Component Middleware
S. Chabridon
September 2020
Outline

1. Introduction
2. Overview of EJB Technology
3. References
1. Introduction
1.1 Limits of object-oriented programming
1.2 Motivations for Component Based Development
1.3 What is a component?
1.4 Runtime environment of a component
1.5 Multi-tier Architecture
1.6 Technologies for component middleware

2. Overview of EJB Technology

3. References
1.1 Limits of object-oriented programming

- A lot of tasks must be done manually
  - Object instantiation
  - Service invocation via direct access to object reference + explicit method call
  - Definition of dependencies between classes
  - Almost no tool for application deployment (installation of executable files on the various sites)

- Applications structure difficult to understand (= set of files)

- Difficult to modify or extend an existing application
  - communication mode
  - modification of system/technical services
  - assembly

- Building an application using black-box classes makes it difficult
  - to introduce new references to other objects
  - to inherit from other classes
1.2 Motivations for Component Based Development

*Programming in the large versus programming in the small*

- Applications are built by assembling existing components
- Notion of connector: Components are connected with one another defining a software architecture
- Formalism to describe interactions between components
- Formalism to describe the deployment of components
- Separation of concerns: Separate **functional** from non-functional or extra-functional aspects to allow for more reusability
- Focus on application concerns (functional) rather than technical problems (extra-functional)
1.3 What is a component?

No consensus on a unique definition. Each platform has its own definition!

According to [1]:

- A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
- Context dependencies: required interfaces and execution environments (platforms)
- A binary unit - not source code!
  - This means that a class library is not a component
- No persistent state - a component is not an instance of itself
  - Much like classes are not objects
1.3.1 Characterization of a component

A software module

- That is a **contractual specification** by exporting some attributes, properties and methods
- That provides interfaces to other components and requires some interfaces from other components
- That has no persistent state
- That has **pre- and post-conditions**
- That is **configurable** by setting properties
- That is independently deployable and composable
1.4 Runtime environment of a component

- Client
- Application Server
- Component
- Container
- Service
- System / Middleware
- Connections
- Component
- Application Server
- Container
- Component
- Service
1.4 Runtime environment of a component II

- **Container**
  - Encapsulates components
  - Provides system/technical services
  - Maintains connections between components
  - Deals with invocations and events

- **Application server**
  - Runtime environment for containers
  - Mediator between the containers and the system/middleware
1.4.1 Technical services

- Resource Management
  - Ressource pooling
  - Activation/deactivation mechanism
- Naming and directory
- Synchronous/asynchronous communication
- Transaction
- Persistence
- Security
1.5 Multi-tier Architecture

**N-tier**
Developing the application from the ground up using distinct tiers (layers) should simplify development.

**Tiers**
- **Presentation**
  - manages the delivery of information the end user.
- **Logic**
  - split into two distinct tiers, the business logic deals with what we want the application to do whilst the application logic deals with how the application works with the platform.
- **Data Access**
  - How to get the information needed out of the database, and how to put it back in.
- **Data**
  - Concerned with data at a low level including performance management, indexing, backup and logging.

**Presentation Tier**
- Public Facing Web Site
- Administrative Web Site

**Logic Tier**
- Business Logic
- Application Logic

**Data Access Tier**
- Database
- Database

**Data Tier**
- File Store
- Archive
- Log

www.jasonslater.co.uk
1.5.1 The 6 roles in component development I

1. **Component Provider**
   - Develops components
   - Provides component metadata: structural information (component logical name, transaction demarcation, persistence requirements...) and component external dependencies
   - Metadata may be expressed in annotations or in an XML deployment descriptor

2. **Application Assembler**
   - Assembles application components into a single deployable unit
   - Defines security roles for application clients, method permissions...
1.5.1 The 6 roles in component development II

3. Deployer

- Uses information provided by the component provider and the assembler
- Resolves component dependencies
- Deploys the application in an operational environment including a container and a server

4. Server Provider

- Responsible of distributed transaction management, distributed objects management, low-level system tasks
- OS vendor, Middleware vendor or DBMS vendor
1.5.1 The 6 roles in component development III

5. Container Provider

- Provides deployment tools and runtime support for components
- Focus on the development of a scalable, secure, transaction-enabled container

6. System Administrator

- Responsible for the configuration and administration of the enterprise’s computing and networking infrastructure
- Oversees the well-being of the deployed applications
- Monitors the log of non-application exceptions and errors logged by the container
- Takes actions to correct the problems caused by exceptions and errors
1.6 Technologies for component middleware I

- **Entreprise Java Beans**
  - Supported by Eclipse Foundation, as part of Eclipse Enterprise for Java (EE4J) initiative [https://projects.eclipse.org/projects/ee4j](https://projects.eclipse.org/projects/ee4j)
  - Initially developed by Sun Microsystems in 2005, then sponsored by Oracle until 2019
  - Application server: Jakarta EE 8 [https://projects.eclipse.org/projects/ee4j.glassfish](https://projects.eclipse.org/projects/ee4j.glassfish)
  - ONE LANGUAGE, MANY PLATFORMS

- **.NET**
  - Supported by Microsoft [https://docs.microsoft.com/en-us/dotnet/](https://docs.microsoft.com/en-us/dotnet/)
  - MANY LANGUAGES (C#, F#, or Visual Basic), MANY PLATFORMS (Initially only on Windows)
1.6 Technologies for component middleware II

- **Spring Framework**
  - Supported by Spring [https://spring.io/projects/spring-framework](https://spring.io/projects/spring-framework)
  - Relies on dependency injection and aspects
  - Lightweight application server enriched with a wide ecosystem

- **CORBA Component Model (CCM)**
  - Supported by the Object Management Group (OMG) [www.omg.org](http://www.omg.org)
  - EJB specification can be seen as a subset of CCM specification
  - MANY LANGUAGES, MANY PLATFORMS
2 Overview of EJB Technology

1. Introduction

2. Overview of EJB Technology
2.1 What is EJB?
2.2 EJB Container
2.3 Java EE at a glance
2.4 Java EE Architecture
2.5 EJB types
2.6 Session Beans
2.7 Entity Beans
2.8 Message-driven Beans (MDB)
2.9 Transaction Service

3. References
2.1 What is EJB?

- **Enterprise Java Beans**

- Java component model for distributed enterprise applications, released by Sun in 1998

- **EJB 3.0 specification (2006) - JSR 220**

- **EJB 3.1 specification (2009) - JSR 318**

- **EJB 3.2 specification (2013) - JSR 345**

- **Definitions [2, 3]:**
  - EJB are standard server-side components for *component transaction monitors (CTM)*
  - EJB technology defines a model for the development of reusable Java server components that encapsulate the *business logic* of an application
2.1.1 The Java Community Process (JCP)

- www.jcp.org
- International developer community whose charter is to develop and evolve Java technology
  - specifications,
  - reference implementations,
  - and technology compatibility kits.
- Company, organization, or individual can be member
2.2 EJB Container

- **Runtime environment** for creation and lifecycle management of bean instances
- Gives access to a set of standardized services to beans
- Provides a context with:
  - Configuration properties
  - References to other components
  - References to technical services
2.2.1 EJB Container — Provided services I

- Includes many **Java technologies**, that can be used independently of EJB
- Java 2 Platform, Standard Edition v8 (J2SE) APIs
  - RMI-IIOP - remote method invocation based on CORBA Interoperable Inter-ORB Protocol
  - JDBC (Java DataBase Connectivity)
  - JSP (Java Server Pages) — Web clients
  - JAXP (Java API for XML Processing)
  - Java IDL — adds CORBA capability to the Java platform
2.2.1 EJB Container — Provided services II

- Current services are frozen
- Research initiatives (s.a. Objectweb JOnAS) provide extensible containers with pluggable services
- EJB 3.2 APIs (javax package), including Java Persistence (JPA 2.1)
- Asynchronous communication: Java Messaging Service (JMS 2.0), JavaMail 1.5
- Connector
- Transaction: UserTransaction interface of JTA 1.2, Java Transaction Service (JTS) (specification based on CORBA Object Transaction Service)
- Security: Java Security API
- Web Services: JAX-RPC 1.1, JAX-WS 2.0, JAX-RS 2.0
Lifecycle service — Java Naming and Directory Interface

- Instances passivation
  - Temporary saving of a bean when container needs memory
- Instances pooling
  - For performance reasons, the container can instantiate less beans than there are clients
  - Then several clients share the same bean
  - Possible only for beans without instance variables
- Pooling of connections to the Database
  - All the beans of a server share a pool of connections to the DB
  - Connections remain open and are used by beans
2.3 Java EE at a glance

- Java Platform, Enterprise Edition
- Application server technology based on EJBs
- Targets scalability, accessibility, security, integrity, and other requirements of enterprise-class applications
- Java API for RESTful Web Services (JAX-RS)
- Contexts and Dependency Injection for the Java EE Platform (CDI)
- Bean Validation: same set of validations can be shared by all layers of an application
- Java Server Faces (JSF) 2.0 supports Ajax
### 2.3.1 Java EE 8 - Specifications

#### Revised Java EE 8 Proposal

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDI 2.0 (JSR 365)</td>
<td>Bootstrap API for Java SE</td>
<td></td>
</tr>
<tr>
<td>Servlet 4.0 (JSR 369)</td>
<td>HTTP/2 support</td>
<td></td>
</tr>
<tr>
<td>JSF 2.3 (JSR 372)</td>
<td>Small-scale new features</td>
<td></td>
</tr>
<tr>
<td>Security 1.0 (JSR 375)</td>
<td>Authentication/authorization APIs</td>
<td></td>
</tr>
<tr>
<td>JSON-B 1.0 (JSR 367)</td>
<td>JSON &lt;-&gt; object mapping</td>
<td></td>
</tr>
<tr>
<td>JAX-RS 2.1 (JSR 370)</td>
<td>Reactive enhancements</td>
<td></td>
</tr>
<tr>
<td>Management 2.0 (JSR 373)</td>
<td>REST-based APIs</td>
<td></td>
</tr>
<tr>
<td>Bean Validation 2.0 (JSR 380)</td>
<td>Collection constraints</td>
<td></td>
</tr>
<tr>
<td>Health Checking</td>
<td>Standard for client-side health reporting</td>
<td></td>
</tr>
<tr>
<td>JMS 2.1 (JSR 368)</td>
<td>Flexible JMS MDBs</td>
<td></td>
</tr>
<tr>
<td>MVC 1.0 (JSR 371)</td>
<td>Action-based MVC framework</td>
<td></td>
</tr>
<tr>
<td>JSON-P 1.1 (JSR 374)</td>
<td>JSON Pointer and Patch</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>Standard for externalizing application configuration</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.
2.3.2 Java EE evolution

Java EE: Past, Present, Future

- **Java EE 8**: Servlet 4, JSON-B, JSON-P 1.1, JSF 2.3, CDI 2.0, JAX-WS 2.1, Security
- **Java EE 7**: JMS 2, Batch, TX, Concurrency, Web-Sockets, JSON
- **Java EE 6**: Pruning, Ease of Use, JAX-RS, CDI, Bean-Profile
- **Java EE 5**: Ease of Use, EJB 3, JPA, JSF, JAX-WS
- **J2EE 1.4**: Web Services, Mgmt, Deploy
- **J2EE 1.3**: CMP, JCA
- **J2EE 1.2**: Servlet, JSP, EJB, JMS, RMI
2.3.3 Families of Java EE APIs

- **Web Application**
  - Servlet, WebSocket, JavaServer Faces, JSON-P, JSON-B, etc.

- **Enterprise Application**
  - EJB, CDI, BV, JPA, Batch, JMS, JTA, JavaMail, JCA, Concurrency, etc.

- **Web Services**
  - JAX-RS, JAX-WS, SAAJ, JAXP, JAXB, StAX, etc.

- **Management / Security**
  - JMX, Management, Security, JACC, JASPIC, etc.
2.3.4 From Java EE to Jakarta EE
2.4 Java EE Architecture
2.4.1 Java EE — 3-tier Architecture I
2.4.1 Java EE — 3-tier Architecture II

- **Client**
  - Heavy weight client — Java application (or possibly other language)
  - Light weight client — Web navigator

- **Application Server**
  - Reference implementation: GlassFish (Eclipse Foundation)
  - Commercial products: WebSphere (IBM), WebLogic (BEA)...
  - Open source distributions: jBoss, JOnAS (Objectweb), Geronimo, OpenEJB...

- **DBMS (DataBase Management System)**
  - Provide storage support for application data
  - Mostly using a relational DBMS (Oracle, SQL Server, PostGreSQL...)
2.5 EJB types I

- **Entity Beans**
  - Model real-world objects (e.g. Owner, Account) that exist in persistent storage (DBMS or other storage accessible using JDBC [Java Database Connectivity])
  - Persistent state is maintained through all method and server invocations
  - Identified by a primary key
  - Object-Relational mapping
  - Implementation using JPA (Java Persistence API)
2.5 EJB types II

■ Session Beans

□ Model client activities
□ Perform a task or process, and are therefore transient
□ Do not exist outside a client session
□ No persistent state
□ Two kinds of session beans: stateless and stateful
□ Manage actions that may cross entity beans or go outside the concern of an entity bean
  □ e.g. Teller may authenticate the user and transfer funds between accounts
  □ e.g. Statement may include transactions from multiple accounts

■ Message-Driven Beans (since EJB 2.0)

□ Listener processing messages asynchronously
□ Only a bean class. No interface.
2.5.1 Main EJB3 Annotations

- **@EJB**: Denotes a reference to an EJB business interface or home interface.
- **@PersistenceContext**: Used to express a dependency on an EntityManager.
- **@Stateful**: Used to annotate a class as a stateful session bean component.
- **@Stateless**: Used to annotate a class as a stateless session bean component.
- **@Remote**: Applied to the session bean class or remote business interface to designate a remote interface of the bean.
- **@MessageDriven**: Specifies a message-driven bean. A message-driven bean is a message consumer that can be called by its container.
@TransactionManagement: Declares whether a bean will have container-managed or bean-managed transactions.

@TransactionAttribute: Applies a transaction attribute to all methods of a business interface or to individual business methods on a bean class. Can be specified on the bean class or on methods of the class that are methods of the business interface.

Possible values:

- MANDATORY
- REQUIRED (default)
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- NEVER

@WebService: Used on a class or an interface to define a Web service.

@WebMethod: Indicates whether the method is part or not of the interface service endpoint interface (SEI) of the web service.
2.5.2 Bean development

- An EJB has a **remote interface** to be accessed by clients
  - Describes the **provided services (methods)**
  - No longer required for session beans

- Possibly an EJB may provide an interface for local access
  - Describe the provided services offered to local clients
  - Same as remote services, or different ones (enables optimisation)
  - Can only be used within the same JVM as the EJB
  - Gets compiled by the ejb compiler to create local stubs for container to interpose transactions, access control, etc. on invocations.

- An **implementation class**
2.5.3 Interfaces

- **Remote Interface**
  - Interface presented to the outside world (contract definition) specifying the business methods provided by the bean
  - Gets compiled by the ejb compiler to create RMI stubs and skeletons
  - Stubs are used by RMI to translate a method invocation to wire format
  - Skeletons are used by RMI to translate wire format to a method invocation

- **NB:** A client application never interacts with a bean class directly; It uses the methods of the bean’s interface.
2.6 Session Beans

- Model **business process** being performed by a single client involving one or more entity beans
- Life duration linked to client’s one
- Two types of session bean
  - **Stateful session bean**
    - maintains the conversational state between a client and the session bean
    - may be serialized out and passivated to conserve system resources
    - will be serialized in and activated when needed in the future
    - e.g. Teller session bean logged into and transfers funds between accounts
  - **Stateless session bean**
    - does not maintain conversational state
    - to be used for generic tasks, to read persistent data
    - e.g. Statement that is given a list of accounts or an owner to generate a textual report for
    - consumes the least amount of resources among all the bean types
Stateless Session Bean — Calculator Example

- Calculator session bean: Simple calculator with 4 operations
- Implementation code:
  - Remote business interface (Calculator)
  - Session bean class (CalculatorBean)

```java
import javax.ejb.Remote;
@Remote
public interface Calculator {
    public double add(double n1, double n2);
    public double sub(double n1, double n2);
    public double mul(double n1, double n2);
    public double div(double n1, double n2);
}
```
Stateless Session Bean — Calculator Example - Implementation class

Possible to name a bean: `@Stateless(name = "myCalculator")`

```java
import javax.ejb.Stateless;

@Stateless(name = "myCalculator")
public class CalculatorBean implements Calculator {
    public double add(double n1, double n2) {return n1+n2;}
    public double sub(double n1, double n2) {return n1-n2;}
    public double mul(double n1, double n2) {return n1*n2;}
    public double div(double n1, double n2) {return n1/n2;}
}
```
**Stateless Session Bean — Calculator Example - Client side**

2 ways to get the reference of the business interface

- dependency injection:
  ```java
  @EJB  Calculator myCalc;
  ```

- look-up in JNDI directory using the `lookup` method provided by `EJBContext` interface and the bean interface name

```java
import javax.naming.*;
public class myClient {
    public static void main(String args[]) throws Exception {
        Context myContext = new InitialContext();
        Calculator myCalc = (Calculator) myContext.lookup("myCalculator");
        double result = myCalc.mul(2,4);
    }
}
```
Stateless Session Bean — No-interface view

- When a bean does not have a remote interface, possible to access directly to the bean implementation class via the no-interface view.
- But never use the new operator to acquire the reference.
- A no-interface view is a variant of a local view that exposes the non-static public methods of the bean class.
- 2 ways to get the reference of the no-interface view of a session bean:
  - dependency injection:
    ```
    @EJB CalculatorBean myCalc;
    ```
  - look-up in JNDI directory using the `lookup` method provided by `EJBContext` interface and the bean interface name:
    ```
    @Resource SessionContext myContext;
    ...
    CalculatorBean myCalc =
    (CalculatorBean) myContext.lookup("myCalculator");
    ```
Cart session bean: represents a shopping cart in an online bookstore.

The bean’s client can add a book to the cart, remove a book, or retrieve the cart’s contents.

Implementation code:
- Remote business interface (Cart)
- Session bean class (CartBean)
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id) throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
package straal.session;

import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
public class CartBean implements Cart {
    String customerName;
    String customerId;
    List<String> contents;

    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else { customerName = person; }
        customerId = "0";
        contents = new ArrayList<String>();
    }
}
... public void addBook(String title) { contents.add(title); }

public void removeBook(String title) throws BookException {
    boolean result = contents.remove(title);
    if (result == false) {
        throw new BookException(title + " not in cart.");
    }
}

public List<String> getContents() { return contents; }

@Remove // The container will remove the bean
public void remove() { contents = null; }
}
Stateful Session Bean — Cart Example — Client side

From the client’s perspective, the business methods appear to run locally, but they actually run remotely in the session bean.

cart.create("Duke DeEarl");
...
cart.addBook("Bel Canto");
...
List<String> bookList = cart.getContents();
...
cart.removeBook("Gravity’s Rainbow");
2.6.1 Asynchronous Method Invocation

- Control returned to the client by the container before the method is invoked on the session bean instance.
- Use Java SE concurrency API to retrieve the result, cancel the invocation, or check for exceptions.
- Useful for long-running operations or to improve application response time.
- The result implements `java.util.concurrent.Future< V >` interface, "V" is the result value type.
Asynchronous Method Invocation — Session bean side

- Annotate a method or a class with `@Asynchronous` (javax.ejb.Asynchronous)
- Asynchronous methods return either void or an implementation of the Future `<V>` interface
- Result is returned to the container, not directly to the client

```java
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    String status = ...;
    return new AsyncResult<String>(status);
}
```
Asynchronous Method Invocation — Session bean side

- Check whether the client requested the invocation to be cancelled with method `javax.ejb.SessionContext.wasCancelled`

```java
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    if (SessionContext.wasCancelled()) {
        // clean up
    } else {
        // process the payment
    }
    ...
}
```
2.6.2 Asynchronous Method Invocation — Client side

- Retrieve result using `Future <V>.get()` methods (synchronous method)
- Use `Future <V>.isDone` to check whether processing has completed
- Call `Future <V>.cancel(boolean mayInterruptIfRunning)` to cancel the method invocation
- Method `Future <V>.isCancelled` returns true if the invocation was cancelled
2.7 Entity Beans I

- Represent a **business object** in a **persistent** storage mechanism
- Can be shared by multiple clients
- Can be linked to other entity beans (like relations in a relational DBMS)
- Primary key required
  - Defined using `@Id` annotation,
  - Possible key types (or of the properties or fields of a composite primary key): java primitive types (and associated wrapper classes), String, Date
2.7 Entity Beans II

- Object/relational mapping annotations to map entities and entity relationships to relational tables
  - Each EB class is mapped to one relational table
  - table name = class name by default
  - or use annotation @Table(name = "...")

- 2 exclusive modes for the definition of table columns
  - property-based access: annotate getter methods
  - field-based access: annotate attributes
@Entity
public class Book implements Serializable {
    private String bookId;
    private String author;
    private String title;
    public Book() { }
    public Book(String author, String title) {
        this.author = author;
        this.title = title;
    }
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public String getBookId() { return this.bookId; }

    public String getTitle() { return this.title; }
    public void setTitle(String title) { this.title=title; }
    ...
}
2.7.1 Multiplicities in Entity Relationships

1. **One-to-one**: Each entity instance is related to a single instance of another entity.

2. **One-to-many**: An entity instance can be related to multiple instances of the other entities.

3. **Many-to-one**: Multiple instances of an entity can be related to a single instance of the other entity.

4. **Many-to-many**: The entity instances can be related to multiple instances of each other.
Multiplicities in Entity Relationships — One-ToMany example

```java
@Entity
public class Author {
  private long id;
  private String name;
  private Collection<Book> books;

  public Author() { books = new ArrayList<Book>(); }
  public Author(String name) {this.name = name; }

  @OneToMany
  public Collection<Book> getBooks() {return books; }

  public void addBook(String title) {
    Book aBook = new Book(this.name, title);
    getBooks().add(aBook);
  }
}
```
2.7.2 Persistence management mode

Persistence can be managed in two ways:

- **Container-managed (CMP)**
  - Simplest to develop
  - Bean code contains no database access calls

- **Bean-managed (BMP)**
  - The client is required to explicitly write persistence logic by providing implementation methods for Home interface
  - More flexibility in how state is managed between the bean instance and the database
  - Used when deployment tools are inadequate
2.7.3 Entity Manager

- Entry point of the persistence service
  - Creates and removes persistent entity instances
  - Finds entities by the entity’s primary key
  - Allows queries to be run on entities

- Associated with a persistence context
  - Defines the scope under which particular entity instances are created, persisted and removed
Container-Managed Entity Manager

- Propagation of the **persistence context** automatically to all application components that use the EntityManager instance within a single JTA (Java Transaction Architecture) transaction.

- To obtain an EntityManager instance, *inject* the entity manager into the application component:

  ```java
  @PersistenceContext
  EntityManager em;
  ```
Application-Managed Entity Manager

- Each EntityManager creates a new, isolated persistence context.
- Life cycle of EntityManager instances managed by the application: The EntityManager and its associated persistence context are created and destroyed explicitly by the application.
- To obtain an EntityManager instance, first get an EntityManagerFactory instance:

```java
@PersistenceUnit
EntityManagerFactory emf;
```
- Then, obtain an EntityManager from the EntityManagerFactory instance:

```java
EntityManager em = emf.createEntityManager();
```
import javax.ejb.*;
import javax.persistence.*

public class BookDBAO {

  @PersistenceContext
  private EntityManager em;

  public void init() {
    Book b1 = new Book("Charles Beaudelaire","Les Fleurs du Mal");
    Book b2 = new Book("Jules Verne","Voyage au centre de la Terre");
    em.persist(b1);
    em.persist(b2);
  }
}
2.7.4 Persistence Unit — persistence.xml file

- Defines the set of all entity classes managed by EntityManager instances in an application
- Represents the data contained within a single data store
- Packaged with the application archive file
- XML elements:
  - `persistence` element: global schema, includes a persistence-unit element
  - `persistence-unit` element: name of a persistence unit and transaction type
  - Optional `description` element
  - `jta-data-source` element: specifies the global JNDI name of the JTA data source
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers. It does not rely on any vendor-specific features and can therefore be deployed to any persistence provider.</description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
2.8 Message-driven Beans (MDB)

- Can implement any messaging type
- Handle asynchronous messages
- Useful for non-blocking calls
- Producer/consumer concept
- Stateless — state is lost between 2 messages processing
- All instances of a same MDB class are equivalent
- Can process messages from several clients
- No remote interface
- The container delivers messages to a MDB using the onMessage() method
- Same lifecycle as a stateless session bean
Message-driven Beans types (MDB)

- 2 communication modes
  - Queue: 1 to 1 or n to 1
  - Topic: 1 to n or n to m
JMS architecture

The Java Message service provides MDB management
MDB development
1. Create a connection using a ConnectionFactory

2. Create a session (possibly several sessions per connection):
   - period of time for sending messages on a queue or topic
   - may be transactional

3. Create a message

4. Send the message

5. Close the session

6. Close the connection
public class myProducer {
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")
private static Queue queue;
public void produce() {
    /* 1 */ Connection connection = connectionFactory.createConnection();
    /* 2 */ Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageProducer messageProducer = session.createProducer(queue);
    /* 3 */ TextMessage message = session.createTextMessage();
    message.setText("This is a message ");
    /* 4 */ messageProducer.send(message);
    /* 5 */ session.close();
    /* 6 */ connection.close();
} } }
@MessageDriven(mappedName="jms/Queue")
public class SimpleMessageBean implements MessageListener {

    public void onMessage(Message m) {
        TextMessage message = (TextMessage) m;
        message.getText();

        ...
    }
}

2.9 Transaction Service I

- Controls concurrent accesses to data by multiple programs
- In case of a system failure, transactions make sure that after recovery the data will be in a consistent state
- Guarantees ACID properties for transactions
  - Atomicity: Either both operations complete successfully or none
  - Consistency: e.g. an account must never have a negative balance
  - Isolation: several parallel transfers must give the same result as if they were performed successively
  - Durability: Account state must be persistent and stored on stable storage
- Example: banking transfer with debit then credit operations
- Fully integrated within the EJB server
- Main advantage compared to the CORBA middleware
2.9 Transaction Service II

- Specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system
  - Resource manager
  - Application server
  - Transactional applications

- Transaction manager
  - Decides whether to commit or rollback at the end of the transaction in a distributed system and coordinates various resource managers

- Resource manager
  - Responsible for controlling the access to common resources in the distributed system
3 References

JSR345 - EJB 3.2: https://www.jcp.org/en/jsr/detail?id=345

C. Szyperski.

*Component Software Beyond Object Oriented Programming.*

R. Monson-Haefel.

*Entreprise Java Beans.*

J. Lafosse.

*Développements n-tiers avec Java EE.*
ENI Éditions, March 2011.