EJB

ENTERPRISE JAVA BEANS

INTRODUCTION TO ENTERPRISE JAVA BEANS, JAVA'S SERVER SIDE COMPONENT TECHNOLOGY

Peter R. Egli
peteregli.net

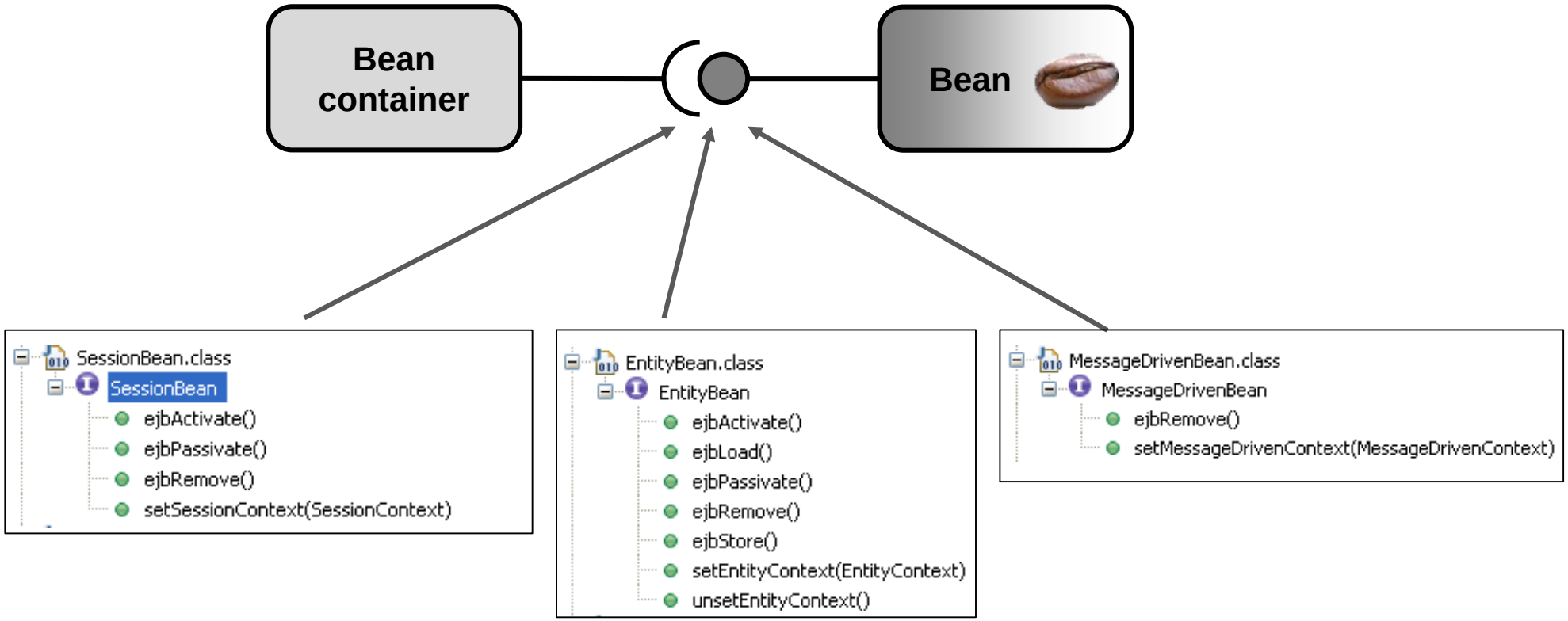
Contents

1. What is a bean?

→ Beans are business logic components that implement a standard interface through which the bean is hooked into the bean container (= runtime object for bean).

→ A Java class implementing one of the standard bean interfaces is a bean.

→ Beans can be accessed remotely, usually from a client tier.



Snapshots from OpenEJB libs in Eclipse package explorer

2. Why EJB?

Common concerns in different applications lead to re-implementing the same functionality for *business logic* components.

Examples of common functionality:

- Persistence
- Transactions
- Security
- Runtime and lifecycle management (create, start, stop and delete component)

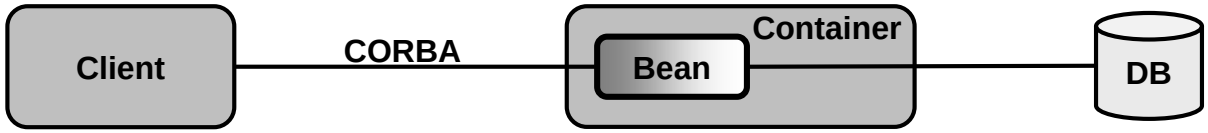
EJB is a framework that provides the following services to applications:

- Persistence
- Transaction processing
- Concurrency control (each client accesses its own bean instance)
- Events using JMS (Java Messaging Service)
- Naming and directory services via JNDI (Java Naming and Directory Interface)
- Security using JAAS (Java Authentication and Authorization Service)
- Deployment of software components to a server host
- Remote procedure calls via RMI (RMI over IIOP)
- Exposing business functionality via web services

3. Evolution of EJB

EJB 1.0 (1998):

- Framework for distributed applications, backed by industry giants IBM and Sun.
- Complexity (difficult to write beans due to high number of interfaces, exception etc. to be used).
- Low performance due to use of CORBA as the only available remoting technology.



EJB 2.0 (2001):

- Performance improvements due to the introduction of local and remote interfaces (local applications use faster local interface).
- Introduction of Message Driven Beans (connect EJB with JMS).
- Complexity (specification: 646 pages).
- Severely limited SQL dialect for entity beans (EJBQL).

3. Evolution of EJB

EJB 3.0 (2006):

- Home interface eliminated.
- Deployment descriptor optional, replaced by Java annotations.
- Implementation of business interfaces as POJI (Plain Old Java Interface) with Java annotations.
- Implementation of beans as POJOs (Plain Old Java Object) with Java annotations.
- Entity beans replaced by POJOs that use JPA-annotations (Java Persistence API) @Resource.
- JNDI-lookup of home interface replaced by dependency injection via annotations (@Inject).
- Beans no longer need to implement callback methods ejbCreate() and ejbActivate(); instead they may use annotations.

EJB 3.1 (since Java 6):

- Support for singletons (@Singleton annotation)
- Asynchronous invocations (@Asynchronous annotation)
- Optional business interface

4. Bean interfaces (1/5)

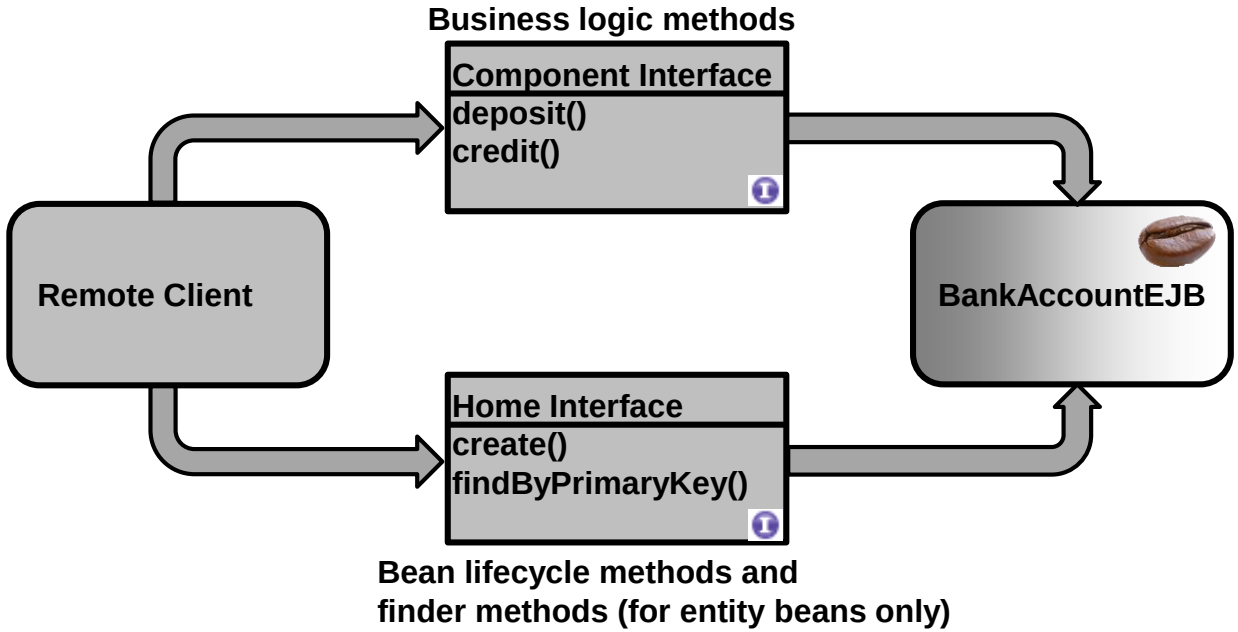
Access to beans is provided through 2 interfaces (in EJB versions EJB 1.x and EJB 2.x):

Component interface:

The *component interface* (also called remote interface) provides the business logic methods (methods that are specific to a bean).
The component interface methods are instance (object) methods.

Home interface:

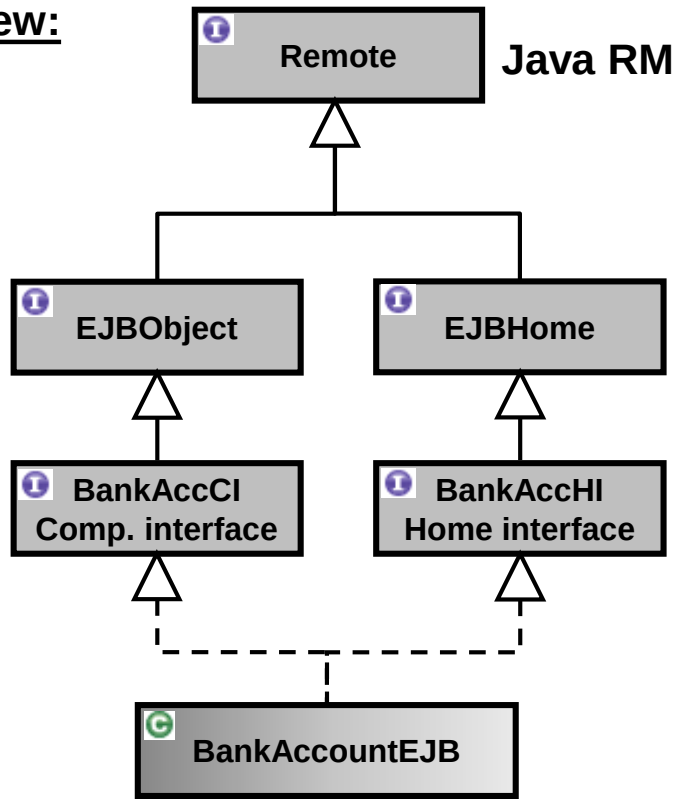
The *home interface* specifies bean lifecycle methods (creation and deletion of beans). These methods are static (class) methods.



4. Bean interfaces (2/5)

Both component and home interface are Java RMI interfaces.
This allows remote access to a Java class that implements these interfaces (bean).
But: A bean can be accessed remotely (client is in a different VM) and locally (client is in the same VM as the bean container), see page 8.

UML view:



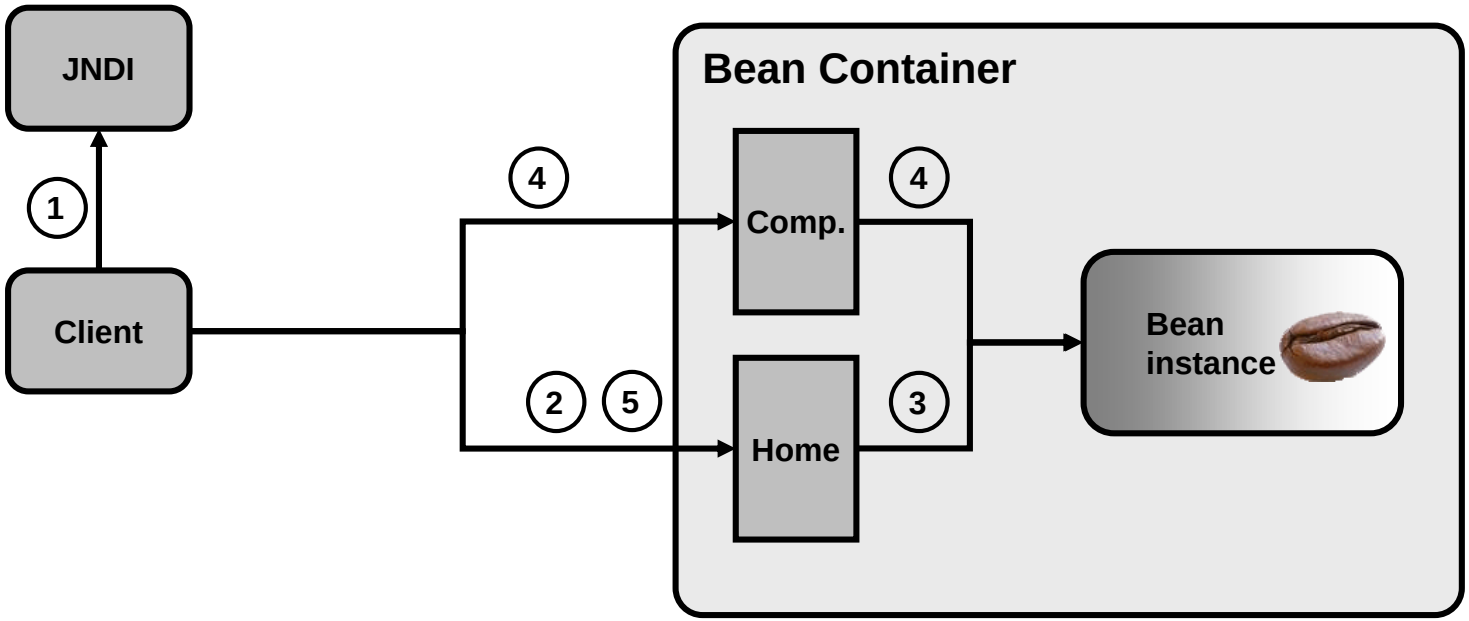
Java RMI remote interface.

EJB component and home interfaces.

User defined component and home interfaces.

4. Bean interfaces (3/5)

Typical access of a client to a bean (EJB 1.x and EJB 2.x):



1. Lookup of the home interface through JNDI (Java Naming and Directory Interface).
2. The Client calls the create() method on the home object.
3. The home object creates an instance of the bean and calls ejbCreate() with the same signature as the client. The container returns a reference to the created bean instance.
4. The client calls a business method on the created bean.
5. When finished the client destroys the bean by calling remove() on the home interface.

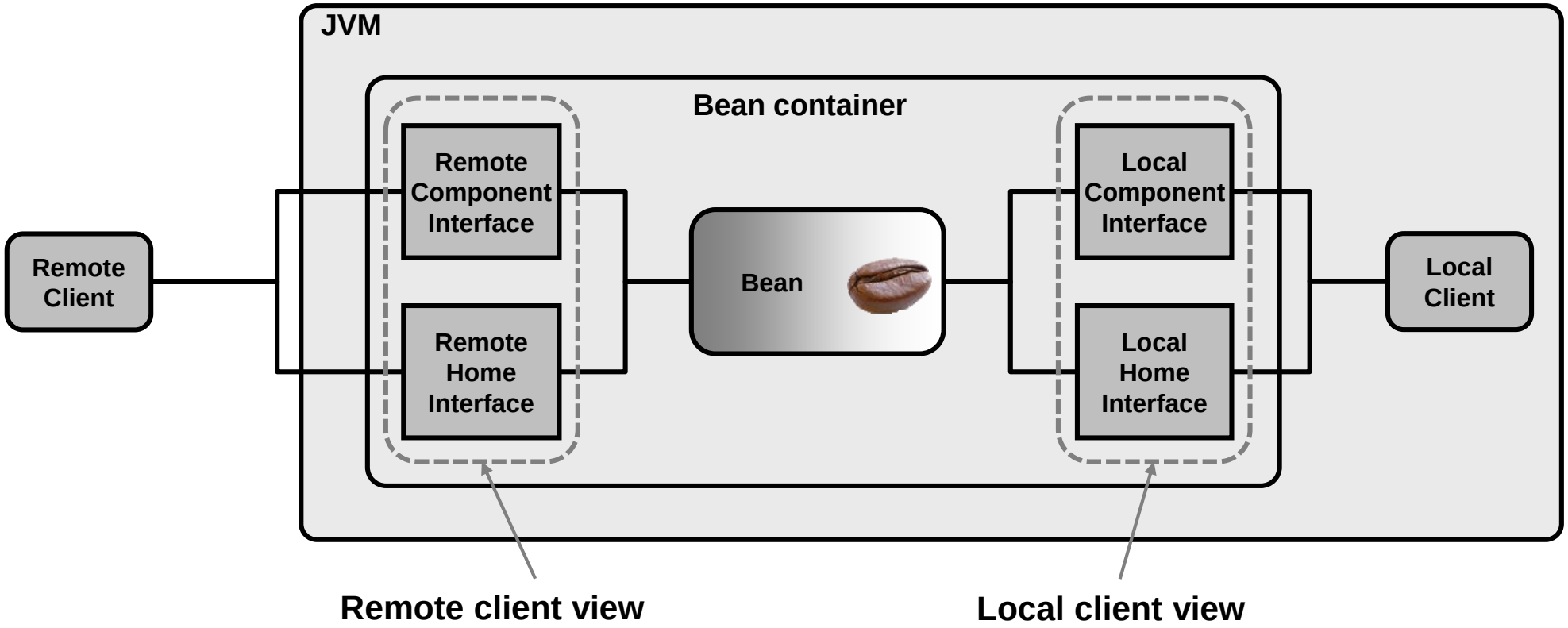
4. Bean interfaces (4/5)

Local versus remote bean access (1/2):

EJB 2.0 introduced local and remote access interfaces.

Clients can run in a different JVM (= remote client) or in the same JVM as the bean (= local client).

Arguments are passed-by-value in calls made by the remote client and passed-by-reference in calls made by the local client.



4. Bean interfaces (5/5)

Local versus remote bean access (2/2):

Remote access:

- → Loose coupling between client and server (=bean).
- → Call-by-value arguments in calls (copy-semantics).
- → Potentially slow (network latency, network stack processing, parameter marshalling etc.).
- → Remote interface = RMI interface.
- → Location transparency (client does not know where server bean resides).
- → Recommended usage: Coarse-grain access client to server (only occasional accesses).

Local access:

- → Tight coupling between client and server (=bean).
- → Call-by-reference arguments (no copying).
- → Plain Java object interface (direct method calls).
- → No location transparency.
- → Recommended usage: May be used when client and bean have a tight interaction.

5. EJB bean types (1/3)

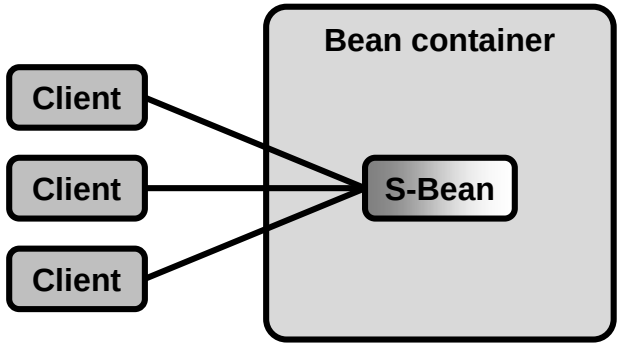
5.1. Session bean (EJB 1.x, 2.x and 3.x):

A session bean contains and represents some business logic.

a. Stateless session bean:

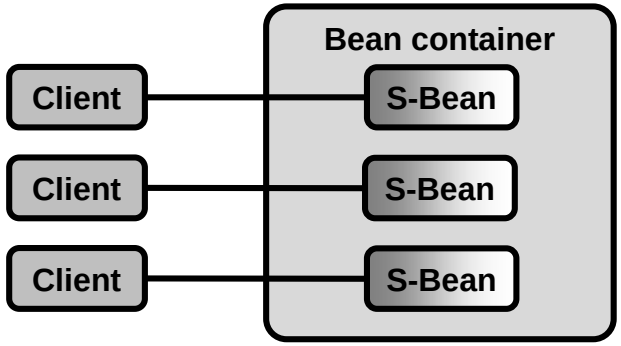
→ The instance variables of the bean are guaranteed to be maintained only for the duration of the client method invocation.

→ Stateless session beans provide better scalability (beans may support multiple clients).



b. Stateful session bean:

→ The bean maintains a conversational state throughout the session with the client, until the session terminates.



5. EJB bean types (2/3)

5.2. Entity bean (EJB1.x, EJB 2.x):

An entity bean represents persistent data maintained in a database (DB).
Typically an entity bean represents a row (=entry) in a DB table.
An entity bean is identified by its primary key.

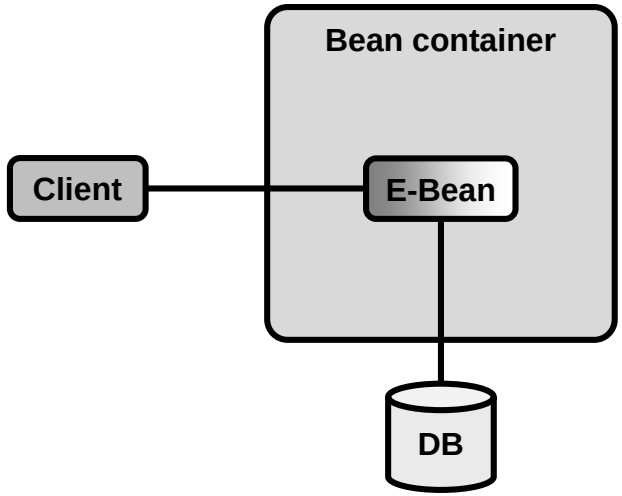
Types of persistence:

a. BMP – Bean Managed Persistence:

→ The bean manages persistence on its own (bean developer must write the DB access calls).

b. CMP – Container Managed Persistence:

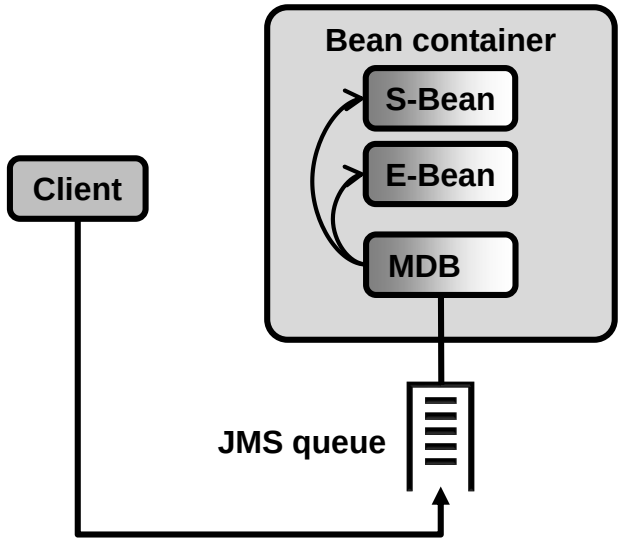
→ The persistence is managed by the bean container, i.e. the container generates the DB access calls.



5. EJB bean types (3/3)

5.3. MDB - Message Driven Bean (EJB 2.x, EJB 3.x):

A message driven bean acts as a listener on JMS message queues (Java Message Service).
Clients do not have direct access to MDBs through home or component interfaces.
MDBs are similar to stateless session beans in that they may receive messages from multiple clients.
MDBs allow asynchronous interaction between client and server which reduces the coupling between them.
MDBs may be used as an asynchronous interface to a server application. The MDB receives messages and converts these to method calls on session and entity beans in the bean container.



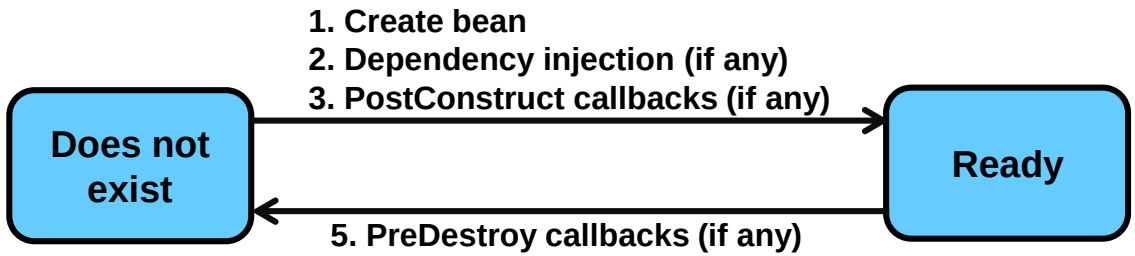
6. Lifecycle of EJBs (1/4)

6.1. Stateless session bean (EJB 1.x, 2.x, 3.x):

Stateless session beans have a very simple lifecycle.
They either do not exist or are ready for receiving method invocations.

Lifecycle steps:

1. The client obtains a reference to a stateless session bean (JNDI lookup).
2. The EJB container performs dependency injection (evaluation of @EJB, @Resource, @Inject annotations if such are provided). The bean goes into the state Ready.
3. The EJB container invokes methods annotated with @PostConstruct.
4. The client invokes business methods on the bean.
5. When the client reference goes out of scope, the lifecycle of the bean ends. The EJB container calls methods annotated with @PreDestroy and then disposes of the bean.



6. Lifecycle of EJBs (2/4)

6.2. Statefull session bean (EJB 1.x, 2.x, 3.x):

In addition to the lifecycle states of stateless beans, stateful beans have the state Passive. The EJB container may move (evict) unused beans to secondary storage, e.g. disk for saving resources (RAM). When a client invokes a method on a passivated object, the container resurrects a bean of the requested type, loads it with state data that it had before passivation (state data is stored separately on disk) and then performs the method invocation.

Lifecycle steps:

The bean creation process (steps 1. through 3.) is the same as for stateless session beans.
4. The client invokes business methods on the bean.
5. The server may, when it detects that the bean is not invoked for some time, passivate the bean. Before passivating the bean, the container calls the @PrePassivate callback.
6. When a new client invocation arrives, the EJB container retrieves a bean of the requested type, fills it with state data the bean had before passivation, calls the @PostActive annotation methods and then the called business method.
7. The actions for bean destruction are the same as for stateless beans.

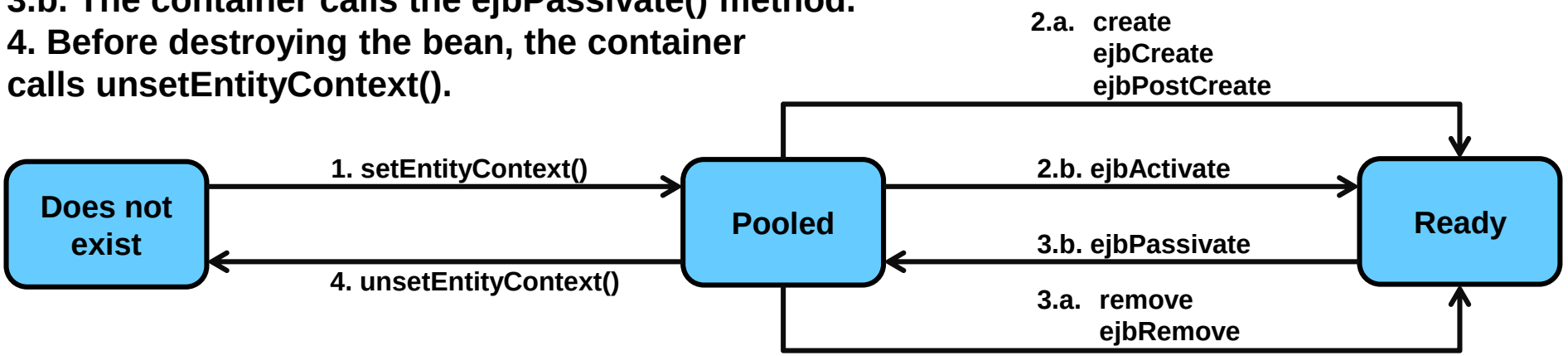
6. Lifecycle of EJBs (3/4)

6.3. Entity bean (EJB 1.x and 2.x):

Entity beans are moved between *Pooled* and *Ready* states, either by client or container request.

Lifecycle steps:

1. The EJB container creates the instance, calls setEntityContext() (set the entity context that may be used in transactions) and moves the bean to the bean pool. At this stage the bean is not associated with any particular object identity.
There exist 2 paths for moving from the *Pooled* state to the *Ready* state:
2.a. The client calls the create() method which causes the container to move the bean to the *Ready* state. Before doing so, the container calls the ejbCreate() and ejbPostCreate() methods.
2.b. The container itself calls the ejbActivate() method and moves the bean to the *Ready* state.
Likewise there are 2 paths to move a bean from the *Ready* to the *Pooled* state:
3.a. The client calls the remove() method, the container then calls the ejbRemove() method.
3.b. The container calls the ejbPassivate() method.
4. Before destroying the bean, the container calls unsetEntityContext().



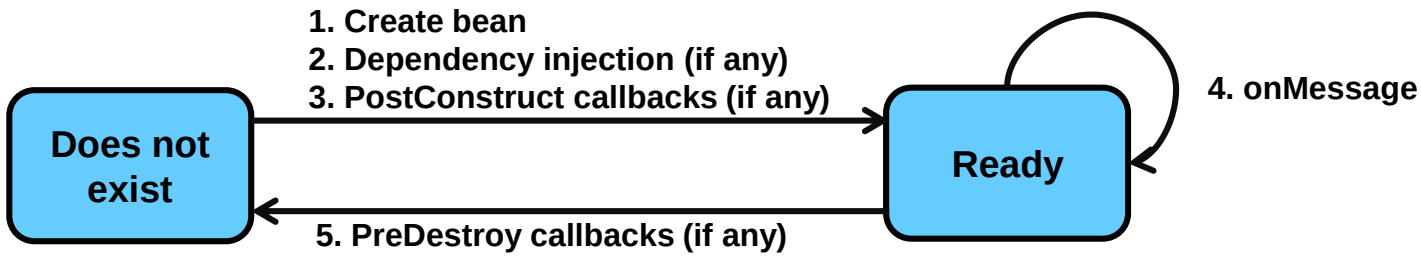
6. Lifecycle of EJBs (4/4)

6.4. Message driven beans (EJB 2.x and EJB 3.x):

The lifecycle of MDBs is similar to the lifecycle of stateless session beans.

Lifecycle steps:

1. The container creates a pool of MDBs.
2. The container injects dependencies into the bean.
3. The container calls methods annotated with @PostConstruct and activates the bean.
4. Upon message reception, the onMessage() method is called in which the bean processes the message.
5. Before the bean is destroyed, the container calls any @PreDestroy callback methods.



7. Session facade pattern for uniform bean access

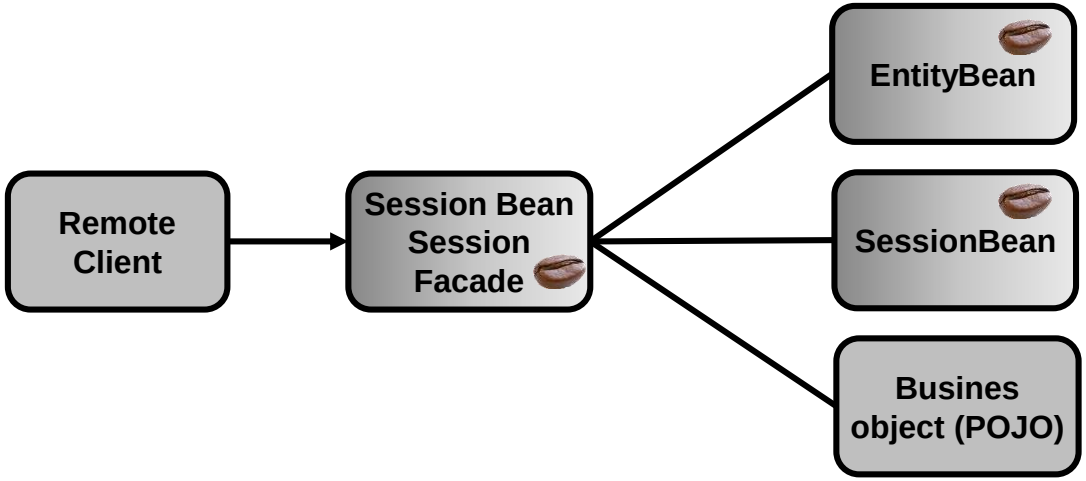
Common problems with beans:

→ Tight coupling between client objects and beans (direct dependency between clients and business objects).

→ Too many method invocations between client and server (network performance problems).

Proposed solution by Sun:

→ Use of a session bean as a facade that hides complexity from the client and manages business and data objects.



The facade provides a uniform interface to the client and hides the access to the business objects (entity and session beans).

The facade session bean provides high-level methods that dispatch individual method invocations to the attached beans.

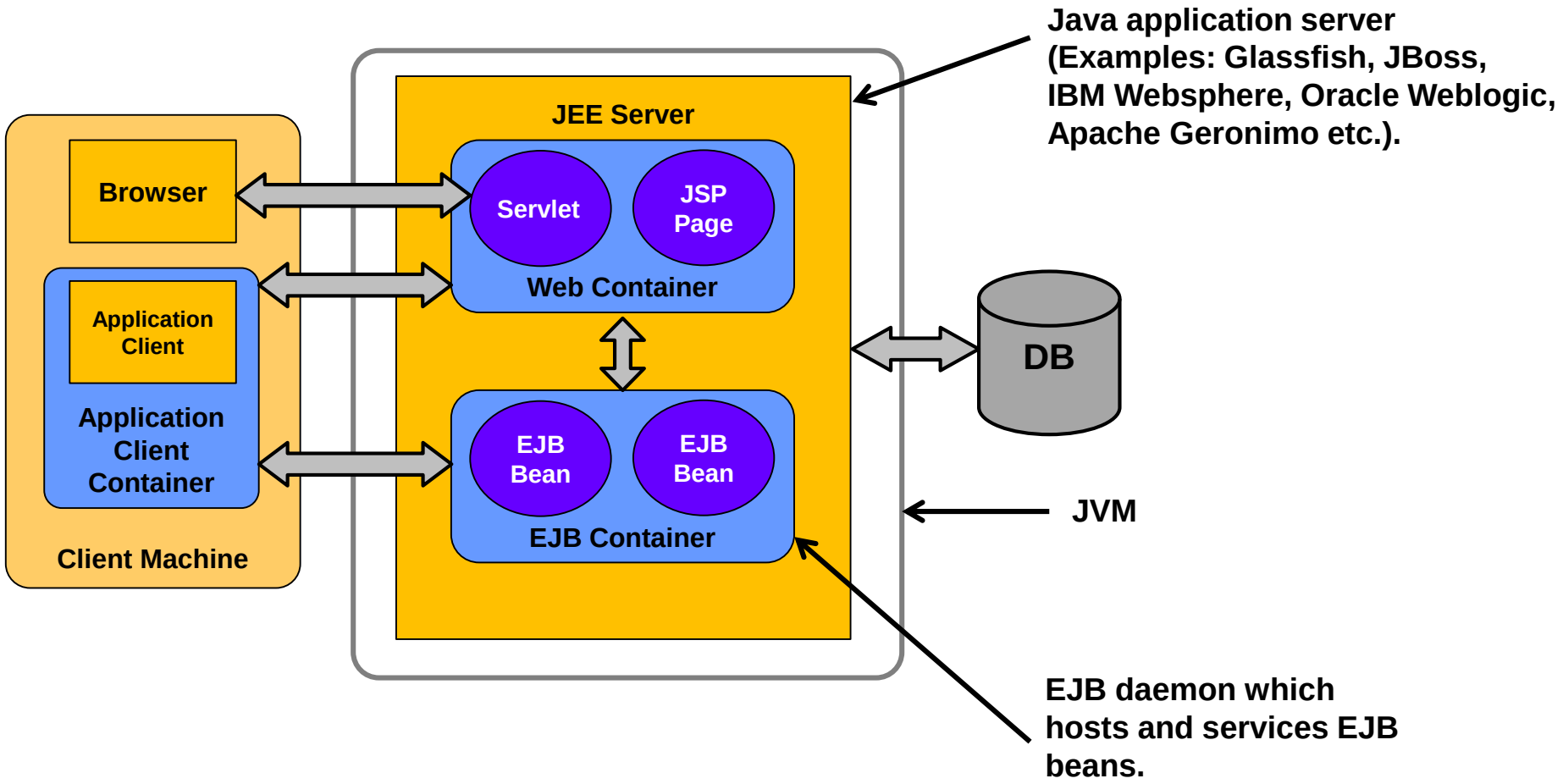
8. Bean deployment

In EJB 1.x and EJB 2.x, a deployment descriptor (XML-file) was required to inform the bean container (JEE application server) about the classes implementing the home and component (remote) interfaces, the type of bean etc.
In EJB 3.0, most of the XML elements were replaced by annotations.

```
<?xml version="1.0" encoding="UTF-8"?>
  <application-client version="5"
xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/application-
client_5.xsd">
    <description>My super application</description>
    <display-name>MrBean</display-name>
    <enterprise-beans>
      <session>
        <ejb-name>MySuperBean</ejb-name>
        <home>com.indigoo.wsmw.ejb.MySuperBeanHomeInterface</home>
        <remote>com.indigoo.wsmw.ejb.MySuperBeanInterface</remote>
        <ejb-class>com.indigoo.wsmw.ejb.MySuperBean</ejb-class>
        <session-type>Stateless</session-type>
      </session>
    </enterprise-beans>
  </application-client>
```

9. EJB container

The EJB container is the place where Java beans „live“ (are hosted).
EJB containers are part of the J2EE (application) server which in turn runs in a JVM.



10. Comparison of EJB with other DOT technology like CORBA

EJB and CORBA are similar in many ways. There are some notable differences:

Feature	CORBA	EJB
Middleware type	CORBA = explicit middleware: Client accesses directly the CORBA API	EJB = implicit middleware: Client is shielded from the specifics of the EJB-API
Language support	CORBA = language-neutral (there are various IDL-language-mappings)	EJB = Java only
Server-side integration	CORBA-POA-model is more flexible CORBA leaves it open how to integrate the server-side objects	EJBs are tightly integrated into the EJB-container
Interface specification	IDL (language-neutral syntax and semantics)	Interface specification = Java interface
Configuration	CORBA does not provide interfaces or concepts for configuration	EJB 1 and 2: EJB configuration is placed into a deployment descriptor (XML file) EJB 3: Configuration through annotations

CORBA: Common Object Request Broker Architecture
DOT: Distributed Object Technology
IDL: Interface Description Language
POA: Portable Object Adaptor

11. When to use EJB

EJB may be overkill in many applications and alternatives (namely Spring) may often be better suited to fulfill a certain task. The following table compares EJB and Spring:

Feature	EJB	Spring Framework
Multi-tiered application (distributed application, remoting)	Yes (client and server tier, separation of client and business logic is one of the main goals of EJB). Container-managed remote method calls.	Remoting through RMI, HTTPInvoker, JAX-RPC web services, JMS.
Standard platform, vendor independence	Yes (defined by JCP, supported by all major JEE vendors)	No (vendor = SpringSource)
Application server	JEE application server needed.	Spring comes with its own (lightweight) object container.
Dependency injection	Yes (EJB 3.0), but less powerful than Spring (only JNDI-objects, not POJOs).	Yes (even between POJOs).
Distributed transactions	Yes (must use JTA).	Yes (JTA and other transaction managers possible).
Persistence / ORM	Programmatic bean-managed persistence. Vendor specific ORM.	Integration with different persistence frameworks (Hibernate etc.)

JEE: Java Enterprise Edition
JCP: Java Community Process
JMS: Java Messaging Service
JTA: Java Transaction API
ORM: Object Relational Mapper
POJO: Plain Old Java Object