty beans.xml CDI deployment descriptor.

8. To run the test, the test case runs on the JBoss server as it requires a container that can host the EJB. To facilitate this a profile needs to be created in the pom file. We have provided an entry, so using Windows Explorer, open the file <LAB_HOME>\labs\arquillian\profilePOMSnippet.txt in a text editor, and paste the contents into your project POM file just underneath the *packaging* element.
9. To run the test successfully, Arquillian needs to know where JBoss is installed. To do this Arquillian has a config file called arquillian.xml. This file is provided for you and it needs to be placed in src/test/resources. So copy <LAB_HOME>\arquillian\arquillian.xml into src/test/resources. Review the file and change the location of JBoss if you have it installed somewhere else.
10. To switch the project to use the profile, right click on the project, click **Maven**, and then click **Select Maven Profiles**. At the dialog box, select the profile you added and then click **OK**.
11. Run the test by right clicking and then select **Run As / JUnit Test**. If set up correctly, you will see green as the test passes.

Chapter 10: Combining JAXRS, CDI, EJB, and JPA

You will now create a session bean that will work with an entity manager to return data to client applications.

Create the DataSource

1. Using Windows Explorer, copy the <LAB_HOME>\labs\jdbc\ mysql-connector-java-5.1.x-bin file into <WILDFLY_HOME>\standalone\deployments.
2. We will need to set up a DataSource on the WildFly Server. So start the WildFly server through the Server view in Eclipse or by double clicking on <WILDFLY_HOME>\bin\standalone.xml .
3. In a browser, login to the server admin console at <http://localhost:8080> using admin/admin.
4. Click on the **Configuration** tab and then click **Datasources** and click **Add**.
5. Set the name to be MySQL and set the JNDI name to be java:mySQL.
6. Click Next. Choose the MySQL driver JAR and click Next.
7. Set the URL to be jdbc:mysql://localhost:3306/conygre. Set the username to be root and the password to be c0nygre. If you are using a remote database, then use the appropriate connection settings for that.
8. Click **Done**, and then click **Enable** to start the connection pool.

Create the Project

1. To create this project we will actually be modifying our CompactDiscDAO project so it can be deployed to an EJB container (JBoss) instead of running standalone. So you can have both versions of this project to contrast. In the **Project Explorer**, Right click on the **CompactDiscDAO** project and then click **Copy**.
2. In the **Project Explorer**, right click again and click **Paste**. Set the new name to be **CompactDiscEnterprise**.
3. Open the POM file for the new project and change both the name and the Artifact ID from CompactDiscDAO to CompactDiscEnterprise.
4. Add a new packaging element directly under the name element, and set the value to war.
5. Also in the pom file, add a dependency on JEE using the following entry from before:

```
<dependency>
    <groupId>org.jboss.spec</groupId>
    <artifactId>jboss-javaee-6.0</artifactId>
    <version>1.0.0.Final</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

6. Also in the POM file, comment out the two dependencies on weld-core and weld-se since it will be running in JBoss now so it will have a container available at runtime so will not require Weld any longer.

7. One problem that you can have with Maven is that it will use its default version of Java which may well not be the version you want it to be using (at the time of writing it was defaulting to Java 5). To make sure this does not happen to you, add the following entries (also found in <LAB_HOME>\labs\arquillian\pomSnippetToSpecifyJavaVersion.txt):

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <inherited>true</inherited>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-resources-plugin</artifactId>
      <configuration>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-ejb-plugin</artifactId>
      <inherited>true</inherited>
      <configuration>
        <ejbVersion>3.1</ejbVersion>
      </configuration>
    </plugin>
  </plugins>
</build>
```

8. Save and close the pom.xml file.
9. In the **Project Explorer**, right click on the project and then click **Properties**.
10. Click **Project Facets** and change the Java version to 1.6.
11. Using the **Project Explorer**, delete the com.conygre.training.services.test package.
12. Using the Project Explorer, create a new folder in **src/main** called webapp.
13. In the new **webapp** folder, create a folder called WEB-INF.
14. Now copy into WEB-INF, the **web.xml** file from your earlier JAX-RS project.
15. Check that your web.xml file contains the content following this step. Make the required changes if it is different. The file will be different if you did the JAX-RS exercise on Tomcat.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" 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">
    <servlet-mapping>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Modify the DAO Class

The DAO no longer needs to be as complex as before since it will now be hosted by the JBoss server and can therefore take advantage of the transaction management capabilities and also it can have the EntityManager injected.

1. Open the **JPACompactDiscDAO** class and remove all references to the EntityManagerFactory and change it so that the only property in the class is an EntityManager. This can be annotated with the @PersistenceContext annotation (and it will therefore be injected by JBoss).
2. Convert the class into a local Stateless session bean. This will involve adding the @Stateless and @Local annotations and also annotating the interface @Local.
3. Add the appropriate @Transactional annotations.
4. Remove the constructor.
5. Finally, go through the various methods and remove all references to transactions and commit() calls. The methods should all now be more like this:

```
public void addCompactDisc(CompactDisc disc) {
    Query query = em.createQuery("from CompactDisc");
    List<CompactDisc> discs = query.getResultList();
    if (!discs.contains(disc))
        em.persist(disc);
}
```

6. You will also need to update persistence.xml to use your new datasource, so replace whatever is in between the <persistence> tags with the following entry:

```
<persistence-unit name="conygreChapter8">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/mySQL</jta-data-source>
    <class>com.conygre.training.entities.CompactDisc</class>
    <properties>
        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5Dialect"/>
    </properties>
</persistence-unit>
```

Create the EJB

1. This will be trivial. You currently already have a `CompactDiscService` class which we will turn into an EJB. So open the `com.conygre.training.services.CompactDiscService` class and add the `@Stateless` annotation. Also, rename it to `CompactDiscServiceEJB`.

Note the `@Inject` annotation for the DAO. That can be left in and the DAO will be injected when the bean is deployed to JBoss.

2. Create an interface called `CompactDiscService` and annotate with `@Local`. Add the two methods signatures from the EJB as abstract methods.
3. Implement the interface in the EJB class.

Create the REST API

1. Using the approach you took in the earlier JAX-RS exercise, create a JAX-RS class that can use your EJB. You can add a variable to it to reference your `CompactDiscService` EJB interface:

```
@Inject  
private CompactDiscService service;
```

2. In your JAX-RS class, add a method to return the list of compact discs to the client. It can do that using the service reference injected. So the method would be something like this:

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public CompactDiscList getCompactDiscs() {  
    CompactDiscList discs = new CompactDiscList();  
    discs.setDiscCollection(service.getCatalog());  
    return discs;  
}
```

3. One final change you will need to make is to the `CompactDisc` entity class itself. If you look at the code above it will need to serialize the `CompactDisc` object as JSON. However, when it tries to serialize the `trackTitles` from the `Track` class, you will no longer be connected to the database – the `CompactDisc` instances will be detatched at this point. Therefore, we need to suppress the serialization of the track titles. In order to do that, annotate the `getTrackTitles()` method as `@XmlTransient`. This is a serialization annotation marking the field as one that should not be converted to XML or JSON, so therefore the field is not accessed when marshalling takes place. The code will look something like the below:

```
@XmlTransient  
public Set<Track> getTrackTitles() {  
    return trackTitles;  
}
```

4. Finally, deploy your application to JBoss and test your RESTful endpoint in a browser to see if you can successfully retrieve all the compact discs from the database.

Optional: Test the EJB using Arquillian

Applying the same techniques as you applied using with the BookRetriever bean create an Arquillian test for the CompactDiscService.

Chapter 14 Java Messaging Service (JMS 1.0 and JBoss 7)

Aims

For these exercises you shall be using the messaging services of JBoss.

In this exercise, you will create a producer of messages, which will create and send a simple text message. You will then create a simple consumer, which will receive the message that was sent.

Part 1 Create an Application User

You will now need to set up an application user. This is because JMS endpoints are secured and you will need a user with the relevant permissions in order to connect. Users in and Application realm have access by default if they are in the role of ‘guest’, so these next steps will show you how to set a user up with the relevant permissions.

1. To create an application user, from Windows Explorer, double click on <WILDFLY_HOME>\bin\add-user.bat.
2. When asked what type of user, select application user.
3. When prompted about the realm, leave the option blank and hit enter.
4. When prompted for a username, enter a name (don’t forget it!)
5. When prompted for a password, enter a password (add-user will warn you if you use the username). (don’t forget this either!)
6. Re-enter the password when prompted and then when asked to confirm, confirm your choices.
7. When prompted for the role the user is to be in, type ‘guest’, since by default this role is allowed to send and receive messages, so any user in this role will have access to your JMS queue that will be using shortly.

Part 2 Creating a Message Producer

1. For this exercise, you will use a normal Java project, so in your IDE (Eclipse or IntelliJ), create a new Java Project. Add the following external Jar to the build path.

<JBoss_HOME>\bin\client\jboss-client.jar

2. Create a new Java class called **MessageProducer** and add a main method.

Using the course notes as a guide, you will need to complete the following:

3. Set up a Properties object with the appropriate settings and create an **InitialContext**.
4. Lookup a **QueueConnectionFactory** called jms/RemoteConnectionFactory.
This is the default connection factory on JBoss.
5. Create a **QueueConnection** with your username and password
6. Create a **QueueSession**
7. Lookup a **Queue** called jms/queue/test

8. Create a **QueueSender**
9. Create a **TextMessage**
10. Send the **TextMessage**
11. Close the **QueueSession**

Part 3 Creating a Message Consumer

Now you have created a producer of messages, you need to create a consumer of messages. Create a new class called **MessageConsumer**. This class will be very similar to your previous class. In this case however, you will not be creating a **QueueSender**, but a **QueueReceiver**. You will also need to create a **MessageListener**. You can do this as an inner class, or if you are unsure of inner classes feel free to use a normal class.

In summary, you will need to:

1. Lookup the **QueueConnectionFactory** called `jms/RemoteConnectionFactory`
2. Create a **QueueConnection** with your username and password
3. Create a **QueueSession**
4. Lookup the **Queue** called `jms/queue/test`
5. Create a **QueueReceiver**
6. Receive the **TextMessage** within the **MessageListener**
7. Display the **TextMessage** content
8. Ensure that your main thread continues to run using a `while(true) {}` at the end of the main block.

Part 4 Running your code

1. Launch JBoss by running a command line in `<JBoss_HOME>/bin` and entering `standalone.bat -c standalone-full.xml`.
This will launch JBoss with the messaging subsystem enabled.
2. Run the message producer and then run the consumer.
3. You will see the consumer will consume the message sent by your producer.
4. The consumer will still be running, so if you now run another producer, your consumer will see the message.
5. If you now stop the consumer and then run the producer two additional times, you will now review the queue status in the admin console.
6. Launch the admin console from a browser by visiting <http://localhost:9990>.
7. Enter your admin username and password.
8. Go to the **Runtime** tab at the top right and then locate the **Messaging** pane.
9. Ensure that **Queues** are selected at the top (it defaults to Topics) and then select the test queue.
10. You will see **that** there are two messages waiting and number of messages already processed.
11. Run your consumer again and then refresh the page in the browser. You will see the number of messages waiting drop to zero and the number of messages processed increment by two.

Part 5 Optional Using a Topic

If you have time, modify the code to work with a Topic instead of a Queue. There is a Topic running in JBoss with the JNDI name of jms/topic/test.

Chapter 14 Java Messaging Service (JMS 2.0 and WildFly 8)

Aims

For these exercises you shall be using the messaging services of WildFly.

In this exercise, you will create a producer of messages, which will create and send a simple text message. You will then create a simple consumer, which will receive the message that was sent.

Part 1 Create an Application User and a Queue

You will now need to set up an application user. This is because JMS endpoints are secured and you will need a user with the relevant permissions in order to connect. Users in and Application realm have access by default if they are in the role of ‘guest’, so these next steps will show you how to set a user up with the relevant permissions.

1. To create an application user, from Windows Explorer, double click on <WILDFLY_HOME>\bin\add-user.bat.
2. When asked what type of user, select application user.
3. When prompted about the realm, leave the option blank and hit enter.
4. When prompted for a username, enter a name (don’t forget it!)
5. When prompted for a password, enter a password (add-user will warn you if you use the username). (don’t forget this either!)
6. Re-enter the password when prompted and then when asked to confirm, confirm your choices.
7. When prompted for the role the user is to be in, type ‘guest’, since by default this role is allowed to send and receive messages, so any user in this role will have access to your JMS queue that will be using shortly.
8. You will now configure a Queue. To do this, using a text editor, open <WILDFLY_HOME>\standalone\configuration\standalone-full.xml.
9. Locate the JMS Destinations section and add the section below in bold:

```
<jms-destinations>
    <jms-queue name="ExpiryQueue">
        <entry name="java:/jms/queue/ExpiryQueue" />
    </jms-queue>
    <jms-queue name="DLQ">
        <entry name="java:/jms/queue/DLQ" />
    </jms-queue>
    <b><jms-queue name="TestQueue">
        <entry name="java:jboss/exported/jms/queue/test"/>
        <entry name="queue/test"/>
    </jms-queue>
</jms-destinations>
```

10. Save the standalone-full.xml file. You have now configured a queue.

Part 2 Creating a Message Producer

1. For this exercise, you will use a normal Java project, so in your IDE (Eclipse or IntelliJ), create a new Java Project. Add the following external Jar to the build path.

<WILDFLY_HOME>\bin\client\jboss-client.jar

2. Create a new Java class called **MessageProducer** and add a main method.

Using the course notes as a guide, you will need to complete the following:

3. Set up a Properties object with the appropriate settings and create an InitialContext.
4. Lookup a **QueueConnectionFactory** called `jms/RemoteConnectionFactory`. This is the default connection factory on WildFly.
5. Create a **QueueConnection** with your application username and password
6. Lookup a **Queue** called `jms/queue/test`
7. Create a **JMSContext** from the **QueueConnection**.
8. Create a **JMSPublisher** and send a text message to the queue.

Part 3 Creating a Message Consumer

Now you have created a producer of messages, you need to create a consumer of messages. Create a new class called **MessageConsumer**. This class will be very similar to your previous class. You will also need to create a **MessageListener**. You can do this as an inner class, or if you are unsure of inner classes feel free to use a normal class.

In summary, you will need to:

1. Lookup the **QueueConnectionFactory** called `jms/RemoteConnectionFactory`
2. Create a **QueueConnection** with your username and password as before.
3. Lookup the **Queue** called `jms/queue/test`
4. Create a **JMSContext** and then create a **JMSConsumer**.
5. Create a class implementing **MessageListener** than can receive the message and print it. This can be an inner class or normal class.
6. Create an instance of your **MessageListener** and assign it as the listener for your consumer.
7. Ensure that your main thread continues to run using a `while(true) {}` at the end of the main block.

Part 4 Running your code

1. Launch WildFly by running a command line in `<WIDLFLY_HOME>\bin` and entering `standalone.bat -c standalone-full.xml`. This will launch WildFly with the messaging subsystem enabled.
2. Run the message producer and then run the consumer.
3. You will see the consumer will consume the message sent by your producer.
4. The consumer will still be running, so if you now run another producer, your consumer will see the message.

5. If you now stop the consumer and then run the producer two additional times, you will now review the queue status in the admin console.
6. Launch the admin console from a browser by visiting <http://localhost:9990>.
7. Enter your admin username and password.
8. Go to the Runtime tab at the top and then click the JMS Destinations subsystem.
9. Ensure that Queues are selected at the top (it defaults to Topics) and then select the test queue.
10. You will see that there are two messages waiting and number of messages already processed.

Part 5 Optional Using a Topic

If you have time, modify the code to work with a Topic instead of a Queue. There is a Topic running in JBoss with the JNDI name of jms/topic/test.

Chapter 15 Message Driven Beans

Aims

In this exercise, you will be creating a message driven bean and deploying it in the JBoss application server. You will then reuse your messaging client application to send messages to the queue to check that your message driven bean receives the messages.

Part 1 Creating the Message Driven Bean

1. Create a new Maven project of type EJB called MyMDBProject.

Add the JBoss JEE Spec dependency you have been using up to now as a provided dependency.

```
<dependency>
    <groupId>org.jboss.spec</groupId>
    <artifactId>jboss-javaee-6.0</artifactId>
    <version>1.0.0.Final</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

2. Create a new Java source file called `MyMessageBean` in the package `com.conygre.message`.
3. Within `MyMessageBean`, declare the class and implement the `MessageListener`.
4. Add annotations to the code to specify that the bean will be using the `jms/queue/test` as the destination, and that it will be using automatic acknowledgements. Finally specify that the destination is a queue.
5. Provide the `onMessage` method for the `MessageListener` interface.
6. Within the `onMessage` method, cast the `Message` argument to a `TextMessage`, and extract the text, displaying it using a `System.out.println` statement.
That's it for an MDB, there are no interfaces, just an implementation class to receive messages.
7. Deploy the bean to a running JBoss/Wildfly server.

Part 2 Test your client and MDB

1. Ensure that JBoss is running with your jar file deployed.
2. Return to your earlier exercise where you created a `MessageProducer` class that sends messages and run the `MessageProducer`. It will send the message as before, but this time it will be received by the MDB. You will know if it has worked because the message will appear in the JBoss/Wildfly console.