Q

What is EJB?

Jakarta Enterprise Beans

Jakarta Enterprise Beans (former Enterprise Java Beans) is one of several Java APIs for modular construction of enterprise software.

EJB is a server-side software component that encapsulates business logic of an application.

.

Jakarta Enterprise Beans

The current version (4.0 – May 2020) is a Jakarta (Apache) porting of the. Enterprise Java Beans

Former major versions:

EJB 3.0 (major restructuring)

EJB 2.0 (+ local beans)

EJB 1.0 (designed by IBM in 1998)

Introduction to Session beans

Enterprise Java Beans

(plain) Java Beans

JavaBeans are reusable software components for Java.

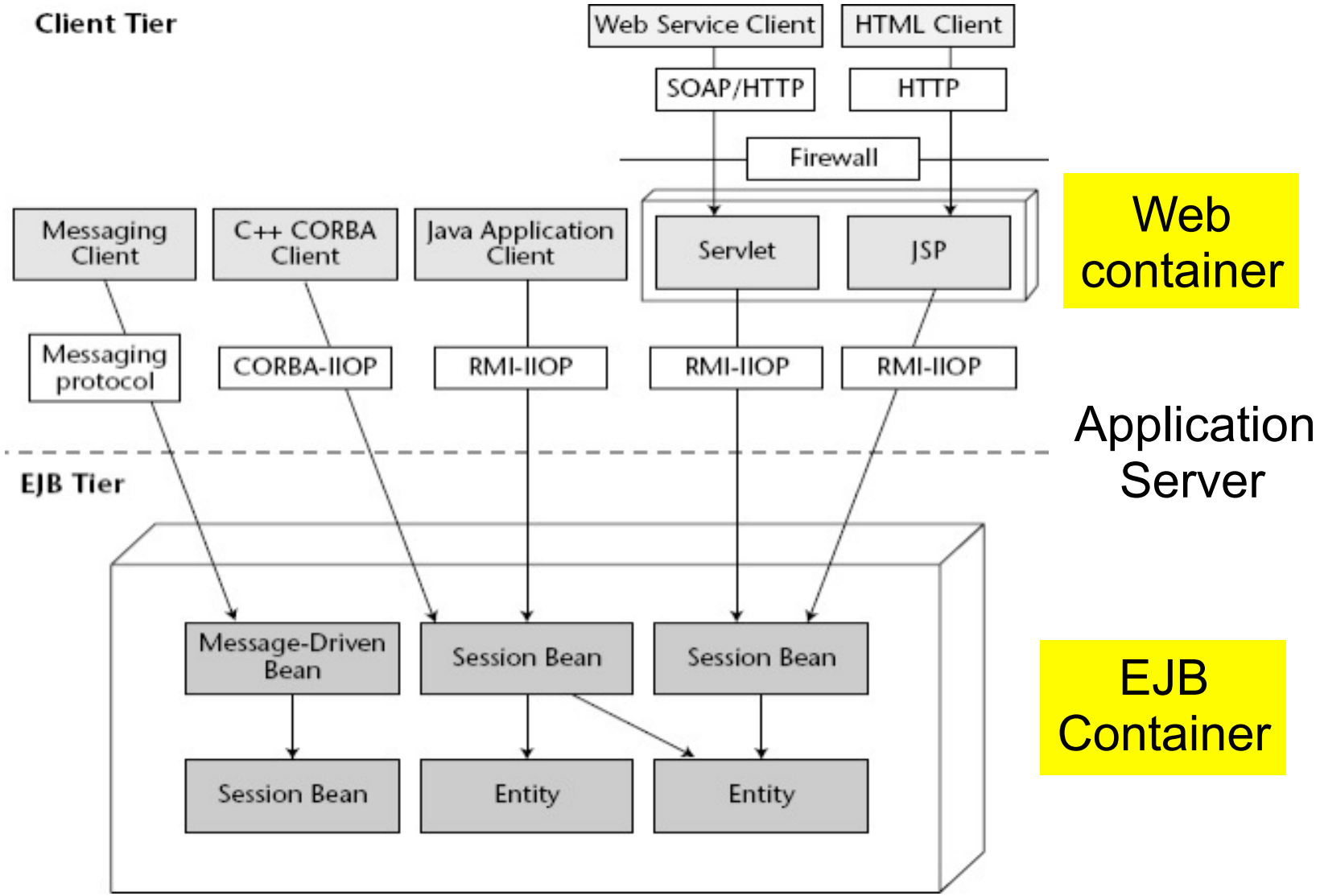
They are classes that encapsulate information and behavior into a single object (the bean).

They are serializable, have a 0-argument constructor, and allow access to properties using getter and setter methods.

Enterprise Java Beans

Entreprise Java Beans are reusable software components for Java, which live in a container that manages their lifecycle.

Architecture



Q

What are Stateless and Stateful beans?

Stateless vs. stateful session Beans

- **All instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.**

  => Stateless session beans can support multiple clients, and offer better scalability for applications that require large numbers of clients.

Stateful session Beans

- **A stateless session bean does maintain a conversational state for a particular client.**

  They are NOT reusable across different clients.

  Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

Stateful session Beans

- **A stateless session bean does maintain a conversational state for a particular client.**

  They are NOT reusable across different clients.

  Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

Examples: stateless

Sending an e-mail to customer support might be handled by a stateless bean, since this is a one-off operation and not part of a multi-step process.

A user of a website clicking on a "keep me informed of future updates" box may trigger a call to an asynchronous method of the session bean to add the user to a list in the company's database (this call is asynchronous because the user does not need to wait to be informed of its success or failure).

Fetching multiple independent pieces of data for a website, like a list of products and the history of the current user might be handled by asynchronous methods of a session bean as well (these calls are asynchronous because they can execute in parallel).

.

Special case: Singleton

Singleton Session Beans are business objects having a global shared state within a JVM. Concurrent access to the one and only bean instance can be controlled by the container (Container-managed concurrency, CMC) or by the bean itself (Bean-managed concurrency, BMC).

Loading a global daily price list that will be the same for every user might be done with a singleton session bean, since this will prevent the application having to do the same query to a database over and over again

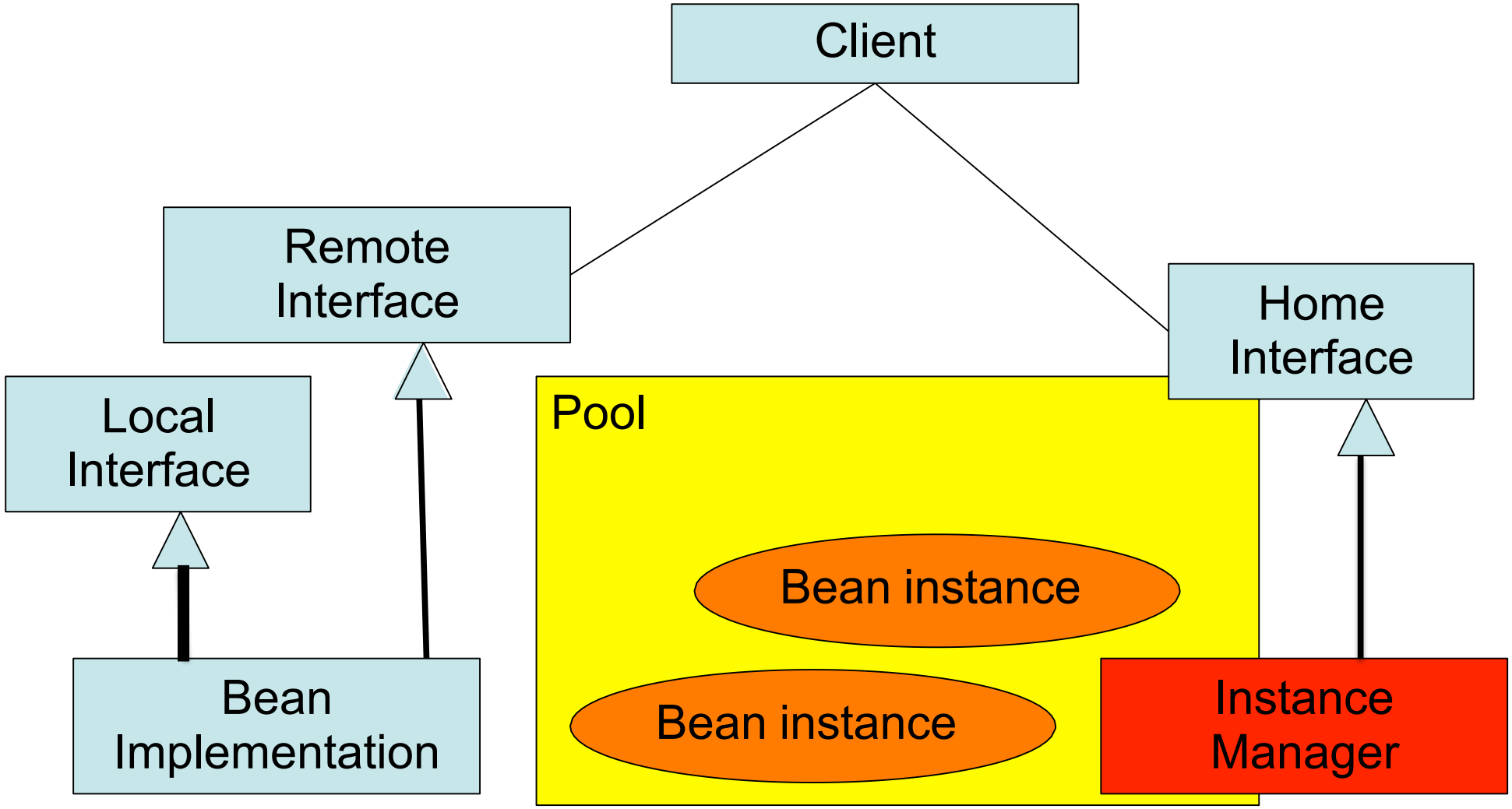
Examples: Stateful

Checking out in a web store might be handled by a stateful session bean that would use its state to keep track of where the customer is in the checkout process, possibly holding locks on the items the customer is purchasing (from a system architecture's point of view, it would be less ideal to have the client manage those locks).

Q

What is the overall architecture, and which are the ingredients?

Logical structure



EJB ingredients

- **Interfaces:** The remote and home interfaces are required for remote access. For local access, the local and local home interfaces are required.

- **Enterprise bean class:** Implements the methods defined in the interfaces.

- **Helper classes:** Other classes needed by the enterprise bean class, such as exception and utility classes.

- **Deployment descriptor:** see later

Remote Interface

```
/**
* This is the HelloBean remote interface.
*
* This interface is what clients operate on when
* they interact with EJB objects. The container
* vendor will implement this interface; the
* implemented object is the EJB object, which
* delegates invocations to the actual bean.
*/
public interface Hello extends javax.ejb.EJBObject
{
/**
* The one method - hello - returns a greeting to the client.
*/
   public String hello() throws java.rmi.RemoteException;
}
```

Must throw RemoteException

Home Interface

```
/**
* This is the home interface for HelloBean. This interface
* is implemented by the EJB Server's tools - the
* implemented object is called the Home Object, and serves
* as a factory for EJB Objects.
*
* One create() method is in this Home Interface, which
* corresponds to the ejbCreate() method in HelloBean.
*/
public interface HelloHome extends javax.ejb.EJBHome
{
/*
* This method creates the EJB Object.
*
* @return The newly created EJB Object.
*/
  Hello create() throws java.rmi.RemoteException,
                        javax.ejb.CreateException;
}
```

Bean Implementation

```
/**
* Demonstration stateless session bean.
*/
public class HelloBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext ctx;
    //
    // EJB-required methods
    //
    public void ejbCreate() { System.out.println("ejbCreate()"); }
    public void ejbRemove() { System.out.println("ejbRemove()"); }
    public void ejbActivate() { System.out.println("ejbActivate()");}
    public void ejbPassivate() {System.out.println("ejbPassivate()");}
    public void setSessionContext(javax.ejb.SessionContext ctx) {
      this.ctx = ctx; }
    //
    // Business methods
    //
    public String hello() {
        System.out.println("hello()");
        return "Hello, World!";
    }
}
```

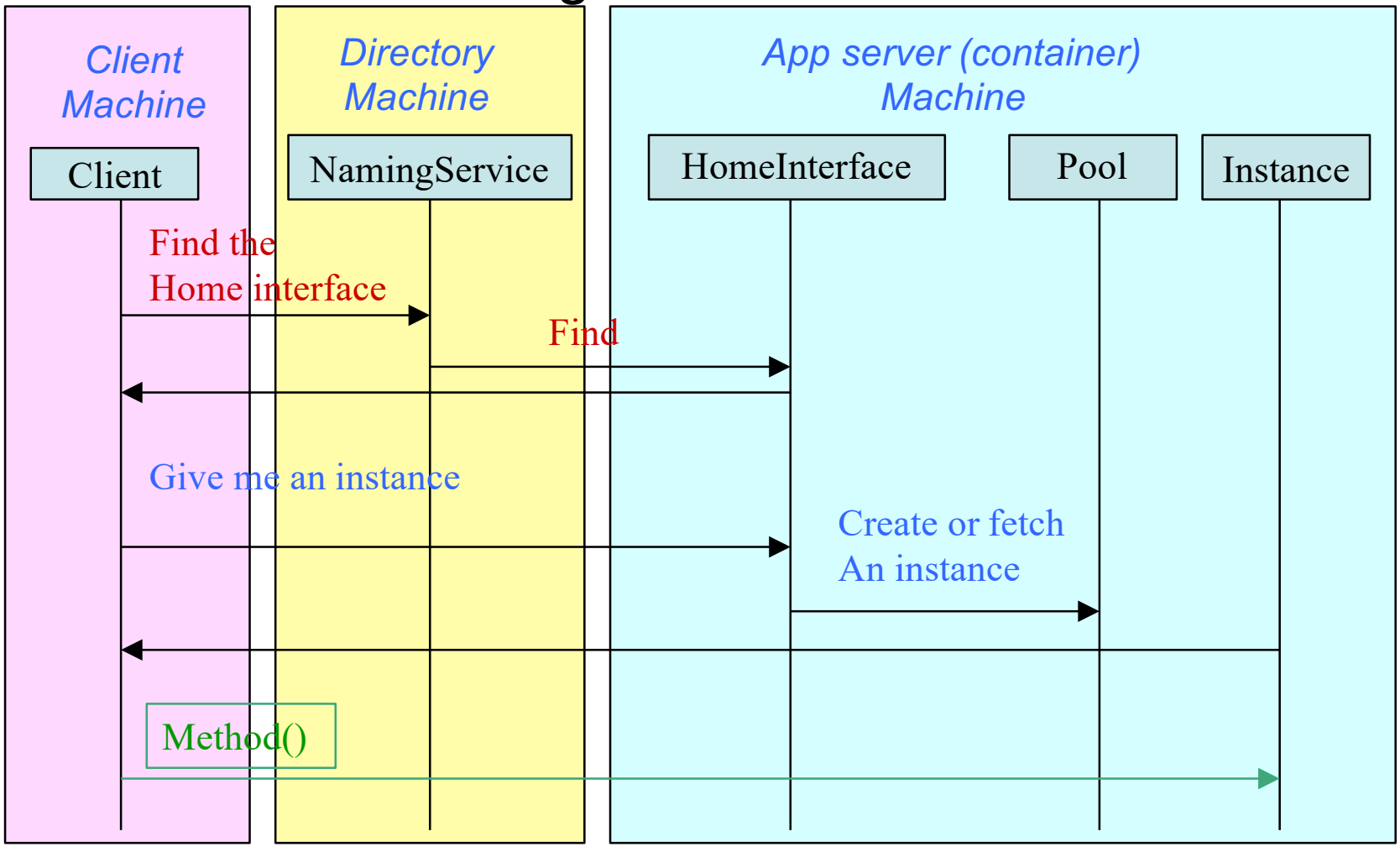
Client Implementation

```
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;
/**
* This class is an example of client code that invokes
* methods on a simple stateless session bean.
*/
public class HelloClient {
    public static void main(String[] args) throws Exception {
        /*
        * Setup properties for JNDI initialization.
        * These properties will be read in from the command line.
        */
        Properties props = System.getProperties();
        /*
        * Obtain the JNDI initial context.
        * The initial context is a starting point for
        * connecting to a JNDI tree. We choose our JNDI
        * driver, the network location of the server, etc.
        * by passing in the environment properties.
        */
        Context ctx = new InitialContext(props);
```

Client Implementation

```
        /* Get a reference to the home object - the
        * factory for Hello EJB Objects
        */
        Object obj = ctx.lookup("HelloHome");
        /* Home objects are RMI-IIOP objects, and so they must be cast
        * into RMI-IIOP objects using a special RMI-IIOP cast.
        */
        HelloHome home = (HelloHome)
        javax.rmi.PortableRemoteObject.narrow(obj, HelloHome.class);
        /* Use the factory to create the Hello EJB Object
        */
        Hello hello = home.create();
        /*Call the hello() method on the EJB object. The
        * EJB object will delegate the call to the bean,
        * receive the result, and return it to us.
        * We then print the result to the screen.
        */
        System.out.println(hello.hello());
        /*
        * Done with EJB Object, so remove it.
        * The container will destroy the EJB object.
        */
        hello.remove();
    }
}
```

The logical architecture



Deployment Descriptor

Deployment descriptor: An XML file that specifies information about the bean such as its transaction attributes.

- **You package the files in the preceding list into an EJB JAR file, the module that stores the enterprise bean.**
- **To assemble a J2EE application, you package one or more modules--such as EJB JAR files--into an EAR file, the archive file that holds the application.**

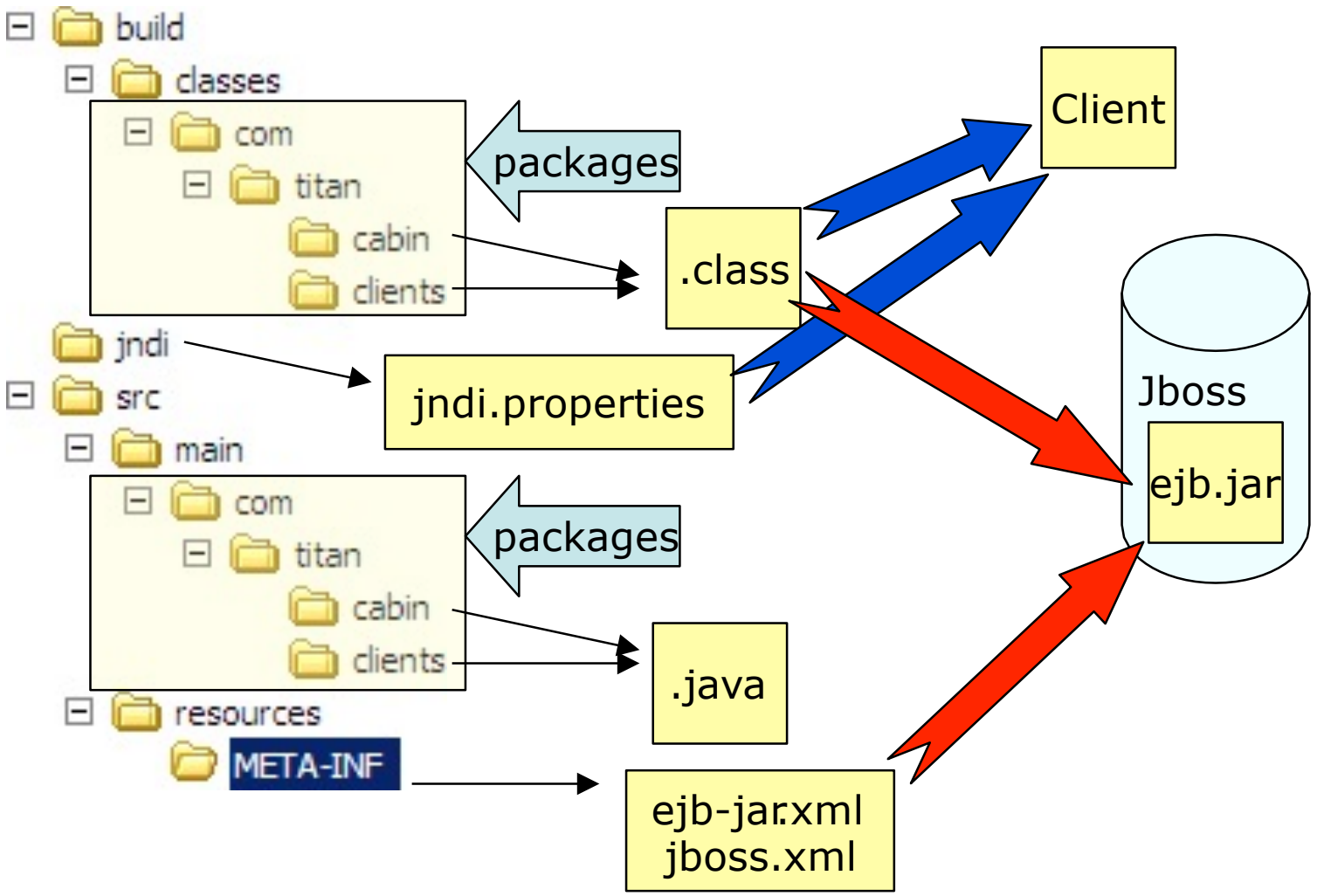
ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
    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/ejb-jar_2_1.xsd"
    version="2.1">
    <enterprise-beans>
        <session>
            <ejb-name>HelloWorldEJB</ejb-name>
            <home>examples.ejb21.HelloHome</home>
            <remote>examples.ejb21.Hello</remote>
            <local-home>examples.ejb21.HelloLocalHome</local-home>
            <local>examples.ejb21.HelloLocal</local>
            <ejb-class>examples.ejb21.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    ...
</ejb-jar>
```

ejb-jar.xml (continued)

```
<assembly-descriptor>
   <security-role>
      <description>   This role represents everyone who is allowed
                   full access to the HelloWorldEJB. </description>
     <role-name>everyone</role-name>
   </security-role>
   <method-permission>
     <role-name>everyone</role-name>
     <method>
         <ejb-name>HelloWorldEJB</ejb-name>
         <method-name>*</method-name>
     </method>
   </method-permission>
   <container-transaction>
     <method>
        <ejb-name>HelloWorldEJB</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
 </assembly-descriptor>
```

The file structure



Q

How did EJB 3.0 simplify the development?

Remote Interface

```
EJB 2.1 ================================================

public interface Hello extends javax.ejb.EJBObject
{
/**
* The one method - hello - returns a greeting to the client.
*/
   public String hello() throws java.rmi.RemoteException;
}

EJB 3.0 ================================================

package examples.session.stateless;
public interface Hello {
    public String hello();
}
```

business interface

Bean Implementation

```
EJB 2.1 ============================================================
public class HelloBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext ctx;
    public void ejbCreate() { System.out.println("ejbCreate()"); }
    public void ejbRemove() { System.out.println("ejbRemove()"); }
    public void ejbActivate() { System.out.println("ejbActivate()");}
    public void ejbPassivate() {System.out.println("ejbPassivate()");}
    public void setSessionContext(javax.ejb.SessionContext ctx) {
      this.ctx = ctx; }
    public String hello() {
        System.out.println("hello()");  return "Hello, World!";
    }
}
EJB 3.0 ============================================================
package examples.session.stateless;
import javax.ejb.Remote; import javax.ejb.Stateless;
@Stateless
@Remote(Hello.class)
public class HelloBean implements Hello {
    public String hello() {
        System.out.println("hello()"); return "Hello, World!";
    }
}
```

enterprise bean instance

The remote client – 3.0

```
package examples.session.stateless;
import javax.naming.Context;
import javax.naming.InitialContext;
public class HelloClient {
    public static void main(String[] args) throws Exception {
    /*
    * Obtain the JNDI initial context.
    *
    * The initial context is a starting point for
    * connecting to a JNDI tree.
    */
        Context ctx = new InitialContext();
        Hello hello = (Hello)
        ctx.lookup("examples.session.stateless.Hello");
    /*
    * Call the hello() method on the bean.
    * We then print the result to the screen.
    */
        System.out.println(hello.hello());
    }
}
```

ejb-jar.xml – 3.0

```
<?xml version="1.0" encoding="UTF-8" ?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_3_0.xsd"
version="3.0">
<enterprise-beans>
</enterprise-beans>
</ejb-jar>
```

Q

What are Local Beans?

Local Interface

```
/**
* This is the HelloBean local interface.
*
* This interface is what local clients operate
* on when they interact with EJB local objects.
* The container vendor will implement this
* interface; the implemented object is the
* EJB local object, which delegates invocations
* to the actual bean.
*/
public interface HelloLocal extends javax.ejb.EJBLocalObject
{
/**
* The one method - hello - returns a greeting to the client.
*/
   public String hello();
}
```

May throw EJBException instead of RemoteException

Local Home Interface

```
/**
* This is the home interface for HelloBean. This interface
* is implemented by the EJB Server's tools - the
* implemented object is called the Home Object, and serves
* as a factory for EJB Objects.
*
* One create() method is in this Home Interface, which
* corresponds to the ejbCreate() method in HelloBean.
*/
public interface HelloLocalHome extends javax.ejb.EJBLocalHome
{
/*
* This method creates the EJB Object.
*
* @return The newly created EJB Object.
*/
  HelloLocal create() throws javax.ejb.CreateException;
}
```

Local Client

```
Object ref = jndiContext.lookup("HelloHome");
HelloHome home = (HelloHome)
        PortableRemoteObject.narrow(ref,HelloHome.class);
...
HelloHome cabin_1 = home.create();
```

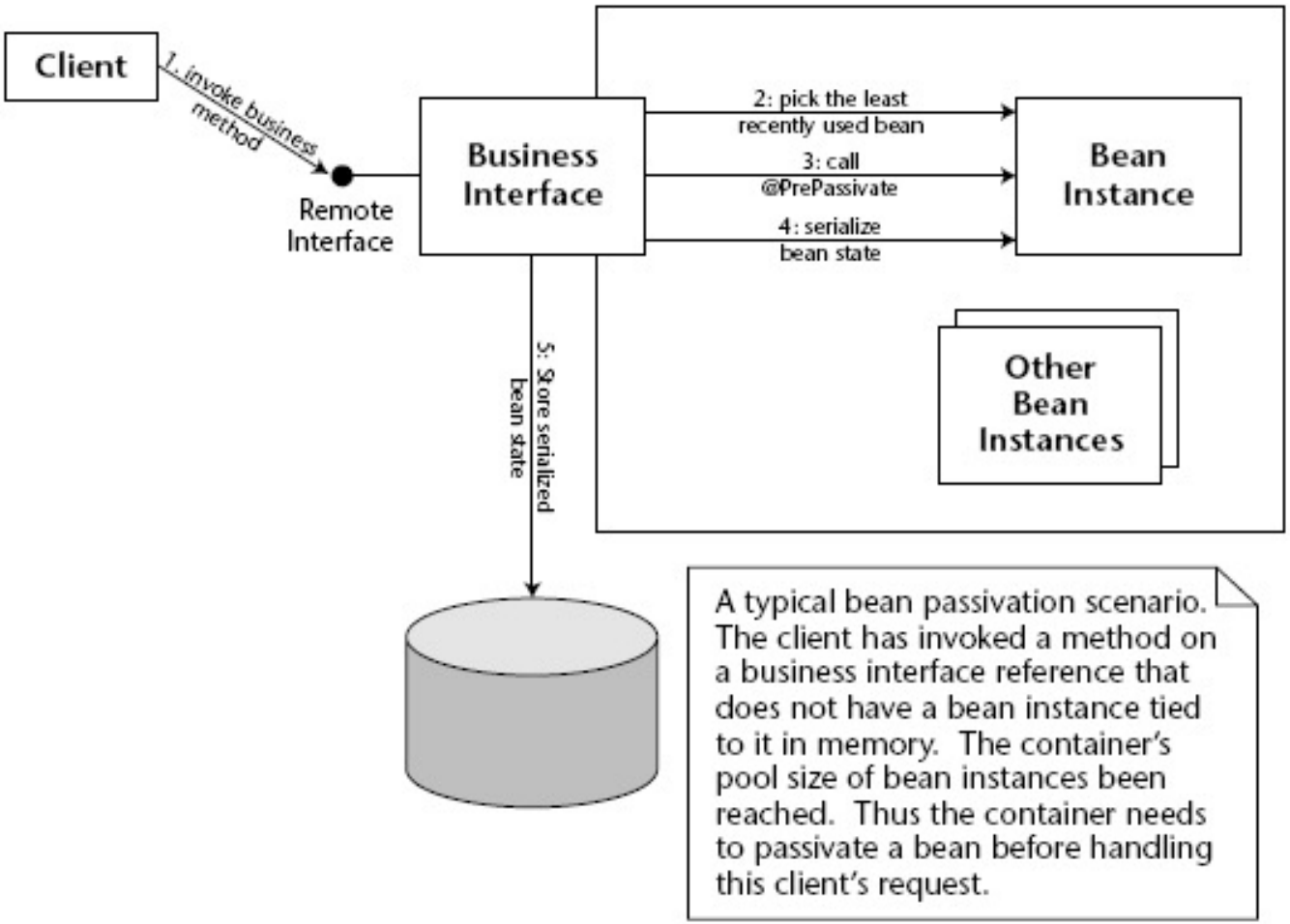
```
HelloLocalHome home = (HelloLocalHome )
        jndiContext.lookup("java:comp/env/ejb/ HelloLocalHome
");
...
HelloLocalHome cabin_1 = home.create();
```

We looked up a bean in *java:comp/env/ejb*.
This is the JNDI location that the EJB specification recommends (but does not require) you put beans that are referenced from other beans.

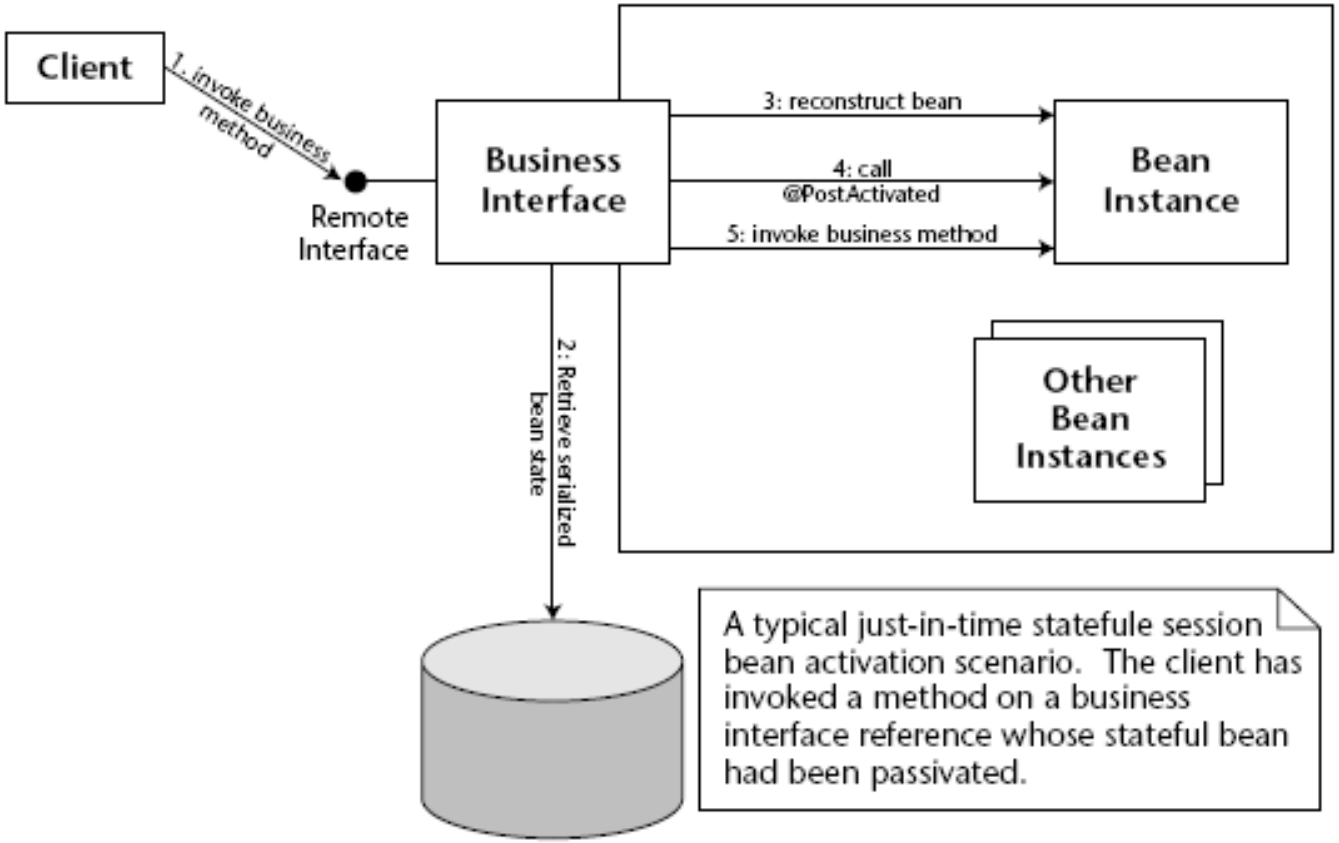
Q

How do we manage EJB lifecycle?

Passivation



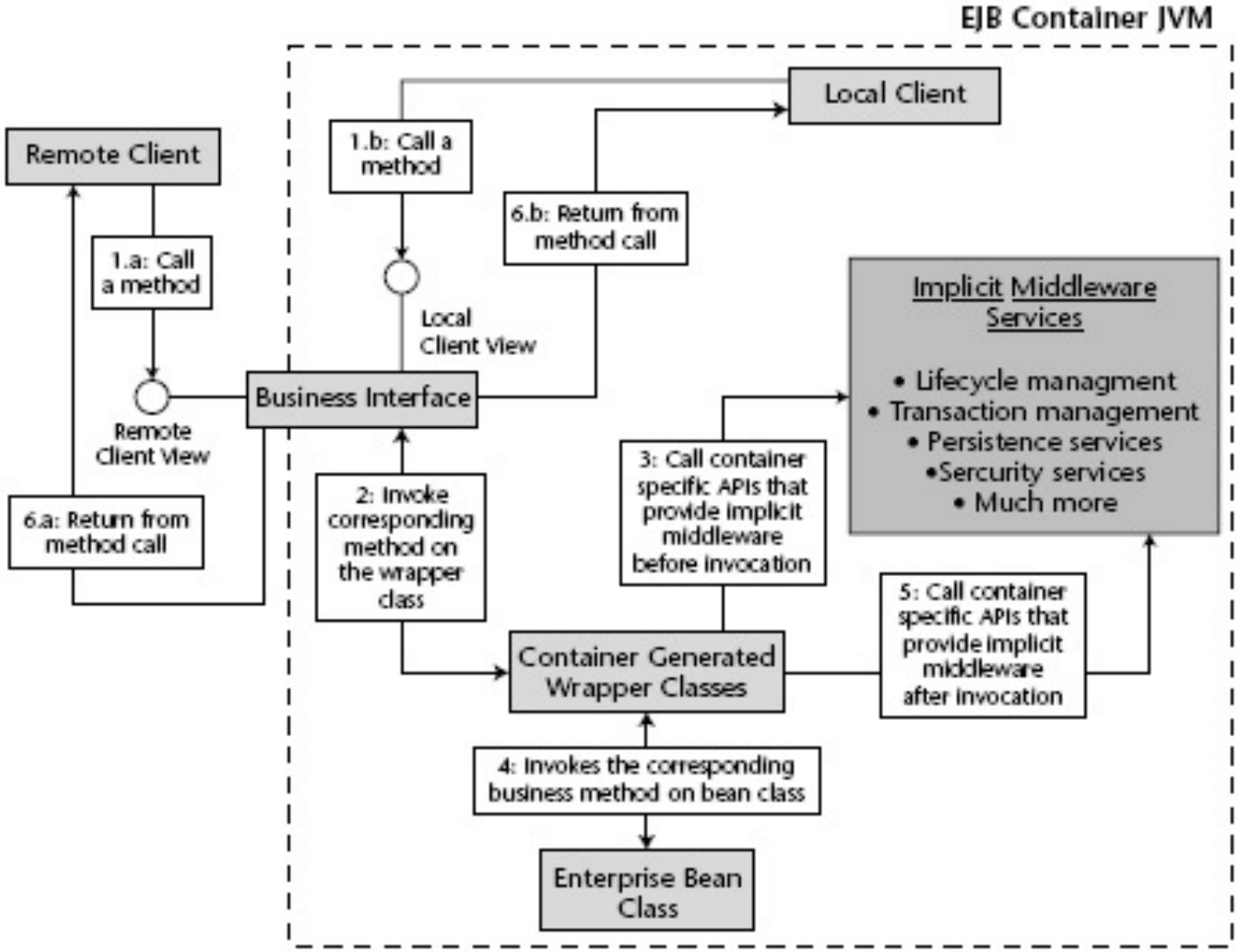
Activation


Client
1. invoke business method
Remote Interface
Business Interface
3: reconstruct bean
4: call @PostActivated
5: invoke business method
Bean Instance
Other Bean Instances
2: Retrieve serialized bean state
A typical just-in-time statefule session bean activation scenario. The client has invoked a method on a business interface reference whose stateful bean had been passivated.

Managing the lifecycle – 3.0

```
@Stateful
public class MyBean {
@PrePassivate
        public void passivate() {
        <close socket connections, etc...>
        }
        ...
@PostActivate
        public void activate() {
        <open socket connections, etc...>
        }
        ...
}
```

3.0 Lifecycle



Q

How do we deploy an EJB application?

jar & jar files

jar command

Java Archive

- inherits from tar : Tape Archive

commands:

jar cvf filename | jar tvf filename | jar xvf filename

java –jar filename.jar

jar file

A JAR file can contain Java class files, XML descriptor files, auxiliary resources, static HTML files, and other files

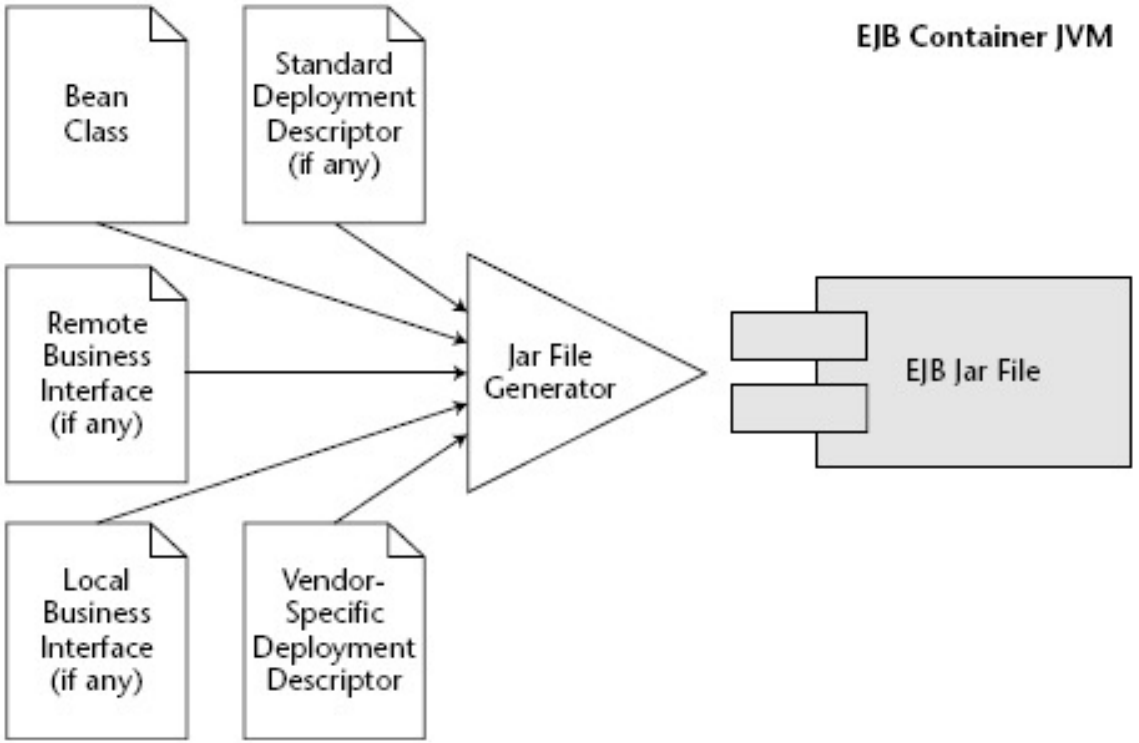
META-INF - Manifest

see http://docs.oracle.com/javase/tutorial/deployment/jar/

specialized jars:

- war
- ear

3.0 Packaging



Keep in mind these terms…

- The ***enterprise bean instance*** is a plain old Java object instance of an enterprise bean class. It contains business method implementations of the methods defined in the remote/local business interface, for session beans.

- The ***business interface*** is a plain old Java interface that enumerates the business methods exposed by the enterprise bean. Depending on the client view supported by the bean, the business interface can be further classified into a local business interface or a remote business interface.

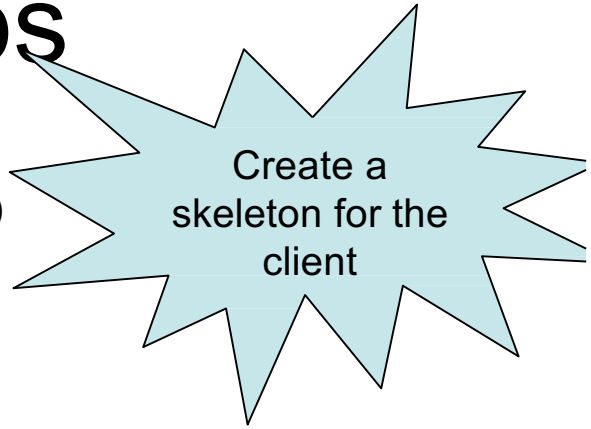
- The ***deployment descriptor*** is an XML file that specifies the middleware requirements for your bean. You use the deployment descriptor to inform the container about the services you need for the bean, such as transaction services, security, and so on. Alternatively, you can specify the middleware requirements using deployment annotations within the bean class as well.

Keep in mind these terms…

- The ***Ejb-jar*** file is the packaging unit for an enterprise bean, consisting of all the above artifacts. An EJB 3.0 Ejb-jar file can also consist of the old-style beans, if your application uses components defined using pre–EJB 3.0 technologies.

- The ***vendor-specific deployment descriptor*** lets you specify your bean's needs for proprietary container services such as clustering, load balancing, and so on. Avendor can alternatively provide deployment metadata for these services, which, like standard metadata, can be used within the bean class to specify the configuration for these services. The vendor-specific deployment descriptor's definition changes from vendor to vendor.

Development steps

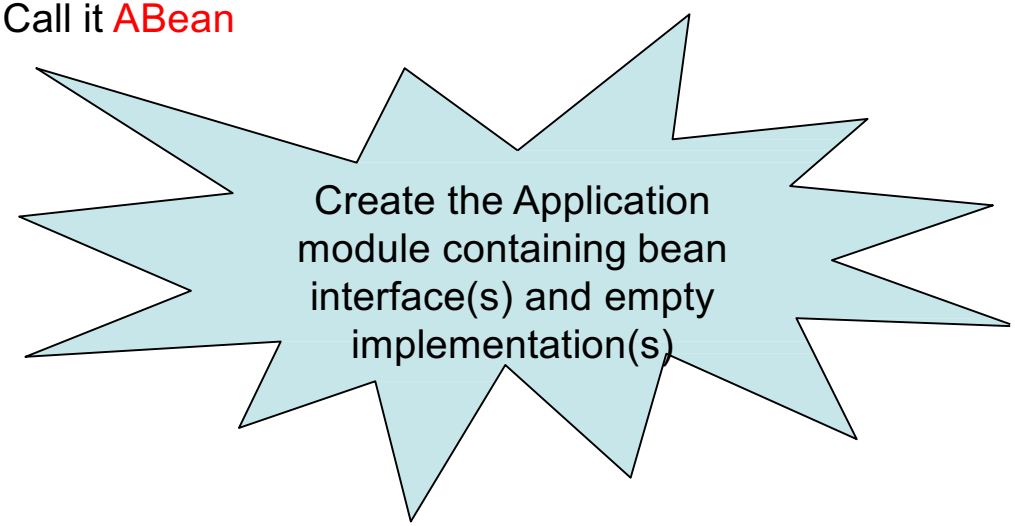


Create a new project for an empty EJB Application Client (Class AppCli)

- New Project -> Java →→Java Application
- name it AppCli in package pack1

Create New Project for the EJB Application

- New Project -> Java EE -> Enterprise Application
- name it EntApp
- choose ONLY “Create EJB Module” (not “Web Application Module”) and name it AppServ, create a Remote Interface in project AppCli. Call it ABean



Development steps

Fill the body of the bean(s)

Go to the Project AppServ

- go to the class ABean, right click “insert code”
- add a Business method called method, with a param String and return type String

- Clean and build the project

Create the jar for the Application module

Go to the EntApp

- Clean and build the project

Create the ear for the application

- Look in its dist folder: take the EntApp.ear file and drop it in the JBOSS standalone/deployments folder.
- Look at the JBoss console to find the JNDI reference

(java:jboss/exported/EntApp/AppServer/ABean!package_name.BBeanRemote)

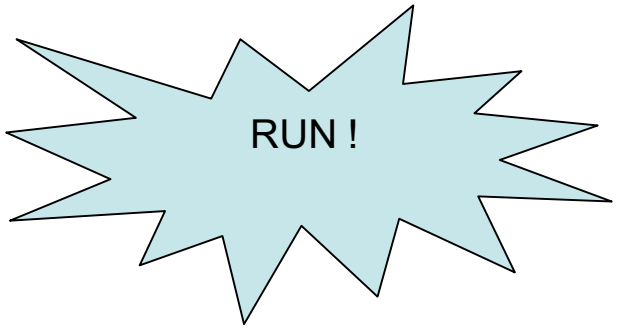
Deploy the application

Development steps

Go to AppCli ,

- fix the JNDI Access to the bean (using the info above).
- Remove any unneeded library
- Add the to the library the jboss-client.jar

Run the client!



Q

What is important to remember about session beans?

Let's play a bit…

```
package session;
import javax.ejb.Stateful;
@Stateful
public class StatefulSessionBean implements
StatefulSessionBeanRemote {
    int counter=0;

    @Override
    public String ping() {
        counter++;
        return "SF hits ="+counter;
    }
}
```

Let's play a bit…

```
package session;
import javax.ejb.Stateless;
@Stateless
public class StatelessSessionBean implements
StatefulSessionBeanRemote {
    int counter=0;

    @Override
    public String ping() {
        counter++;
        return "SL hits ="+counter;
    }
}
```

A client with stateful and stateless

```
package _client;
public class _Client {
public static void main(String[] args) throws NamingException {
 Properties jndiProps = new Properties();
 jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jboss.naming.remote.client.InitialContextFactory");
 jndiProps.put(Context.URL_PKG_PREFIXES,"org.jboss.ejb.client.naming");
        jndiProps.put(Context.PROVIDER_URL, "remote://localhost:4447");
 jndiProps.put(Context.SECURITY_PRINCIPAL, "user");
 jndiProps.put(Context.SECURITY_CREDENTIALS, "pw");
 jndiProps.put("jboss.naming.client.ejb.context", true);
 Context ctx=new InitialContext(jndiProps);
```

A client with stateful and stateless

```
StatelessSessionBeanRemote bean = (StatelessSessionBeanRemote)
       ctx.lookup("_Server/_Server-ejb/StatelessSessionBean!
       session.StatelessSessionBeanRemote");
StatefulSessionBeanRemote sf_bean = (StatefulSessionBeanRemote)
       ctx.lookup("_Server/_Server-ejb/StatefulSessionBean!
       session.StatefulSessionBeanRemote");
StatelessSessionBeanRemote bean1 = (StatelessSessionBeanRemote)
       ctx.lookup("_Server/_Server-ejb/StatelessSessionBean!
       session.StatelessSessionBeanRemote");
StatefulSessionBeanRemote sf_bean1 = (StatefulSessionBeanRemote)
       ctx.lookup("_Server/_Server-ejb/StatefulSessionBean!
       session.StatefulSessionBeanRemote");
```

A client with stateful and stateless

```
        System.out.println(bean.ping());
        System.out.println(bean.ping());
        System.out.println(bean.ping());
        System.out.println(sf_bean.ping());
        System.out.println(sf_bean.ping());
        System.out.println(sf_bean.ping());
        System.out.println(bean1.ping());
        System.out.println(bean1.ping());
        System.out.println(bean1.ping());
        System.out.println(sf_bean1.ping());
        System.out.println(sf_bean1.ping());
        System.out.println(sf_bean1.ping());
    }
}
```

Execution results

```
SL hits =1
SL hits =2
SL hits =3
SF hits =1
SF hits =2
SF hits =3
SL hits =4
SL hits =5
SL hits =6
SF hits =1
SF hits =2
SF hits =3
```

No instance variables in stateless!

```
package session;
import javax.ejb.Stateless;
@Stateless
public class StatelessSessionBean implements
StatefulSessionBeanRemote {
    int counter=0;

    @Override
    public String ping() {
        counter++;
        return "SF hits ="+counter;
    }
}
```