EJB 3.1 Fundamentals

This course covers the following aspects of the EJB 3.1 specification:

- What are the key concepts behind the EJB
- Recognizing the different types of EJB
- Understanding when one should use various types of EJB components
- Creating simple EJB components
- Invoking EJB periodically
- Making EJB components transactional
- Understanding basic dependency injection rules that can be applied to EJB
- Basic deployment rules of EJB components

Please note that Entity Beans have been replaced by the JPA specification since the creation of EJB 3.0 onwards. Due to this reason, Entity Beans are not included in the scope of this course. And JPA is not covered in this course either because it's not part of the EJB 3.1 certification. Its a separate certification on its own.

1. Introduction to EJB

In this lesson, we will create a first EJB and deploy it on the Glassfish application server. Then, we will review the distributed architectures and the services provided by EJB containers.

For example, your may have a complex business logic for computing the salary of an employee taking many parameters into account such as working hours and taxes.

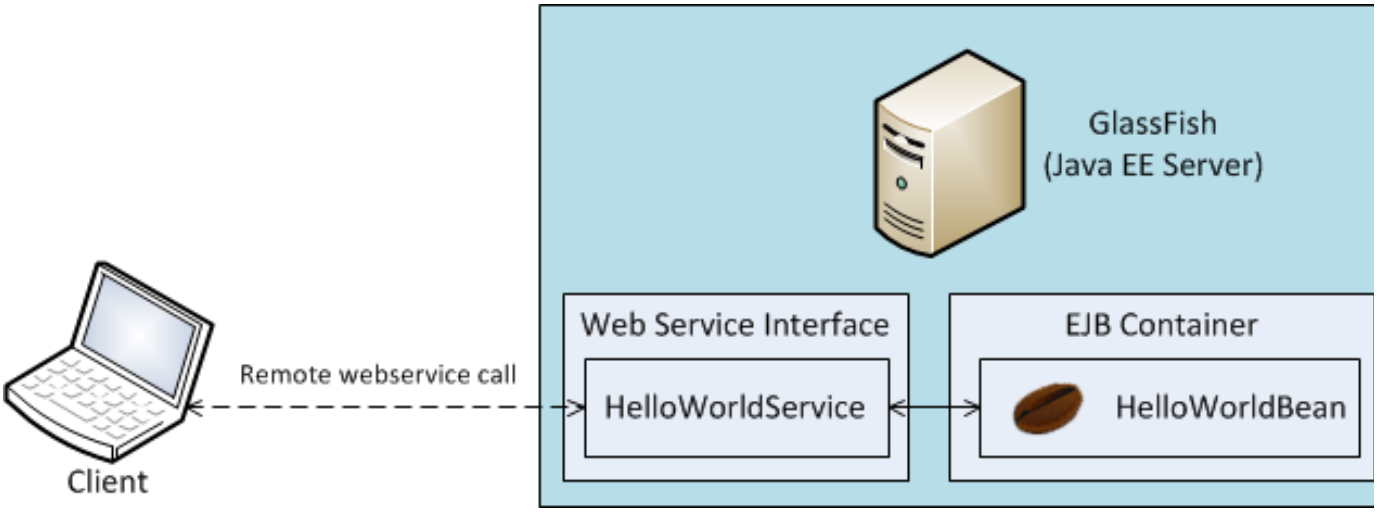
1.1. Creating a First EJB

Java programmers using the Enterprise Java Beans (EJB) technology need to have an EJB server and an IDE setup to work together.

In this topic, you will learn the simplest way of creating your first EJB. We choose to create stateless session beans because this is the simplest kind of EJB components. The bean will be exposed via web service in order to make the testing easier.

We will deploy and run our component using the Glassfish server. Then, we will develop the component using the Eclipse or NetBeans IDE. GlassFish, Eclipse, and NetBeans are free tools, available for download from the internet.

You will end up with the following architecture :



1.1.1. EJB - An Overview

Java EE includes:

- Servlet & JSP for developing web applications inside a web container
- EJB

EJBs are components (Java classes) running inside an EJB container to perform some business logic.

There are two main kinds of EJB:

- **Session Beans**. They are activated through the method calls (as usual Java classes) either from another EJB or from a remote client.
- **Message Driven Beans**. They listen to a queue and are activated asynchronously when a message arrives on that queue.

Note, for those who have information about EJB in the past, that *entity beans* are not in the EJB specification anymore, but in the JPA specification covered by the JPA/Hibernate courses.

The first EJB that you will write now is a session bean and will run on the *GlassFish* EJB container. An EJB container is a software server environment where EJBs are deployed. An EJB cannot run outside an EJB container.

1.1.2. Activity: Configuring an EJB Container

In order to create your first EJB 3.1, you need to download, configure, and run an EJB container. In this course, we will use the open source application server named Glassfish. We will run an embedded instance of it using the GlassFish Eclipse plug-in.

By installing the Eclipse plug-in, we will get two softwares:

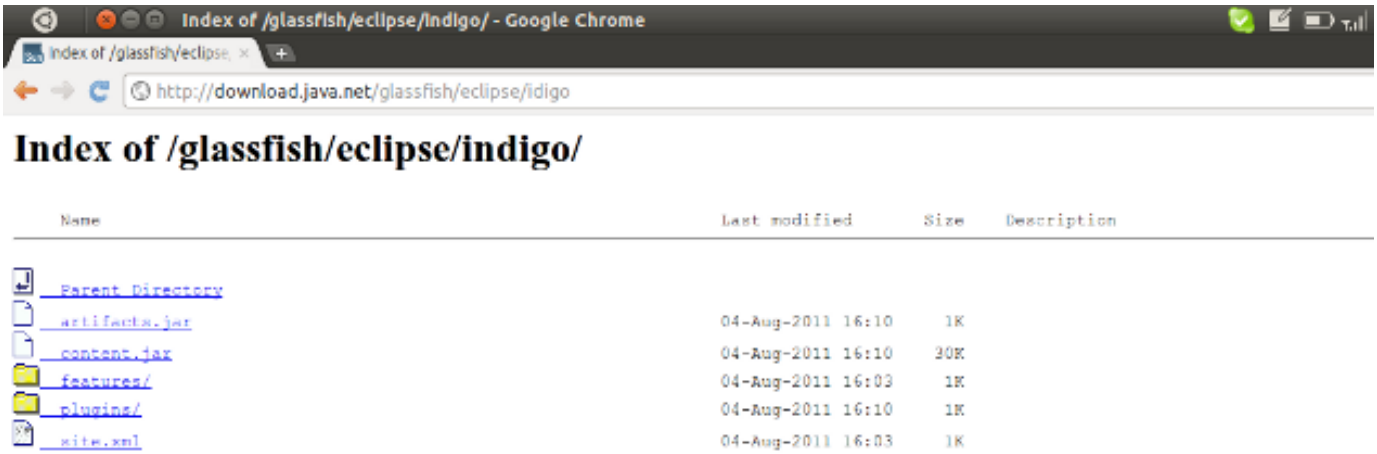
1. GlassFish application server (that we could have downloaded separately)
2. GlassFish plug-in for Eclipse

Follow these steps:

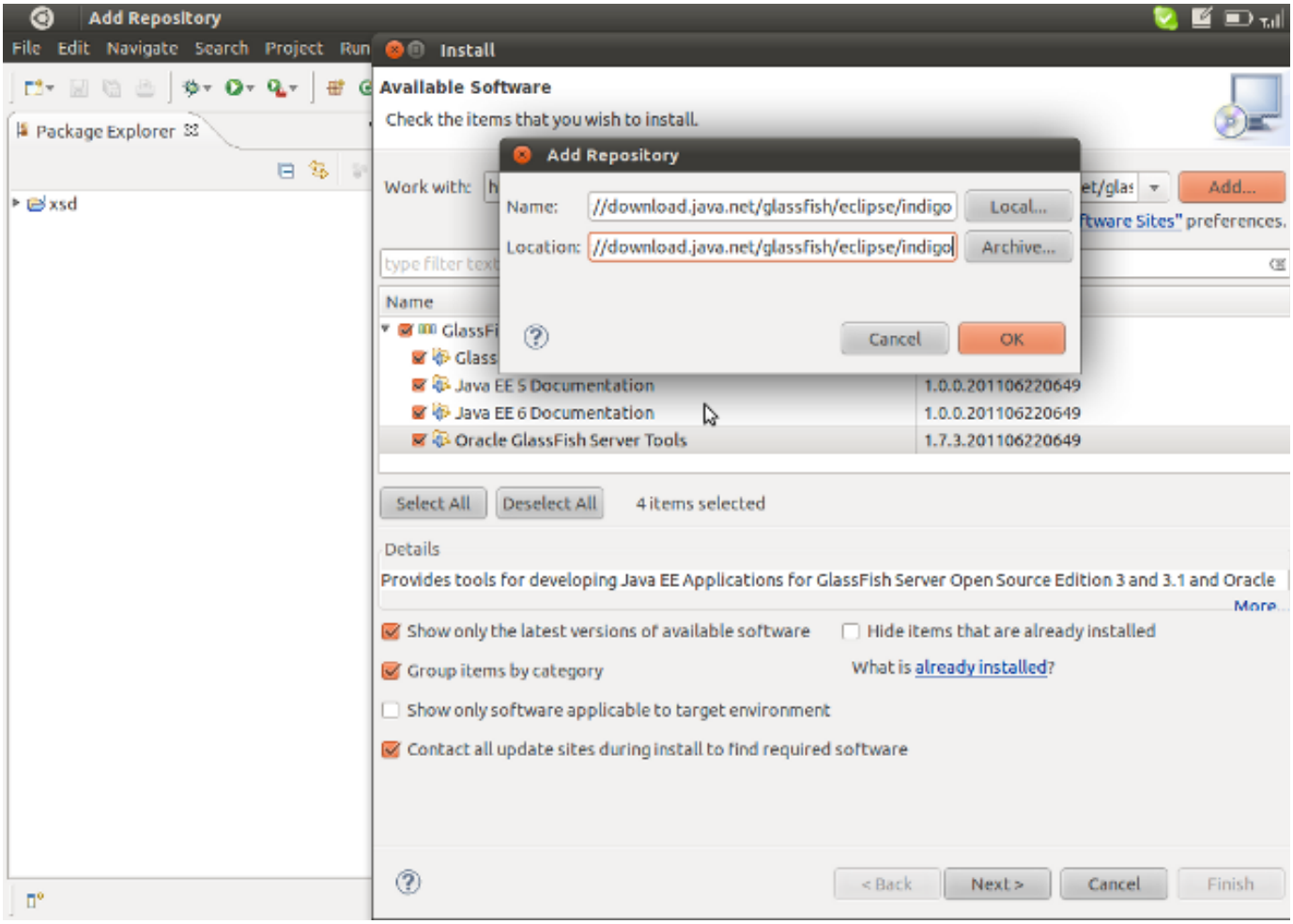
1. Open a web browser to the GlassFish plugin site: http://glassfishplugins.java.net
2. Note the update site URL for Eclipse. For Eclipse 3.6 (Helios), it is:
   http://download.java.net/glassfish/eclipse/helios
3. In Eclipse, select the menu Help > Install New Software
   Click the Add... button, and type the update site URL, you just noted, in the location field.

4.
5. Type "GlassFish" in the first field and click OK.
6. In the tree, select *GlassFish 3.1 Application Server runtime*, and *Oracle GlassFish Server Tools*. Press Next.
7. Continue the Wizard steps until you restart Eclipse.

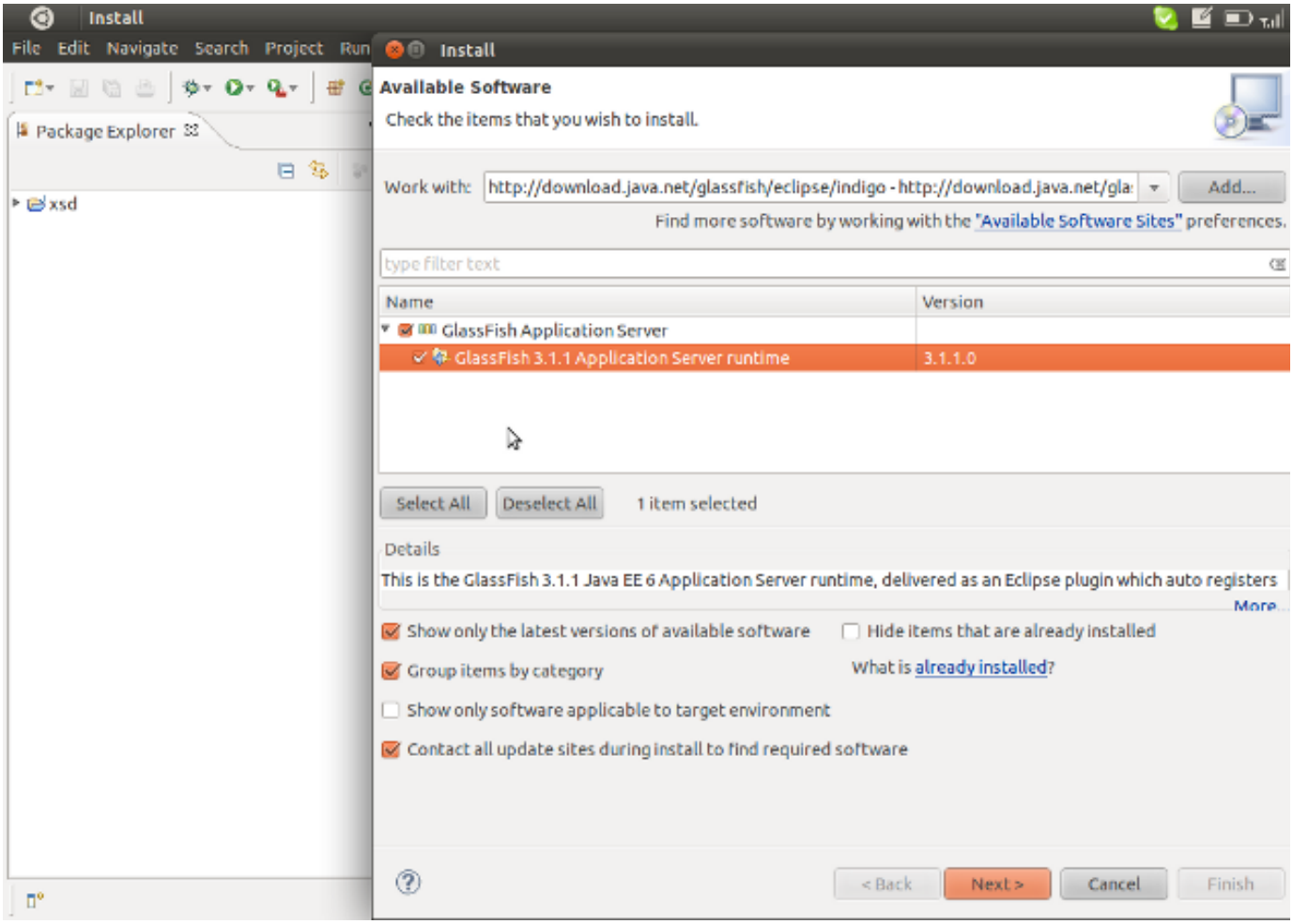
Some screenshots of the activity described above:



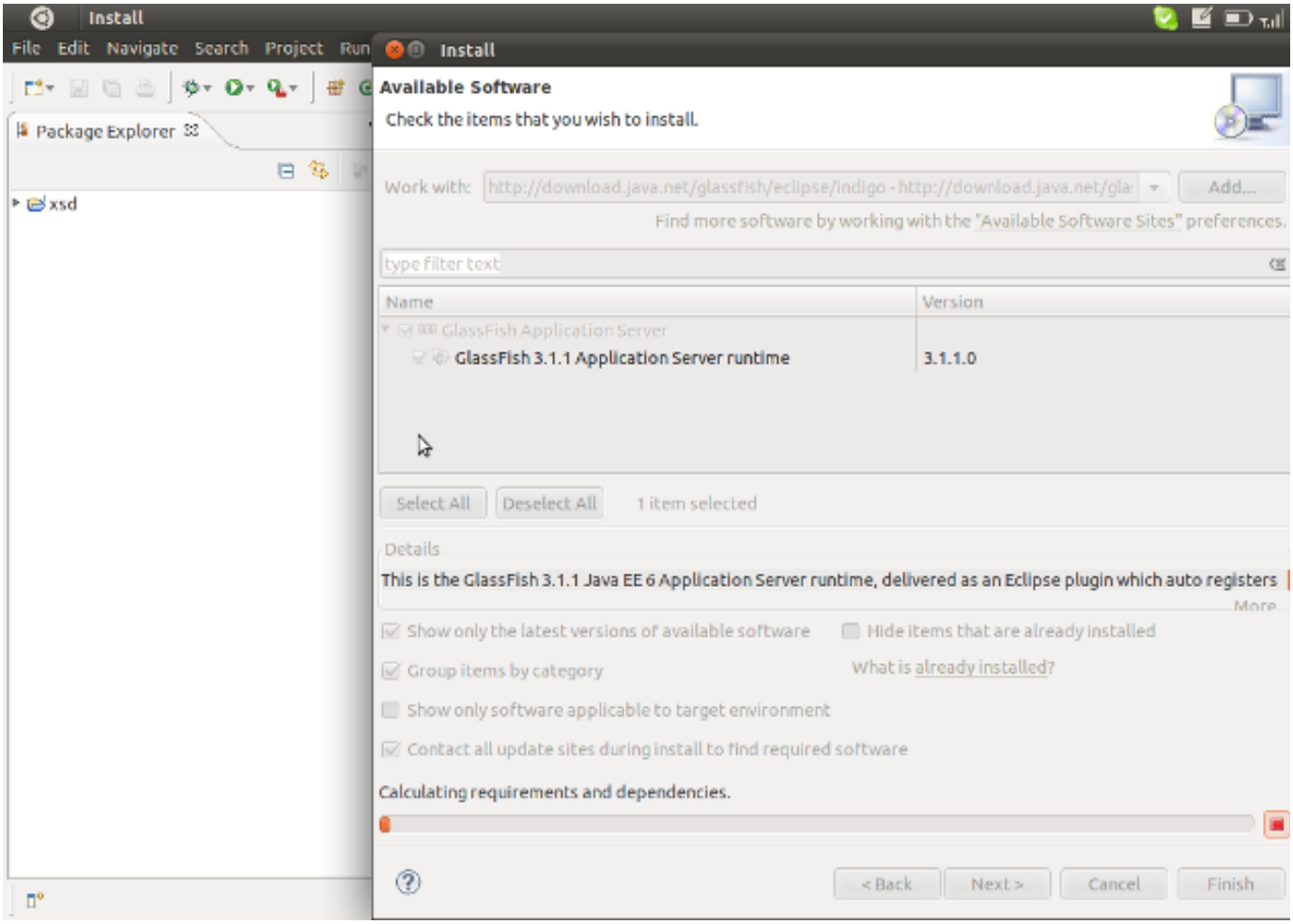
Add the latter URL to your Eclipse discovery site catalog.



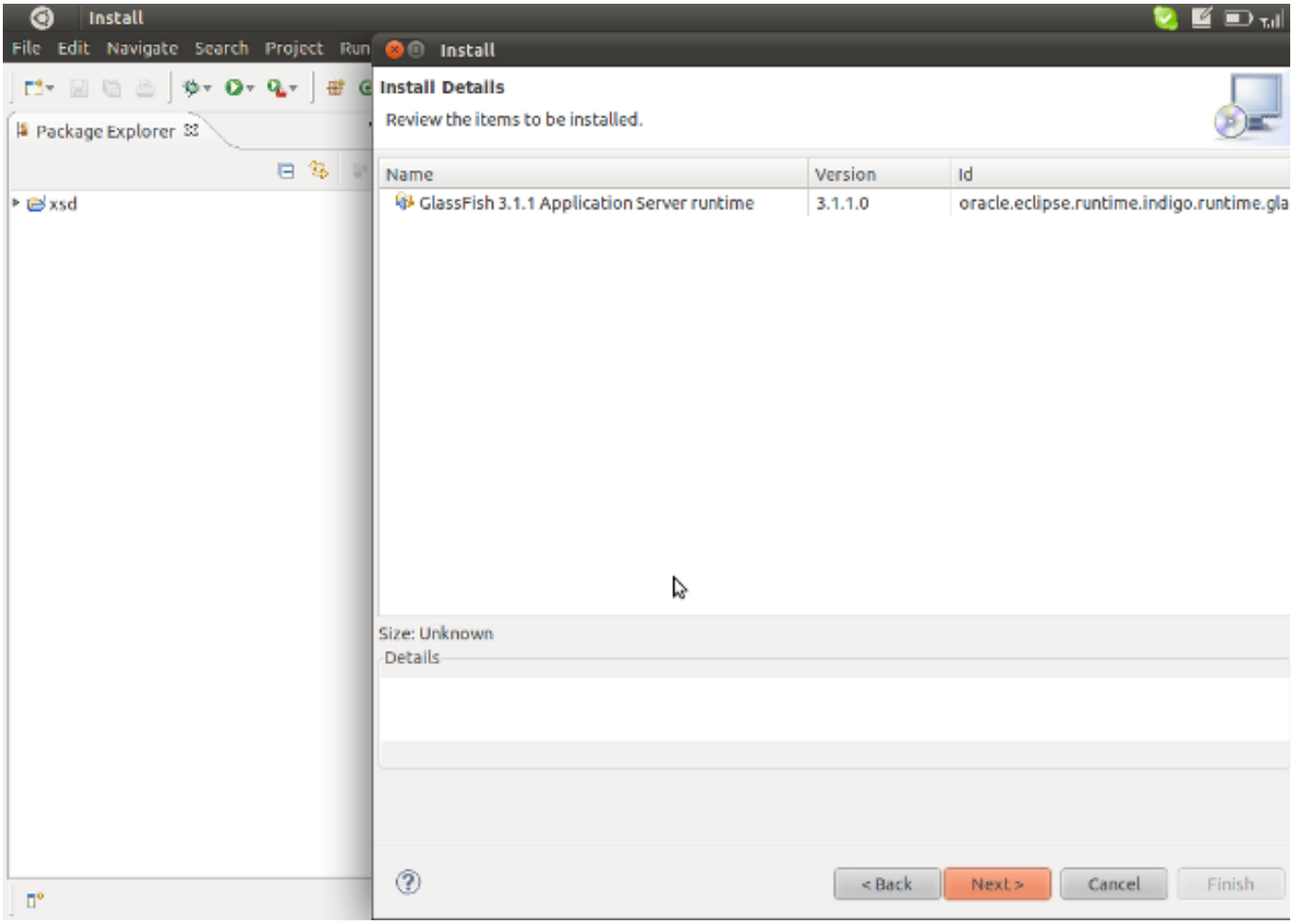
Now, download the Glassfish Application Server Runtime plugin.



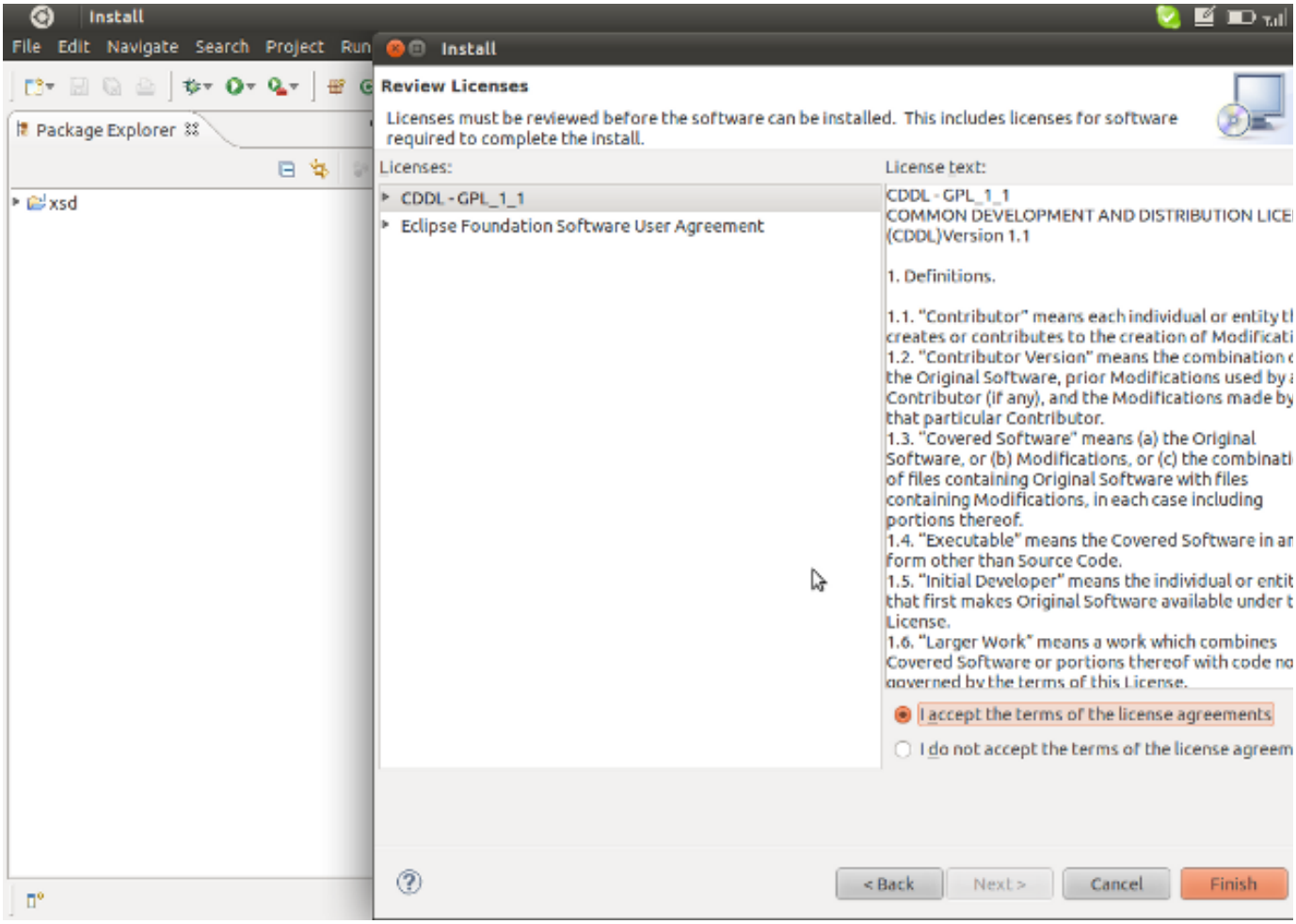
Give Eclipse some time to calculate the dependencies of the plugin.



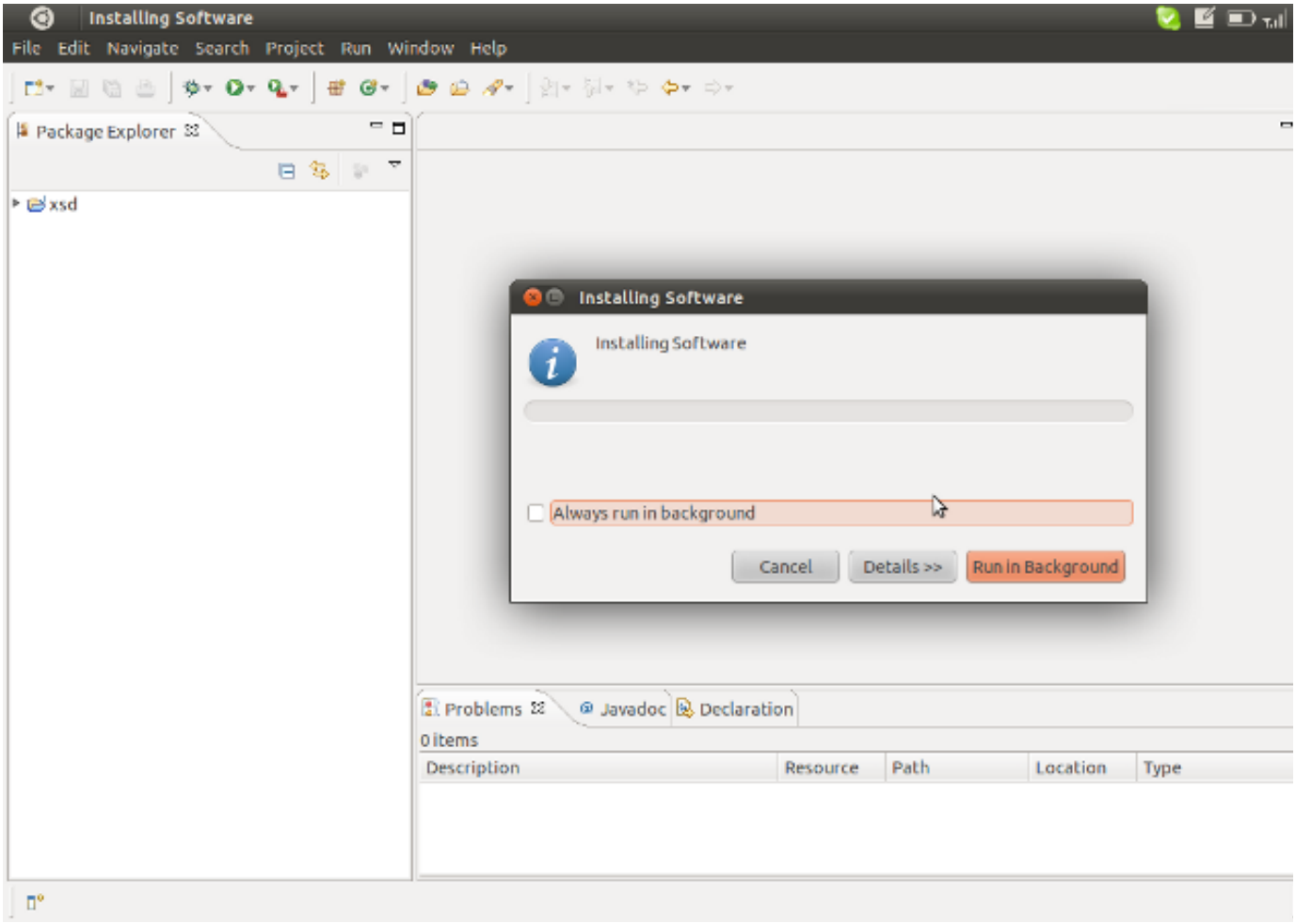
Confirm that you want to install it.



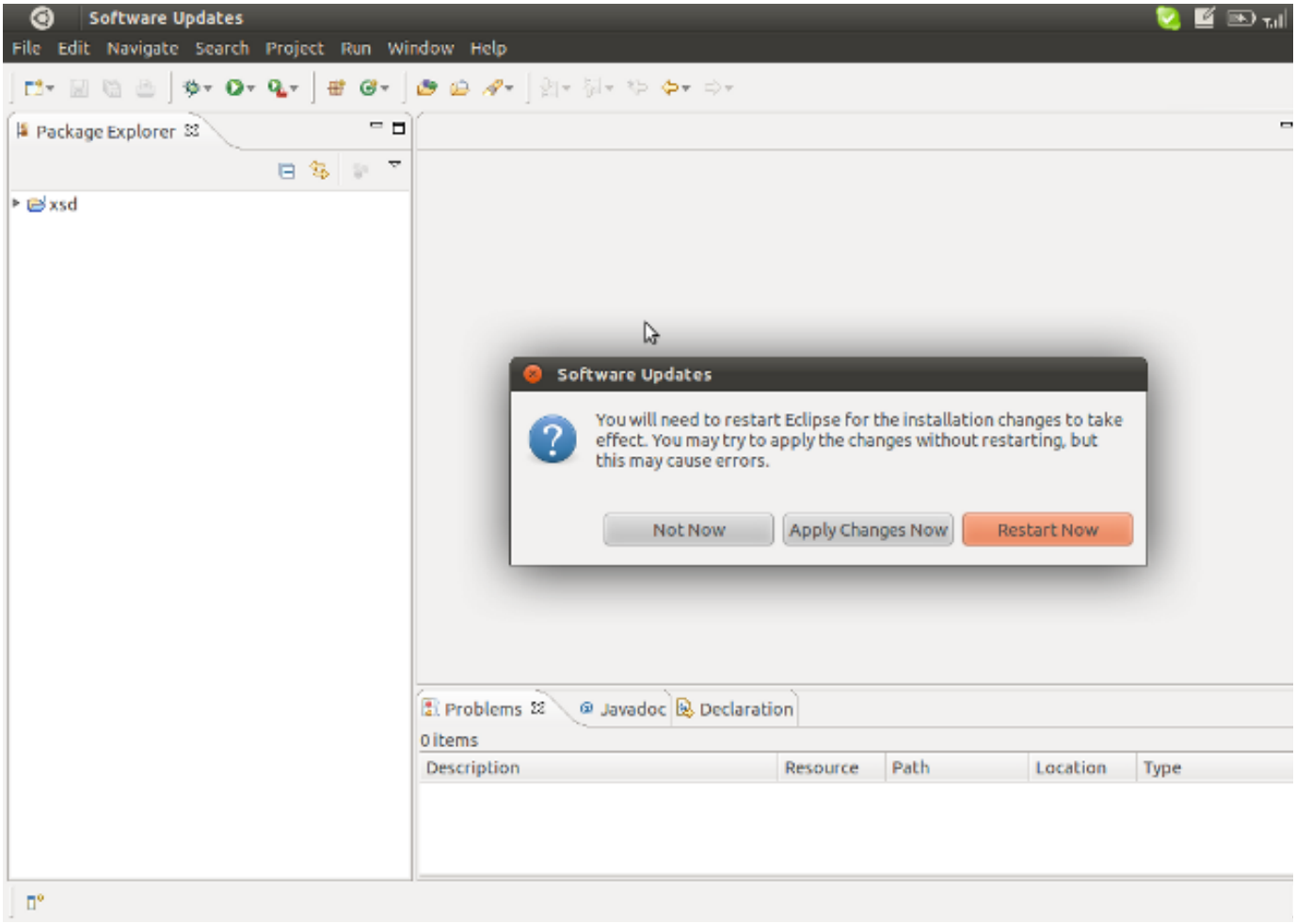
Accept the Glassfish plug-in license.



Let Eclipse download the plug-in and Glassfish bundle.



Now, restart the IDE as Eclipse itself suggests.

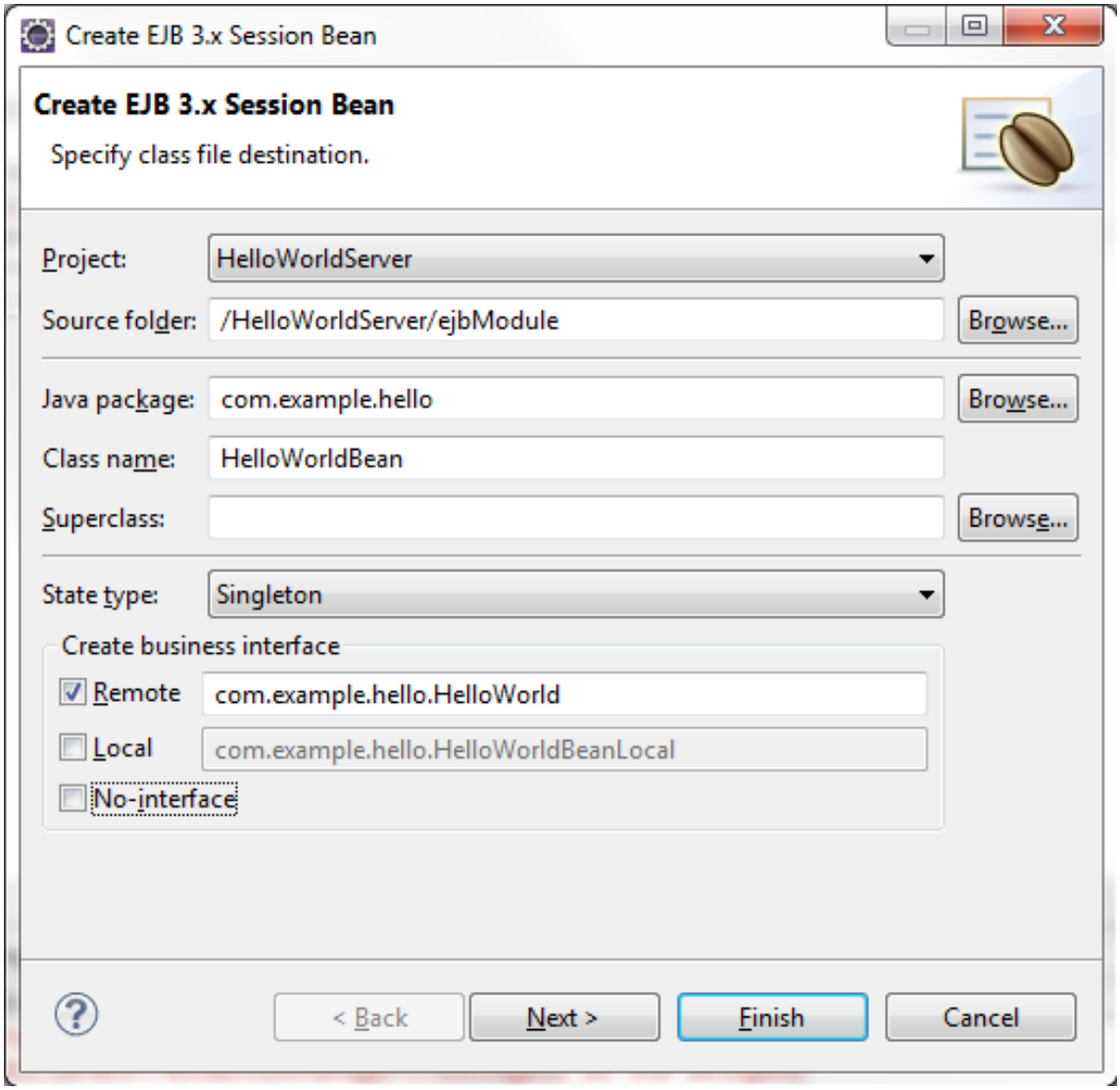


1.1.3. Activity: Creating a Hello World Bean with Eclipse

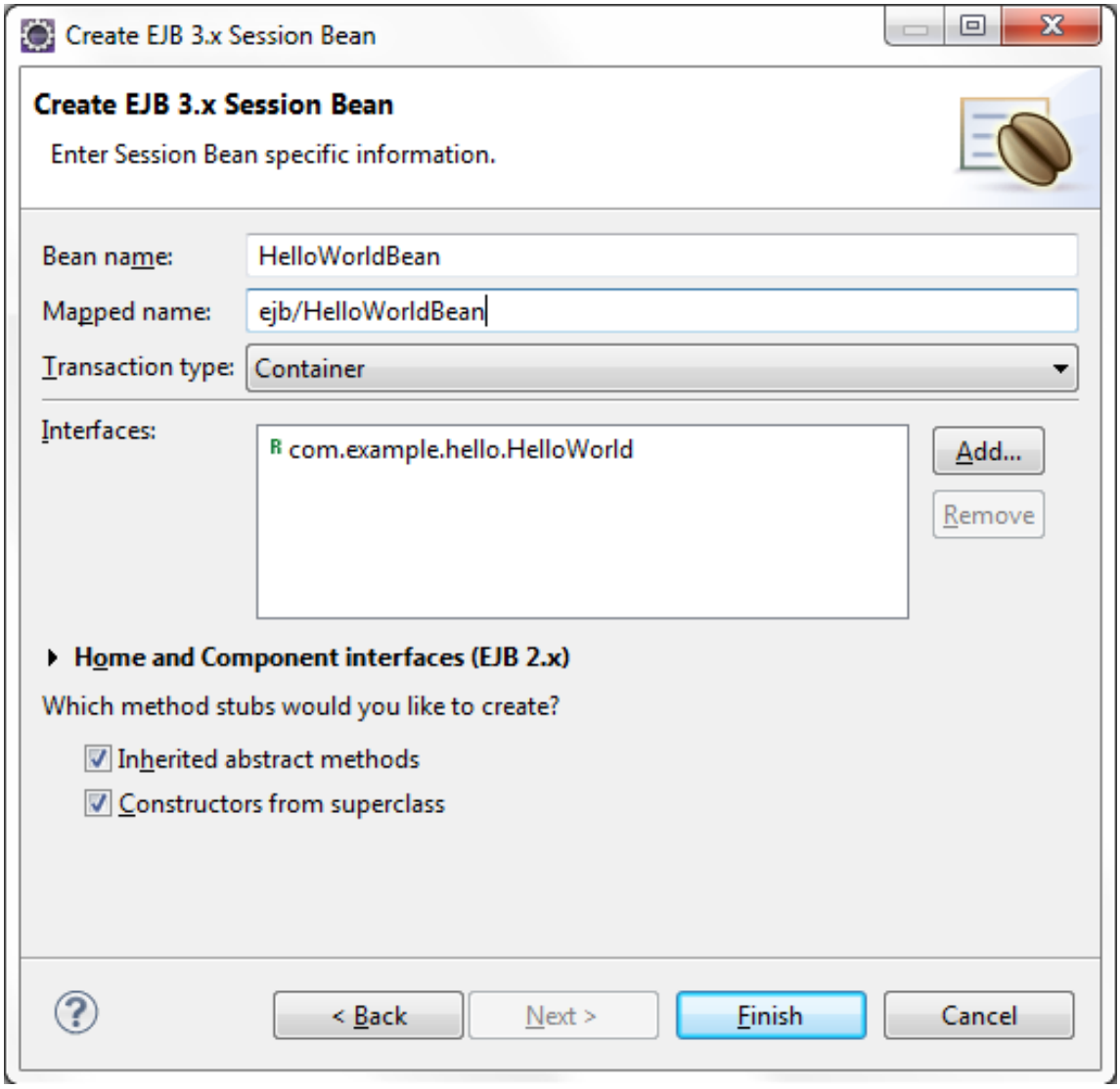
In the next lesson, you will be developing a simple banking application. However, before you proceed to such complex example, let's start with the simple *Hello World* EJB application. The EJB technology and architecture are not explained in the steps below. The purpose is to make you experience the creation of a first EJB to get a concrete idea of what an EJB is, before you go through the theoretical architecture and definition topics.

In the next lesson, you will redo a similar example step-by-step with all the necessary detailed explanation.

1. At the beginning, create an EJB project in the Eclipse. The latter project will hold the very first EJB. To create the project, select File > New > EJB > EJB Project. Name project *HelloWorldServer*.
2. Start the wizard for creating an EJB class of type *Session bean*. With the project selected, select the menu File > New > EJB > Session Bean.
3. In the package field, type *com.example.hello*, and the class name is *HelloWorldBean*. Stay on the wizard page.
4. On the same wizard page, select the state type *Singleton* from the drop-down box. This option will be explained in the next lesson. Stay on the wizard page.
5. Check the *Remote* option for creating the business interface, and name it *com.example.hello.HelloWorld* (the default name "HelloWorldBeanRemote" doesn't suit well).
6. Uncheck the *No-interface* check box.



7. Press the *Next* button.
8. In the *Mapped name* field, enter "ejb/HelloWorldBean".



9. Press the *Finish* button.
10. Find your new HelloWorldBean class in your project and check that it is the same as below:

Source

```
package com.example.hello;

import javax.ejb.Singleton;

/**
* Session Bean implementation class HelloWorldBean
*/
@Singleton(mappedName = "ejb/HelloWorldBean")
public class HelloWorldBean implements HelloWorld {

    /**
     * Default constructor.
     */
    public HelloWorldBean() {
        // TODO Auto-generated constructor stub
    }

}
```

11. In this class HelloWorldBean, add a public sayHello() method that returns any String:

Source

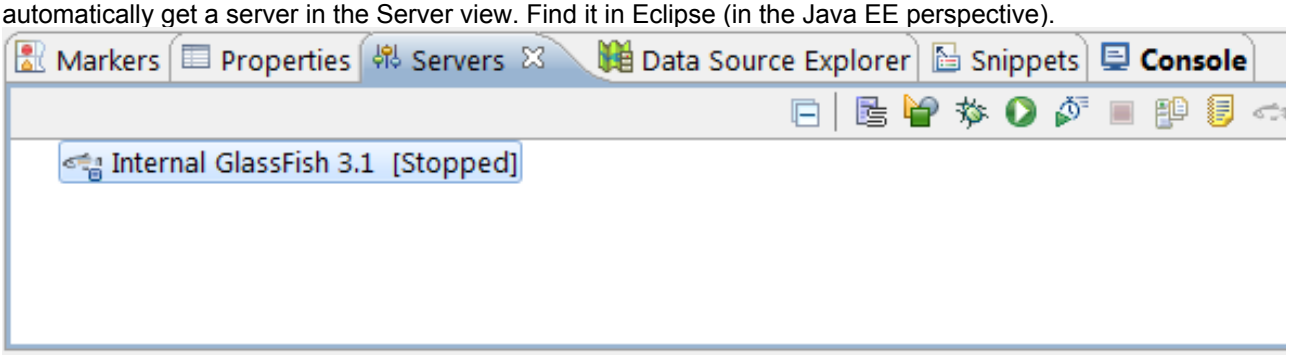
```
    public String sayHello() {
        return "Hello World!";
    }
```

12. Add the method signature to the HelloWorld interface that has been created by the EJB wizard in the same package.

Source

```
@Remote
public interface HelloWorld {
   public String sayHello();
}
```

13. Your code is ready, it's now time to deploy it. Because you did install the GlassFish Eclipse plugin, you automatically get a server in the Server view. Find it in Eclipse (in the Java EE perspective).



14. On the *Internal GlassFish* server in that view, right click and select *Add and Remove...* in the pop-up menu.
15. Select your server project in the left column and put it in the right column with the *Add* button, and click *Finish*
16. Start your server with the green *Start the Server* button within the server view. It is supposed to work. But, if on the contrary, you see error messages, don't worry. Having your EJB work is not that important for this lesson. The intention of this exercise is to have an idea of what a concrete EJB is. In the next lesson, you will get to explore more details and you will have the necessary knowledge to debug your project and make it work.

Congratulations, your first EJB is now deployed and running on the Glassfish application server. Don't try to fully understand the above code at this moment. This is only a "Hello World" example - we will discuss the EJB annotation details later on. In the next lesson, we will redo a similar example and explain every detail at each step. The important thing for now is that you created your first remote EJB component with simple business logic and deployed it to the application server.

1.1.4. Activity: Creating Hello World client

We started our EJB server with our HelloWorldBean EJB inside, but we have not tested it yet. In this activity, we will create another project with a Java SE application that will perform a remote call to the HelloWorldBean.sayHello() method for running the EJB server.

1. In Eclipse, start the wizard for creating a new Java project. Select the menu File > New > Project... > Java Project > Next.
2. Fill *HelloWorldClient* as the project name, and click *Next*.
3. In the second wizard step, you can configure projects settings. You need your client code to have access to the HelloWorld interface from the HelloWorldServer project. You could make a JAR file containing the interface and put it in the client project, but it would force us to do that every time we change the interface on the HelloWorldServer project. An alternatively easier way is to make the HelloWorldClient project depend on the HelloWorldServer project in Eclipse. In the *Projects* tab of the wizard, click the Add... button and select *HelloWorldServer* from the list (check the checkbox) and press *OK*.
4. Click *Finish* to close the wizard and make Eclipse create your project. Eclipse may redirect you to go to the Java perspective, but you should probably reply no as you prefer to stay in the *Java EE* perspective.
5. Your client application will need to connect to the GlassFish server. The inner communication details will be executed by the ready-to-use client classes from GlassFish. These client classes are in the appserv-rt.jar file. You will need to locate that file and put it in your client project.
6. The appserv-rt.jar file is in that sub-folder (or similar, maybe you have a newer version of GlassFish):

eclipse\plugins\oracle.eclipse.runtime.glassfish_3.1.0.0\glassfish3\glassfish\lib

7. Add this JAR file to your project's build path: Right-click on the HelloWorldClient project > Build Path > Configure Build Path...
8. In the open dialog box, click the *Add External JARs...* button. In the tree, select the plugins\oracle.eclipse.runtime.glassfish_3.1.0.0\glassfish3\glassfish\lib\appserv-rt.jar file, then click *OK*.
9. Your HelloWorld project is now ready to get your Java code. Create a new (normal Java SE) class named HelloMain, with a main method, in the package com.example.hello.client:

Source

```
package com.example.hello.client;

public class HelloMain {

public static void main(String[] args) {

}

}
```

10. In the main method, add the following code (that will not be compiled):

Source

```
        InitialContext ctx = new InitialContext();
        HelloWorld hw = (HelloWorld) ctx.lookup("ejb/HelloWorldBean");
        String s = hw.sayHello();
        System.out.println(s);
```

As we will explain in the next lesson, the first two lines get a client object that implements your EJB business method *sayHello()*. The third line performs a remote call and activate the business logic on your EJB to get the result (the string "Hello World!").

11. To remove compilation problems, import the necessary packages (Ctrl + Shift + o) and add a throw clause to your main method.

Source

```
package com.example.hello.client;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import com.example.hello.HelloWorld;

public class HelloMain {

public static void main(String[] args) throws NamingException {
                InitialContext ctx = new InitialContext();
                HelloWorld hw = (HelloWorld) ctx.lookup("ejb/HelloWorldBean");
                String s = hw.sayHello();
                System.out.println(s);
}

}
```

12. Verify in the *Server* view that your GlassFish server is still running (you started in in the previous activity), and execute your client: on the HelloMain class in the Project Explorer, right click > Run As > Java Application.
13. It is supposed to work and you should see "Hello World!" at the client console. Else, don't worry. Having your EJB work is not that important for this lesson. The important learning step for you now is to have an idea of what a concrete EJB is. In the next lesson, you will get every detail explained and you will have the necessary knowledge to debug your project and make it work.

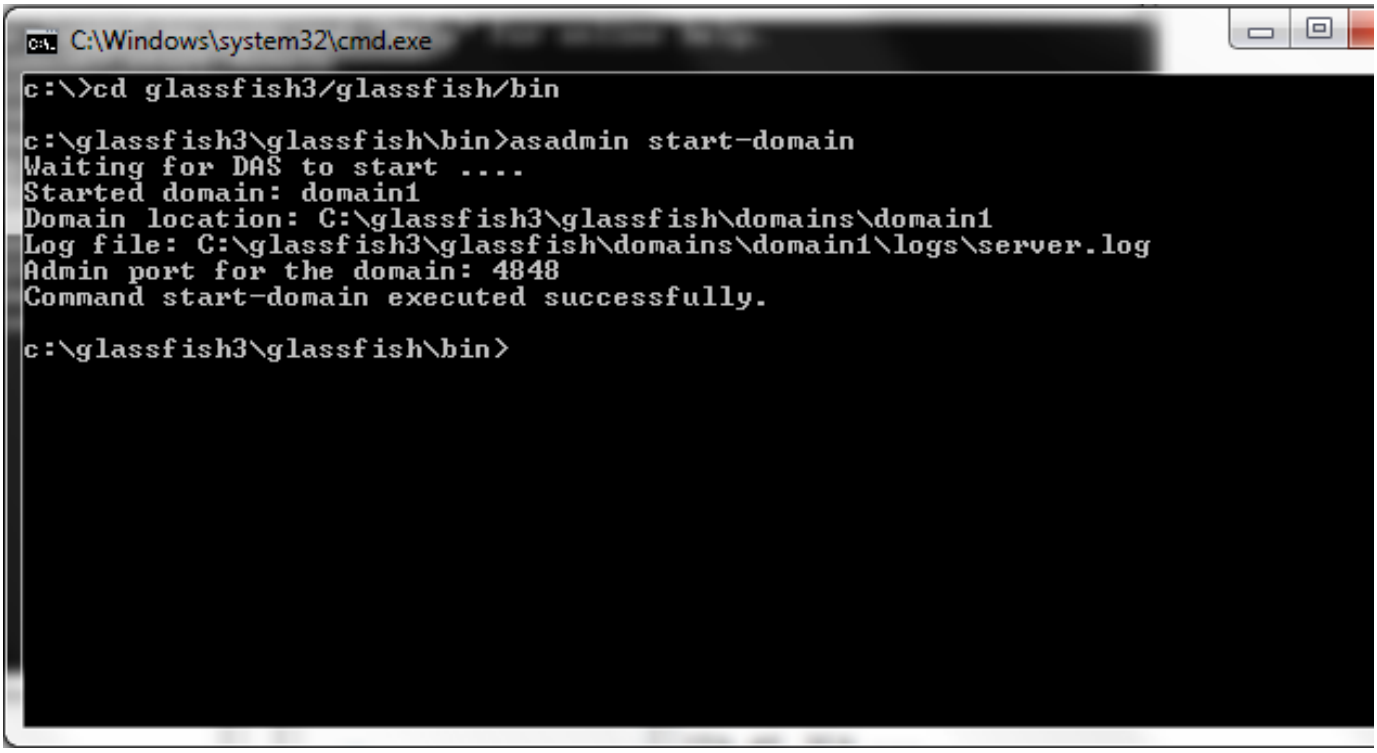
1.1.5. Potential Error

1.1.5.1. Cannot start Glassfish from Eclipse

In some environment, we may have issues to run the Glassfish server directly from Eclipse.

The Workaround to bypass this problem is to start the server with the command line.

Execute the command in the prompt: asadmin start-domain

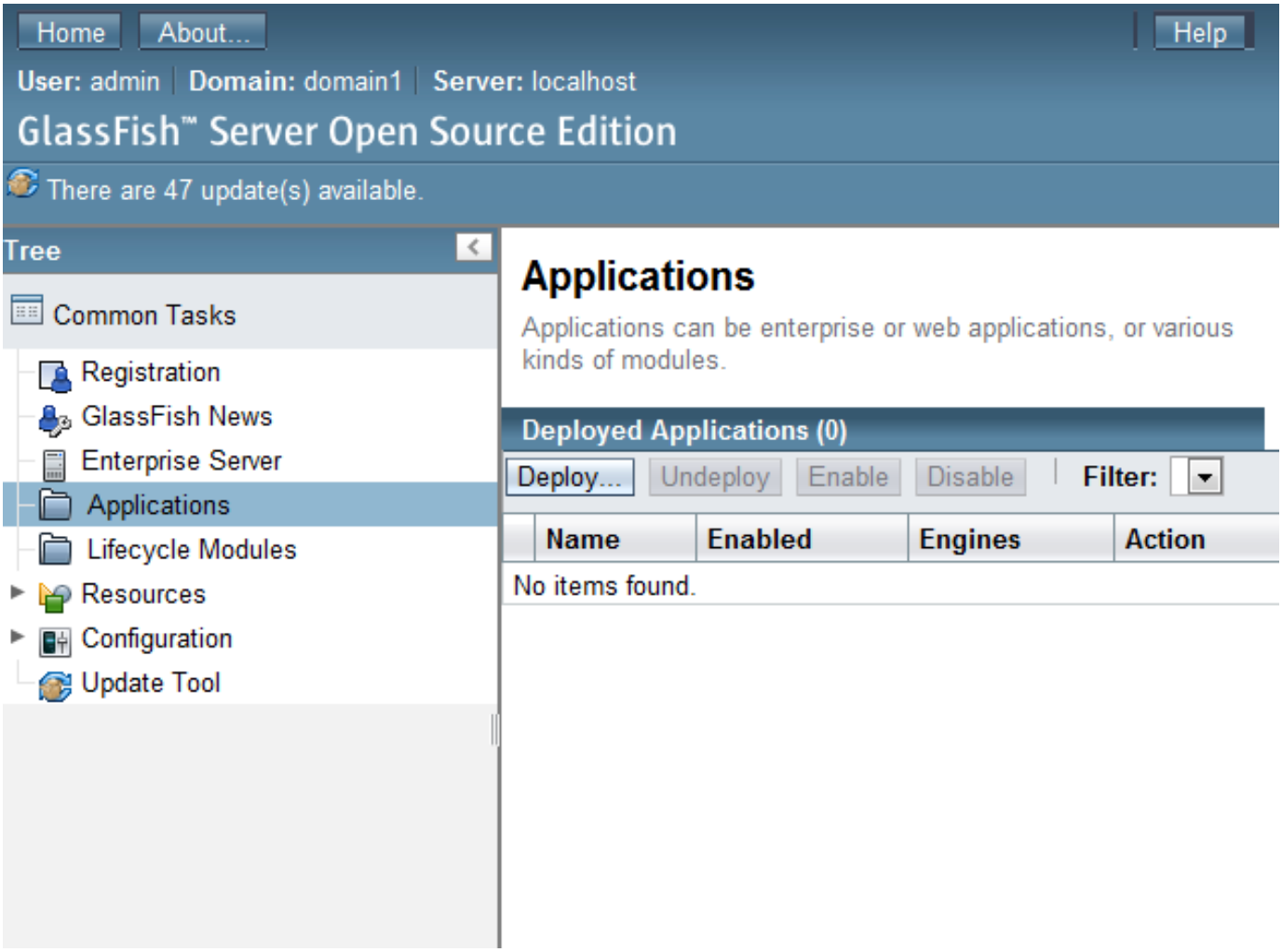


The domain is started, we need to deploy the project on it.

So first, create the EJB Jar file of your HelloWorldServer project, just right-click on the project > Export > EJB JAR file. Select a Destination and click Finish.

Now, we need to access to the admin page of Glassfish, open a browser and type "http://localhost:4848" in the address bar (4848 is admin interface port by default).

To deploy your application, click on "Applications", you will normally get this page.



Click on Deploy button > Select the path of your EJB JAR file and select the correct type : "EJB Jar". Then click ok.

Your server is running and the HelloWorldServer project is deployed on it.

Try to run your HelloWorldClient.

1.2. Examine Distributed Architectures

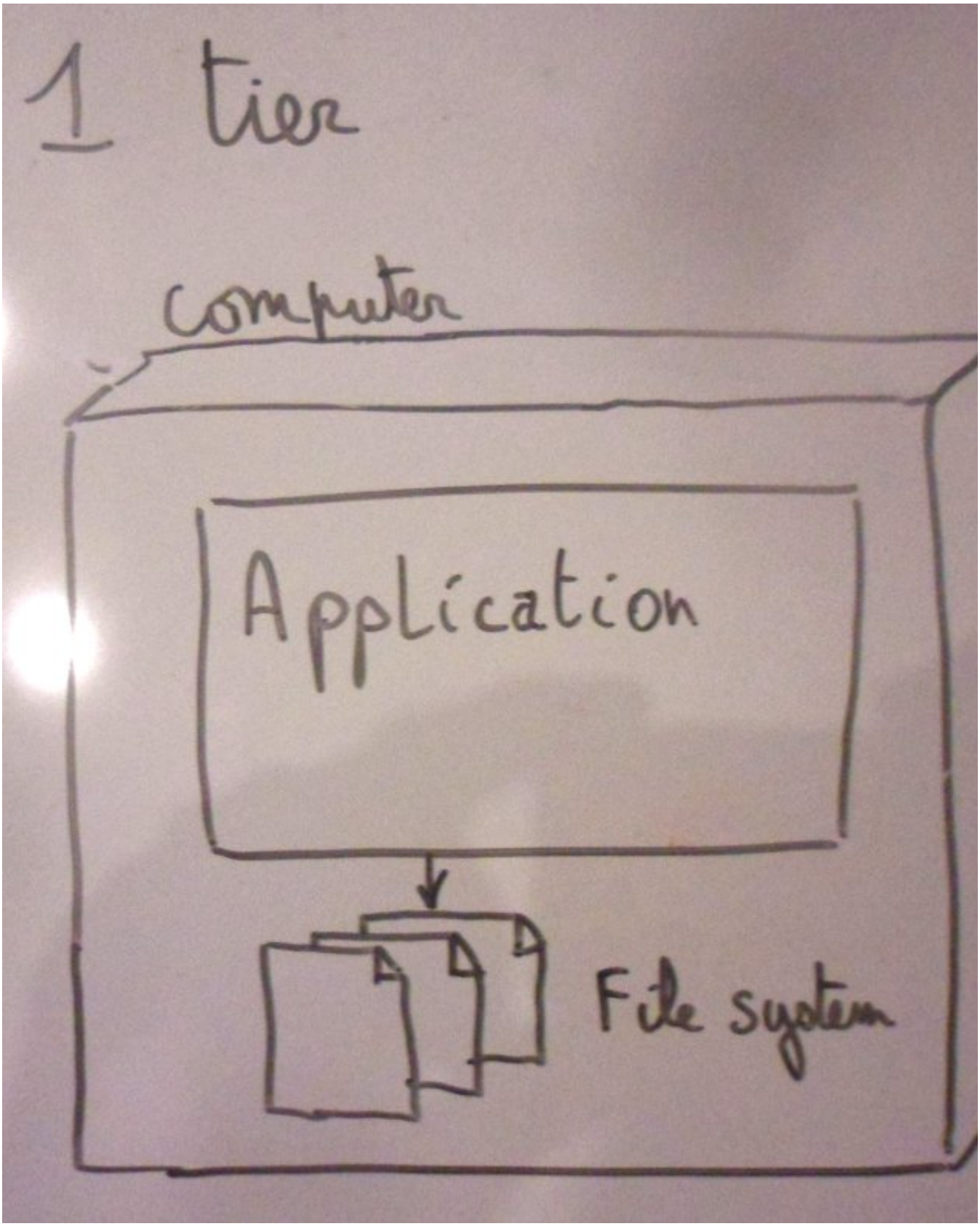
After having installed your EJB container and run your first bean, you get a concrete idea of what an EJB is.

EJBs can be used in many architectures. For example, you can centralize a banking application business logic on an EJB server, and have two kinds of clients making calls to that server. The first kind of client could be a Java web server. The second kind of client could be a fat GUI written in the Visual Basic language.

On a high level, where you decide on which computer will reside the software you build, you identify tiers. A tier is a vertical layer with a specific responsibility. In this topic, you learn multitier architectures with, at the end, a concrete example that is common in large companies.

1.2.1. 1 Tier

Stand-alone applications needing no connectivity with any server are 1 tier applications. They manage the persistence of their data, for example with an embedded relational database.



The 3D box represents a physical computer in UML.

Glossary: UML: Unified Modeling Language is a graphical convention to describe IT systems and architectures.

Typical 1 tier applications are non-connected games, word processors, and image design softwares.

Mainframes with terminals can also be seen as a 1 tier system, if we consider the terminal as not being a computer on which an application runs, but only a direct peripheral of the mainframe.

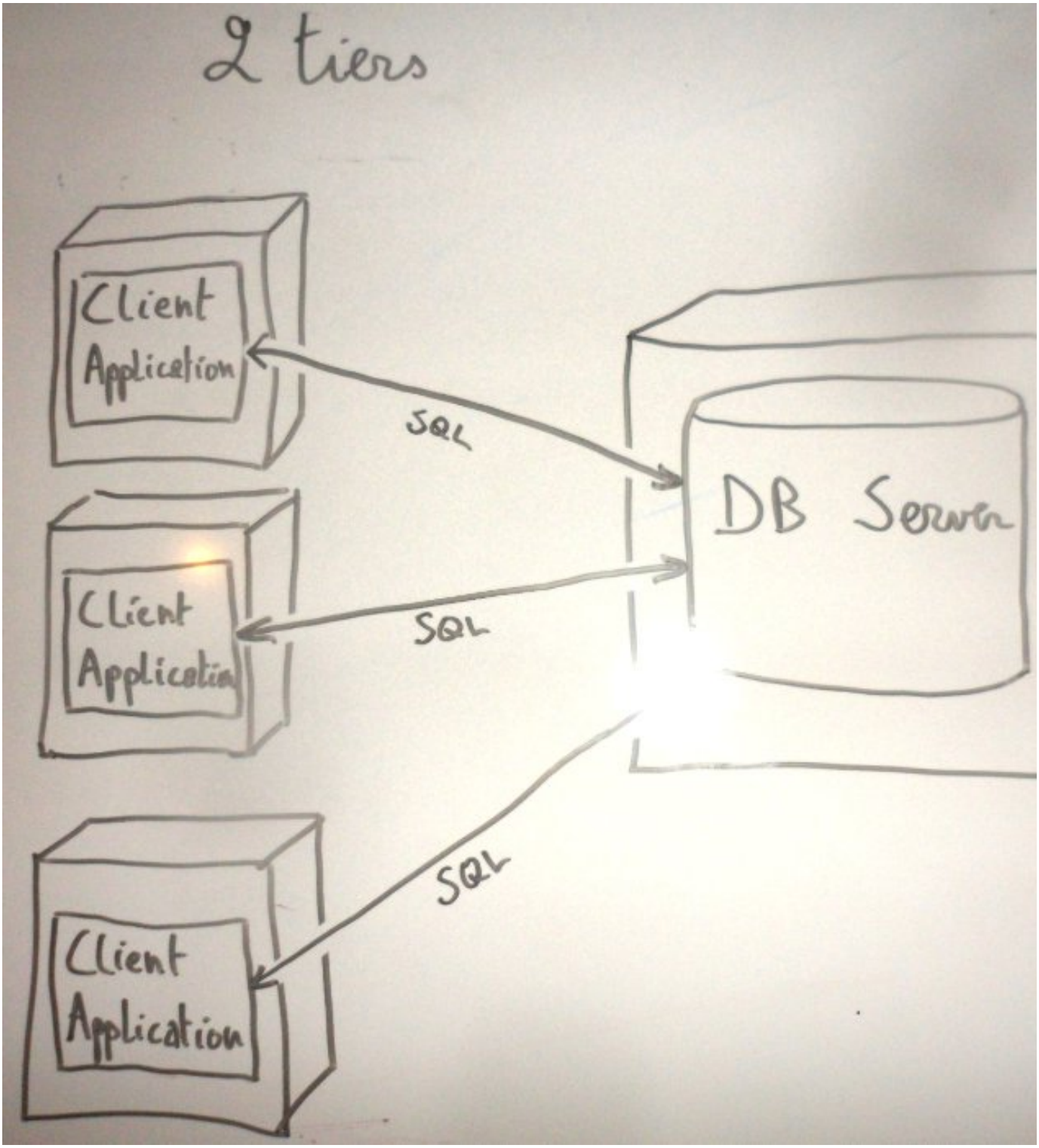
The problem of 1 tier stand-alone PC applications is the lack of scalability for supporting multiple concurrent users. The problem of 1 tier multi-user mainframe applications is the lack of interoperability with other applications while accessing the same data.

1.2.2. 2 Tiers

When multiple users need to work on the same data, each from their PC, we need to store that data on a separated DB server.

Fat UI clients are directly connected to that DB and send SQL statements to read/write relational data. Fat UI clients are executable programs compiled to run under Windows and uses all the widgets provided by the Windows operating system.

The model was particularly popular in the nineties, with the rise of networks to interconnect PCs. Languages such as Powerbuilder, Delphi, and Visual Basic were very successful to implement the fat UI client connecting an Oracle or MS SQL server through ODBC.

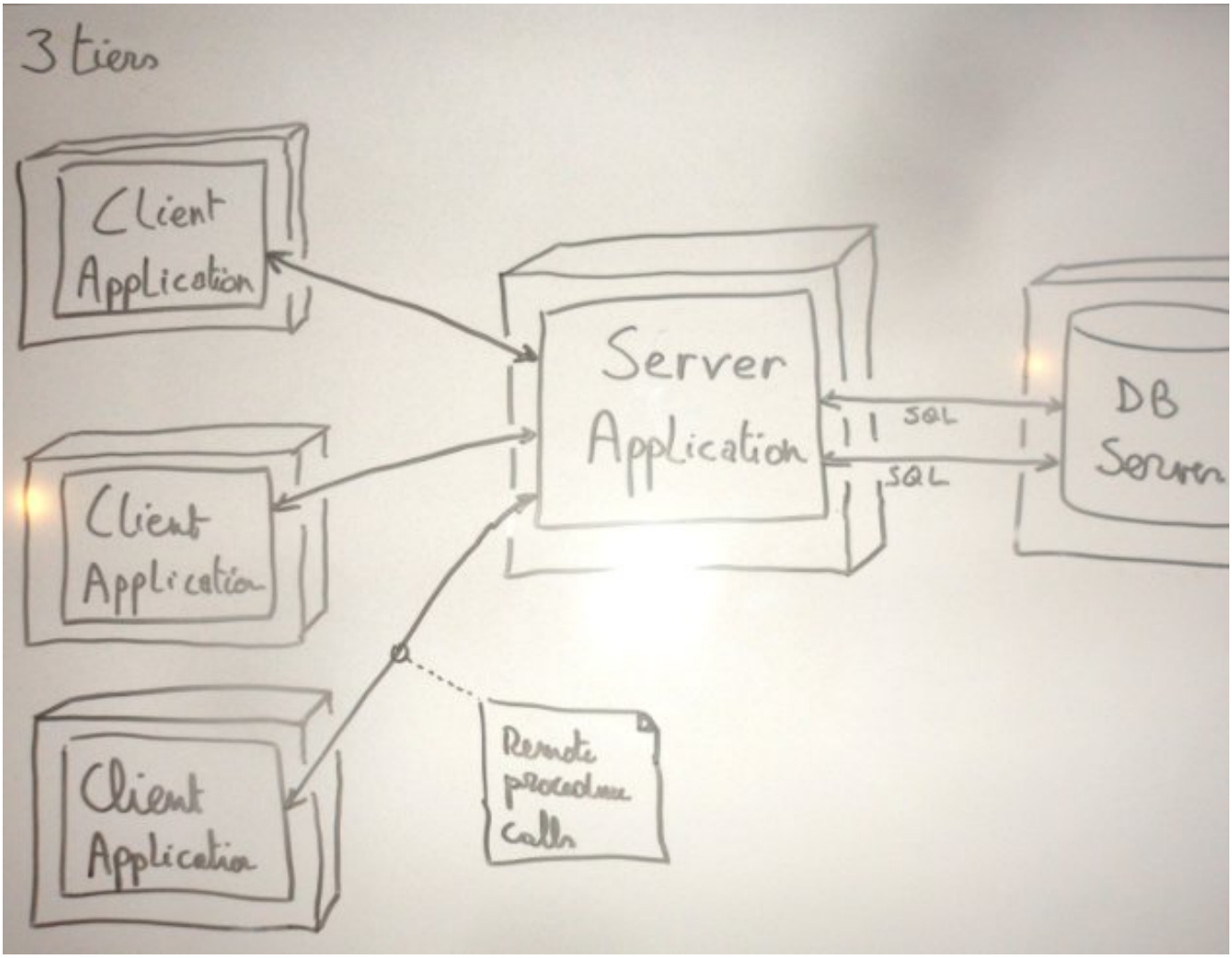


The model loses in popularity because:

- direct access to the DB with no server-side business logic:
  - may cause performance problems (of moving too much data over the network)
  - may cause security/integrity problems
- deploying the client application (for each new version) on many computers is an additional technical hassle.
- web interfaces are richer now (AJAX / JavaScript) and compete with fat UIs on the usability point of view.

1.2.3. 3 Tiers

A 3 tier architecture puts the business logic on a centralized server between the clients and the DB. The application server can protect the integrity of the data of the DB. It is also very close from the DB in terms of network and can get more data for some computations with better performance than a far away client.



A primitive form of application server is the mechanism of PL/SQL stored procedures on an Oracle DB server. Some 2 tier applications started to shield the DB with PL/SQL procedures to avoid exposing the DB tables directly to the client, and to perform server-side computations.

Glossary: PL/SQL: programming language (with procedures, variables, loops, conditions) that is supported by some DB servers.

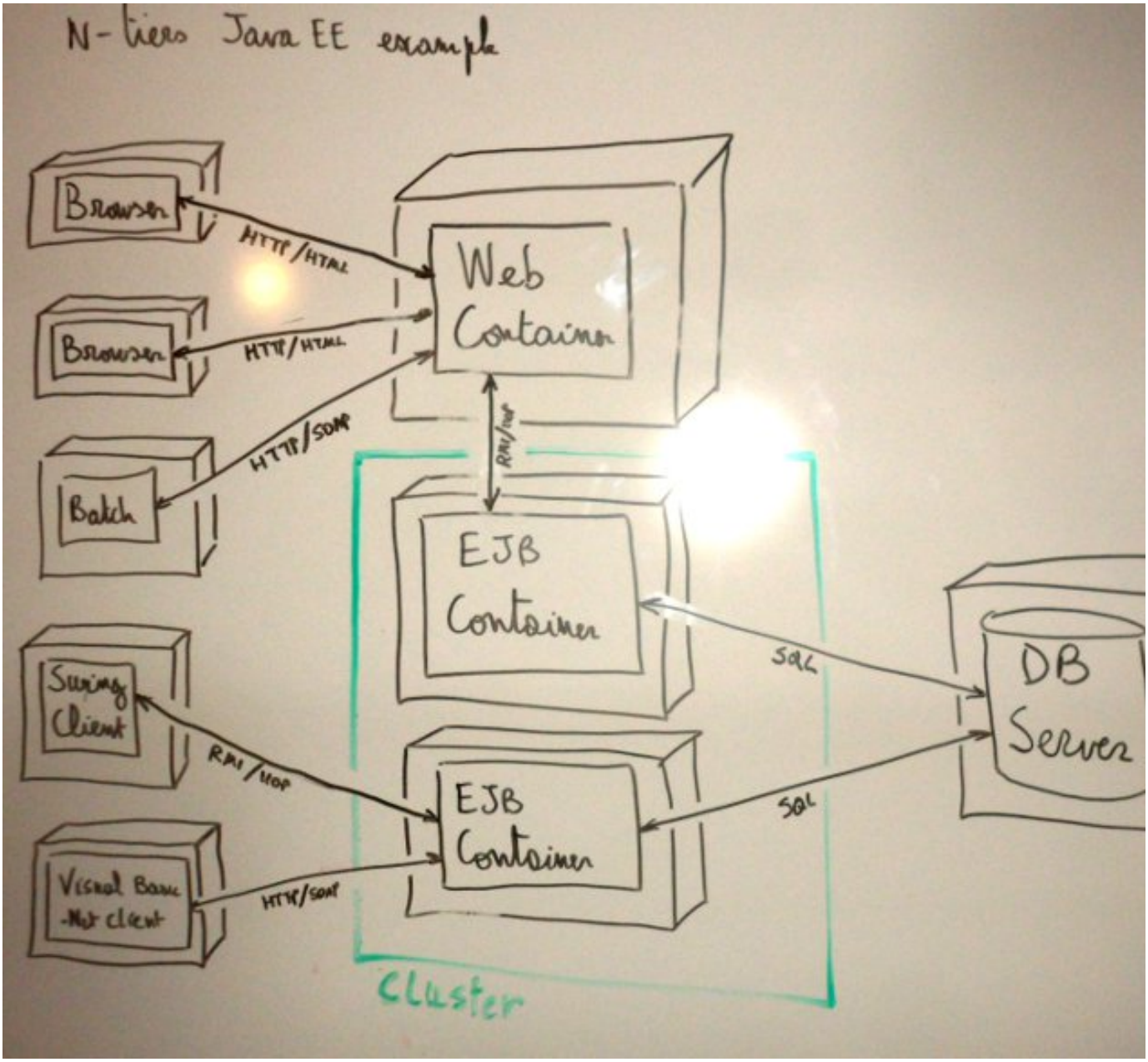
EJB Servers have been created to fulfill that role: host the centralized business logic between the client and the DB server. A disadvantage of the 3 tier model is the higher complexity.

1.2.4. N Tiers Example

The diagram below shows an application that spans over 4 tiers:

1. browser / batch / Swing / .Net
2. Java EE Web Container

3. Java EE EJB Containers
4. DB Server



The two top clients are web browsers as Chrome, Firefox, or Internet Explorer. They communicate through HTTP to the web container, which sends back the HTML pages. If it is an AJAX application, JSON requests/responses are sent over HTTP.

Glossary: AJAX: Asynchronous JavaScript and XML is a technique to update a small part of a web page in the browser, from information obtained from the server. As the web page is not completely refreshed in the browser, the user experience is better than with traditional web applications.

Glossary: JSON: JavaScript Oriented Notation is a lightweight text representation of structured data. It is typically used in AJAX communications.

Glossary: HTTP: Hypertext Transfer Protocol specifies the information headers added to the data exchanged by a web server and a web browser. A typical header of an HTTP response is the type of data sent from the server, such as HTML text, a JPEG image, or a JSON text.

The third client is a batch process. It is, for example, an application from another company that needs data from our web service every hour.

The Web Container is a Java EE server taking care of the UI logic. It generates HTML pages (or fragments) for the browser clients. It hosts technologies such as Servlets, JSPs, Struts, Spring MVC, JSF, and Vaadin.
It connects an EJB container to invoke methods remotely and get the results back.

The EJB Container is another Java EE Server taking care of the business logic such as computing complex interest rates, or mining data from the DB, or applying complex rules for tax immunity.
It is accessed directly by the two bottom clients, which call methods and expect results:

- The Java Swing fat UI uses the RMI/IIOP protocol which works very well between two distant Java programs.
- The .Net client needs a more interoperable protocol. The EJB Container exposes some remote method to be called as a web service. The SOAP protocol transports the web service requests and responses.

1.2.5. Activity: Quiz

A program runs on a PC and connects a DB server. Multiple PCs running this program access the same DB. How many tiers has this architecture:

( ) 1 tier

( ) 2 tiers

( ) 3 tiers

Explanation: Some could argue that it is one tier if the DB is on the same machine as the PC program. But programs from multiple machines access the same DB => these are on different machines and are definitely separate tiers.

Solutiuon: 2 tiers.

1.3. Foundations of EJB

You need to setup a server, create an EJB, and then you will understand where your EJB server and client stand in a complex architecture.

So far we have not really explained the services provided by the EJB container and which you can take advantage of. For example, if your application gets a lot of workload, one physical server may not be enough. Your EJB container will help you to share the load on multiple machines. This is one of the services that you can expect.

In this lesson, we cover the theoretical background of the EJB technology. We discuss object distribution problems, transaction monitors, declarative security, and memory management. This should give you a much better view on whether or not you would like to use an EJB server.

1.3.1. Problems

The EJB technology has been created to manage and solve the following problems in a standard way:

1. Distribution
2. Persistence
3. Transaction
4. Security
5. Memory Management
6. Scalability

7. Dependency Injection
8. Batch Job Trigger

1.3.2. Object Distribution

Objects should be able to be deployed to many machines and operating systems with performance, scalability, and reliability.
Two technologies were initially available in the Java world: CORBA (Common Object Request Broker Architecture) & RMI (Remote Method Invocation).

CORBA is a cross language, cross platform object distribution system. For example, it enables a C++ program on a Unix machine to call a Java method from another program in a Windows machine, and pass its objects as parameters. CORBA requires the programmer to use an additional language, IDL (Interface Definition Language), to describe the objects exposed on the network. CORBA is some kind of ancestor for EJBs as it also includes transaction services. CORBA uses IIOP (Internet Inter-ORB Protocol) as the communication protocol.

RMI is a Java only solution, for a Java program to call another Java program probably on another machine. It has the advantage of being simpler than CORBA. It uses JRMP as the communication protocol.

EJBs enable distributed architectures where clients request remote code to be executed on an EJB server. Internally, it uses a mix of CORBA and RMI named RMI over IIOP to implement that distribution, but it is completely hidden for the programmer. Most EJB servers can also directly use JRMP (RMI native protocol), which is lighter than RMI over IIOP when no CORBA interoperability is needed.

Later, distribution mechanisms include web services, with less focus on performance and more focus on interoperability (anything communicates with anything). Web services communicate with the SOAP protocol that uses XML to structure the request and the response data.

Usually, when a Java to Java communication is needed, RMI over JRMP is used. When the client is a non-Java program, then web services over SOAP is used.

1.3.3. Object Persistence

Since it's first version, the EJB specification has tried to include a solution O/R Mapping.

Object to Relational Mapping addresses the impedance mismatch (i.e. the structural difference) between objects (from object oriented programming) and relational databases (accessible through SQL). It provides a mapping between the two different worlds (OO & Relational) which resembles the OO programming.

Up to version 2 of the EJB specification, its O/R Mapping technology was not used in nontrivial applications. As a reaction, the community of Java programmers has created many O/R Mapping open source technologies, and the most famous is *Hibernate*.

As a reaction, Sun Microsystems, the company behind Java at that time, has standardized these open source initiatives with the *Java Persistency API* (JPA) specification.

JPA was included in EJB v3.0. It has been removed from EJB v3.1 because JPA does not require an EJB container, and many applications benefit JPA without EJB. JPA is not part of the *Oracle Enterprise JavaBean Developer 6* certification. JPA is not part of this course, but it is the subject of another set of courses.

1.3.4. Objects and Transactions

RDBs (Relational Databases) support transactions. The JDBC API, enables to access RDBs with SQL and includes support for communication transaction instructions to the RDB.

But the built-in RDB transaction support is not enough when dealing with multiple RDBs in one transaction. In JDBC, the transaction methods are bound to a connection (.setAutoCommit(boolean), .commit(), .rollback())

The EJB technology includes a two phase commit mechanism to enable one EJB transaction to involve multiple RDBs, even if they are from multiple vendors.

CICS, Encina, and Tuxedo products may be considered as the ancestor of that technology. They are known as *Transaction Processing Monitors* because they provide an execution environment that ensures the integrity, availability, and security of data; fast response time; and high transaction throughput.

Component Transaction Monitor is an OO flavor of transaction monitor. It includes *Microsoft Transaction Server* and EJB.

1.3.5. Security in Enterprise Architectures

EJB containers include the notion of job roles (as User, Manager, Administrator, Accountant,...). You can easily condition the execution of some code to the verification that the concerned user has the rights to do it (has the appropriate job role).

Such business logic security enforcement is much stronger than only placing security conditions in the UI layer such as deciding whether or not to show a button, according to the role of the user.

1.3.6. Memory Management

Creating an object instance and not using it for a long time occupies unnecessary RAM and the time the object instance is not used. Operating systems implements virtual memory with memory swapping.

It is not always enough. You may want to retrieve objects even after an OS crash. Or you may want to share objects representing the same data or behavior among multiple threads not knowing each other. You need a central mechanism to identify objects that may be shared and maybe pool them for reuse.

The EJB specification provides memory management with the beans pooling, activation/passivation, and load/store mechanisms which are covered further.

1.3.7. Scalability

EJB architecture provides an easy way of managing horizontal scalability of the business logic of the application. EJB containers are designed to be easily clustered and replicated. The main idea behind the EJB scalability is to provide bean component with business logic that will be clustered and load-balanced.

This approach allows application architect to scale the business tier of an application regardless of the web tier. This is an important feature in case of applications with very complex business computations and relatively simple web

layer. This also means that deployment of business components can be performed regardless of the deployment of the web components.

For example, complex financial application with heavy back-end computations and thin web layer can take advantage of this kind of scalability.

1.3.8. Dependency Injection

In large applications, it may be useful to prevent classes from finding instances of other classes. Consider these two classes:

- BankService: contains business logic to see if a customer could get a loan.
- CustomerRepository: contains DB queries to retrieve customers from the DB.

BankService needs to call methods on other classes, for example on CustomerRepository. For doing so, the BankService instance needs a reference to an instance of CustomerRepository. How will BankService obtain that reference? In a conventional basic scheme, BankService would use the *new* keyword to create an instance of CustomerRepository.

On the contrary, with dependency injection, BankService does not take care of obtaining a reference to CustomerRepository. The EJB container will give (inject) that reference in BankService. This method has a few subtle benefits that will be explained in the dependency injection lesson.

1.3.9. Batch Job Trigger

Many applications contain code that must be performed on a regular basis. For example, you may need that some DB cleaning tasks start every day at midnight, and you may need that some employee pay processing starts every month, probably the last day of the month.

These tasks are typically named batch jobs. EJB containers provide a very easy way to mark a method as "to be executed every ...", and the EJB container will take care of starting it at the right time. In the EJB world, this mechanism is named *Timer*.

1.3.10. Activity: Quiz

From the services below, what are the services provided by an EJB server to your application?

[ ] Make your code remotely accessible through web services.

[ ] Make your code remotely accessible through RMI.

[ ] Make a single transaction span multiple DBs.

[ ] Secure your application by asking the user a user and password.

[ ] Make coffee

Explanations:

1 & 2. EJB enables object distribution through RMI or SOAP (web services). With the @WebService annotation you

can easily make a class remotely accessible.

3. The EJB container is a transaction monitor to synchronize multiple DBs on the same transaction, such as updating a record on DB 1 and inserting a record on DB 2, in the same transaction.

4. The EJB container provides no UI to ask its credentials to a user (but web containers do). Its security features mainly cover authorization to identify what part of the code can be executed by which users.

5. At the current level of the technology, EJB containers don't make coffee yet.

Solution: the 3 first choices are correct.

1.4. Fundamentals of EJB

You have everything in your hands to get a more formal definition of EJB. You did setup a server, created, and tested Java code and had background information about the services that you benefit within an EJB container.

In this lesson, you will define EJBs, what is new in the version 3.1 of the specification, and list products implementing that specification.

1.4.1. Formal Definition

Formally,

> *EJB is a standard server-side model providing an abstraction for component transaction monitors*

So, EJB is a model. It is not a product. It is a contract that products must implement to claim that they are EJB containers. For example, EJB defines the API that your code can use to access the container. The EJB specification shields you from the internal details and variations of each product. That way, it is an abstraction. The EJB container provides a major service to your application. It is to be a component transaction monitor, or in other terms, an object oriented transaction monitor to organize transactions that span multiple DBs.

1.4.2. Characteristics of an EJB container

EJB containers have the following characteristics:

- Where your EJBs reside.
- Responsible of making the EJB classes available to the client.
- Manages the EJBs providing:
  - Security
  - Concurrency
  - Transaction support
  - Memory management (swapping unused EJBs to secondary storage).
  - Load Balancing
  - Workload Management (tuning the amount of active beans/threads)
- EJBs are deployed into EJB containers and run on EJB servers.

1.4.3. Need for EJB

The EJB technology has been used in many projects where it should not have been used, especially in the first years of the century.

Some projects really benefit the features of the technology. These questions help to position the need for your project:

- Is there an architectural push toward a standard, portable, component-based architecture?
- Distribution: Is there a need for access to enterprise data and shared business logic from different client types ?
- Transaction: Is there a need for concurrent read and update to shared data with a declarative programming model?
- Transaction: Is there a need to access multiple disparate data sources with transactional capabilities?
- Security: Is there a need for method level object security seamlessly integrated with security for HTML documents, servlets, JSPs, and client logins?
- Clustering: Is there a need for multiple servers to handle the throughput or availability needs of the system?

1.4.4. What's new in EJB 3.1

The EJB 3.1 specification comes with several interesting improvements. If you are new to EJB technology and do not know EJB 3.0, feel free to skip the section.

The key differences between the EJB 3.1 and EJB 3.0 are:

- Local session beans can be accessed without the separated local business interface.
- EJB components can be packaged and deployed directly into the WAR file.
- New embedded container API is now available for making testing and local development easier.
- New Singleton stateless bean.
- Improved timer API.
- Possibility of making asynchronous calls to session beans.
- Portable JNDI names.
- EJB Lite profile making easier to deploy simple EJB applications.

1.4.5. EJB 3.1 Lite

EJB 3.1 comes with a Lite version - a subset of full EJB API suitable for applications that do **not** require the following features:

- remote session beans interfaces
- messaging with MDB (Message Driven Bean)
- exposing session beans with web services (JAX-WS, JAX-RS, JAX-RPC)
- timer services
- asynchronous session beans calls
- legacy EJB 1.x/2.x technology support
- legacy entity beans support (replaced with JPA)
- RMI/IIOP interoperability

An application that implements EJB Lite can be deployed and run on any Java EE application server that implements EJB 3.1.

1.4.6. Types of Application servers

There are excellent open source application servers in the Java ecosystem. The most popular ones are:

- GlassFish (promoted by Oracle)
- JBoss (promoted by RedHat)

There are also closed-source products having various reputations among developers. The most known are:

- Weblogic (from Oracle, via a company aquisition)
- WebSphere (from IBM)

These four application servers provide both:

- a Web Container (to run Servlets and JSPs)
- an EJB Container (to run EJBs)

Other popular open source application servers provide only a web container such as Tomcat, Jetty, and JRun.

1.4.7. Portability

If your application complies to the Java EE specification, they can run on any of these application servers. In theory, you are free to change very easily. In practice, large applications often use proprietary features of the application server, making significant changes in the company decision. This is due to the fact that the Java EE specification does not cover 100% of an application's need and leaves room for vendors to provide their proprietary way of solving the uncovered area.
All the examples and activities of this course remain inside the Java EE specification.

On the Operating System, however, Java fully keeps its WORA promise (Write Once Run Anywhere). It's usual that developers use an operating system (typically Windows or Mac OS) to run the application server, while the test and production servers use another (typically Linux/Unix).

1.4.8. Activity: Quiz

Which product(s) have/are an EJB Container?

[ ] Tomcat

[ ] WebSphere

[ ] GlassFish

[ ] JBoss

[ ] Atari

Explanation:

1. Tomcat is no EJB container but a web container only. It runs servlets, no EJB.

2. WebSphere contains both an EJB container and web container. It is a product developped by IBM.

3. GlassFish contains both an EJB container and web container. It is an open source product developped by Oracle, and is used as the reference implementation (to show an example of the new versions of the EJB specification).

4. JBoss contains both an EJB container and web container. It is an open source product developped by RedHat.

5. Atari is a popular game device of the eighties. It is not related to EJB.

Solution: WebSphere, GlassFish, JBoss.

1.5. Using Annotations

During this course, you will use many annotations in your Java code. This topic provides a light explanation about annotations that will help you, in case you are not used to this concept.

Annotations provide (meta)data about your code.

Source

```
class MyClass extends SuperClass {
   @Override
   public String toString() {
       return "Hello World";
   }
}
```

In this example, the @Override annotation indicates that MyClass.toString() method overrides the toString() method defined in an ancestor class. In this example, the annotation is an hint for the compiler about the intent of the programmer: I know there is a toString method in an ancestor and I want to override it. If the programmer is wrong (if there is no toString method in an ancestor to be overridden), then the compiler produces an error message.

Classes, methods, variables, parameters, and packages may be annotated.

Annotations have been introduced in Java 5, together with other important language improvements (as generics, for each loops, enums,...).

Annotations are used by many frameworks. For example, JPA (Java Persistence API) provides mapping annotations to instruct an attribute of a class of the corresponding DB column name.

The annotation mechanism itself is built-in the Java language. Java SE also provides some annotations such as @Override. Most annotations are defined by JARs added to your application's classpath. EJB defines many annotations that you will use in this course such as @Singleton, @Stateless, @Stateful, @MessageDriven, @EJB, @PostConstruct, etc. You can use them in your code once you have the EJB JAR file in your build/classpath. They will not impact the Java compiler. But the EJB container will detect their presence at run-time and act accordingly. For example, if the EJB container detects the @Singleton annotation on a class, it registers that class as a session EJB and knows that it should not create more than one instance of if. You will get an explanation for all these annotations during the corresponding lessons of this course.

Annotations may have attributes. This is not the case of @Override. @Singleton has optional attributes. You are not obliged to provide any value for them. For example, it has a *name* attribute of type String. In the code below, you annotate the class with @Singleton and provide the value "Bob" to the attribute *name*.

Source

```
@Singleton(name="Bob")
class MyClass {
    ...
}
```

Since that attribute is optional, the following code is also valid:

Source

```
@Singleton
class MyClass {
    ...
}
```

2. Working with Session Beans

In the previous lesson, you have seen what architecture needs trigger the usage of an EJB container. You have also setup the GlassFish EJB container and created a first session bean.

Session bean is the most used kind of EJB. It is a business components responsible for processing some business logic. Developer encapsulates the business logic in the bean class, deploys it and then application server is responsible for making the bean instance available for the client.

In this lesson we build a realistic banking scenario of EJB usage where we experiment all the kinds of session beans, and explain architectural details of how session beans are implemented.

2.1. Create a Banking Application

To make our EJB considerations more concrete, we will introduce an example of application that will be developed in this course. We decided to stick with the sample banking application example. Financial operations are an excellent field to demonstrate the advantages of the EJB programming model.

2.1.1. Defining Architecture Requirements

While we will make it simplistic in this course, you can assume that the business logic of our banking application is very complex.

The business logic needs to be used by multiple kinds of clients. Even if you will not write it in this course, you know that a .Net application will need to access that business logic. A Java client will need to access it as well.

The business logic needs to access of lot of data from the database, and you are concerned for the performance. That data does not need to travel to the end-user UI. Most of it is only used for the computing the information displayed to the user. The business logic needs to be physically close from the DB.

For all these reasons you decide that your banking application should be multi-tier and your business logic should be implemented as EJBs.

Additionally, banking systems require the following architectural features:

- distributed transactions
- back-end processing system accessible via clients written on different platforms
- asynchronous processing

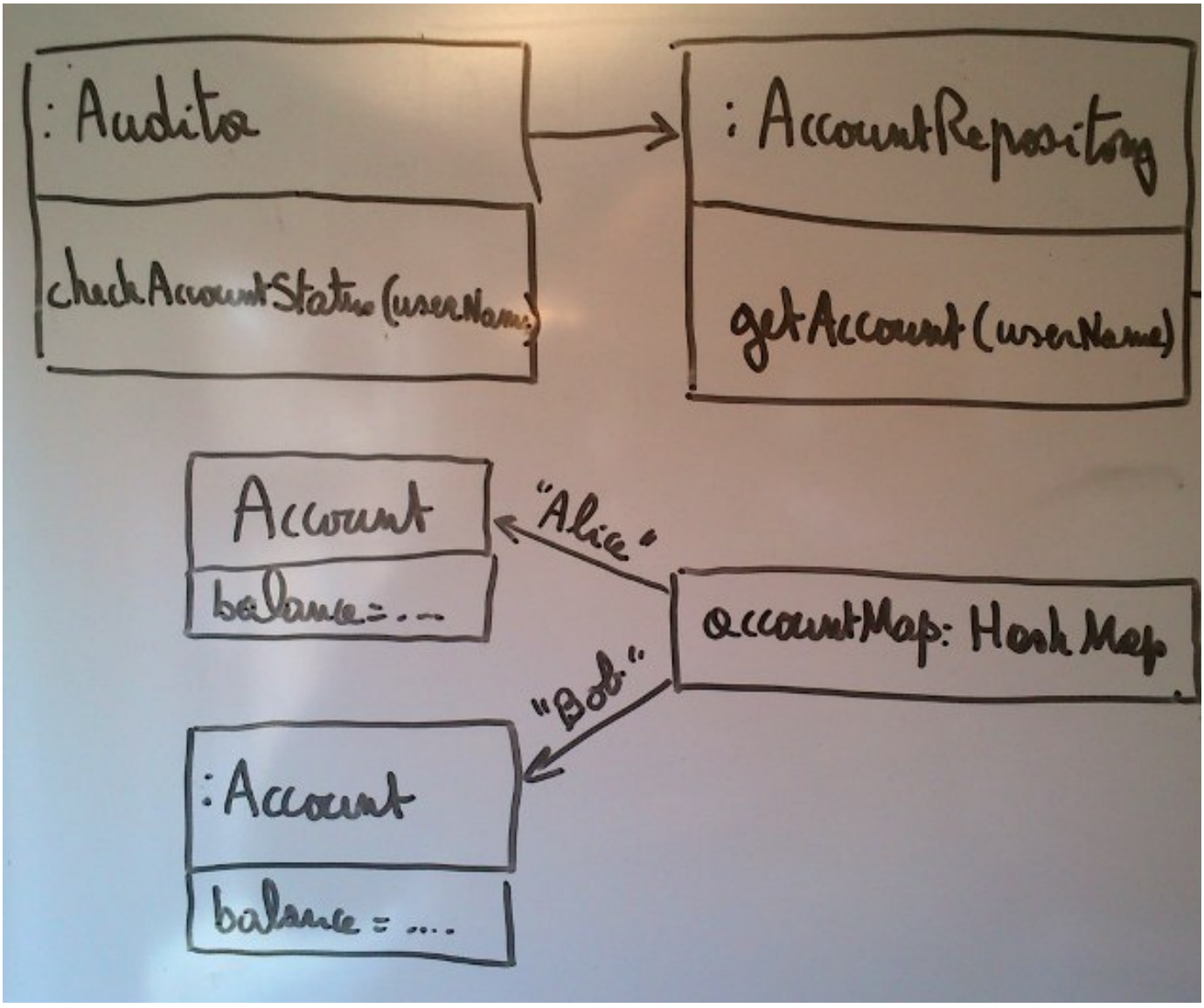
Since EJB model provides all the features mentioned above, the simple banking system will be an excellent example of EJB usage.

2.1.2. Overview

Our banking example needs 3 classes to be created:

- Account: to store the balance of a bank account.
- AccountRepository: to store accounts and retrieve them from the user name.
- Auditor: to perform some business logic on an account.

As shown on the object diagram below, an Auditor instance will have a reference to an AccountRepository instance. That AccountRepository will store two Accounts in its map, one for Alice, and one for Bob.



2.1.3. Domain Model

Account is the domain model class uses throughout the course.

Source

```
package com.example.bank.model;

public class Account {

   private long balance;  // in cents

   public long getBalance() {
       return balance;
   }

   public void setBalance(long newBalance) {
       this.balance = newBalance;
   }

}
```

The balance field represents how much money lays on the account.

2.1.4. Repository

The AccountRepository class will be used to store and retreive accounts. To keep this example simple, no DB is used. Instead, you keep accounts into a map, with the name of the owner as key.

Source

```
package com.example.bank.repository;

import java.util.HashMap;
import java.util.Map;

public class AccountRepository {

// Replaces DB
private Map<String,Account> accountMap = new HashMap<String,Accout>();

public AccountRepository() {
 // Create fake data
 Account account1 = new Account();
 account1.setBalance(100000); // $1000
 this.accountMap .put("Alice", account1);

 Account account2 = new Account();
 account2.setBalance(200000); // $2000
 this.accountMap .put("Bob", account2);
}

public Account getAccount(String username) {
 if (username == null || username.isEmpty()) {
  throw new IllegalArgumentException("Blank user name.");
 }
```

```
 return accountMap.get(username);
}

}
```

The constructor creates two Account instances, and puts them in the Map: one with the key "Alice" and the other with the key "Bob".

The getAccount() method retreives an Account instance from a user name. For example, `getAccount("Alice")` will return the account with the balance 100.

2.1.5. Business Logic

Our business logic verifies the status of the account of a given user, and returns that account.

Source

```
package com.example.bank.bean;

import com.example.bank.model.Account;
import com.example.bank.repository.AccountRepository;

public class Auditor {

   AccountRepository accountRepository = new AccountRepository();

   public Account checkAccountStatus(String userName) {
       Account account = accountRepository.getAccount(userName);

       // Not implemented: make computing intensive checks on account

       return account;
   }
}
```

The checkAccountStatus() method gets a user name (such as "Alice") and returns an account after having performed some business logic on it. That business logic is supposed to be long and make various checks. In our example, we do no check and place a comment where they would happen.

2.1.6. Main

Our program starting point is very simple: it calls the business logic and displays the result at the console.

Source

```
package com.example.bank.client;

import com.example.bank.bean.Auditor;
import com.example.bank.model.Account;

public class Main {
   public static void main(String[] args) {
       Auditor auditor = new Auditor();
```

```
        Account account = auditor.checkAccountStatus("Alice");
        System.out.println("balance = " + account.getBalance() + " cents");
    }
}
```

2.1.7. Activity: Test the Java SE Version

For the moment, this code is pure Java SE and everything runs in the same JVM. Before we transform this code into Java EE - EJB code, we will test it as it is now.

1. Create a new Java (**non**-EJB) project into Eclipse. Name it *BankClient*. You will create the following classes inside that project.
2. Create the Account class as it is in this chapter, in the package com.example.bank.model.
3. Create the AccountRepository class as it is in this chapter, in the package com.example.bank.repository.
4. Create the Auditor class as it is in this chapter, in the package com.example.bank.bean.
5. Create the Main class as it is in this chapter, in the package com.example.bank.client.
6. Execute the Main class as a regular Java program.

2.2. Transform the Application into EJB

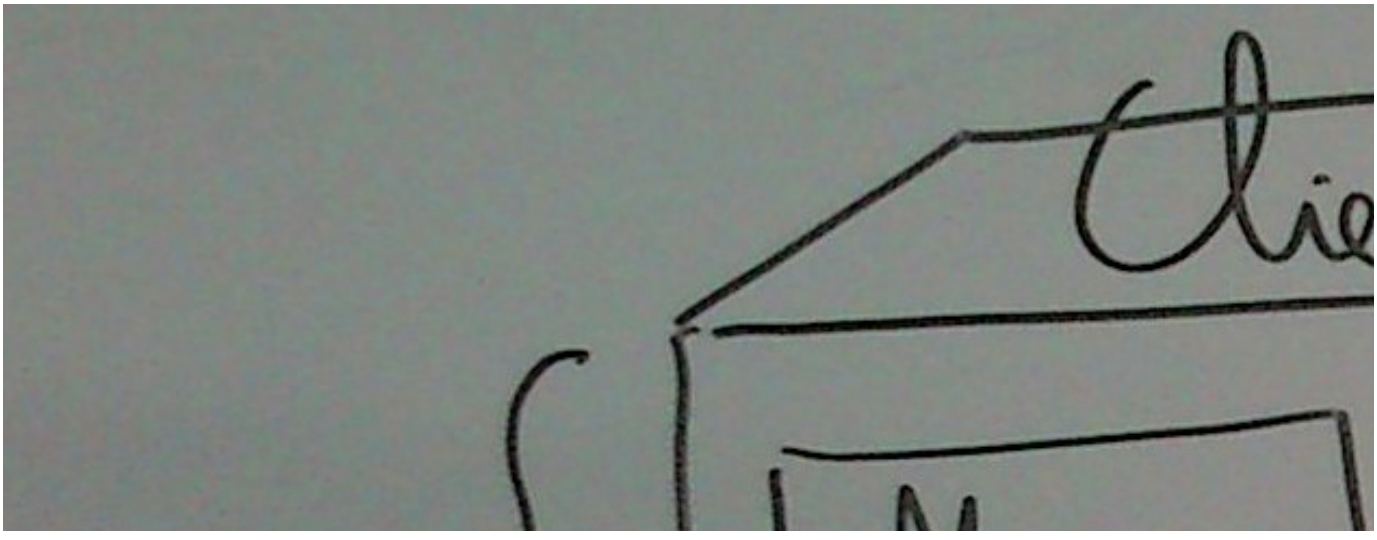
We have written a Java SE banking application. In this topic we will transform that application into the EJB model.

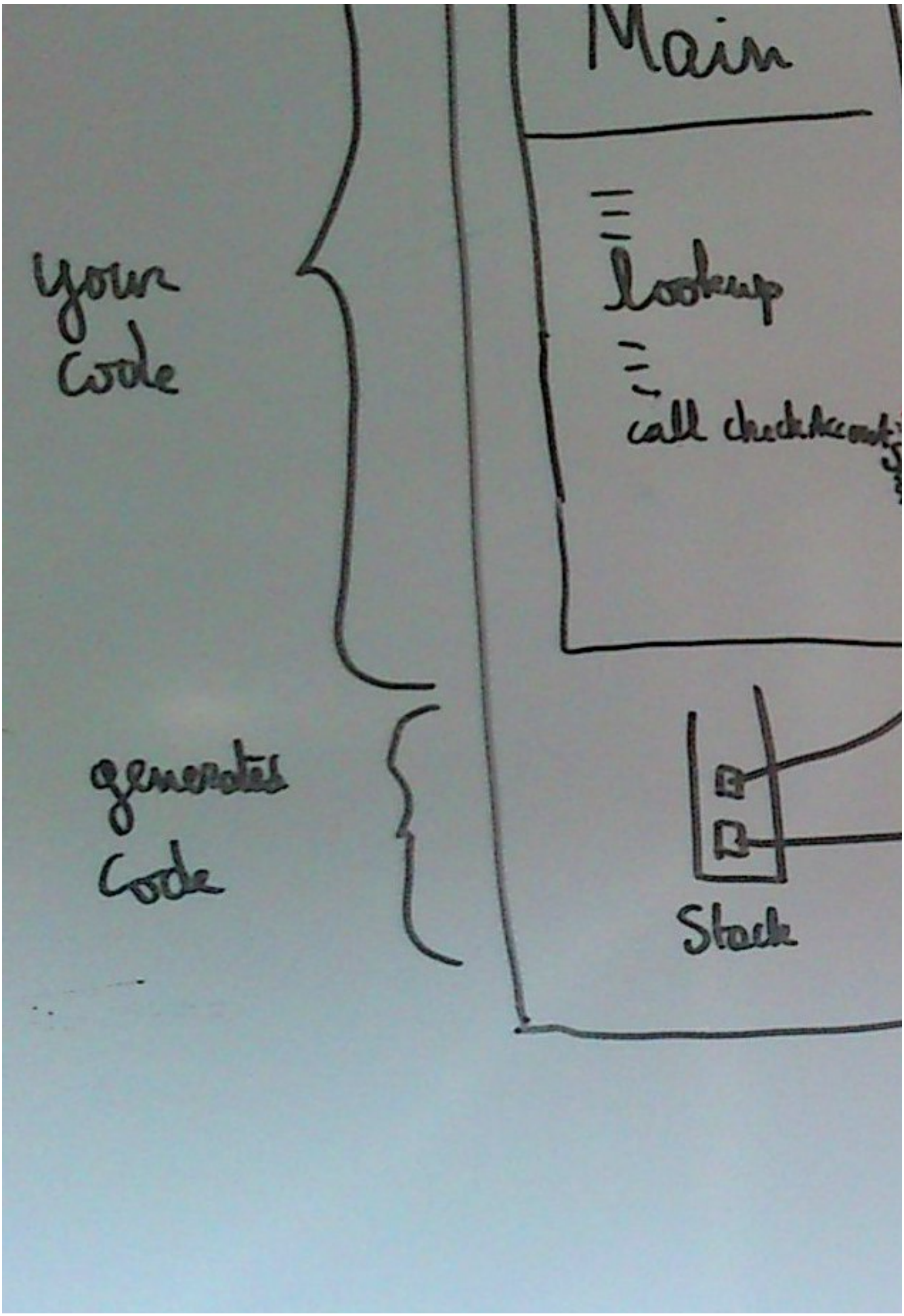
Session bean is the most used kind of EJB. Contrary to message driven beans, Its method can be called directly. The caller is usually a remote client or another EJB.

We will create a remote interface for the Auditor class. Rename the class and annotate it with @Singleton. We will make the client get a remote reference to Auditor, through a JNDI look-up. By doing so, we enable the Auditor business logic to run on a separate machine than the client, and to benefit the services from the EJB container (as transaction management, and scalability).

2.2.1. Architecture

The following diagram gives an overview of our application as it will be in its EJB version, with the internal details of the EJB architecture.





Our classes will have the following roles:

- **AuditorBean**: your EJB that runs inside the EJB container.
- **AccountRepository**: a regular Java class that is used by AuditorBean and runs inside the EJB container. In a further lesson, you will transform AccountRepository into an EJB.
- **Account**: a domain model class holding data. It will be both instantiated on the server and the client.
- **Main**: our Java SE code running in the client JVM, outside the EJB container.

Through this topic, we explain the remaining elements of that diagram, including stub, skeleton and proxy. The steps on the diagram are:

1. The client calls checkAccountStatus() on the stub.
2. The stub performs a remote call to the skeleton, over the network.
3. The skeleton calls the checkAccountStatus() method on the bean's proxy.
4. The proxy calls the checkAccountStatus() method on the bean. Your code executes.
5. AuditorBean calls AccountRepository.getAccount() and returns an Account instance.
6. That Account instance has been returned to the proxy, then the skeleton. It has been serialized and returned to the stub over the network. The stub has deserialized it on the client and returns it to the main method.

2.2.2. Stub and Skeleton

The client code needs to call the Auditor.checkAccountStatus() method. But the Auditor class is on another machine. To shield you from programming any networking communication explicitely, the EJB container will create a remote stub class and a remote skeleton class.

The remote stub is a client-side class exposing the business methods of your EJB. In this case, the stub must provide the checkAccountStatus() method. That method in the stub does not contain your business logic. The EJB container that generates the code fills it with networking communication code to reach the EJB container with the RMI-IIOP protocol.

The stub class communicates with a skeleton class which remains in the EJB server. That skeleton, will get remote networking calls from the stub, and call our Auditor.checkAccountStatus() business method from inside the EJB container.

2.2.3. Creating the Remote Interface

Your client code needs to act as if it has a reference to the Auditor object:

Source

```
Auditor auditor = ... // get remote stub from the EJB container
Account account = auditor.checkAccountStatus("Alice");
```

The client needs to know the business methods of the Auditor class, but does not need their implementations. That's exactly the purpose of interfaces in Java. You will create an interface with the business method of Auditor, that the

client should be able to use.

This will be the *remote interface* and you annotate it with *@Remote*. With this annotation, you indicate to the EJB container the role of that interface: it can be used by client code from other machines, to perform remote call with a a stub/skeleton pair.

Source

```
@Remote
interface Auditor {
   Account checkAccountStatus(String userName);
}
```

2.2.4. Creating the Bean

To avoid name collisions and make roles more clear, you rename the Auditor class *AuditorBean*. This is the most common naming convention for a modern EJB. You also annotate it with @Singleton to indicate that it is an EJB. We also make our bean implement the remote interface.

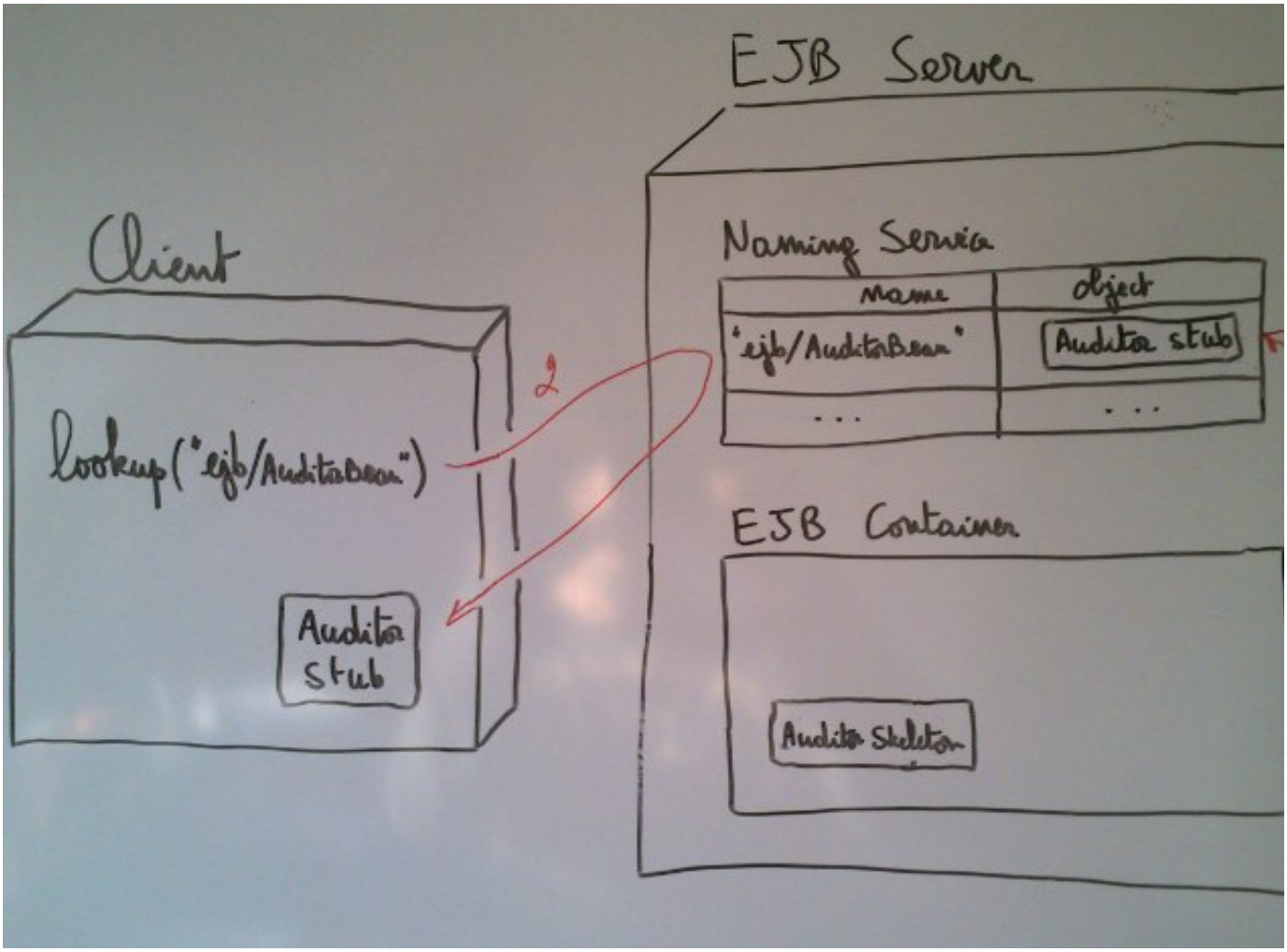
Source

```
@Singleton
class AuditorBean implements Auditor {

   AccountRepository accountRepository = new AccountRepository();

   Account checkAccountStatus(String userName) {
       Account account = accountRepository.getUserAccount(userName);

       // Not implemented: make computing intensive checks on account

       return account;
   }
}
```

2.2.5. Naming Service

The EJB server (GlassFish) contains a naming service that stores pairs of name / object. In the EJB Server, the EJB container do register the stub for the auditor bean. It uses the name referred in the @Singleton annotation of the AuditorBean class. This is the arrow 1 on the diagram below.

That naming service is accessible from outside the EJB server. The client will give the bean name to the naming service and get the stub in return. This is the arrow 2 on the diagram.



2.2.6. Getting a Reference to the Stub

JNDI (Java Naming and Directory Interface) is the API to access that naming service. It provides the javax.naming.InitialContext class. The only method that most Java programmers use on this class is lookup(String). It sends a request to the naming service, to get a reference to an EJB stub.

The client code below make a JNDI lookup to get the client stub from a symbolic name. Then the rest of the code is the same as the initial non EJB version of the Main class: we call the checkAccountStatus() method and print the balance of the returned account.

Source

```
class Main {
   public void main(String[] args) throws NamingException {
        InitialContext ctx = new InitialContext();
        Auditor auditor = (Auditor) ctx.lookup("ejb/AuditorBean");
        Account account = auditor.checkAccountStatus("Alice");
        System.out.println("balance = " + account.getBalance() + " cents");
   }
}
```

This code works in your case because you connect the local machine and default port. If you want to connect to the arbitrary context (not the default one) set the *java.naming.provider.url* variable on the context.

Source

```
InitialContext context = new InitialContext();
context.addToEnvironment("java.naming.provider.url", "192.168.1.1:1234");
```

2.2.7. Narrowing the Stub

To be 100% conform with the EJB specification, you need to add a third line to your client JNDI code: the call to the PortableRemoteObject.narrow() method.

Source

```
InitialContext ctx = new InitialContext();
Object o = ctx.lookup("ejb/AuditorBean");
Auditor auditor =  (Auditor) PortableRemoteObject.narrow(o, Auditor.class);
```

The usage of the narrow method is required by the usage of RMI over IIOP. For Java to Java communication, GlassFish (and JBoss) application server default to RMI over JRMP where the narrow method is not required. The examples of this course do not use the narrow method.

2.2.8. JNDI Name

Our client JNDI code is looking for the stub with a symbolic name "ejb/AuditorBean".

Source

```
Auditor auditor = (Auditor) ctx.lookup("ejb/AuditorBean");
```

When the EJB container starts, it registers your beans in the naming service. Before the v3.1 of the EJB specification, any EJB product (as GlassFish or JBoss) used its own naming conventions. Therefore, it is a good practice to no use the product specific naming convention and to name your beans yourself. You can do that through the annotation on the bean class:

Source

```
@Singleton(mappedName="ejb/AuditorBean")
class AuditorBean implements Auditor { ... ]
```

The EJB 3.1 standardized naming convention is not that simple and not covered in this topic.

2.2.9. Serializing the Parameters

Any value passed over the network between the stub and skeleton must be serializable. These values are transformed into a stream of byte, which is used on the other side to inflate new object instances.

It is the case for our String parameter to the checkAccountStatus(String) method. It should also be the case of the Account returned as result. To make Account serializable, we make it implement the java.io.Serializable interface:

Source

```
public class Account implements Serializable {

   private int balance;

   public int getBalance() {
       return balance;
   }

   public void setBalance(int balance) {
       this.balance = balance;
   }

}
```

When AuditorBean will return an instance of Account, the sekleton will serialize it into a stream of bytes, and pass it to the stub. The stub will create a new instance of Account on the client by deserializing it. The client instance will contain the same values (balance).

2.2.10. Proxies

Now that the client/server communication is configured and explained, let's cover the internals of the EJB container.

Beans always have a proxy. It is a class automatically created by the EJB container, and placed in front of a bean. You never directly call the bean itself. Instead you call methods on the proxy which calls the real bean instance. Additionally to calling the bean, the proxy may provide services as starting and stopping a transaction.

Because of these proxies, you never directly instantiate a bean with the *new* operator. The following code would not be EJB compliant.

Source

```
AuditorBean ab = new AuditorBean();  // NO!
```

Instead, you let the container provide you a reference to the proxy. You can, for example, do a JNDI lookup (InitialContext.lookup() as explained earlier). Another method that is not covered in this topic is to use dependency injection.

Once you have a reference to a proxy, it behaves as if you had a direct reference to the bean. You just call your business method on it.

Proxies are explained in more details in the *Lifecycle*lesson. Dependency injection is explained in the *Dependency Injection* lesson.

2.2.11. Activity: Create a Server Project

Your banking application code is in a Java SE project named *BankClient*. To make the client and server split clear, you create another project to contain the server-side code.

Create an EJB project named *BankServer*. In Eclipse, select New Project > EJB > EJB Project.

Move these packages/classes in that project:

- com.example.bank.bean.Auditor
- com.example.bank.bean.AuditorBean
- com.example.bank.repository.AccountRepository
- com.example.bank.model.Account

The Main class remains in the BankClient project.

Now the Main class should not compile anymore because it has no access to the Auditor and Account classes. A solution could be placing these missing classes into a jar file that you put in the build/class path the client project. To avoid repeating that operation everytime we modify these classes during these courses, we will use another solution: make the server project a dependency of the client project in Eclipse. It will make Eclipse put the code of BankServer in the build/class path of the BankClient project when compiling/executing BankClient/Main.

Select the BankClient project, then select Project > Properties > Java Build Path > Projects > Add > BankServer.

2.2.12. Activity: Transform the Application

In this activity, you execute the steps explained during this topic to transform your banking Java SE application into a EJB application.

1. Rename the class Auditor to AuditorBean
2. Create the remote interface Auditor.
3. Make AuditorBean class implement the Auditor interface and annotate it with @Singleton including mapped JNDI name.
4. Make the client Main method use JNDI to lookup for the stub.
5. Make Account implement Serializable.

Test you code by starting the EJB server and executing the Main.main() method.

You do not change AccountRepository yet, we'll take care of that later in this lesson.

2.3. Singleton Session Beans

We have transformed a Java SE application in to an EJB application and have seen how a session bean works in details.

There are 3 kinds of session beans: singleton, stateless and statefull. In this topic, you learn the specificities of singleton session beans, which is the type we have used so far. Contrarily to other beans, singletons have only one instance in memory. In most business situation it is the session bean of choice for new code, when that code is threadsafe, because it's the less resource consuming.

2.3.1. Singleton Pattern

Singleton design pattern has been in use for a long time, but it has been described for the first time by the Gang Of Four (i.e. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides). The detailed description of the pattern is beyond the scope of this course, but to understand the idea behind the Singleton Session Beans model, one need to have a clue about this design pattern.

In great brief, singleton design pattern describes class that can provide only single instance of itself. We cannot create more than one instance of singleton class. Singleton instance can be usually accessed via some kind of global reference (singleton global entry point).

A basic, commonly used example is :

Source

```
public class MySingleton {
   // Private instance attribute
   private static MySingleton instance;

   private MySingleton() {
       // Empty private constructor to avoid instanciation from outside.
   }

   // Getter of the singleton instance
   public static synchronized MySingleton getInstance() {
       if (instance == null) {
           instance = new MySingleton();
       }
       return instance;
   }
}
```

Within an EJB container, all this boilerplate code will become useless. You will just use a single annotation and the container will manage having one instance only without the need of explicit static variable and method.

2.3.2. Singleton in the context of EJB

Singleton beans have only one instance per EJB container. That instance is shared by all the threads needing to call a method on that bean.

The EJB container (who manages the lifecycle of the beans) will guarantee that given bean will be created only once per application. Whenever client wants to communicate with singleton session bean, container initiate interaction with the same bean instance. Please note that different clients access the same bean instance.

Creating singleton with EJB 3.1 specification is very easy. The only thing you have to do is to mark the bean class with the **@javax.ejb.Singleton** annotation:

Source

```
@Singleton
public class MyBean {
   ....
}
```

2.3.3. Threadsafety

Being threadsafe means that the behaviour of your code is not corrupted if multiple threads happen to execute it concurrently. The best way for a class to be threadsafe is to have no attributes, all the variables being local. A class is also threadsafe if all it's attributes are not modified and sharable.

Source

```
class MyThreadSafeClass {
   final int ATTRIBUTE_1 = 500;

.... methods
}
```

2.3.4. Non Threadsafe Auditor

Explaining a concurrency problem on a non threadsafe class helps defining threadsafty. We have added, to AuditorBean, a method *computeCreditLimit()* that returns how much money a customer can take from his account. That method takes long to execute, and because of that, is more exposed to concurrency. For simulating a slow business process, we have created the *simulateLongOperation()* method that makes the current thread sleep.

The method implementation is very bad, because it uses an attribute (creditLimit) where it should have used a local variable. That bad design will help us to make *computeCreditLimit()* not threadsafe.

The method starts by setting the creditLimit to 0. It waits 1 second, then change the value of the creditLimit, wait again, then if the user is rich, change creditLimit again, and waits a last time before returning the value. During that long execution where the thread waits 1 + 2 + 3 = 6 seconds, chances are that another thread will try to execute the method on that AuditorBean instance.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean {

   AccountRepository accountRepository = new AccountRepository();

   long creditLimit;  // Ugly variable that should better be local.


   public long computeCreditLimit(String userName) {
       Account account = accountRepository.getAccount(userName);

       creditLimit = 0;
       simulateLongOperation(1);

       creditLimit = Math.random();
       simulateLongOperation(2);

       if (account.getBalance() > 150) {
           // That customer is rich.
           creditLimit += (creditLimit*20)/100; // +20%
       }
       simulateLongOperation(3);

       return creditLimit;
   }

   private void simulateLongOperation(int seconds) {
       try {
           Thread.sleep(1000 * seconds);
       } catch (InterruptedException e) {
           throw new RuntimeException(e);
       }
   }
...
```

An possible scenario of failure would be:

1. Thread A starts executing computeCreditLimit(). It assigns a random number the creditLimit attribute, for example 54. Then it starts waiting 10 seconds.
2. During that time, 5 seconds after Thread A started executing computeCreditLimit(), Thread B starts executing that method too. It sets creditLimit to another random number, for example 16. Then Thread B starts waiting 10 seconds.
3. While thread B is waiting, Thread A has finished waiting and returns the value of creditLimit which is 16 (and not 54) because of thread B.
4. Finally, thread B has finished waiting too and returns 16 too, the same value.

This scenario is illustrated below:

Source

```
    public long computeCreditLimit(String userName) {
       Account account = accountRepository.getAccount(userName);

       creditLimit = 0;
       simulateLongOperation(1);

       creditLimit = Math.random();
       simulateLongOperation(2);

// 1 & 2. thread A - creditLimit == 54
// 2.     thread B - creditLimit == 16

       if (account.getBalance() > 150) {
           // That customer is rich.
           creditLimit += (creditLimit*20)/100; // +20%
       }
       simulateLongOperation(3);
// 3. thread B - creditLimit == 16

       return creditLimit;

// 3. thread A - returns 16
// 4. thread B - returns 16
   }
```

If threads A and B did not execute the method concurrently, they would not have returned the same value. Because of concurrency bad luck, they did return the same value. That code is not threadsafe.

The execution scenario below is more lucky:

1. Thread A executes the whole computeCreditLimit() method and returns 54.
2. Later after thread A finished, thread B executes the whole computeCreditLimit() methdo and returns 16.

2.3.5. Concurrency

Because the single bean instance is shared by multiple threads, the chances are big that sometimes, two threads execute a method of the bean concurrently. Singleton session beans should either be threadsafe (multi-thread resistant) or be synchronized (one thread at a time). You'll learn more about threadsafty in the next topic.

The @ConcurrencyManagement annotation enables you to control if multiple threads can execute methods of your bean:

- @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
- @ConcurrencyManagement(ConcurrencyManagementType.CONTAINER) - default

With BEAN, the container will let multiple threads execute on you bean, and your class is supposed to be threadsafe.

With CONTAINER, the container will enable only one thread at a time to execute you bean's methods. It's like synchronizing your bean or implementing the old servlet SingleThreadModel. Since there is only one instance of your bean, this option would probably create a performance bottleneck in your application. Despites this option is the default if you specify nothing, it makes the less sense.

Most of your session beans are by stateless by nature and will be:

- either your bean is threadsafe (as most session beans are or should be), and it is @Singleton with @ConcurrencyManagement(BEAN), multi-thread,
- either your bean is not threadsafe (but stateless), and it should not be a singleton to let the container create multiple instances and prevent it being a performance bottleneck. In that case you annotate it with @Stateless.

2.3.6. @Lock

CMC is the default concurrency model for singletons in EJB 3.1 specification. In CMC model, the responsibility for the concurrency management is delegated to the container. Singletons can be annotated with the @Lock annotation to indicate the type of concurrency for each bean method. There are two types of locking supported by the CMC:

- **@Lock(LockType.WRITE)** Exclusive lock. Method marked with this option cannot be executed concurrently.
- **@Lock(LockType.READ)** Non-exclusive lock. Method marked with this option can be executed concurrently.

If there is no @Lock annotation on the class, LockType.WRITE is the default.

The example below demonstrates overriding default CMC LockType.Write lock type on the bean method. Note the different meaning of override here, as annotations are not inherited.

Source

```
@Singleton
public class RegistryBean {

   // This method has @Lock(LockType.WRITE) by default
   public void modifyRegistry(String key, String value) {
       // perform some operations that require be to synchronized
   }

   @Lock(LockType.READ)
   public void displayStatus() {
       // this operation do not need to be synchronized
   }

}
```

2.3.7. @AccessTimeout

The @AccessTimeout annotation can configure how long a thread will wait to acquire the read or write lock. This annotation can also be used on the class or method level. The annotation maps directly to the java.util.concurrent.locks.Lock interface. The default timeout value is vendor specific.

- A value > 0 indicates a timeout value in the units specified by the unit element.
- A value of 0 means concurrent access is not permitted.
- A value of -1 indicates that the client request will block indefinitely until forward progress it can proceed.

Values less than -1 are not valid.

Examples:
@AccessTimeout(-1) - Never timeout, wait as long as it takes. Potentially forever.
@AccessTimeout(0) - Never wait. Immediately throw ConcurrentAccessException if a wait condition would have occurred.
@AccessTimout(30, TimeUnit.SECONDS) - Wait up to 30 seconds if a wait condition occurs. After that, throw ConcurrentAccessTimeoutExcpetion

2.3.8. History

Prior to Java EE 6, EJB specification distinguished two types of session beans - stateful and stateless. In the meantime other Java Enterprise frameworks (like Spring framework) started to promote using singleton services whenever possible. The latter approach influenced Java EE 6 specification, so now we can create also singleton session beans.

Singleton session beans have been introduced with the version 3.1 of the EJB specification. Most session beans being threadsafe, @Singleton with @ConcurrencyManagement(BEAN) is now the first choice for session beans. Before the version 3.1, stateless session beans were the most used option. Therefore, you will find many examples in the litterature with stateless session beans, even if technically, a singleton would make more sense in most cases.

In new applications, the large majority of session beans should be threadsafe singletons with the @ConcurrencyManagement(BEAN) annotation. It is the closest match to a singleton bean in the Spring framework.

2.3.9. Eager loading

By default singleton session beans are created lazily i.e. the first time they are accessed. This behavior reduces the overall startup time of the container.

Sometimes however we may want to create the singleton as soon as possible (eagerly). This may happen when creating singleton bean take so much time, that it is unacceptable for the client accessing the singleton as the first, to wait. The other reason is to validate that bean is properly initialized at the start-up of the container (to avoid discovering bug later, during the client session).

To tell the container to eagerly initialize the bean, use the **@Startup** annotation. For the **UserRegistry** example it would be:

Source

```
@Singleton
@Startup
public class UserRegistry {
 ... // irrelevant code ommited
}
```

2.3.10. Singleton and clusturing

What happens with singleton in the clustered environment? As mentioned above singleton is unique for the EJB container. That means that each node (machine) in the cluster will have its own instance of the singleton session bean.

It may cause problems if two singletons of different nodes have attributes with different values. You would not like that the execution result of a remote request depends on which node it has been executed on. But since singletons are stateless, it will not happen.

2.3.11. Activity: Create a Performance Bottleneck Singleton

In this activity you will simulate a long operation in our Auditor session bean. Then, you will run two concurrent clients to access it. We will compare the behaviour with the two concurrency management modes.

1. Start from the Auditor application.
2. In the AuditorBean's *checkAccountStatus* method, insert 3 lines of code:
    1. print "ENTER" with the current time at the console (println of new Date())
    2. make the current thread sleep for 10 seconds (Thread.sleep(10000) + exception handling)
    3. print "LEAVE" with the current time at the console.
3. Run the client code. Within the next 10 seconds, start it a second time.
4. Look at the server's console. In Eclipse there are 3 consoles but not all are displayed at the same time. The icon "Display Selected Console" in the top right corner of the console vue enables you to select which console to display.

XXXXXXXXXXX INSERT SCREENSHOT for CONSOLE with button to switch

You should see that the second thread was delayed to enter the method, until the first thread left. Also note the total execution time is 20 seconds. AuditorBean is a bottleneck: its clients have to wait long when there are many simultaneously.

Source

```
ENTER Wed Oct 12 23:23:45 CEST 2011
LEAVE Wed Oct 12 23:23:55 CEST 2011
ENTER Wed Oct 12 23:23:55 CEST 2011
LEAVE Wed Oct 12 23:24:05 CEST 2011
```

Add the @ConcurrencyManagement(ConcurrencyManagementType.BEAN) annotation on the AuditorBean class and execute the client twice again, in the same conditions.

You should see that both threads could be in the method at the same time. The total execution time is 11 seconds.

Source

```
ENTER Wed Oct 12 23:35:11 CEST 2011
ENTER Wed Oct 12 23:35:12 CEST 2011
LEAVE Wed Oct 12 23:35:21 CEST 2011
LEAVE Wed Oct 12 23:35:22 CEST 2011
```

2.3.12. Activity: Use a Non Threadsafe Bean

In this activity, you will implement the non threadsafe computeCreditLimit() method that you have seen in this topic, and run two clients concurrently. You will first make the code return wrong results. Then by changing the concurrency management strategy of the bean, you will make it work correctly.

1. Adapt the AuditorBean and its remote interface as on the the course example: add the computeCreditLimit() method and the creditLimit attribute.
2. Annotate your singleton AuditorBean with *@ConcurrencyManagement(ConcurrencyManagementType.BEAN)*
3. Adapt the client to call the computeCreditLimit method and display the results.
4. Test your code by deploying your server project and executing sucessfully your client project.
5. Run your client code twice. Start it a first time. Do not wait that it ends. You have 10 seconds to start it again. This will generate two client processes on your computer and two threads on the EJB container. Each client process has its own console in Eclipse. When the processes are finished (after 10 seconds), switch between both client's consoles to compare the results. You should see the same number, because you have reproduce the bad luck concurrency scenario explained in the theory. The total execution time should be less than 10 + 10 seconds.
   In Eclipse there are 3 consoles (2 clients and 1 server) but not all are displayed at the same time. The icon "Display Selected Console" in the top right corner of the console vue enables you to select which console to display.
   XXXXXXXXXX INSERT SCREENSHOT for CONSOLE with button to switch
6. On AuditorBean change the concurrency management strategy to *ConcurrencyManagementType.CONTAINER*.
7. Run your code twice again, within less than 10 seconds. Both client consoles should display different numbers, and the total execution time should be around 10 + 10 = 20 seconds, because the second thread had to wait that the first thread finished executing the computeCreditLimit method.

Session beans annotated @Stateless will enable us to run multiple threads on non-threasafe code. You will how in the next topic.

2.4. Stateless session beans

So far, we have used one kind of beans: singletons. Prior the introduction of singleton beans, stateless session beans were the only choice for non conversational beans. Stateless session beans are used for non threadsafe code which is a case where singletons should not be used. The EJB container may create multiple instances of your bean and will ensure that only one thread at a time is active on any instance.

In this topic, you will see the difference between a stateful conversational bean and a stateless bean. In the activity, you will experiment the container instantiating your stateless bean multiple times.

2.4.1. Stateless vs Statefull

Before formally defining statefull/lessness, let's compare both with an example

Source

```
public class CalculatorA {
  public int add(int op1, int op2) {
      return op1 + op2;
  }
}
```

If you call `calculatorA.add(3, 5);`, you always get 8, whatever method (with whatever parameters) you called before on that instance.

On the contrary, the following version of the Calculator is not stateless:

Source

```
public class CalculatorB {
  int result;
  public int add(int op) {
      result += op;
   return result;
  }
  public void clear() {
      result = 0;
  }
}
```

If you call `calculatorB.add(5);` the returned value depends on the previous calls on the calculator instance. For example, it will return 5 if `calculatorB.clear();` has just been called before the add.

CalculatorB is statefull, it's attribute *result* is the conversation state.

2.4.2. Definition

A Stateless session bean is session EJB that has no conversation state. It may have a state (attributes) as long as method behavior does depends on previous method calls. It gets all the necessary information from the parameter values.

Stateless session beans are annotated with @Stateless.

Source

```
@Stateless
public class CalculatorA {
  public int add(int op1, int op2) {
      return op1 + op2;
  }
}
```

Because a stateless session bean is has no conversation state, a client could call a method on whatever instance of the bean. If there are two instances on the EJB container, the call of a client on a first method of the bean could be redirected by the container to the first instance, and the second method call from the client could be redirected by the container to the second bean instance, with no impact on the returned result. Each time the client accesses the stateless bean, it has no guarantee that it will be served by the same instance again.

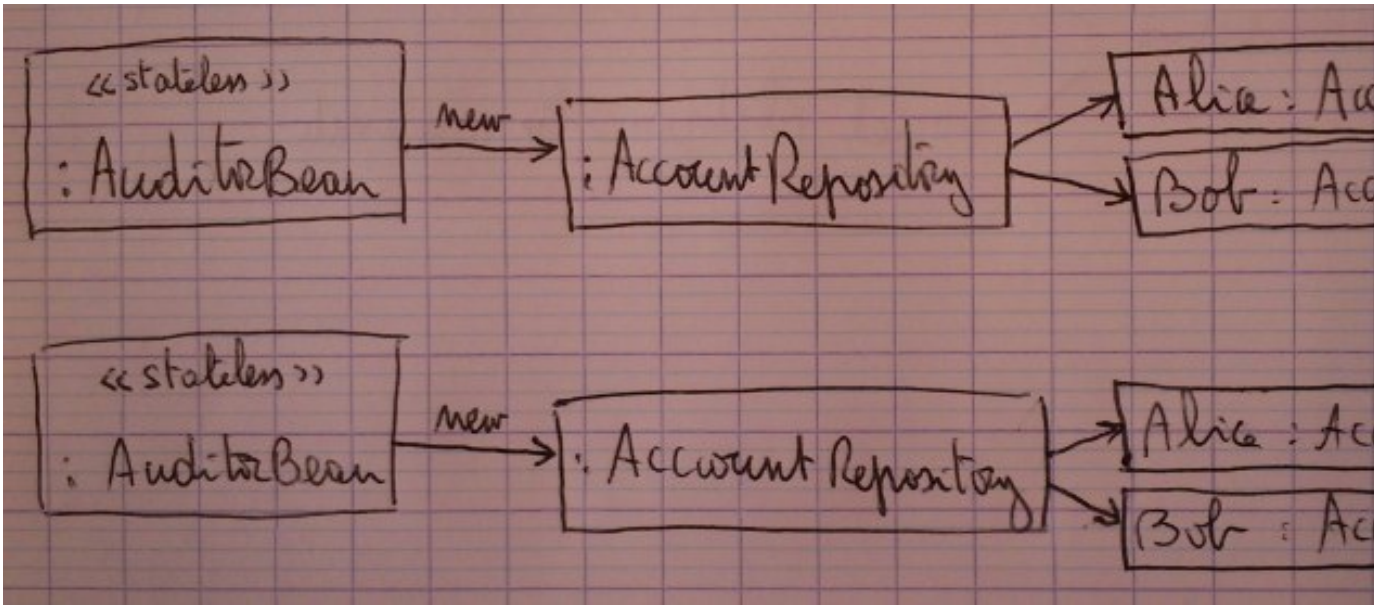
2.4.3. Multiple Instances

Prior the introduction of Singleton session beans, the EJB specification assumed that none of your session bean were threadsafe (while it's usually the contrary).

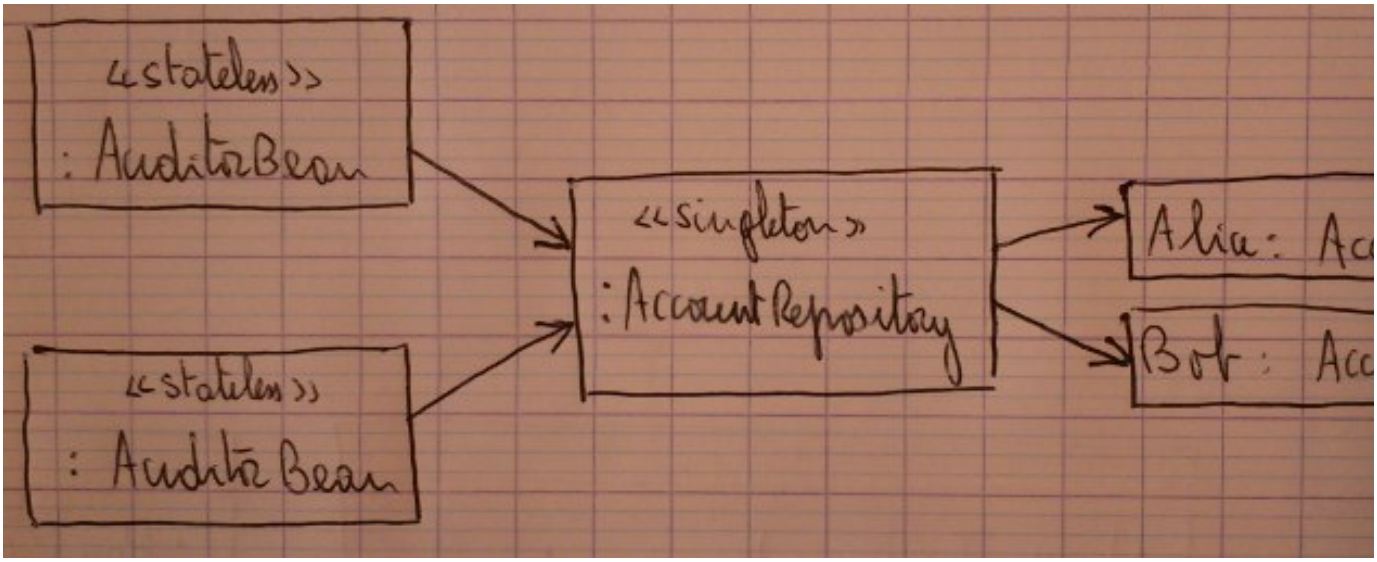
Session beans (annotated with @Stateless) never have two threads concurrently active on a single bean instance. To prevent threads from waiting and creating performance bottlenecks, the EJB container creates multiple instances of the same bean. If an instance is busy with a thread, and another thread would like to execute a method of the bean, then the container will create a new instance of the stateless session bean.

In the case of CalculatorA, annotated with @Stateless, if each computation method takes some time to execute, it is likely that multiple threads would need to be executing a method at a given point of time. In that case the EJB container will create other instances of CalculatorA instead of making these threads wait for each other.

Note that in our banking application example, AccountRepository is instantiated by AuditorBean. If AuditorBean is not a singleton anymore but a statless bean, there might be multiple instances of it, and each AuditorBean instance would instantiate AccountRepository once.



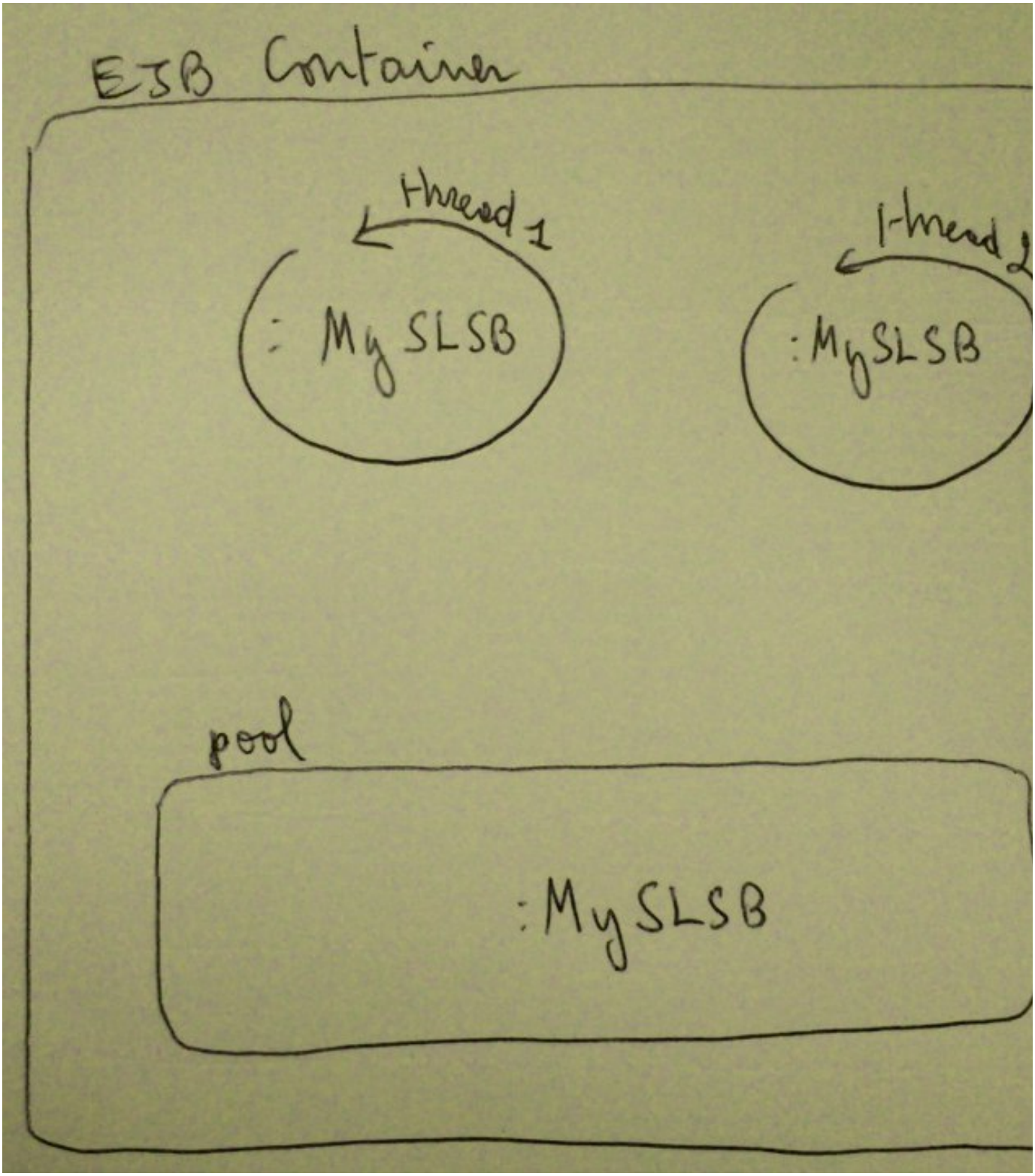
We'll turn AccountRepository into a Singleton EJB in another lesson. That singleton would be shared by all the instance of the stateless version of the AuditorBean.



2.4.4. Pooling

The instantiation of a your session bean could be costly, according to what you put in your constructor. To minimize the amount of instantiations of stateless session beans, the EJB container keeps a pool of reusable instances that are currently not used by any thread. When a thread needs a bean, the EJB container takes one from the pool. If the pool is empty, the EJB container either instantiate a bean, or make the thread wait for a bean to be available. The pool settings are implementation-dependent and usually includes the initial and max size allowed to the pool.

In the following diagram, the EJB container contains 3 instances of MySLSB (StateLess Session Bean). Two of these are being used by two threads. The last one is in the available pool.



2.4.5. Activity: Use a Stateless non-Threadsafe Bean

In the previous topic's activity, you have implemented the method computeCreditLimit in AuditorBean annotated with @Singleton. That method is not threadsafe. When you enabled two thread to work on the bean simultaneously, the method returned wrong results. When you forced the second thread to wait with ConcurrencyManagementType.CONTAINER strategy, the method returned correct results, but the total execution time was much longer. In that case, if many threads need to execute the method they will be need to wait each other and the AuditorBean becomes a performance bottleneck in your application.

Start from the end of that activity.

1. Change the annotation @Singleton to @Stateless.
2. Remove the annotation @ConcurrencyManagement (which is only meaningful for singletons).
3. Run the two client processes as you did in the previous topic's activity. Check that the returned numbers are correct (different from each other).

The EJB container will create a second instance of AuditorBean and both thread will execute the method simultaneously, each in its own instance. Note the total execution time which should be less than 20 seconds.

2.5. Stateful Session Beans

In the previous topic, we have given the Calculator class example with two version: stateless and stateful. We have defined stateless/fullness, and the notion of conversational state.

Sometimes, a business problem is simpler to solve with a statefull conversation. In this topic you create statefull bean to fulfill these needs.

2.5.1. Definition

Stateful session beans are business components communicated using the conversational style. Session bean is created once for every new client. Each time the client connects to the bean, the application server always associates the same bean with given client. This feature allows developer to hold the mutable conversation state within the session bean. It prevents any share/reuse and therefore, there is no pooling contrary to stateless session beans.

Use the @Stateful annotation to indicate a stateful session bean.

Please note that the bean holds the mutable state private for the client. That means that each time client accesses the bean, the application server needs to serve the same instance. It may have a significant impact on the server's memory according to:

- the number of clients,
- the average duration of the conversations,
- the memory footprint of (the business state of) each bean.

2.5.2. Stateful Calculator

The following calculator is statefull, as we have seen in the previous topic. It is stateful, because the result/behavior or one method call depends on the previous method calls on the same instance. The result of add() depends on the previous calls to add() and clear().

Source

```
@Stateful
public class CalculatorB {
   int result;
   public int add(int op) {
       result += op;
       return result;
   }
   public void clear() {
       result = 0;
   }
```

```
}
```

In the usage example below, you see that calc1.add(0) and calc2.add(0) do not return the same result, because of the past method calls on calc1 and calc2.

Source

```
CalculatorB calc1 = ...
calc1.clear();
calc1.add(8);

CalculatorB calc2 = ...
calc2.clear();

int result1 = calc1.add(0);  // returns 8
int result2 = calc2.add(0);  // returns 0
```

2.5.3. Activity: Stateful LoanAdvisor

You need a module in your banking application, to advise the conditions for loans. You decide to create a LoadAdviser EJB. Here is a typical scenario:

1. The customer identifies himself. For example "Alice".
2. The LoanAdvisor replies with a range of possible amounts for that customer. For example between $100 and $1500.
3. The client sends the desired amount, for example $1200.
4. The LoadAdvisor replies with a list of rates-duration. For example, 1 year at 5%, and 6 years at 4%.
5. The client sends the selected rate-duration to confirm the loan with the bank.

Note that for every client step, the customer takes a decision and sends a value. This would typically happen in a web application in a wizard form: the customer provides some data through a form, presses next, and get additional information as reply.

The following class implements such a business logic. These methods will be called by the client, that specified order:

1. getAmountRange(), takes a user name, retrieves the user's account and decides a minimum and maximum possible amounts for the loan.
2. getDurationRange(), takes the amount decides by the customer
3. confirmLoan(), takes the selected duration and rate. It returns nothing, this is the final step.

Source

```
public class LoanAdvisorBean {

   AccountRepository accountRepository = new AccountRepository();

   // Conversation state.
   Account account;  // null when not identified yet.
   long[] amountRange;
   Long amount;  // null when not set yet.
   Map<Integer, Double> rateRange;  // possible range. key = duration in years. va



   public long[] getAmountRange(String userName) {
       if (account != null) {
           throw new IllegalStateException("Bug: please, do not reuse Stateful ses
       }
```

```
        this.account = accountRepository.getAccount(userName);

        amountRange = new long[2];  // 2 longs, a min and max for the range.
        // Maximum twice the money on the customer's account.
        amountRange[1] = account.getBalance() * 2;
        // Minimum $100, and no more than the max.
        amountRange[0] = Math.min(10000, amountRange[1]);

        return amountRange;
    }

    public Map<Integer, Double> getRateDurationRange(long desiredAmount) {
        // First verify the parameter and conversation state.
        if (account == null || amountRange == null) {
            throw new RuntimeException("Bug: getAmountRange() method should be call
        }
        if (desiredAmount < amountRange[0] || amountRange[1] < desiredAmount) {
            throw new IllegalArgumentException("Bug: parameter out of possible rang
        }
        amount = desiredAmount;

        // Simplistic business logic.
        rateRange = new HashMap<Integer, Double>();
        double baseRate = 0.06;  // 6%
        if (amount > 100000) { // More than $1000
            baseRate -= 0.005;  // better rate.
        }
        rateRange.put(1, baseRate);
        rateRange.put(6, baseRate - 0.01);  // Longer = cheaper.

        return rateRange;
    }

    public void confirmLoan(int desiredDuration, double desiredRate) {
        if (account == null || amount == null || rateRange == null) {
            throw new RuntimeException("Bug: getAmountRange() and gerRateDurationRa
        }

        Double rateFromMap = rateRange.get(desiredDuration);
        if (rateFromMap == null) {
            throw new IllegalArgumentException("Bug: Invalid duration");
        }
        if (!rateFromMap.equals(desiredRate)) {
            throw new IllegalArgumentException("Bug: Invalie rate");
        }

        // Push things to the DB and send confirmation mail to client (not coded he
    }
}
```

Please start a working version of your the banking application. This may be your current version, or the solution of the first EJB version of it. Follow these steps:

1. Create the LoanAdvisorBean class, from the code proposed above.
2. Annotate it with @Stateful.
3. Create a remote interface for that bean and make the bean implement it.
4. Make the client call the 3 methods of LoanAdvisorBean, in the correct order. Before each call you may use the console to input values from the keyboard. After each call, please display the data returned by the bean.

Optionnal: Try a Stateless Session Bean.

Change the annotation from @Stateful to @Stateless and make it run again. With one client alone it should work

because the EJB container will put the bean in the pool after each method call, and reuse it for the next call. But if you execute your client twice, and if that client gets the old dirty bean instance for its second execution, the bean will start throwing exceptions because of its defensive coding.

2.6. Determine the Appropriate Session Bean type

You have learned all the kind of session beans. When you create a new bean, you will face the choice of its type: singleton, stateless or stateful. How to make the right choice? In this topic, we review the typical reasons for each possible choice of session bean.

2.6.1. State Level

Your choice depends on the "statelessness" of the code inside your bean. There are 3 levels of statelessness:

1. **Threadsafe**. This is the ultimate stateless type. It is so stateless, that even in the middle of its execution, another thread can use it with no risk.
2. **Non-Threadsafe but Stateless**: The bean is stateless: methods can be called in any order and by any client. But only one client at a time because the code is not threadsafe.
3. **Statefull**. By nature, a statefull, conversation-oriented class is not thread safe at all.

2.6.2. ThreadSafe Session Beans

In a business application session beans can typically be divided into to categories: service and repository/dao.

Service beans contain business logic. AuditorBean of our bank application is a typical example of service bean. Their method get parameters. They typically compute return values using these parameters only, or data retreived from the DB inside the method. They usually store no value in no attribute and are threadsafe.

Repository or DAO (Data Access Object) beans contain DB access logic. They use JPA or JDBC to execute queries and return results. A typical method of an AccountRepository bean would be findAccountsWithNegativeBalance() for example. It would contain a query and would return a collection of Accounts (those with a negative balance).

2.6.3. Repository Beans are ThreadSafe

If you have worked with Java code accessing a relational database, you have proably either worked with a JDBC connection or a JPA EntityManager (which are similar on a very high level point of you). A repository bean will typically have the JDBC connection or JPA EntityManager as attribute. Multiple threads calling methods on the bean instance will share that attribute. If that attribute was a simple EntityManager, the bean would not be threadsafe at all. But that attribute is in fact a proxy to a collection of EntityManager and the EJB container associates one EntityManager instance with each thread behind the cover. Repository beans are typically threadsafe and their only attribute is that threadsafe multi-thread EntityManager.

2.6.4. Stateful Session Beans Characteristics

Stateful session beans hold the conversational state for the client between two method calls. They are a kind of extention of the client on the server side which may be convenient, but has also drawbacks:

Stateful beans are slower. Because they are bound to one specific client and the EJB container must instantiate and maintain many of them without any possibility of sharing and reuse. To spare memory, the EJB container may passivate them, which also make these bean appear as being slower.

Stateful beans use more system resources. Since each client has its own bean instance, more server memory is used. They also consume CPU and disk activity when being passivated/activated.

2.6.5. Session Bean Settings

According to the sate level of your code, you select the corresponding bean type:

1. **@Singleton @ConcurrencyManagement(BEAN)**: This option makes the most sense for most session beans. They are stateless and threadsafe.
2. **@Stateless**: If your bean is stateless but not threadsafe, this option will enable the container to create multiple reusable instances (one per concurrent thread).
3. **@Stateful**: If your bean is conversation oriented. Stateful beans are resources consuming and should be avoided. Cases where that option is needed for other reasons than bad design are rare. You try to avoid stateful beans, but it is a design possibility.

@Singleton @ConcurrencyManagement(CONTAINER) should never be used. Or only in the rare case where you only want one client at a time, active on non-thread safe code. The reason for that need to happen is probably related to very obscure and badly designed legacy code. Being forced to use CMC (Container Managed Concurrency) Singletons never happens in most projects.

2.6.6. Activity: SalaryTaxComputer

For an employee pay application, you are turning some legacy classes into EJB. One of them is named *SalaryTaxComputer* and seems stateless, not conversational. However you have no guarantee that it is threadsafe and it seems hard to find out. This class does not seem resource consuming to instantate, but you noticed that some methods are very slow to return a value. What is the best and safe choice for this class:

( ) @Singleton @ConcurrencyManagement(BEAN)

( ) @Singleton @ConcurrencyManagement(CONTAINER)

( ) @Stateless

( ) @Stateful

Explanation: It is not @Stateful because the class is not conversational and is stateless. It is not @Singleton @ConcurrencyManagement(BEAN) because the code may not be threadsafe and you don't want to risk multiple threads executing on the same instance. It is not @Singleton @ConcurrencyManagement(CONTAINER) because some methods are slow and if only one thread at a time can be active on the single instance, it might cause a performance bottleneck. It is @Stateless, since creating additional instances seems not resource consuming for that bean.

Solution: @Stateless

2.7. Session Bean Clients

The client of a session bean is either a remote client or another bean.

To make your bean usable by a remote client, you must either create a remote interface or use a web service.

To make your bean usable by another bean within the same EJB container, you need no interface. Optionnally you can use a local interface.

2.7.1. Remote Interface

Up to now, every bean you have created in his course has a remote interface annotated @Remote.

On the client side, you upcast the stub (returned by JNDI) to that remote interface. In the code below, *Auditor* is the remote interface implemented by both AuditorBean and the stub returned by the lookup.

Source

```
Auditor auditor = (Auditor) ctx.lookup("ejb/AuditorBean");
```

All the parameters and return value of the methods of a remote interface must be Serializable. In the following example, the String "Alice" and Account must be Serializable.

Source

```
Account account = auditor.checkAccountStatus("Bob");
```

2.7.2. Web Service

Another way to enable remote calls to your seteless or singleton session bean is to expose it as a web service. You need no remote interface. Just annotate it with *@WebService* and they can be accessed using the JAX-WS client (i.e. using SOAP messaging).

Source

```
@Stateless
@WebService
public class HelloService {

  public String sayHello() {
     return "Hello!";
  }

}
```

That's is as far as the EJB technology is involved. The rest of the story is the same for any web service (EJB based or not).

The client code (in any language supporting web services) will access the web service with the WSDL file. This is the WSDL file generated by the EJB container for the HelloService above:

Source

```
<definitions targetNamespace="http://hello.example.com/" name="HelloService">
   <types>
       <xsd:schema>
           <xsd:import namespace="http://hello.example.com/" schemaLocation="http:
       </xsd:schema>
   </types>
   <message name="sayHello">
       <part name="parameters" element="tns:sayHello"/>
   </message>
   <message name="sayHelloResponse">
       <part name="parameters" element="tns:sayHelloResponse"/>
   </message>
   <portType name="HelloService">
       <operation name="sayHello">
           <input wsam:Action="http://hello.example.com/HelloService/sayHelloReque
           <output wsam:Action="http://hello.example.com/HelloService/sayHelloResp
       </operation>
   </portType>
   <binding name="HelloServicePortBinding" type="tns:HelloService">
       <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="docum
       <operation name="sayHello">
           <soap:operation soapAction=""/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
       </operation>
   </binding>
   <service name="HelloService">
       <port name="HelloServicePort" binding="tns:HelloServicePortBinding">
           <soap:address location="http://localhost:8080/MyProject/HelloService"/>
       </port>
   </service>
</definitions>
```

2.7.3. No Interface

Some beans do not need to be accessed remotely. These beans are access from within the EJB container from other beans. Since the v3.1 of the EJB specificiation, it is possible to have no local interface for these beans. Just annotate them with @Singleton, @Stateless or @Stateful annotation, and they are accessible from other beans.

Beans having the role of Repository/DAO have typically not remote client. They are local beans.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountRepositoryBean {
   ...
}
```

The client bean obtains a reference to the local bean through dependency injection as explained in the Dependency

Injection lesson.

2.7.4. Local Interface

Alternatively, a local bean can implement an interface annotated with with @Local as in this example:

Source

```
@Local
public interface HelloServiceLocal {
  public String sayHello();
}
```

This interface was mandatory for local beans prior the version 3.1 of the EJB specification. Nowadays, there is no reason to use that option unless you want to systematically create interfaces for each beans. It could be the case if you mock them for unit testing.

2.7.5. Activity: Quiz

When EJB such as AuditorBean needs to access another EJB, such as AccountRepositoryBean, what is true (select all the possible answers).

[ ] AccountRepositoryBean class needs to be annotated with @Local

[ ] AccountRepositoryBean class needs to be annotated with @Remote

[ ] AccountRepositoryBean must implement an interface annotated with @Local

[ ] AccountRepositoryBean must implement an interface annotated with @Remote

Explanation:

Choices 1 & 2: the @Local and @Remote annotations are not directly used on the bean class but on an interface that the bean class implements.

Choices 3 & 4: Since the EJB 3.1 specicification, your bean class does not need to implement any interface to be referred by another bean.

Solution: none of the choice is true.

On what kind(s) of EJB can the @WebService annotation be placed? Select all the beans that applies (if any).

[ ] Singleton

[ ] Stateless

[ ] Statefull

[ ] Message Driven

Explanation:

Choice 3: Statefull beans cannot be web services because HTTP is a stateless protocol and it would require some extra mechanism for a client to stick to a specific web service EJB instance. Web services are stateless by definition.

Choice 4: Message driven bean get messages, not remote calls as web services do.

Solution: 1 & 2 (Singleton & Stateless).

3. Message Driven Beans

Session beans allow us to execute business logic encapsulated in the components. The execution of the business logic is performed synchronously on client demand. But how to handle logic that should be executed asynchronously, without the direct call from the client?

Calls to session bean instances are always bound to their interfaces and this accounts for high coupling. In case low coupling is important, synchronous communication is not the best solution.

Another disadvantage of synchronous communication is occupation of server ressources for the time the connection between client and server is alive and may be the session bean holds additional connections to a database (OK, the application server takes care for these ressources but you know there is not such a thing like limitless ressources).

If one of the problems above matters, this is point where JMS (Java Messaging Service) together with the asynchronous processing should be used.

3.1. Create a Message Driven Bean

This topic explains the main concept of message driven beans, as Messaging Oriented Middleware and Asynchronous communications.

You will implement the case of a tax return application where tax forms goes through message queues and are consumed by the EJB container, where some resource intensive processing happens.

You will setup the necessary infrastructure to experiment your first bean. The next topics goes into more details with the concepts and technology.

Message driven beans are used in applications based on messaging for chaining processing of heavyweight jobs.

3.1.1. Definition

Many business applications use messaging in their back-end tiers. They rely on a MOM (Messaging Oriented Middleware) to persist the messages in queues. Software components that need to consume these messages can register the MOM to be notified when a message arrives.

A message driven bean is an EJB that consumes messages from a MOM. It is annotated @MessageDriven, and implements the MessageListener interface. That code is explained further.

Source

```
@MessageDriven(...)
public class MyMessageDrivenBean implements MessageListener {
   @Override
   public void onMessage(Message message) {
     ...
   }
}
```

Because a message driven bean is only activated by the EJB container when a message arrives, it needs no remote or local interface.

3.1.2. Asynchronous

A MOM is a way for a software components to call another one asyncrhonously. The software component sending the message does not expect an immediate reply and contiunes it's processing. The messages resides on the MOM, and the software component that should consume the message may be busy or even down. The MOM acts as a reliable message store and will deliver the message when the consumer is up and ready to process it.

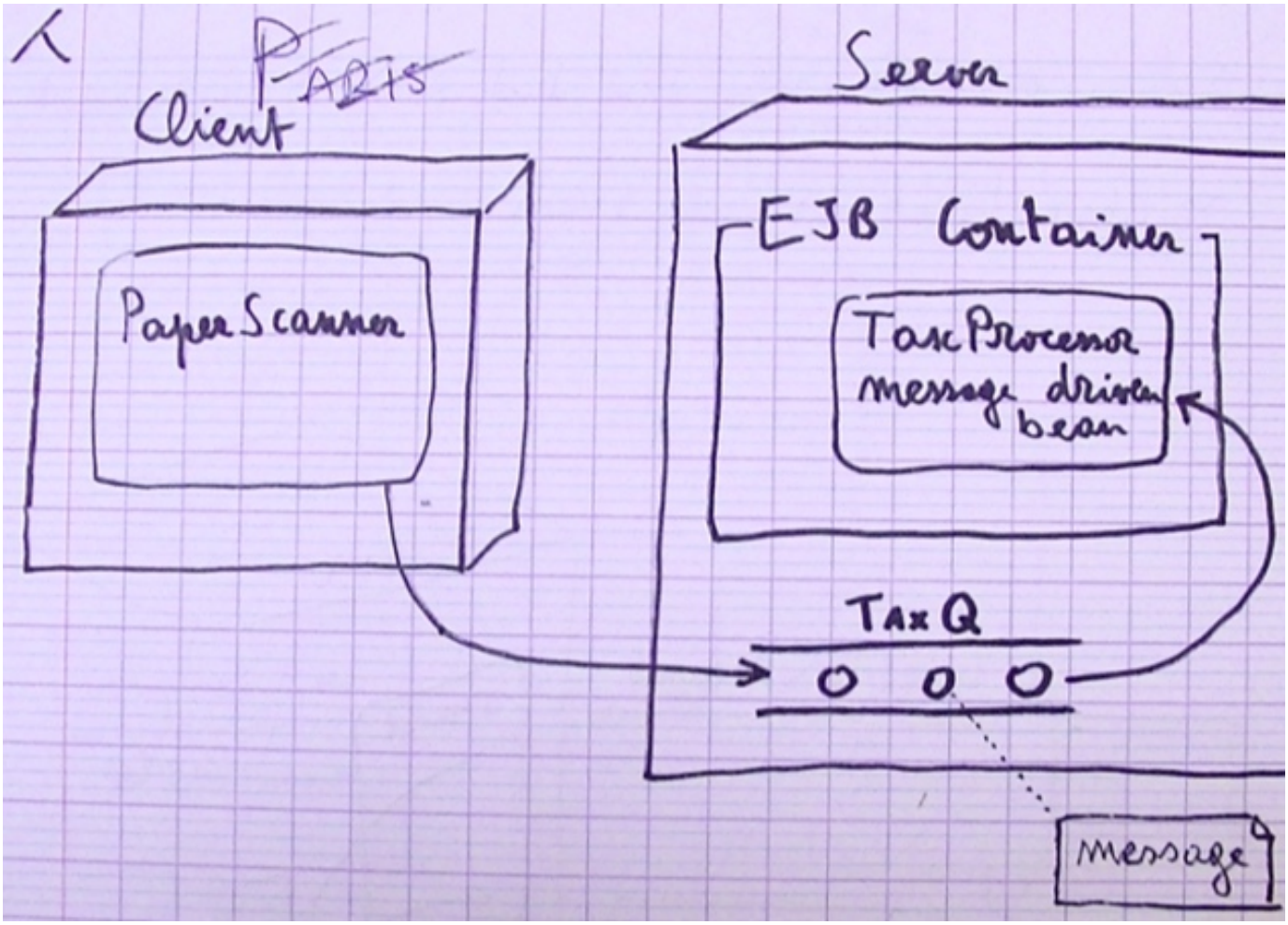
On the contrary, the session beans rely on syncrhonous communication: when a client calls a method on a session bean, it waits for the session bean to finish its processing and to return a value. During the processing on the session bean, the client is blocked.

3.1.3. Tax Return Application

Our business scenario for needing a message driven bean, is a tax processing application.

The client is a machine that reads paper tax return forms sent by the citizens. The main purpose of that machine is to scan paper, not to compute taxes. There maybe multiple client machines in many cities.

When a client machine has sucessfully scanned a tax return, it puts the data on a queue. In our configuration, that queue resides on the EJB server machine. In the EJB container, a message driven bean consumes the message (the tax return form data) and processes the business logic, as validating that data, cross checking it with data from employers, computing tax rates, etc. That process will probably end up with some update in the government DB and the printing of a letter to be sent to the citizen.



The key part of this diagram is the Queue. It is named TaxQ and is accessed by:

- PaperScanner, the client.
- TaxProcessor, the message driven bean on the server.

3.1.4. Client Sending a Message

The client writes a message on the queue named "TaxQ" with the JMS API. Java Messaging Services (JMS) is the Java API to access queues of a MOM. The code below may seem cryptic. The key code fragment is:

Source

```
message = session.createTextMessage();
message.setText("John Doe earned $100000 this year.");
mPoducer.send(message);
```

The rest of the code around it is boilerplate code to find a reference to the queue TaxQ, and open a connection to it. It will be explained in details during the next topic. This is the full code:

Source

```
public class PaperScanner {
   public static void main(String[] args) throws Exception {
       Connection connection = null;
       MessageProducer mProducer = null;
       Session session = null;
       try {
           Context context = new InitialContext();
```

```
            ConnectionFactory cFactory =
              (ConnectionFactory) context.lookup("ConnectionFactory");
            Destination destination =
              (Destination) context.lookup("jms/TaxQ");
            connection = cFactory.createConnection();
            session = connection.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            mProducer = session.createProducer(destination);
            message = session.createTextMessage();
            message.setText("John Doe earned $100000 this year.");
            mProducer.send(message);
        } finally {
            if (mProducer != null) { mProducer.close(); }
            if (session != null) { session.close(); }
            if (connection != null) { connection.close(); }
        }
    }
}
```

3.1.5. Server Listening to the Queue

On the server, a Java class is configured to be activated if a message comes. It is annotated as a message driven bean, with the necessary information to be linked to TaxQ.

This TaxProcessor class implements the JMS MessageListener interface and provides an onMessage() method that the EJB container will call for each new message available on TaxQ. In that method, our TaxProcessor just prints the message at the console. Our tax return processing business logic would come here.

Source

```
@MessageDriven(activationConfig = {@ActivationConfigProperty(propertyName = "desti
               propertyValue = "javax.jms.Queue")}, mappedName = "TaxQ")
public class TaxProcessor implements MessageListener {
    @Override
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            System.out.println("Tax return received: "
                    + textMessage.getText());
        }
    }
}
```

3.1.6. Activity: Setup the Queue

In order to make our tax return application work on your machine, you need to:

- Create the TaxQ through Glassfish admin interface.
- Write the Java SE client class in one project.
- Write the Java EE message driven bean in another project.

In this activity, we focus on creating TaxQ and associated JMS configuration on the GlassFish administration console:

1. Create the Queue (JMS physical destination).
2. Create the ConnectionFactory.
   Create the JMS destination resource.

3.

Make sure that your EJB server is started (*Servers* view in Eclipse).

To login the GlassFish administration console, open a web browser and go to that addess:

Source

```
http://localhost:4848
```

In the Tree pane on the left, select the node Configuration > Java Messaging Service > Physical Destinations.

XXXXXXXX IMAGE SCREENSHOT

In the JMS Physical Destinations pane, click the New button.

Enter the name of the queue (respect case): TaxQ

Select *java.jms.Queue* as type of JMS physical destination.

XXXXXXXX IMAGE SCREENSHOT

Click the save button.

After you have created the Queue, you need to create a ConnectionFactory. This is a mandatory JMS object that will enable your client to remotely connect the TaxQ.

In the Tree pane on the left, select the node Resources > JMS Resources > Connection Factories.

XXXXXXX IMAGE SCREENSHOT

In the JMS Connection Factories pane, click the New button.

Enter "ConnectionFactory" as the pool name. Remember, this string is used by the client to look for the remote connection factory object:

Source

```
            ConnectionFactory cFactory =
              (ConnectionFactory) context.lookup("ConnectionFactory");
```

Still in the GlassFish form, select *javax.jms.ConnectionFactory* as resource type.

XXXXXXX IMAGE SCREENSHOT

Click the save button.

After you have created the Queue and the ConnectionFactory, we need to complete a last configuration step: defining a JMS destination resource. It is like giving a public name to TaxQ.

In the Tree pane on the left, select the node Resources > JMS Resources > Destiniation Resources

XXXXXXX IMAGE SCREENSHOT

In the JMS Destination Resources pane, click the New button.

Enter "TaxQ" as JNDI name. This unique name will be used the the client to identify the queue. Remember the client

code:

```
Source
            Destination destination =
              (Destination) context.lookup("jms/TaxQ");
```

It is used in the message driven bean annotation as well. Remember the server code:

```
Source
        @ActivationConfigProperty(propertyName = "destination",
               propertyValue = "jms/TaxQ")})
```

Still in the GlassFish form, enter the Physical Destination name *TaxQ*. This is the internal name of the queue that we created in GlassFish at the beginning of this activity. By luck (or because we try to keep things consistent), it is the same name as the public name you are defining.

Specify the resource type: *javax.jms.Queue*

Click the Ok button.

3.1.7. Activity: Send a Message

Now that the TaxQ queue has been setup on your GlassFish server, it is time to write the client code that will create a new message into the queue.

Create a new Java project. It is not a Java EE project, just a regular Java SE project. From Eclipse, select File > New > Project... > Java > Java Project.

In that project, create the PaperScanner class as described earlier in this topic.

Before executing your project, you will verify that TaxQ is empty on GlassFish.

Make sure that your GlassFish server is still started and display the administration console. In the Tree view, select XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ????? how to display the content of queues ??????????????????????

Go back to Eclipse and execute your PaperScanner.

Go to the GlassFish administration console to display the content again.

XXXXXXXXXXXXXXXXXXXXX SCREENSHOT.

3.1.8. Activity: Consume a Message

You managed to put a message in TaxQ. It is time to write the server side code to make the message driven bean consume it.

In a Java EE project (your project used for creating the session beans, or a new project), create the TaxProcessor class as described earlier in this topic.

Restart your GlassFish server to deploy your new EJB.

Go to the GlassFish administration console to display the content of TaxQ. It should be empty.

XXXXXXXXXXXXXXXXXXXX SCREENSHOT.

And the server console should have displayed the message:

Source

```
Tax return received: John Doe earned $100000 this year.
```

Congratulations! You created a queue on GlassFish, then the client code to produce a message and a message driven bean to consume it.

3.1.9. Optional Activity: Configuring GlassFish with the Command Line

You have defined a queue in GlassFish using the admin console UI. You may prefer using the command line.

You can use the *asadmin* command line tool in the /bin folder of your GlassFish installation folder.

This command creates the queue.

Source

```
asadmin create-jms-resource --restype javax.jms.Queue jms/TaxQ
```

If you want to be sure it worked, you can list all the JMS resources using another command (when your server is running):

Source

```
asadmin list-jms-resources
```

3.2. JMS

Creating a message driven bean is as simple as annotating a class. With messaging, all the science is in the client and in the MOM. This topic, explains in details, the client code using JMS for sending a message, that you have written in the previous topic. It lists MOM products, then explains the architecture difference between topics and queues. It explains how JNDI is used to store JMS objects reference, and details the scenario for the tax return application. Finally, it reviews the JMS message types.

3.2.1. Reasons for Messaging

Three main advantages of a messaging solution (message driven bean), rather than to a remote call solution (session bean) are:

- **Coupling**: It relaxes the coupling between the client and the server component. The client does not have to know the server's remote interface.
- **Asynchronousity**: The client does not have to wait for the server to have processed the request. It is not blocked in case that process takes time.
- **Dependency**: The client is less dependent on the availability and workload on the server. If the server is down, the stored message will not be lost, and will be processed later.

A disavantage of messaging over remote call is the lack of immediate reply. In some cases, the client needs return values from the server in order to show an immediate feedback to the user (through the UI).

3.2.2. Products and Standards

Many messaging products coexist on the market in both commercial and opensource categories.

Commercial products:

- MQ-Series (IBM)
- MSMQ (Microsoft)
- BEA Weblogic (Oracle)

Open source products:

- RabbitMQ (SpringSource)
- ActiveMQ (Apache)
- JBoss Messaging (JBoss)
- GlassFish Message Queues (Oracle) - part of GlassFish Application Server

Historically, each product has its own way of communicating with its clients (message producers and consumers). On the Java platform, we are lucky to have JMS (Java Messaging Services), a standard API to access any messaging product that supports it. Most products nowadays support JMS.

The JMS API is part of Java EE.

3.2.3. Queues and Topics

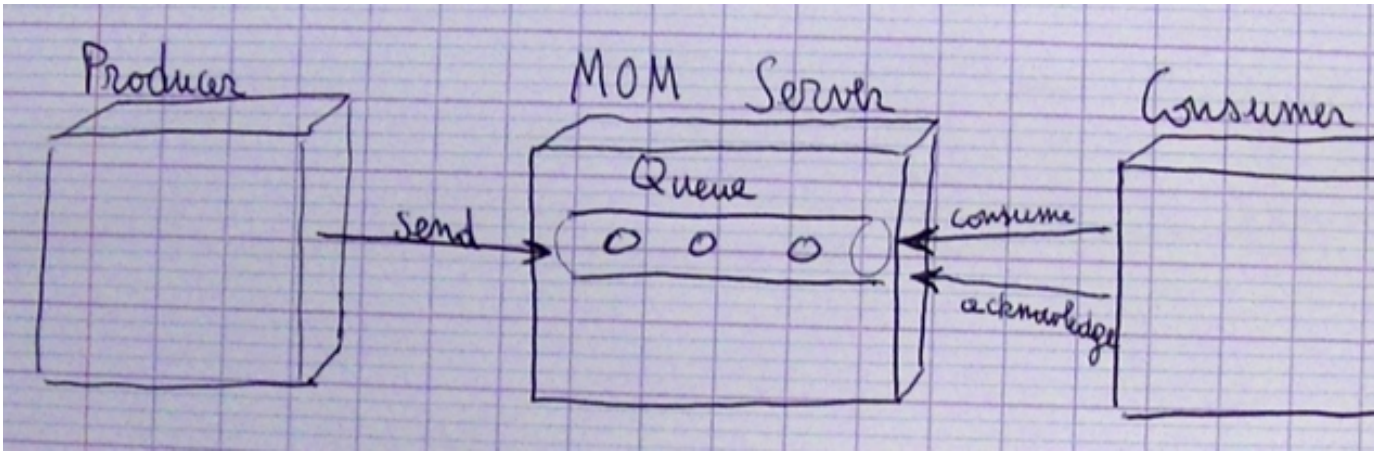
Messaging middleware propose two main message stores, which are named places to store a message:

1. Queues, for the point to point method,
2. Topics, for the publish/subscribe method.

3.2.3.1. Queues

In Point-To-Point communication JMS producer sends message to the queue. Then JMS consumer reads the messages from the queue and processes them. In this type of communication each message will be consumed only once.

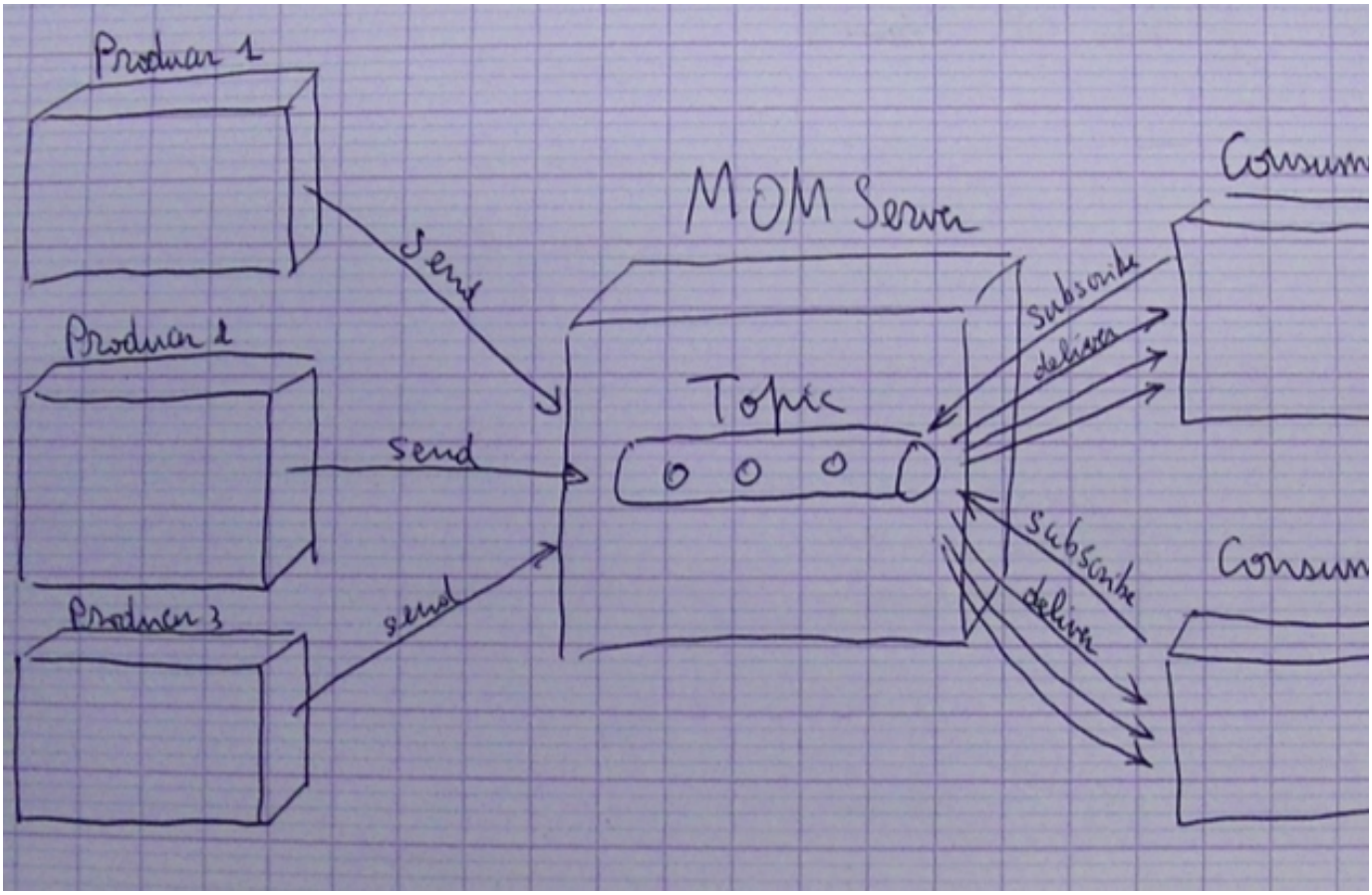
The other way for a message to be cleared from the queue is a timeout to occurs. A timeout can indeed be associated with each message by the client, or set at the queue level.



A Queue is used when only one consumer program is getting the messages and no message can be missed (even if a message arrives when the consumer is down).

3.2.3.2. Topics

On the other hand, in the Publish-Subscribe model, multiple consumers register to the given topic. Then producers send messages to that topic. Each message sent to the topic is received by all subscribers. In this type of communication single message can be (and usually is) processed by many consumers.



Message driven beans can consume a queue or a topic.

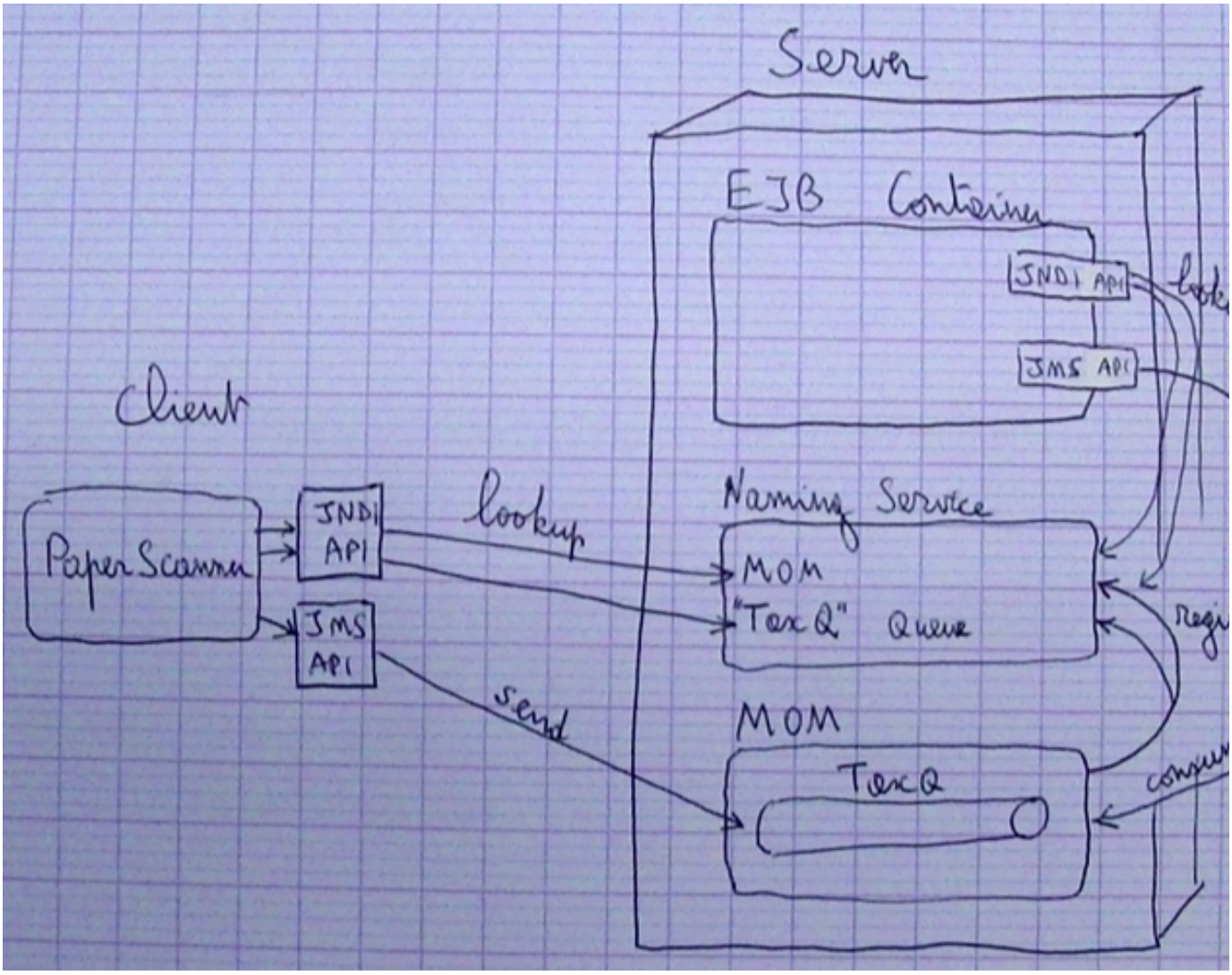
3.2.4. JNDI and JMS

Queues and topics are identified by a name, hold on a directory service. JNDI (Java Naming and Directory Interface) is the Java API to access such a directory service. System administrators can change the remote names of these queues and topics. Message senders and consumers use JNDI to get a remote reference to a queue or topic from that remote name.

In our example, the Java SE client did explicitely used JNDI to find the ConnectionFactory and the Destination. This code starts with instantiating a new InitialContext. This is the same JNDI object we used to lookup for session beans.

Source

```
Context context = new InitialContext();
ConnectionFactory cFactory =
  (ConnectionFactory) context.lookup("ConnectionFactory");
Destination destination =
  (Destination) context.lookup("queue/TaxQ");
```



3.2.5. JMS Complexity

Some students ask why we need to lookup two object (ConnectionFactory and Destination), and then write so many lines of boilerplate code. Why looking up the Destination is not enough? The reply is simple: that's the way the JMS specification has been made. We could imagine that in a further version, JMS API provides a simplified procedure for connecting to a queue, that suits simple needs.

Source

```
Context context = new InitialContext();
ConnectionFactory cFactory =
  (ConnectionFactory) context.lookup("ConnectionFactory");
Destination destination =
  (Destination) context.lookup("queue/TaxQ");
connection = cFactory.createConnection();
session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
mProducer = session.createProducer(destination);
```

Note that EJB shields us partially from that complexity. On the server-side, message driven beans are configured just with the remote name of the ConnectionFactory and Destination. We don't need to write the JMS code to get a message from the queue.

Source

```
@MessageDriven(activationConfig = {
       @ActivationConfigProperty(propertyName = "destinationType",
              propertyValue = "javax.jms.Queue"),
       @ActivationConfigProperty(propertyName = "destination",
              propertyValue = "queue/TaxQ")})
```

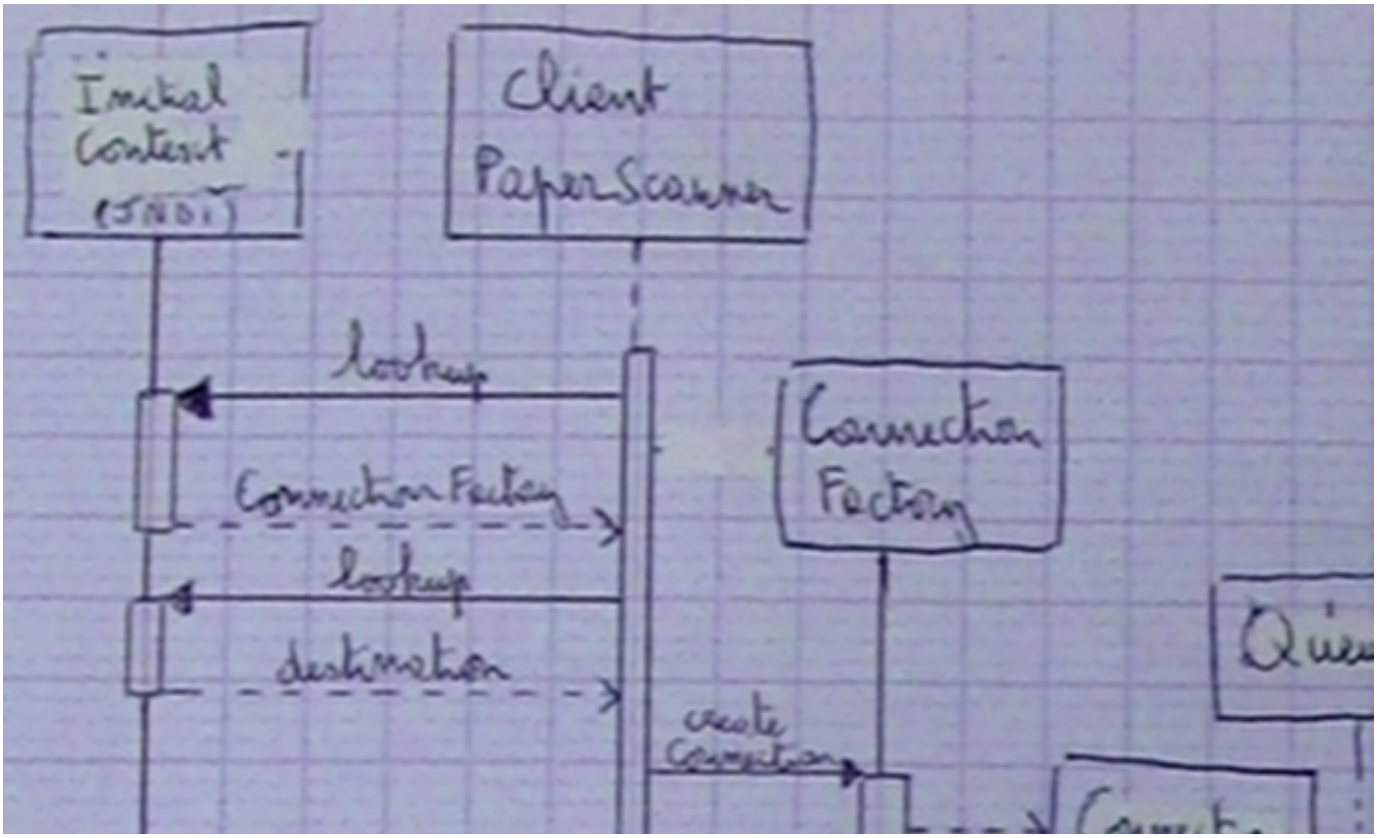
Other frameworks, such as Spring, propose an API to simplify JMS for us.

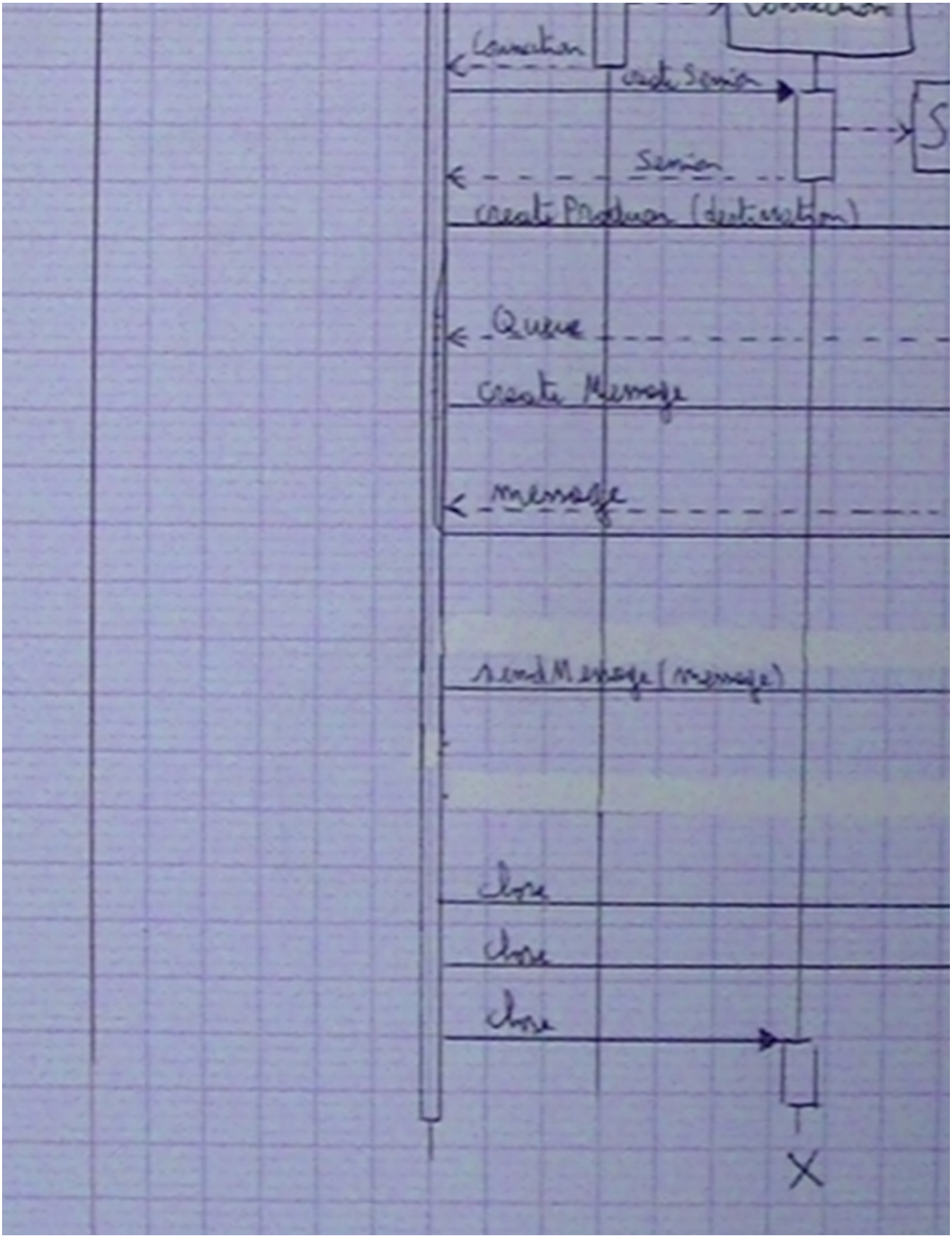
3.2.6. JMS API

In our client code, we used quite a few interfaces from the javax.jms package. Let's review them.

- **Message**: This is the object containing one message. It is created by your client. Your message driven bean's method gets one as parameter.
- **Session**: It is your factory for instantiating new messages.
- **MessageProducer**: It is the object through which you send messages to the queue/topic.
- **Destination**: It represents your queue or topic. Queue and Topic are two child interfaces of Destination (not explicitely used in our code).
- **Connection**: This is you connection with a specific JMS server. It the equivalent to a JDBC connection to a DB server. It typically an open TCP/IP socket between the client and the messaging server. With that connection, you could connect many queues. Usually in you client code, you only want to connect one, which makes the need for that object and it's factory, less obvious in that case.
- **ConnectionFactory**: It is your way to instantiate connections. It encapsulates a set of connection configuration parameters that has been defined by an administrator. Note that these configuration parameters are set and stored on your messaging server.
  It is rare that a client needs multiple connections to the same messaging server. We could imagine a scenario where the client wants to cut the conection off for a while, then reconnects the JMS server, using the same ConnectionFactory.

The following UML sequence diagram shows what happen when executing the client code.





3.2.7. Message Structure

A JMS Message container 3 parts:

- **Header**: automatically filled by JMS, it contains a message identifier. For a reply message, it may also contain the identifier of the initial message.
- **Properties**: application-defined values. It's a kind of header that you can customize. It is sometimes used to filter messages. Some consumers may be interested into specific kind of messages. In our example, PaperScanner could send two kinds of tax return messages on the queue: tax returns for regular employees and tax returns for CEOs. Our TaxProcessor message driven bean may not be interested in CEO tax returns (which will be consumed by another consumer that we would have setup). In that example, we could have a *CEO* boolean property in each message.
- **Body**: content of the message. In our example it was text, but according to the kind of content, a specific sub-interface of Message would be used:
  - TextMessage: contains text (maybe XML),
  - MapMessage: contains key-value pairs,
  - BytesMessage: contains raw bytes (maybe an image),
  - StreamMessage: contains a stream of primitive values,
  - ObjectMessage: contains a serialized graph of objects.

We can print a message to the standard output with:

Source

```
System.out.println("Message: " + msg);
```

The format in which the message contents appear is implementation-specific. In the GlassFish Server, the message format looks like this:

Source

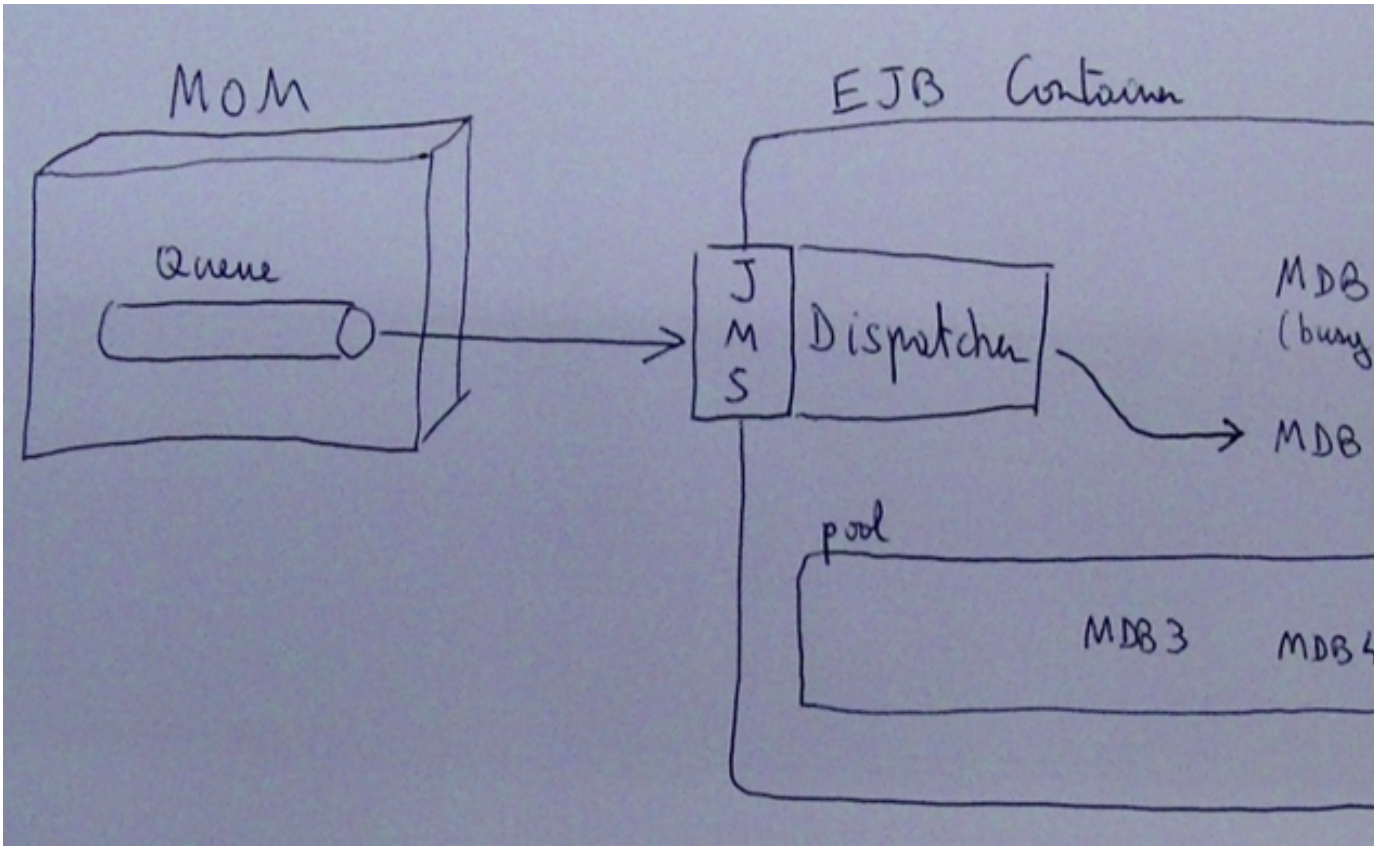
```
Message contents:
Text: This is message 3 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID(): ID:14-128.149.71.199(f9:86:a2:d5:46:9b)-40814-1255980521747
getJMSTimestamp(): 1129061034355
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

3.3. MDB Configuration

While most configuration is done at the MOM level (and is product specific), there are some parameters that you can configure at the EJB container level. In this topic, you will first learn how pooling for message driven bean works, and how you configure it. Then you will see how to associate a bean to a queue or topic and what parameter you can specify.

3.3.1. Pooling

In the EJB container architecture, the message driven bean is not directly connected to the message server. The EJB container connects to the message server and gives the message to the bean. The processing of a message can take some time and involves some I/O (as DB access which are not immediate). If the bean is still busy to handle a previous message when a new message arrives, the container can instantiate another bean (of the same class) to handle the new message. Two beans would be active simultaneously for message processing. It is up to the container to decide if it starts additional bean, and that decision is based on the current workload of the server (as free memory available and CPU load). It is a way to acheive load balancing, and a good reason to use an EJB server to consume messages (which could alternatively be consumed by a regular Java SE application). Beans that finished processing a messages are placed in a pool and reused when further messages arrive.



In most cases you just let the container decides on the pool size according to available resources. Setting MDB pool sizes manually is not standardized and is provider specific.

This is a JBoss's example using annotations:

Source

```
@MessageDriven(
   @Pool(value = PoolDefaults.POOL_IMPLEMENTATION_STRICTMAX, maxSize = 100, timeou
)
```

In Weblogic we must to configure WebLogic-specific behaviors for the MDB in the message-driven-descriptor element of weblogic-ejb-jar.xml. For example:

Source

```
<weblogic-ejb-jar>
 <weblogic-enterprise-bean>
   <ejb-name>ExampleMDB</ejb-name>
   <message-driven-descriptor>
       <pool>
           <max-beans-in-free-pool>500</max-beans-in-free-pool>
           <initial-beans-in-free-pool>250</initial-beans-in-free-pool>
       </pool>
   </message-driven-descriptor>
 </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

3.3.2. Configuration

We did annotate our message driven bean class with annotations from the javax.ejb package.

Source

```
@MessageDriven(activationConfig = {
       @ActivationConfigProperty(propertyName = "destinationType",
               propertyValue = "javax.jms.Queue"),
       @ActivationConfigProperty(propertyName = "destination",
               propertyValue = "queue/TaxQ")})
```

The @MessageDriven annotation marks the class as a message driven bean (instead of a session bean). It takes an array of @ActivationConfigProperty as attribute. Each @ActivationConfigProperty is a key-value pair. A property is a configuration point that has a name (such as "destinationType") and takes a value (such as "javax.jms.Queue"). The main properties are:

- **destinationType**: The two possible JMS destinations are *javax.jms.Queue* and *javax.jms.Topic*
- **destination**: JNDI name for the queue or topic.
- **messageSelector**: Conditional expression (String) to filter messages based on message's properties values
- **acknowledgeMode**: Either *Auto-acknowledge* or *Dups-ok-acknowledge*. It specifies when the EJB container will acknowledge the message server that the message has well be received (and should not be re-send). In most cases, the default value *Auto-acknowledge* is used, and within a transaction managed by the container, means: the message will be acknoladge if and when the transaction commits.
- **subsriptionDurability**: Either *Durable* or *NonDurable*. It is used with topics. Durable means that if the EJB container is not available (failure, shutdown,...) when the message server notifies the topic listeners that a new message arrives, then the message is kept in the queue and will be consumed wuen the EJB container comes back.

4. Deployment

After you developed your bean, you need to know how to deploy it. This lesson will guide you through the deployment process of the Enterprise Java Beans.

To deploy your beans, you may need a deployment descriptor. As it is optional we have not covered it yet. It is covered in the first topic of this lesson.

You will also need to provide your project files in a way that is understood by the EJB container. You will do that in the packaging topic with the creation or EAR and JAR files.

Finally, in the last topic, you will learn how to embed an EJB Lite container into your Java application instead of deploying your application in a server.

4.1. Deployment Descriptor

In this course, you mainly use annotations to provide (meta-)data about our beans, interceptors, the resource they use, transactions and security. Before EJB v3.0, these meta-data were described in a deployment descriptor file named ejb-jar.xml. Since EJB v3.0 you have the choice between using annotations, the deployment descritor or both.

4.1.1. Example

The following deployment descriptor defines our AuditorBean as singleton through the `<session>` tag. `<ejb-name>` contains the JNDI name of the EJB. `<remote>` contains the remote interface fully qulalified name if any and the `<ejb-class>` contains the bean class name. `<session-type>` contains the type (Singleton, Stateless or Stateful) of session bean.

Source

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1"
          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_1.xsd">
 <enterprise-beans>
   <session>
      <ejb-name>AuditorBean</ejb-name>
      <remote>com.example.banking.Auditor</remote>
      <ejb-class>com.example.banking.AuditorBean</ejb-class>
      <session-type>Singleton</session-type>
   </session>
 </enterprise-beans>
</ejb-jar>
```

We did not need the `<session>` element so far because we defined AuditorBean through the @Singleton annotation. And if we defined more beans through the deployement descriptor we would have multiple `<session>` elements.

4.1.2. Deployment Descriptor Elements

For any EJB annotation, there is a corresponding XML element in the deployment descriptor and the mapping is fairly trivial.

This table maps the annotations you have seen, or will see in this course and the corresponding deployment descriptor xml element.

Annotation	Deployment Descriptor Element	Description

@Stateless	<session-type>Stateless <ejb-name>	Define a stateless session bean.
@Stateful	<session-type>Stateful <ejb-name>	Define a stateful session bean.
@MessageDriven	<message-driven> <ejb-name>	Define a message driven bean.
@Remote	<remote>	Identify a remote interface.
@Local	<local>	Identify a local interface.
@PostConstruct	<post-construct>	Identify a call-back method to be called after a bean construction an dependency injection.
@PreDestroy	<pre-destroy>	Identify a call-back method to be called before the bean destruction.
@PostActivate	<post-activate>	Identify a call-back method of a stateful session bean to be called after it has been activated (after passivation).
@PrePassivate	<pre-passivate<>	Identify a call-back method of a stateful session bean to be called before it is passivated to secondary storage.
@Resource	<resource-ref> <resource-env-ref> <message-destination-ref> <env-ref>	Makes the value of an attribute injected by the container.
@EJB	<ejb-ref> <ejb-local-ref>	Make a bean reference (object attribute) injected by the container.
@Interceptors	<interceptor-binding> <interceptor-class>	Define which interceptor should be applied on a class.
@AroundInvoke	<around-invoke>	Define a method as being an interceptor.
@Transaction-Management	<transaction-type>	Defines the type of transaction control: BMT or CMT.
@Transaction-Attribute	<container-transaction> <trans-attribute>	Defines the transaction propagation attribute (REQUIRED, NEVER, SUPPORTS,...) on a method or class.

4.1.3. Choosing Annotation or Deployment Descriptor

There no need to define deployment descriptor in XML since application container will scan the classpath of your project in order to find the classes annotated as the Enterprise Java Beans. The rule for providing a value through annotation or deployment descriptor is the following. Most developer prefer using annotations by default, except:

- When you use an existing code that you don't own and cannot (don't want to) change.
- For values that could possibly change during deployment (to separate concerns of development/deployment).
- For settings that cannot be set through annotations, as the default interceptor.

If contradictory information is given, the deployment descriptor overrides the annotation value.

4.1.4. Activity: Quiz

The annotation and the deployment descriptor for a bean give condraticatory information for AuditorBean. For example, they provide a differnt JNDI name: "AAA" (annotation) and "DDD" (deployment descriptor).

Source

```
@Singleton("AAA")
public class AuditorBean ...
```

Source

```
   <session>
      <ejb-name>DDD</ejb-name>
      <remote>com.example.banking.Auditor</remote>
      <ejb-class>com.example.banking.AuditorBean</ejb-class>
      <session-type>Singleton</session-type>
   </session>
```

What will happen when the application is deployed?

( ) The EJB container will show an error message because of the conflict between both names.

( ) The JNDI name will be "AAA"

( ) The JNDI name will be "DDD"

Explanation: the deployment descriptor supersedes annotations in case of conflict.

Solution: "DDD"

4.2. Packaging

Enterprise JavaBeans are packaged into a Jar archive. Usually, deploying EJB applications is placing a jar file in a specific directory of your EJB container. That file jar contains your code and additional information. The process of making this jar file is known as *packaging*.

4.2.1. EJB-JAR Files

EJBs are delivered to the EJB container as a jar-file commonly referred as the "EJB-JAR". It contains the compiled class files and the jar dependencies if any. The deployment descriptor (if any) is located in the META-INF sub-folder and is named "ejb-jar.xml".

Our *hello word* project from the first lesson would produce a server-side jar file with two .class files in their package directory:

Source

```
com/example/hello/HelloWorldBean.class
                  HelloWorld.class
```

Below, an example bigger project contains a deployment descriptor and a the jar file of *bankframework*. This file is named banking-ejb.jar

Source

```
META-INF/ejb-jar.xml
        manifest.mf
bankframework.jar
com/example/banking/AuditorBean.class
                    Auditor.class
                    AccountRepositoryBean.class
                    AccountRepository.class
```

The manifest.mf file refers bankframework.jar

Source

```
Manifest-Version: 1.0
Created-By: 1.4.2 (Sun Microsystems Inc.)
Class-Path: bankframework.jar
```

Most IDEs, such as Eclipse, supports creating an EJB-JAR file for you.

4.2.2. EAR Files

Java EE includes both servlet (web) and EJB specifications. In this course, you focused on the EJB container deploying EJBs. Web applications are packaged as WAR files. As we have seen, EJB modules are packaged as JAR files. Both aspects (web and EJB) of a Java EE application can be grouped in EAR (Enterprise ARchive) files.

Below, the structure of an example EAR file:

Source

```
META-INF/application.xml
banking-ejb.jar
banking.war
lib/bankframework.jar
```

The EAR file contains some ejb jar files (if any), such as banking-ejb.jar. It also contains some war files (if any), such as banking.war file. Here, our Java EE application is "only" composed of one ejb-jar file and one war file. The lib folder contains Java SE jar files with classes used by multiple modules. For example, our bankframework.jar contains .class files needed by both the EJB and the web modules. If it was only needed by the EJB module it would have been placed within banking-ejb.jar.

In META-INF, the application.xml file is the EAR deployment descriptor that lists/describes the jar/war module files. It is listed below:

Source

```
<application>
 <module>
   <ejb>banking-ejb.jar</ejb>
 </module>
 <module>
   <web>
     <web-uri>banking.war</web-uri>
     <context-root>/bank</context-root>
   </web>
 </module>
</application>
```

For a web module, the context-root information is the URL fragment to access the web application (such as http://localhost:8080/**bank** in this case).

This application.xml file is optional since Java EE 5. The Java EE server will figure out the role of each jar file without the need for a EAR deployment descriptor.

4.2.3. EJB in WAR File

Note that since EJB 3.1, it is possible to pack an EJB inside a war file. You either

- put your EJB classes with the other web classes in WEB-INF/classes and put the ejb-jar.xml in the WEB-INF directory, or
- put an EJB-JAR file in the WEB-INF/lib directory.

4.2.4. Standard Bean JNDI Name

EJB 3.1 introducted portable JNDI names of the EJB components. All portable names are embedded in the java:global namespace. The beans are mapped to the JNDI according to the following namining convention:

java:global[/<app-name>]/<module-name>/<bean-name>

That means if our AuditorBean is packaged in the ejb.jar file and bankapp.ear file, the application server will map it to the following name:

java:global/bankapp/ejb/AuditorBean

If you inject bean using the @EJB annotation, it tells the container to look up for the bean in the JNDI registry using the naming convention above. Before EJB 3.1 every EJB container product had its own naming conventions. Some bean JNDI name could be written as String in your Java classes or deployment descriptor using that proprietary convention. It made your code not being portable to another product (for example, from JBoss to GlassFish).

4.2.5. Activity: Quiz

What is the name of the file, playing the role of deployment descriptor in a EAR file?

( ) application.xml

( ) applicationContext.xml

( ) ejb-jar.xml

( ) persistence.xml

( ) web.xml

Explanation: applicationContext.xml is a typical name for a configuration file of the Spring framework. ejb-jar.xml is within the JAR file (not EAR) of an EJB application. persistence.xml is the configuration file name for JPA (Java Persistence API). web.xml is the deployment descriptor file name for web applications.

Solution: application.xml

What is the correct assertion?

( ) EJB JAR and WAR files are inside EAR files.

( ) EJB JAR and EAR files are inside WAR files.

( ) EAR and WAR files are inside EJB JAR files.

Solution: EJB JAR and WAR files are inside EAR files.

4.3. EJB Lite

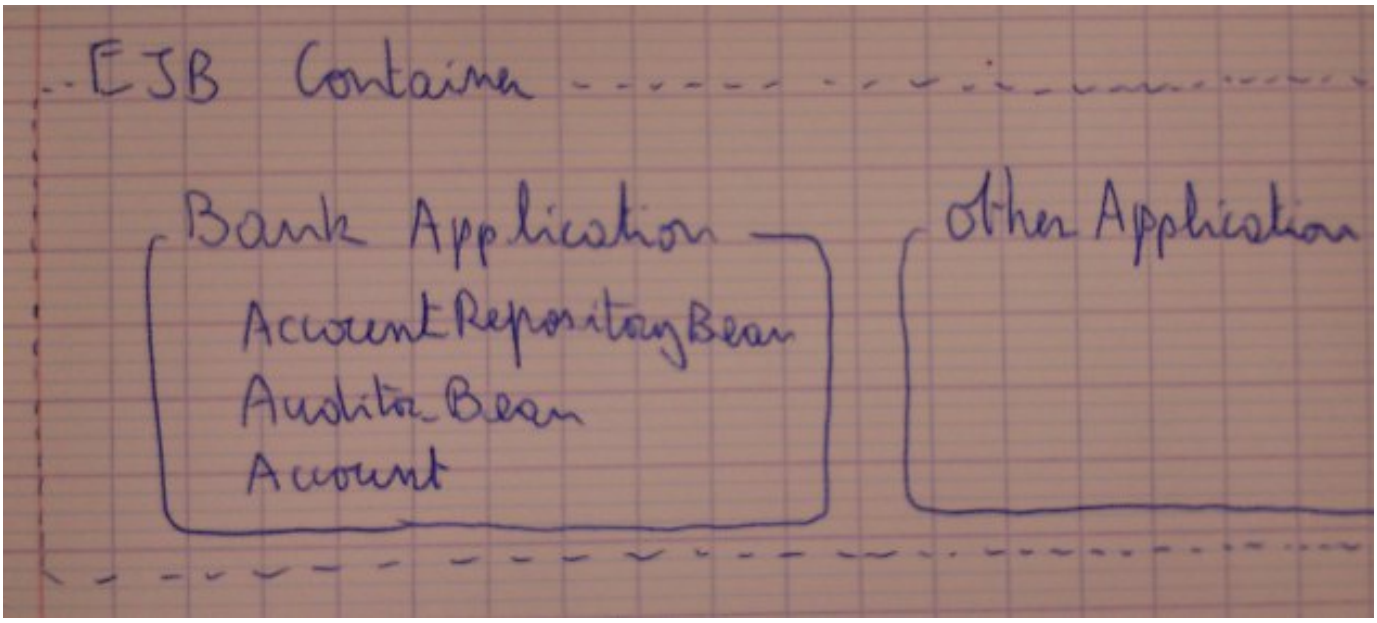
So far, you have deployed your EJB application within a running EJB container. You may also benefit some EJB features from within a regular Java SE application. EJB Lite is a lightweight edition of EJB. In this topic you learn what features it contains and transform your banking EE application to run it into an embedded EJB container.

4.3.1. Embedded Containers

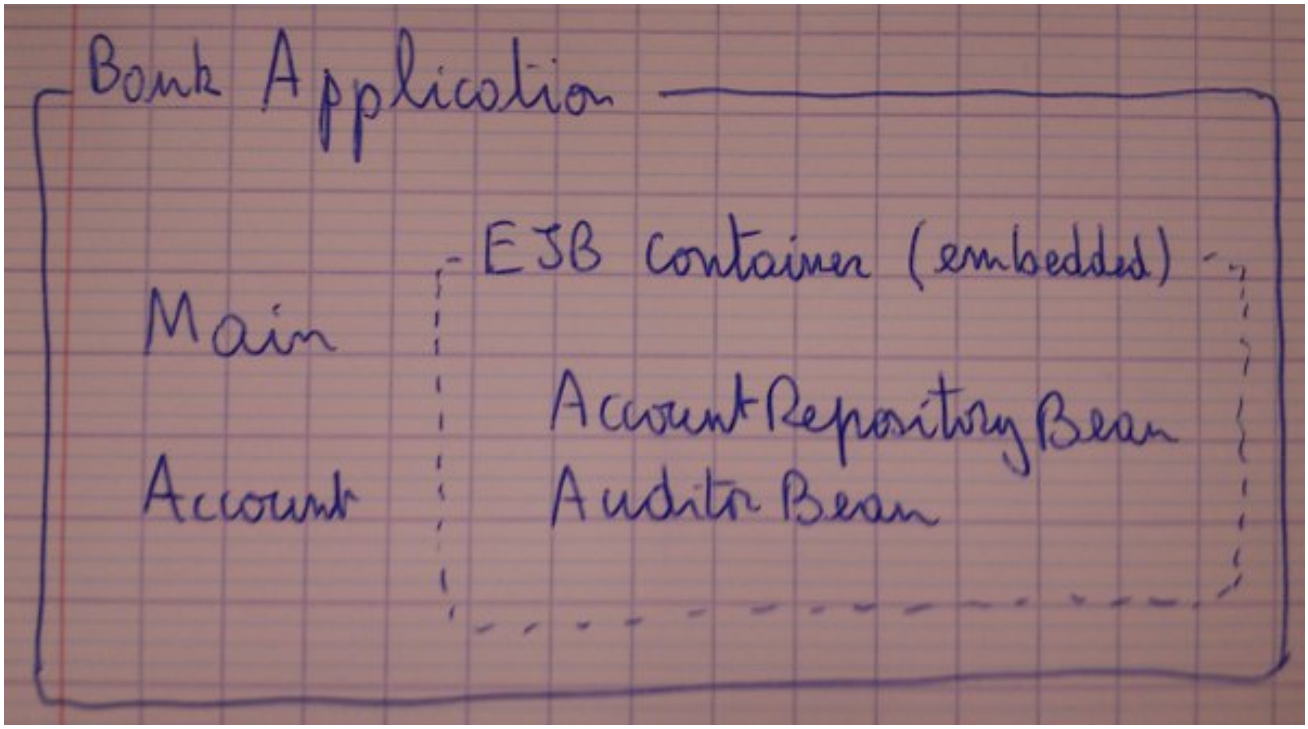
Historically, EJB containers are considered by many developers as *heavyweight*. The simple fact to deploy the application within a running server contributes to that impression. As a reaction, the Spring framework proposed an alternative of embedded (non-EJB) container to manage bean's lifecycle. With Spring, you do not deploy your application in a Spring container. Rather, you simply add a few jar files to your Java SE application classpath and start an usual main method.

Since the version 3.1, the EJB specification proposes that way of doing with *EJB Lite*. The main characteristic of EJB Lite is to be an *embeddable EJB container* to be places in your Java SE application (starting with a main() method). Additionnally, EJB Lite restricts the features you have access to.

The diagram below shows a very conceptual view of a full (not lite) EJB application within the EJB container. Every class is in the EJB container.



The diagram below shows an EJB lite application. Only beans can be considered as being in the EJB container, which itself is embedded into your application.



4.3.2. Subset of Features

Many applications do not need the full functionnalities of the EJB specification, but only a subset. EJB Lite is centered around the session bean component model. Note that EJB Lite is not a product or an implementation. GlassFish has jar files that implement the EJB Lite specification.

EJB Lite includes these features:

- Embeddable API
- Session beans (stateless, stateful, and singleton).
- Local EJB interfaces or no interfaces.
- Dependency Injection
- Interceptors.
- Transactions (container-managed and bean-managed).
- Security (declarative and programmatic).

EJB Lite excludes these features:

- Message driven beans.
- Asynchronous processing and scheduling.
- Remoting (web services and RMI)
- Backward compatibility with EJB 2 and 1.

4.3.3. Using the EJBContainer Class

With EJB Lite, we can use any session bean, such as our HelloWorldBean:

Source

```
@Singleton("ejb/HelloWorldBean")
public class HelloWorldBean {
   public String sayHello(String name) {
       return "Hello " + name;
   }
}
```

The Main class below will start an embedded EJB container via the javax.ejb.embeddable.EJBContainer class. For those who know the Spring framework, it works similarily, EJBContainer being the ApplicationContext. The first bean is obtained via a JNDI lookup, but it is not a remote bean running on a server. It is a local instance and HelloWorldBean implements no remote interface. HelloWorldBean could easily obtain references to other beans through dependency injection (@EJB).

Source

```
import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import javax.naming.NamingException;

public class Main {
  public static void main(String args[]) throws NamingException {
      EJBContainer container = null;
      try {
          container = EJBContainer.createEJBContainer();
          Context namingContext = container.getContext();
          HelloWorldBean helloWorldBean = (HelloWorldBean) namingContext.lookup(
                  "ejb/HelloWorldBean");
          System.out.println( helloWorldBean.sayHello("World!) );
      } finally {
          if(container != null) {
              container.close();
          }
      }
 }
}
```

In the code above, we obtain an instance to an EJBContainer via the static factory method EJBContainer.createEJBContainer(). At the end, we close it with container.close(). The JNDI Context is obtained through the EJBContainer instance.

4.3.4. EJB Lite Jar File

To use classes such as javax.ejb.embeddable.EJBContainer in your Java SE application, you need to provide the jar file in which they are defined. The jar file depends on the product you are using. For glassfish it is:

Source

```
$GLASSFISH_HOME/lib/embedded/glassfish-embedded-static-shell.jar
```

4.3.5. Activity : Embedding the Banking Application

Your project manager plans to run part of the banking application within a Java SE process. He would like to reuse some business logic in a Java SE application with a Swing UI (non web fat client). You have been asked to program of prototype of your EJBs running in an embedded EJB container and test that features such as dependency injection still work as expected.

1. Create a new Java SE project with a Main class and a main() method.

2. Test your main() method
3. Change your build path to add the following jar file. $GLASSFISH_HOME/lib/embedded/glassfish-embedded-static-shell.jar. In Eclipse, select Project > properties > Java Build Path > Libraries > Add External JARs... In the dialog box, start from the GlassFish installation directory, go down to lib/embedded and get the jar file.
4. Recreate the java code example of this topic with HelloWorldBean: add the annotated HelloWorldBean class and fill your existing main method. Execute your main method to test it.
5. Copy AuditorBean and AccountRepositoryBean from a previous EJB project, into your Java SE project. Do not take any remote interface along. AuditorBean has a field @EJB AccountRepositoryBean.
6. Modify your main method to lookup AuditorBean (not the remote interface) and call a method on it.
7. Test your code by executing the main method again.

5. Dependency injection

You have seen all kinds of beans in the previous lessons. In business applications these beans are usually bound to each other. In this lesson you will learn how to use dependency injection to bind EJBs together easily. You will turn AccountRepository into a singleton bean and inject it into AuditorBean.

You can also inject configuration values into beans, such as remote server addresses or passwords: anything that you prefer separated from the source code.

5.1. Injecting Environement Entries

Some constant values should be placed out of your source code, because their value need to be defined by those who will deploy your application, not by programmers. Values that are not the same for development, test and production environments are typical examples. Data to access other systems, such as a user name, a password or an IP address need to be configurable without changing/recompiling the source code.

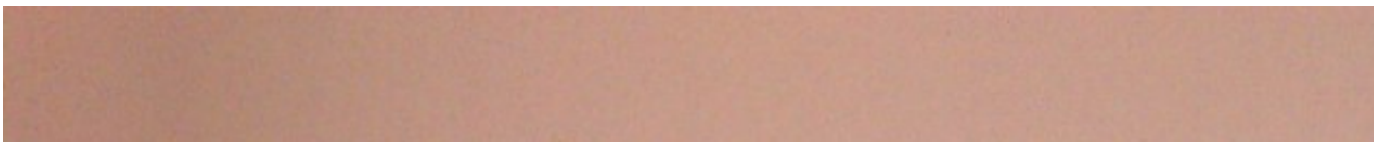
In this topic, you will create a new bean with an ip address and externalize the value of that ip address into the deployement descriptor.

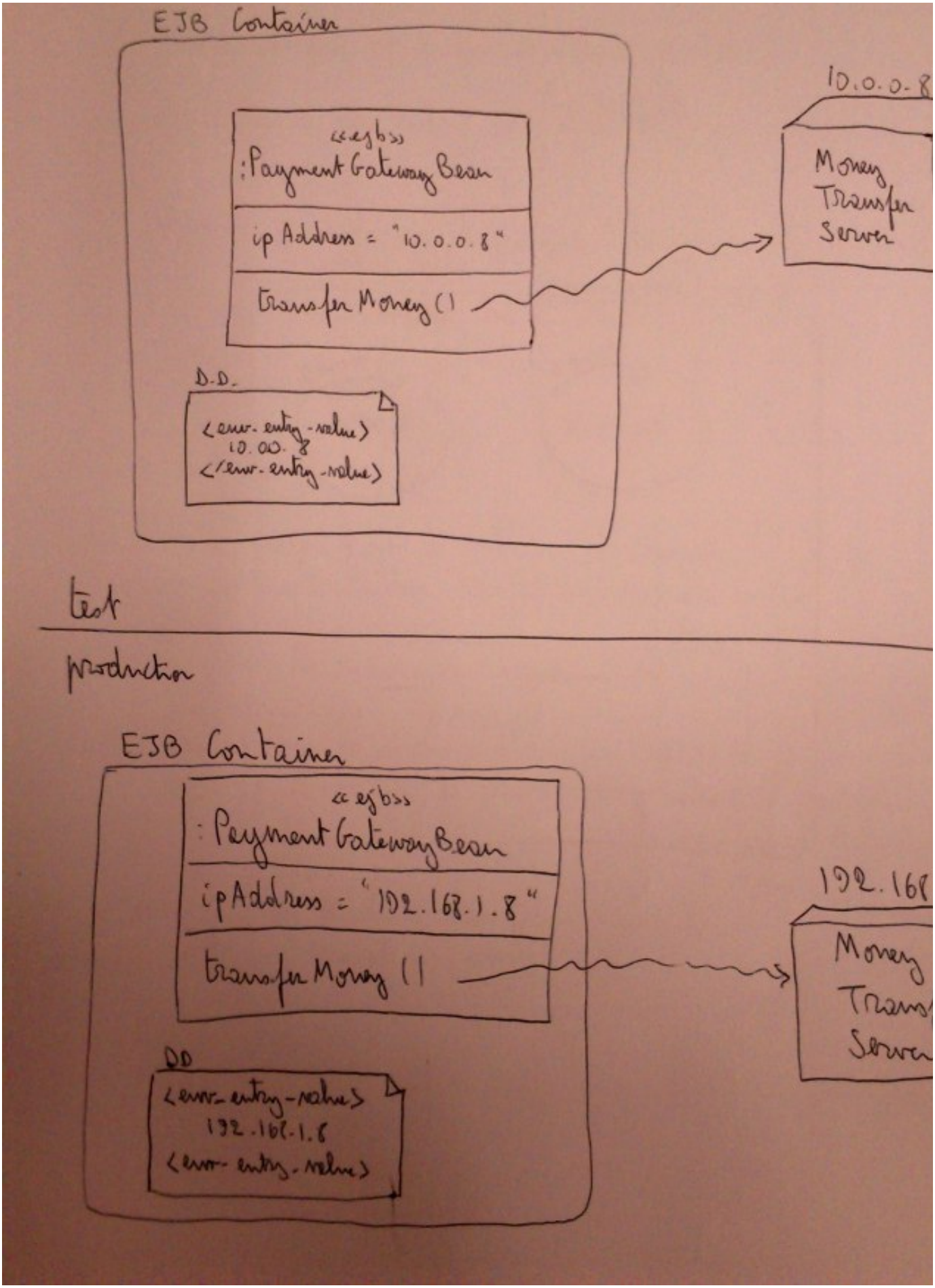
5.1.1. Value Injection

Injecting environment entries is making the EJB container initialize attributes from values specified outside your Java code, in the XML deployment descriptor. This separation enables these values to be changed without changing the Java code, probably after the compilation and before the deployment.

5.1.2. Scenario With Multiple Environements

Our banking application needs to issue payements with other banks. A money transfer is not performed by our application but by an external server accessible through a proprietary/legacy mechanism. Our application contains the PaymentGatewayBean class responsible to access the legacy server. PaymentGatewayBean needs to know the IP address of that server. But the server is not the same for a test environement and a production environement. Indeed, during tests, we don't issue payement orders on the production money transfer server.





In the diagram above, you see two environment: test and production. Our application first runs in the test environment where it is used by testing people. Once a version of our application is tested and ready, it is deployed in the production environment.

The top part is the test environement with a test money transfer server at the address 10.0.0.8. In the bottom part, the production money transfer server is at the address 198.168.1.8.

5.1.3. Injecting an IP Addess in PaymentGatewayBean

In the PaymentGatewayBean, we declare an attribute `String ipAddress`, that will contain the ipAddress to the money transfer server, and that our code can manipulate to open a connection to it.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class PaymentGatewayBean {
   @Resource(name="ipAddress")
   private String ipAddress;

   ...
}
```

That attribute is annotated with javax.annotation.Resource, with a symbolic name as parameter. That symbolic name "ipAddress", is the link with the value in the deployement descriptor. The @Resource annotation, when applied on a method or attribute, indicates that the EJB container should inject the specified resource into the bean when it is initialized.

5.1.4. Value in Deployement Descriptor

Values are not provided in the source code, because the purpose of the system is to externalize the value from the source code. Values are provided in the deployment descriptor in `<env-entry>` elements.

In the deployement descriptor below, within the bean declaration element `<session>`, we specify:

- The symbolic name associated in the value: "ipAddress". This is the same name as the annotation attribute: @Resource(name="ipAddress"). It does not have to be the same name as the Java attribute PaymentGatewayBean.ipAddress.
- The type of value. This can be any primitive type or String.
- The value.

Source

```
<ejb-jar ...>
<enterprise-beans>
 <session>
   <ejb-name>PaymentGatewayBean</ejb-name>
   <env-entry>
      <env-entry-name>ipAddress</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>10.0.0.8</env-entry-value>
   </env-entry>
 </session>
</enterprise-beans>
</ejb-jar>
```

5.1.5. Activity: Create PaymentGatewayBean

Create and test the PaymentGatewayBean described in this topic.

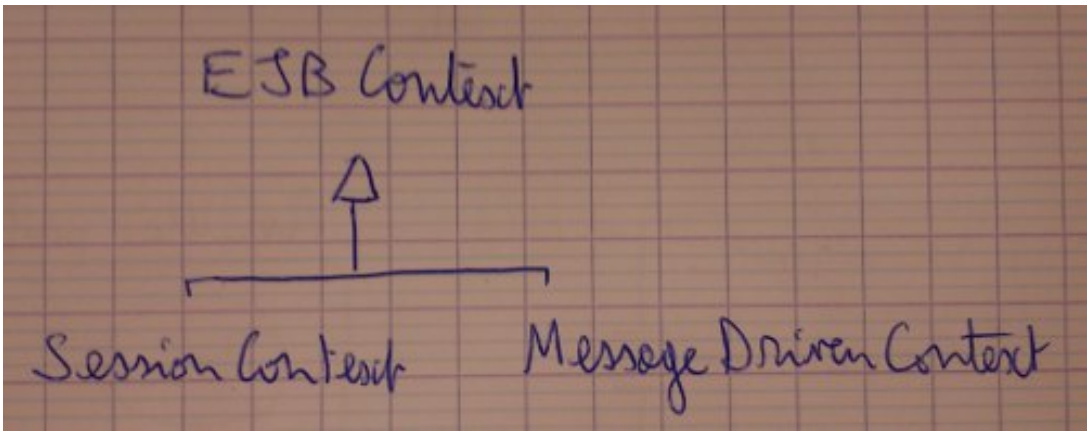
1. Start from any working version of your banking application, for example the solution of the first version with AuditorBean.
2. Create the bean class with the appropriate annotations.
3. Create the deployment descriptor file if it does not exist yet, and add the appropriate `<env-entry>` element to it.
4. Add a transferMoney(long amount) method to PaymentGatewayBean class. That fake method only prints a message with the ipAddress of the server at the console, such as "Sending $100 through server 10.0.0.8".
5. Create a client to test calling your PaymentGatewayBean.transferMoney() method.

5.2. Injecting EJBContext

EJBContext is an interface which provides access to some container managed resources about your bean. Sometimes, you may want to call an API about your bean. It is the case for programmatic transactions, programmatic timers and programmatic security covered in the next lessons.

5.2.1. Class Hierarchy

EJBContext is a common ancestor for SessionContext and MessageDrivenContext.



MessageDrivenContext defines no additional method. SessionContext define a few additional methods which are rarely used. The useful methods are defined at the ancestor level and are explained in their respective lessons:

method	category	description
getCallerPrincipal	Security	Returns the user name associated with the current thread.
isCallerInRole()	Security	Test if the current user is associated with the given role.
getRollbackOnly()	Transaction	test if the current transaction has been marked for rollback only.
setRollbackOnly()	Transaction	Mark the current transaction for rollback.
getUserTransaction()	Transaction	Returns the UserTransaction interface to set a transaction as begun, committed and rolled back.

The EJBContext provides a couple of other methods that are out of scope for this course.

5.2.2. Injection

The @Resource annotation enables your bean to get a reference to it's EJB context very easily, thanks to dependency injection.

Source

```
@Singleton
public class AuditorBean {
   @Resource
   SessionContext sessionContext;

...
}
```

5.3. Injecting Beans

As your application grows, it contains more and more beans. Some beans need other beans and need references to them. The @EJB annotation will help you to make the EJB container inject bean references to wire up a graph of beans.

For example, in our banking application, we will turn AccountRepository into a bean and inject it into AuditorBean. That way, AccountRepository will benefit the services from the EJB container, as all other beans.

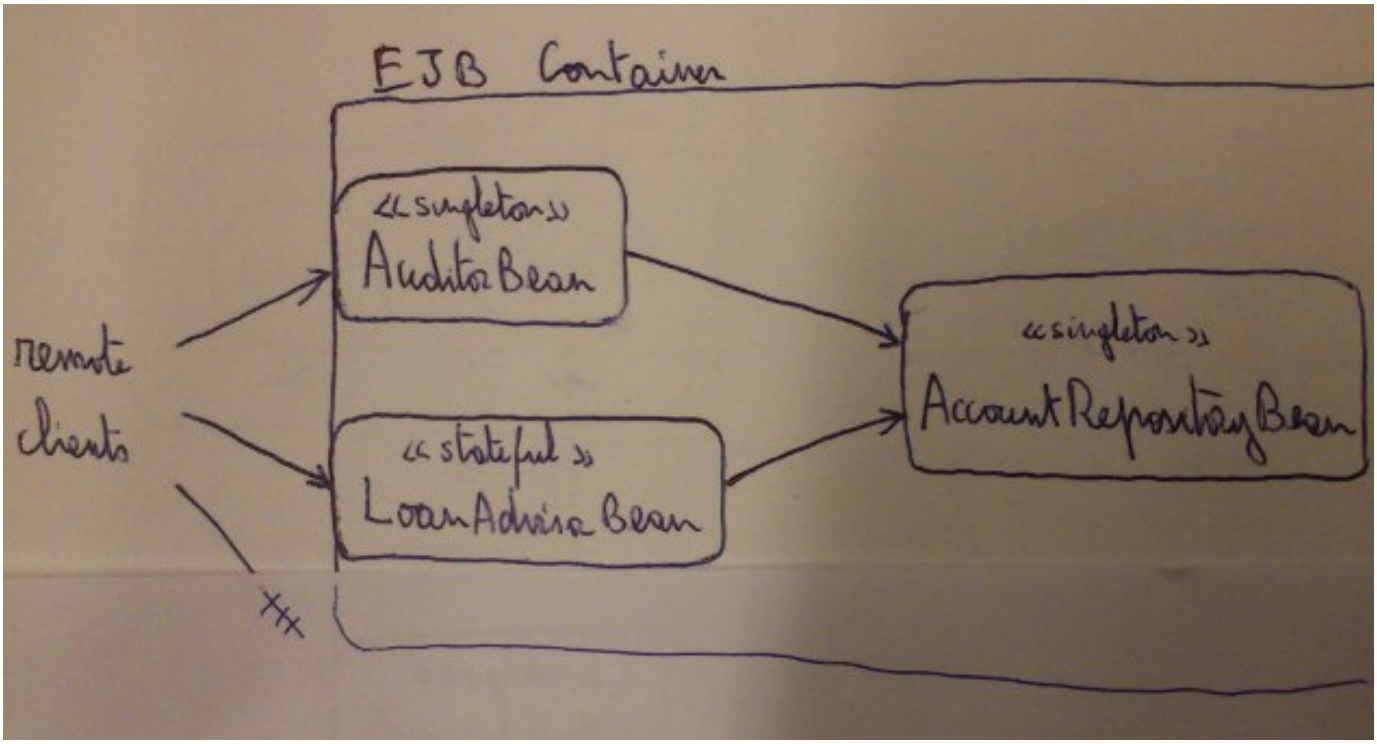
5.3.1. Defining Bean Dependency Injection

Dependency injection is a design pattern describing how graph of objects are wired together. Dependency injection frameworks (such as the Spring framework, a competitor or EJBs) provide some mechanisms to configure dependencies between the objects.

EJB 3.1 comes with its own dependency injection framework. It is designed to make integration of JNDI resources easier.

5.3.2. Turning AccountRepository Into a Bean

So far, we have used a non-EJB java class, AccountRepository, to store a map of accounts in memory. It would make sense for AccountRepository to be a bean too, in order to benefit services from the EJB container. It probably does not need to be accessed remotely, so it does not need a remote interface.



Non-EJB version:

```
public class AccountRepository {
    ...
}
```

EJB version:

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountRepositoryBean {
    ...
}
```

For naming conventions, we renamed the class *AccountRepositoryBean*.

5.3.3. Wirering AuditorBean and AccountRepositoryBean

Currently, AuditorBean instantiates the AccountRepository class:

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean {

    AccountRepository accountRepository = new AccountRepository();

    ...
}
```

You should not use the *new* operator to get an instance of AccountRepositoryBean because it's forbidden by the EJB specification to directly instantiate beans. Only the EJB container can instantiate EJBs. AuditorBean could do a JNDI lookup to get a reference to AccountRepositoryBean, but it would take several lines and add boilerplate technical code to our AuditorBean. JNDI calls from within the EJB container were the common way for getting beans before the v3.0 of the specification.

Asking the EJB container to wire the beans together is a much more elegant solution. The EJB container will inject the instance of AccountRepositoryBean (the dependency) into AuditorBean. This is dependency injection. To make that happen, annotate the reference AuditorBean.accountRepository with @EJB.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean {

   @EJB
   AccountRepositoryBean accountRepository;

   ...
}
```

When the EJB container will instantiate AuditorBean, it will detect the @EJB annotation and also assign the attribute AuditorBean.accountRepository with the reference to the AccountRepositoryBean.

5.3.4. Injecting an EntityManager

If you use JPA for persisting your data to a relational DB, you will need a reference to an EntityManager. The EJB specification provides a special annotation for injecting that: @PersistanceContext.

If AccountRepositoryBean was using JPA, you would inject an EntityManager like this:

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountRepositoryBean {

   @PersistenceContext
   EntityManager em;

   ...
}
```

5.3.5. Activity: AccountRepositoryBean

It is time to turn AccountRepository into a bean. It will finalize the EJBification of our banking application.

1. Start from any working EJB version of the banking application, such as the first version with AuditorBean.
2. Modify the AccountRepository class as shown in this topic. Rename it an annotate it to turn it into an EJB.
3. Modify the way AuditorBean obtain a reference to AccountRepositoryBean. Use dependency injection with the @EJB annotation.
4. If the version of the project you started from include the LoanAdvisorBean, modify the way it obtains its AccountRepositoryBean reference to use dependency injection. Note that now, AuditorBean and LoanAdvisorBean will share the same singleton AccountRepositoryBean instance. It was not the case when

each of them created their instance.

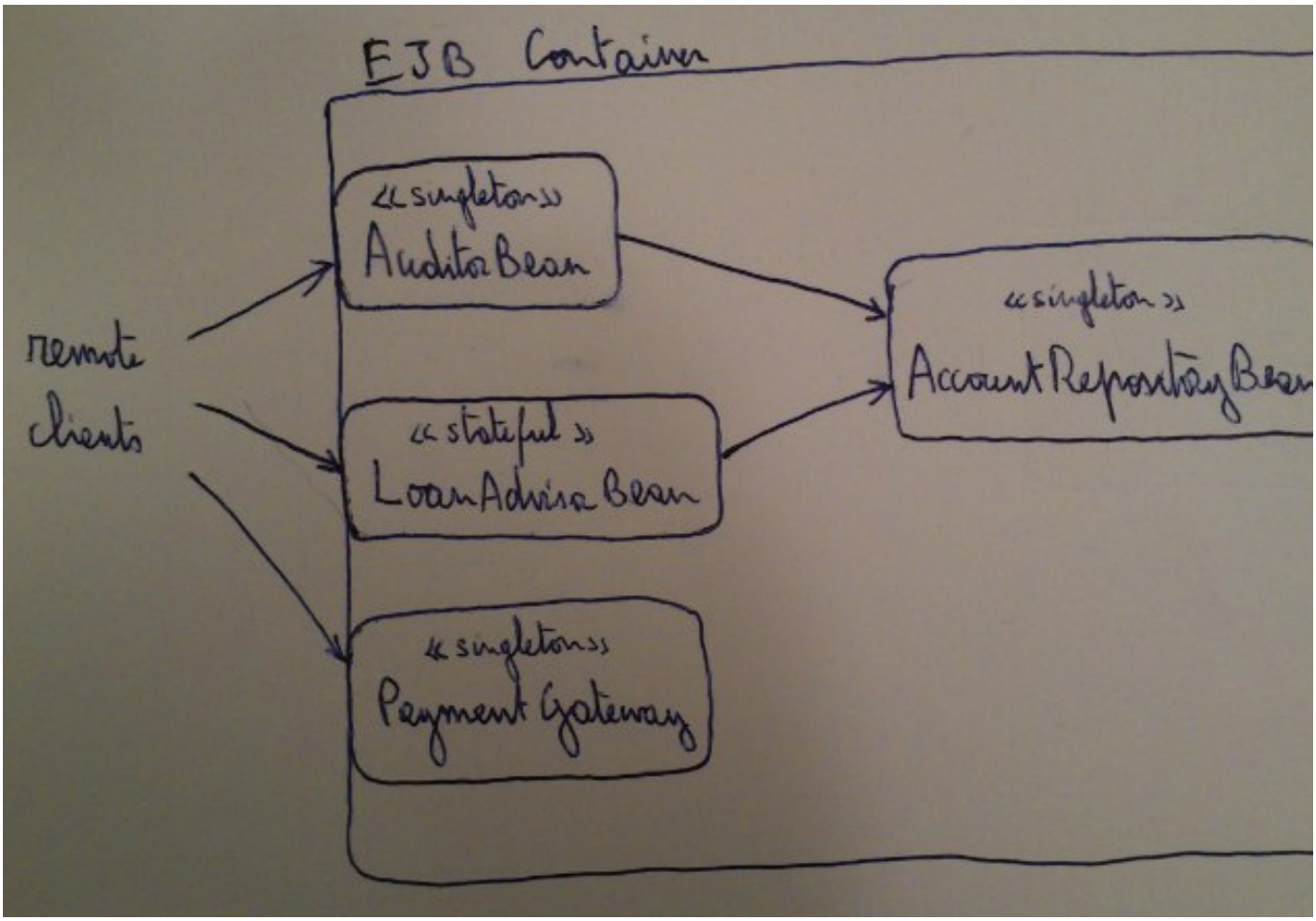
5. Test your project.

5.4. Interfaces

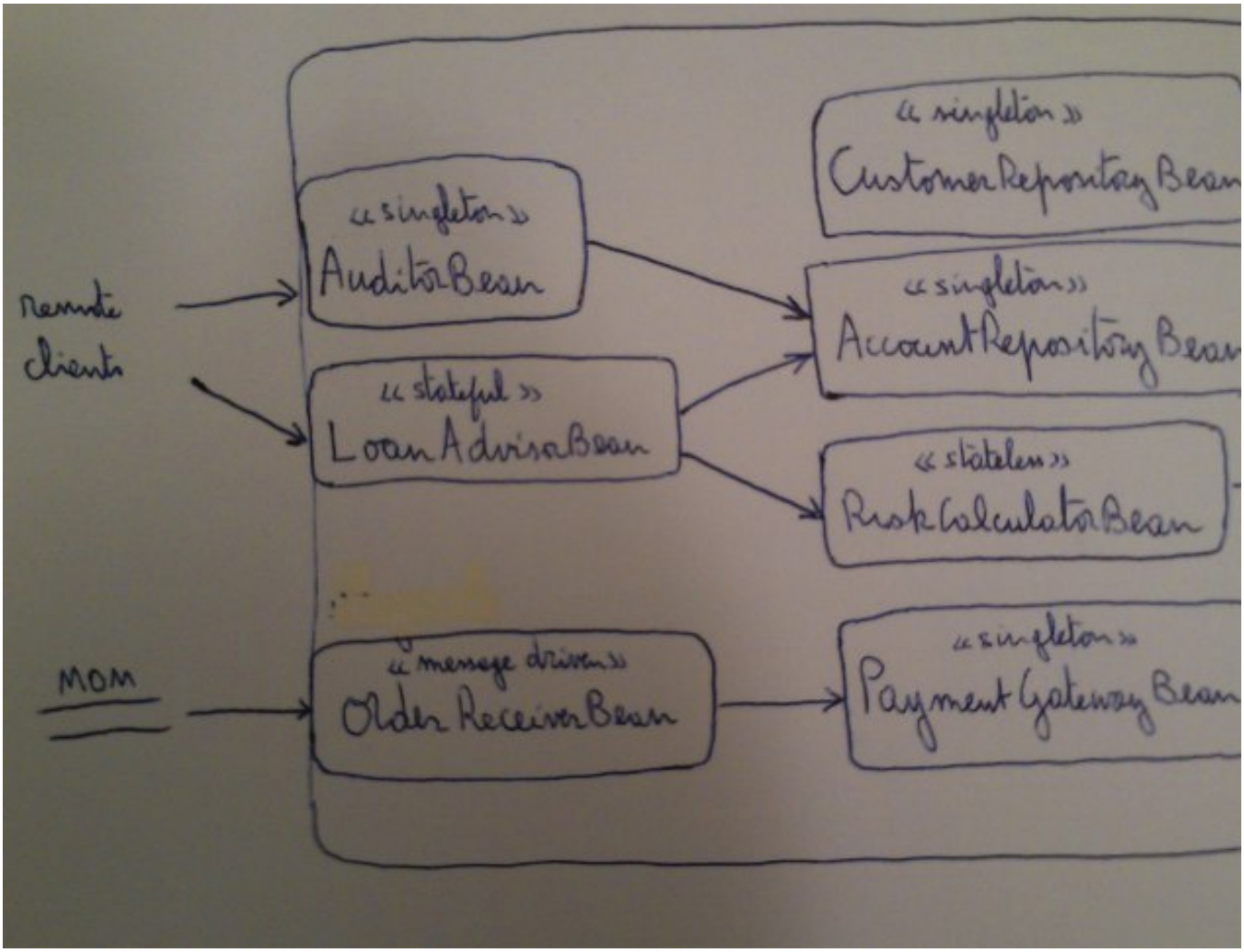
Since the first version of the EJB specification it is mandatory to develop Java interfaces for every EJB class. It changed only since EJB 3.1. Now it is possible to define bean with no interface, which makes your code simpler. In this topic, we review the notion of remote interface, local interface and no-interface bean.

5.4.1. Graph and Boundaries

Beans are often bound to each other and they make a graph. The diagram below shows an EJB dependency graph for our banking application. Real business applications have many many more beans and the graph is much bigger.



For example, the following diagram adds beans to our banking application to make a bigger graph.



In this graph, some beans are accessible to remote clients, and some are only accessed by other beans within the EJB container. For example, AuditorBean and LoanAdvisorBean are accessible by remote clients. AccountRepositoryBean is only accessed by other beans.

5.4.2. Remote Interface

Beans that need to be access by remote client must implement a remote interface. That interface contains the signature of the methods used by the remote client, and is annotated with @Remote.

Source

```
@Remote
interface Auditor {
   Account checkAccountStatus(String userName);
}
```

5.4.3. Remote Interface From a Local Client

In the first version the EJB specification, the only way for a bean to access another bean was to get its remote interface. Going through the remote interface as a remote client would do, was a performance issue. The following diagram shows AuditorBean accessed by OtherBean within the EJB container through a remote interface. Going through the stub, network and skeleton is useless and is a performance overhead.

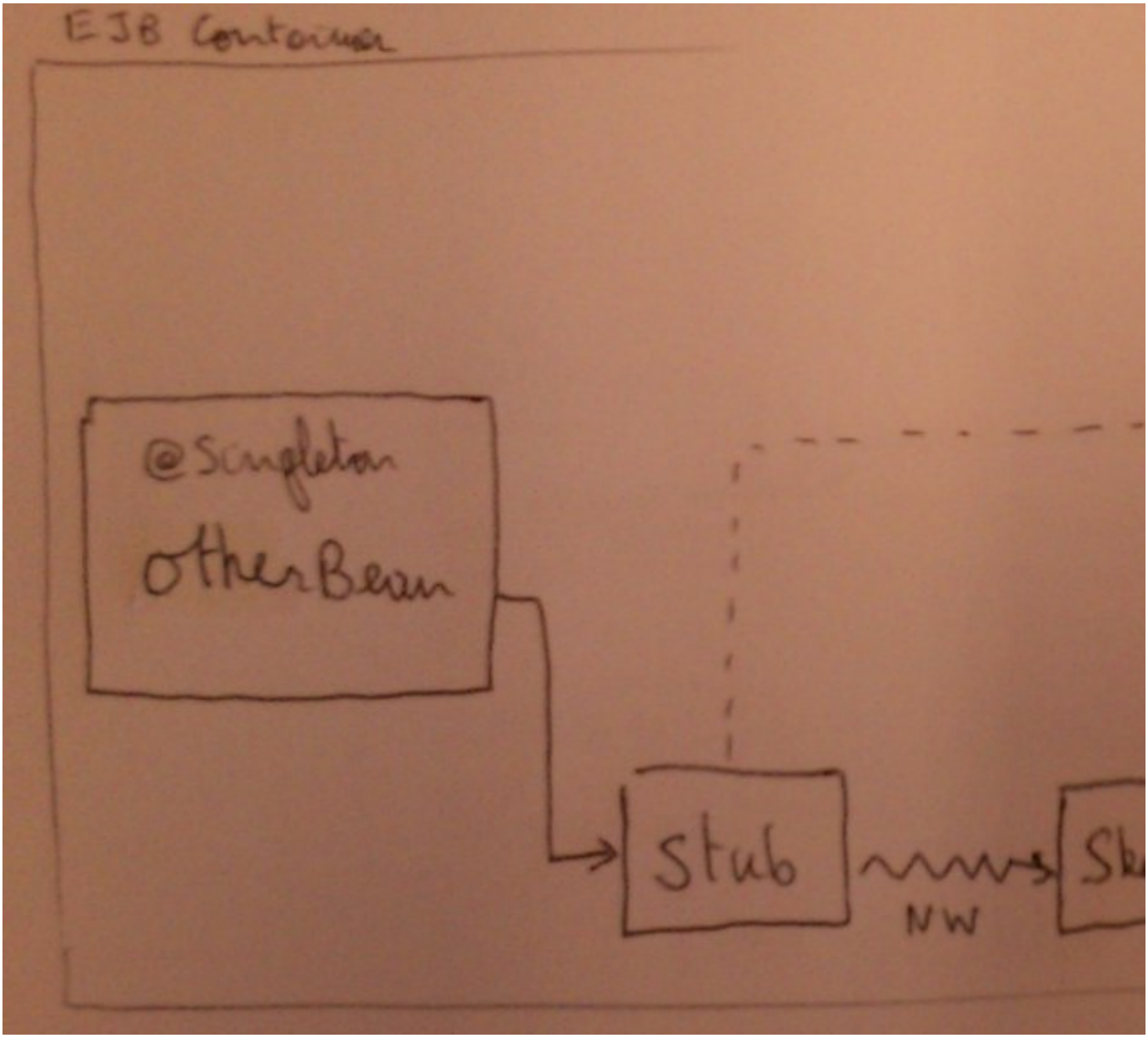
This is the code you know from previous lessons with AuditorBean and its remote interface.

Source

```
@Remote
interface AuditorRemote {
   Account checkAccountStatus(String userName);
}
```

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean implements AuditorRemote {
   ....
}
```



5.4.4. Local Interface

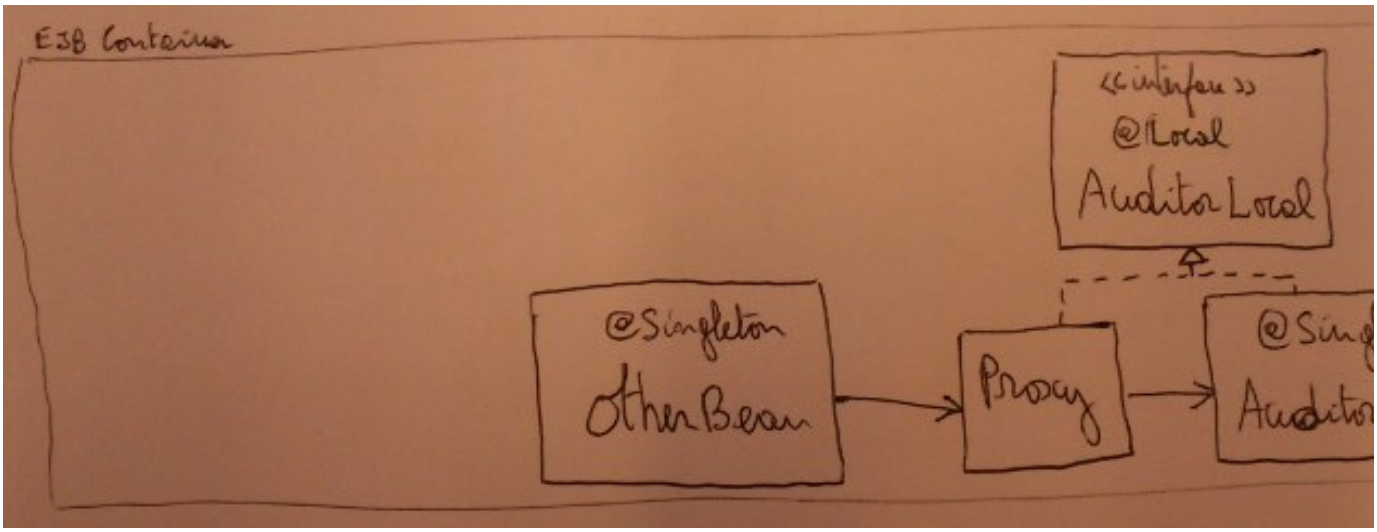
Version 2.0 of the specification introduced the notion of local interface to enable faster access to beans from within the EJB container. In the code and diagram below, we replaced the remote interface by a local interface annotated with @Local.

Source

```
@Local
interface AuditorLocal {
   Account checkAccountStatus(String userName);
}
```

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean implements AuditorLocal {
    ....
}
```



5.4.5. Local and Remote Interfaces

If it had to be accessed by both a remote client and another bean, a session bean could have two interfaces: a remote and a local.

The Auditor bean with two interfaces would be:

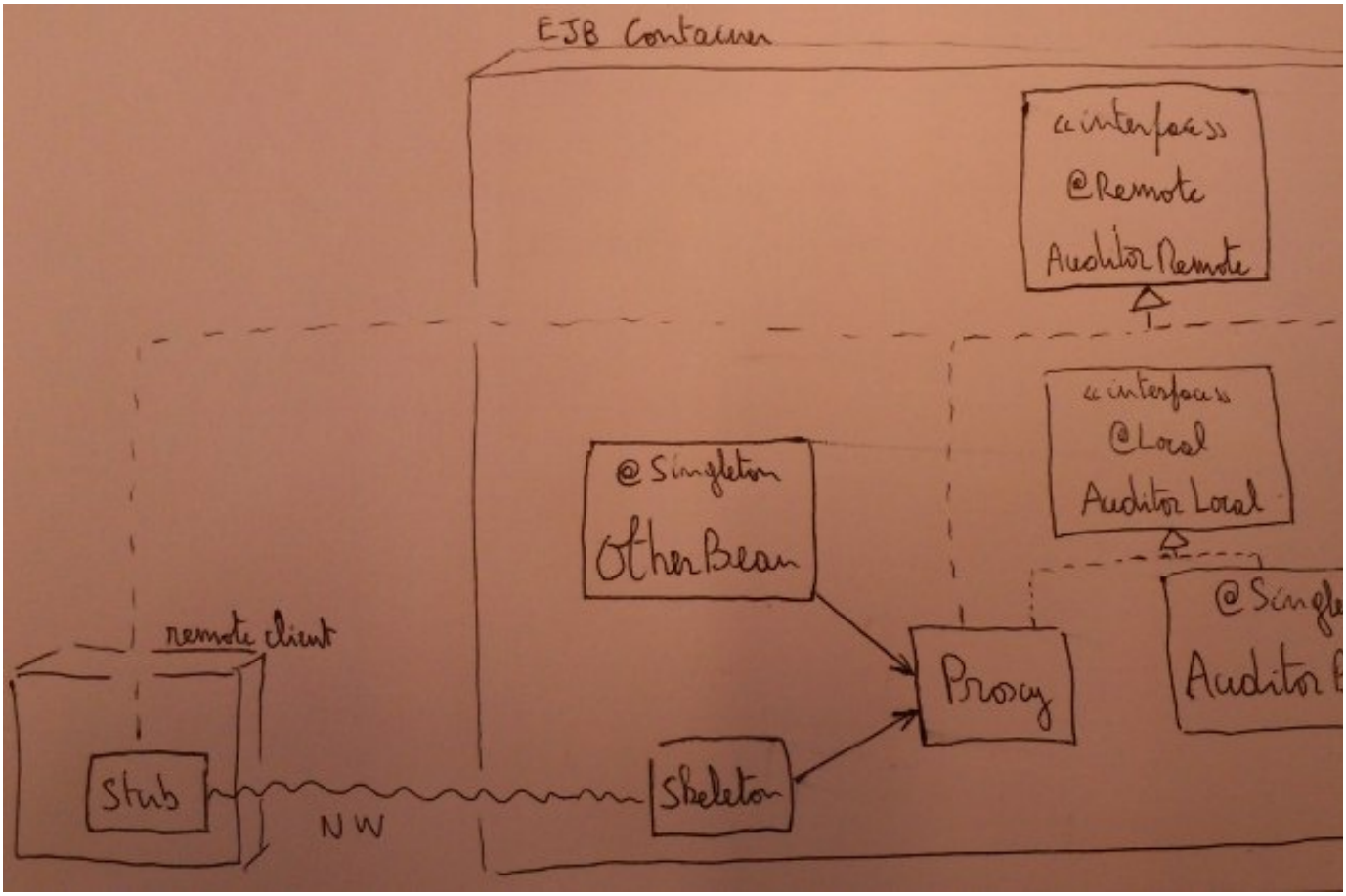
Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean implements AuditorRemote, AuditorLocal {
    ....
}
```

This is OtherBean getting the AuditorBean injected through its local interface:

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class OtherBean {

   @EJB
   AuditorLocal auditor;

   ....
}
```



5.4.6. No Interface

Since the v3.1 of the EJB specification, no interface at all is needed to access a bean from another bean. That is what you have done in the previous topic with AccountRepositoryBean, accessed from AuditorBean. AccountRepositoryBean has no interface.

5.4.7. Interface Choices

So, you have the choice between local interface and no interface. By default, having no interface is easier. If for any non-EJB related design reasons you want to program with an interface, simply annotate it with @Local. It may be the case for example, when you want to create a mock of the bean for unit testing. The mock would implement the local interface.

Here are a few rules for selecting what interface to create:

- **Remote**: When a remote client needs to access your bean.

- **Local**: When a local client needs to access your bean and you prefer to have your business methods defined in an interface.
- **None**: When only local client needs to access your bean.

Because remote interface calls imply network communications, you should try to limit the amount of calls done to a remote interface. For the reason, remote interfaces are usually coarse grained, which means having fewer methods but each of them doing more.

5.4.8. Activity : AccountRepository Interface

A coworker of you wants to introduce unit testing in the banking application. He will create a class *AccountRepositoryMock* with a fake implementation of your AccountRepository methods. He needs you to provide a local interface to implement, and to upcast the reference in AuditorBean to that local interface. In this activity you create the local interface and adapt the bean.

1. Start from the banking application version where AccountRepositoryBean is an EJB injected in AuditorBean.
2. Create an interface named AccountRepository that contains the signature of the methods used by AuditorBean.
3. Make AccountRepositoryBean implement that interface.
4. In AuditorBean, change the type of the referring attribute annotated with @EJB, from AccountRepositoryBean to AccountRepository.
5. Test your application.

6. Lifecycle

Each type of bean have its own lifecycle. At some point the bean is created and then somehow configured/initialized by the container. This is the moment when component is ready to serve the client request. Later on the bean is available to be destroyed and is disposed by the application server.

Now we will take a closer look at the lifecycle of each type of bean components.

6.1. Lifecycle and Callbacks

As the EJB container creates and destroys your beans, it is useful to understand what step is done when. For example, you may ask the container to call some of your bean methods at specific steps of the bean lifecycle. These are named the callback methods.

In this topic, you will see how to use a callback method to execute initialization code after dependencies have been injected and before the first method is called. For example a singleton PaymentGatewayBean may need to ping a remote server to check its availability. You will do such initialization code in a call back method.

6.1.1. Callback Methods

EJBs are instantiated by the container. Additionally to calling the new operator, the container performs some operations on the beans. One of them is calling some callback method. A callback method is a method of your bean that you point out as such with a call back annotation. The choice of the annotation determines the kind of call back: when the container should call your method in the bean's lifecycle. Every bean can at least have these two call back

methods:

- @PostConstruct: called after the constructor's execution and after the dependencies have been injected.
- @PreDestroy: called before the garbage collection frees the bean instance.

Your methods annotated for call back must return void, have no parameter, throw no checked exception. They can be private.

Callback methods are a way for you to be notified of some events. You do not call them yourself. They are called by the EJB container when the corresponding event happens.

6.1.2. Initialization Sequence

The EJB container instantiates your beans. The sequence for a bean instantiation is the following:

1. Instantiate the object (new). Your constructor (if any) is executed.
2. Inject the dependencies.
3. EJB container calls @PostConstruct methods (if any).

The main usage of @PostConstruct call back methods is to perform initialization tasks that need injected values. When the constructor is executed, these dependencies have not been injected yet. In the example below, the field *ipAddress* is filled by the EJB container after the call to the constructor. Because of that, the method *pingIpAddress()* could not be called from the constructor, it would not know what address to ping. That method verifies that the server at ipAddress is up and responding.

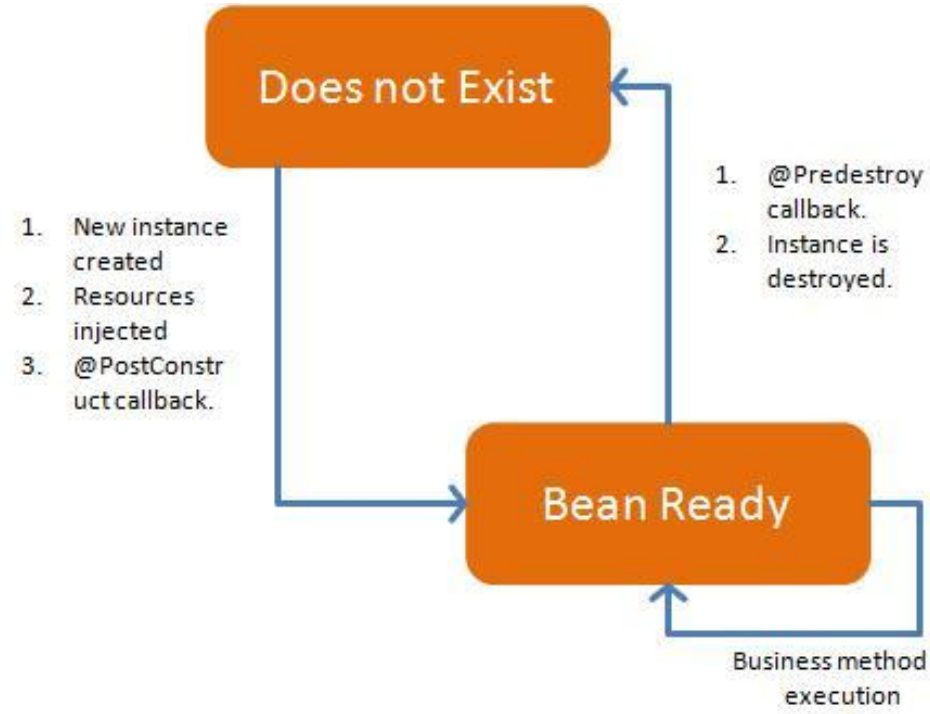
Source

```
@Stateless
public class PaymentGatewayBean {
   @Resource(name="ipAddress")
   private String ipAddress;

   PaymentGatewayBean {
       // Cannot call pintIpAddress() now (from constructor)
       // because ipAddress has not been injected yet :-(
   }

   @PostConstruct
   private void pingIpAddress() {
       try {
           InetAddress address = InetAddress.getByName(ipAddress).isReachable(3000
       } catch (UnknownHostException e) {
           throw new RuntimeException("Panic: Unable to lookup " + ipAddress, e);
       } catch (IOException e) {
           throw new RuntimeException("Panic: Unable to reach " + ipAddress, e);
       }
   }

   ...
}
```

The java.net.InetAddress class from Java SE, has a static method to create an instance from a name (or IP address): *getByName(String)*. It returns a InetAddress object on which we can call *isReachable(timeout)*.

6.1.3. State Diagram

In the EJB culture, it is very common to show the following state diagram.



The diagram concerns one bean. Both boxes represent a state and the bean is either in the "does not exist" or the "bean ready" state. The arrows are the actions to go from one state to another. Business methods can only be called from the "bean ready" state.

When we put this state diagram and our code example of the PaymentGatewayBean together, we see the following sequence:

1. The bean instance does not exists yet (top state on the diagram).
2. The container decides to instantiate the bean.
3. The constructor is called.
4. The container injects the value of ipAddress (thanks to the @Resource annotation).
5. The container calls @PostConstruct method and our code pings the remote server.
6. The bean is ready (bottom state on the diagram).

6.1.4. Destroying Beans

When the business method is finished, the bean is ready again.

For stateless session beans and message driven beans, the container may create multiple instances of the same bean class, maybe many in case of high need. When fewer threads (stateless beans) or messages (message driven beans) need an instance of that bean, the EJB container may decide to recover resources by lowering the amount of instances for that bean. It would call the @PreDestroy call back method and let the instance go to the garbage collector. This can be helpful in case one wants to get or release a database connection before the EJB container destroys a bean instance.

Statefull session beans are destroyed when their client calls a method annotated with @Remove. Contrarily to stateful session beans and message driven beans it is not the EJB container who takes the initiative of destroying instances. It is the client when the conversation with its bean is over. The container will also call @PreDestroy before

destorying the stateful instance.

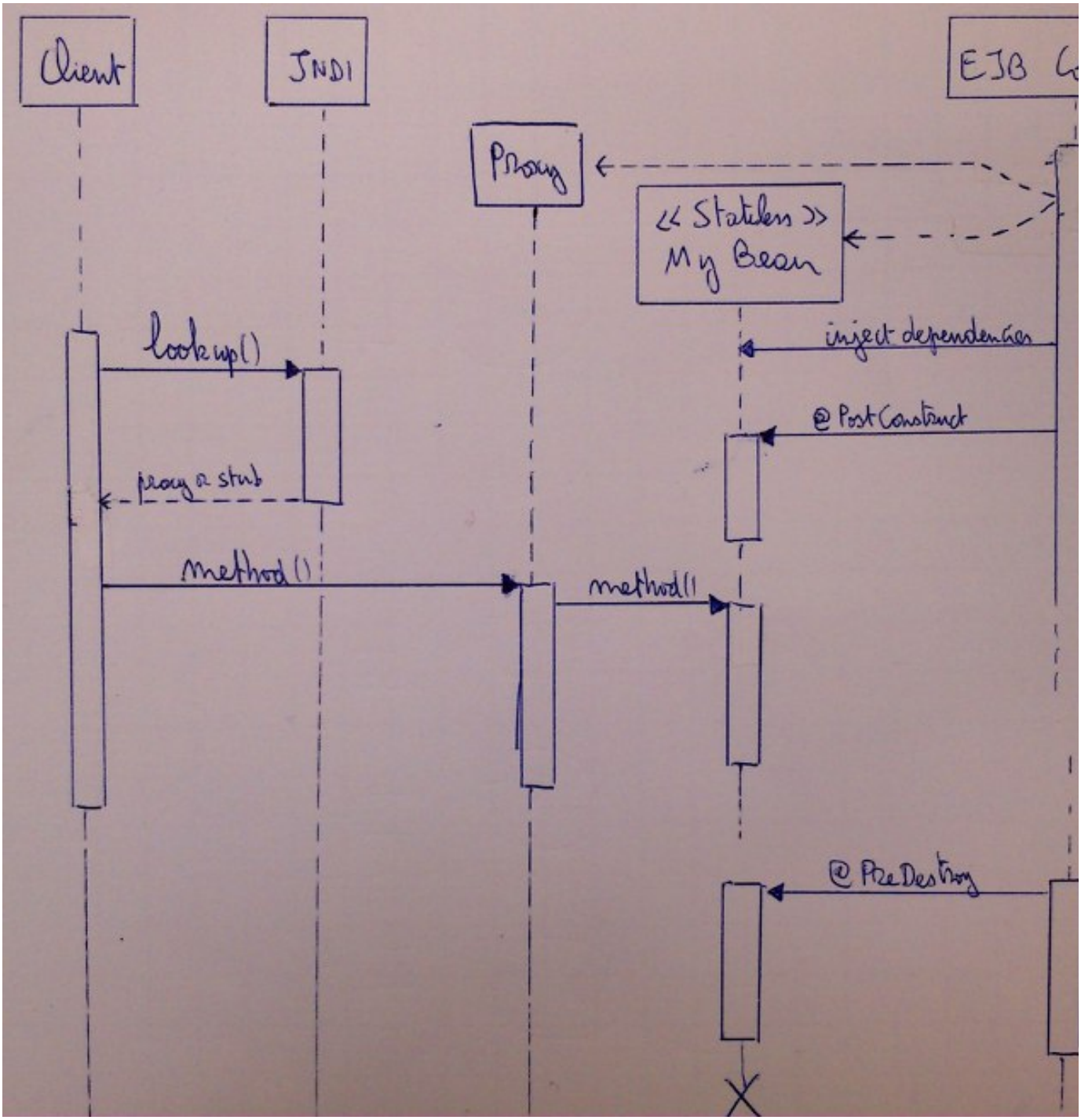
Singletons are similar to stateless session beans, but since there is one instance only, there is no reason for the container to destroy them before the EJB application shuts down.

6.1.5. Sequence Diagram

All the beans follow this basic sequence:

1. Bean construction.
2. Business method call (probably repeated multiple times, maybe serving different clients). In the case of message driven bean, it is the onMessage() method.
3. Bean destruction.

The following diagram shows them for a stateless session bean. It is very similar for a singleton or a message driven bean. For a stateful session bean, it would be different: the client lookup probably triggers the construction of the bean which is not reused once the connection is closed.



The container, on the right, first instantiates the bean and its proxy. It injects dependencies and calls any @PostConstruct method of the bean. This part would be different for a stateful session bean because it would be the client that triggers the creation of the bean when it needs it.

Later, a client needs the bean. It uses JNDI (or dependency injection) to get a reference to a stub (remote client) or proxy (client within the EJB container). It calls a business method on the proxy (directly or via the stub) which calls the corresponding method on the bean. Later, the same client or other clients may use that same proxy/bean instance to call business methods.

At some point, the bean needs to be destroyed. The EJB container calls any @PreDestroy method and the bean is ready for garbage collection.

6.1.6. @Remove

Stateful session beans are not shared or reused by the container. The client may help the container to know when it does not need the bean instance anymore. By calling a business method annotated @Remove, the client of a stateful session bean says: *I do not need this stateful bean anymore, my conversation with it is over.* Then the container knows it can destroy that bean and recover memory.

Our stateful session beans topic had a calculator as example and a load advisor as activity. CalculatorBean could have a new method named *turnOff()* with the @Remove annotation.

Source

```
@Stateful
public class CalculatorBean {
    ...
    @Remove
    public void turnOff() {
        ... // cut the power ;-)
    }
}
```

You can annotate the existing *confirmLoan()* method of LoanAdvisorBean with @Remove.

Source

```
@Stateful
public class LoanAdvisorBean {
    ...
    @Remove
    public void confirmLoan(int desiredDuration, double desiredRate) {
        ...
    }
}
```

After the call to an @Remove method,

- the container can destroy the bean any time (and will call @PreDestory methods first).
- any further usage of the bean by the client is forbidden (and would result in an exception thrown by the proxy).

@Remove is called by the client of a stateful session bean. It is forbidden on other beans.

6.1.7. Call-Back Methods by Bean Type

The following table lists the call-back methods that we have seen or will see in this lesson and for which type of bean they are valid.

@PostConstruct	Y	Y	Y	Y
@PreDestrory	Y	Y	Y	Y
@PrePassivate			Y	
@PostActivate			Y	

Please note that @Remove (valid for stateful session beans) is not in the table because it is not a call-back method (it is not called by the container but by your code).

6.1.8. Activity: Displaying the Lifecycle Sequence

In this activity, we create call back methods on a bean and verify in which sequence they are called.

1. Start from a working hello world application with a singleton session bean.
2. Add a public constructor with no argument to your HelloWorldBean class. It prints a message at the console "Processing constructor".
3. Add a method annotated @PostConstruct. It prints "Processing PostConstruct" at the console.
4. Add a method annotated @PreDestroy. It prints "Processing PreDestroy" at the console.
5. Test your application. You should see at least the message from the constructor and from the @PostConstruct methods at the console.
6. Add a method annotated with @Remove to your bean and call it at the end of your client code.
7. Test your application. The EJB container should refuse to start because only stateful session beans can contain @Remove methods and your HelloWorldBean is singleton.
8. Change your HelloWorldBean to a stateful session bean.
9. Test your code again. You should see the additional message from the @Remove method at the console, and probably the @PreDestroy message right after if the container decided to destroy the bean (no rule in the specification prevents EJB containers to be inneficient and keep unusable garbage).

6.2. Activation and Passivation

Because they are not sharable, stateful session beans may have many instances on the server, if many clients need them. The EJB specification contains a mechanism to temporarily store unused instance onto hard disk to free up some RAM. This is named passivation and helps applications to be more scalable. With activation, the EJB container brings back a passivated instance into memory in case a client needs its stateful bean again.

6.2.1. Serialization

Serialization is a Java SE mechanism to store (a graph of) object(s) in a stream, typically connected to a file or a network connection. For example, parameters passed from a remote client to a session bean are serialized over the network. In the case of statefull session beans passivation, the bean is serialized on secondary storage by the EJB container.

The bean attributes that are primitive or serializable types will automatically be saved into secondary storage unless they have the *transient* keyword.

Bean attribute of a special types recognized by the EJB container are automatically properly handled (and restored at activation) by the EJB container:

- javax.ejb.SessionContext
- javax.persistence.EntityManagerFactory
- javax.persistance.EntityManager
- javax.sql.DataSource
- javax.jta.UserTransaction
- javax.naming.Context

6.2.2. Passivation Callback Methods

When the container decides to passivate the bean in order to recover some memory, it calls your callback method annotated with @PrePassivate. This enables your to perform any needed cleanup operation before the beans is saved to secondary storage and that the current bean instance is taken by the garbage collector. It would typically be the closing of a remote connection.

If the clients calls a method again on the bean, it will be activated by the container before the business method call. During the activation, the container will call any method annotated with @PostActivate, to enable you to restore some resources that could not be saved. It would typically be the re-opening of a remote connection, or restoring any non serializable attribute.

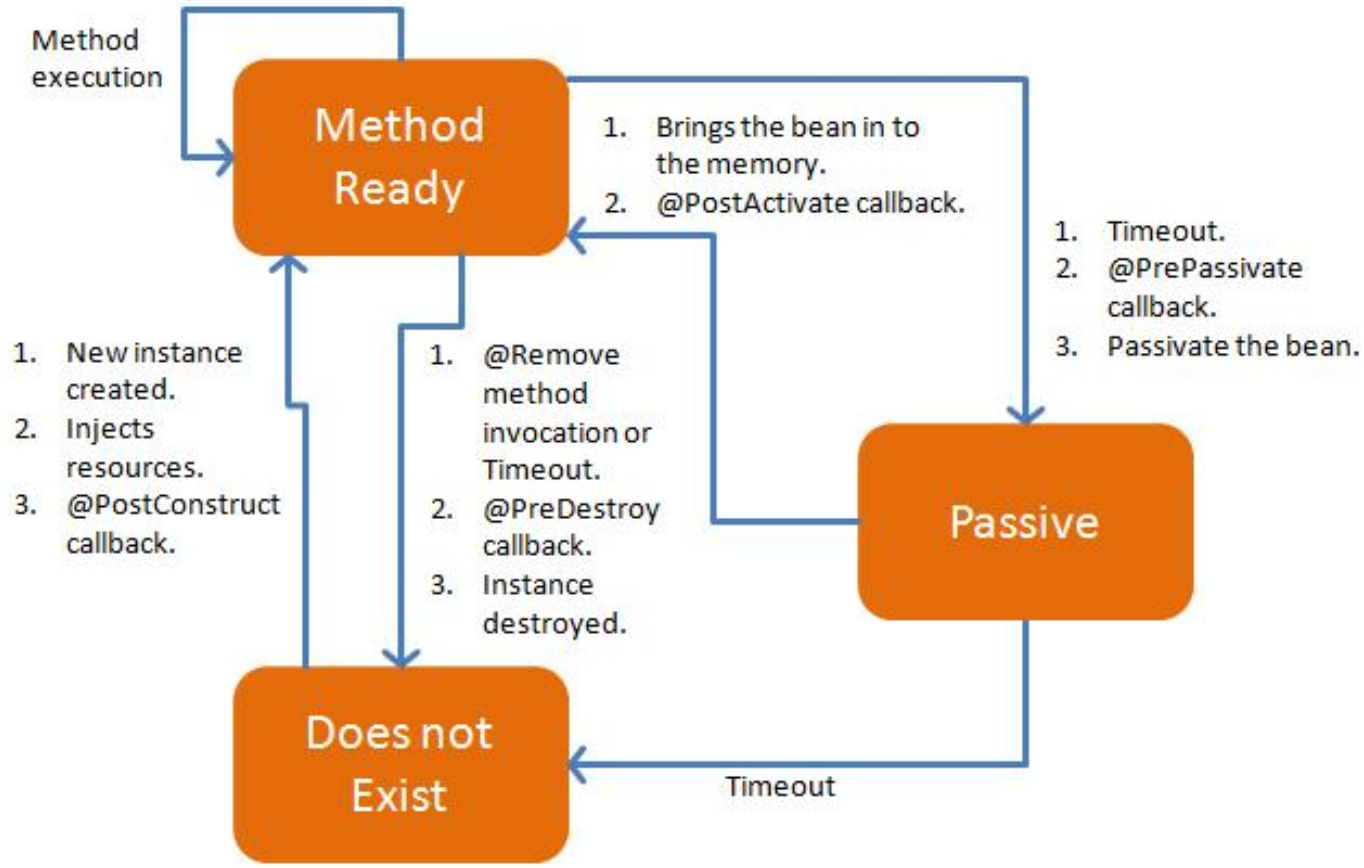
The code below is a stateful session bean with 4 attributes as conversational state. The 5th attribute is a reference to the LegacyConnection class (a class made by the programmer to access a mainframe). This reference is managed by the bean that knows how to obtain it (with a new). LegacyConnection should not be serialized in when the bean is passivated. It is marked as transient. The programmer prefers the reference to be dropped and to make a new instance of LegacyConnection when the bean activates back in memory.

Source

```
@Stateful
public class LoanAdvisorBean {

   // Conversation state (that will be serialized)
   Account account;
   long[] amountRange;
   Long amount;
   Map<Integer, Double> rateRange;


   // This one should not be serialized along.
   transient com.example.LegacyConnection legacySystem;

   @PostActivate
   public void restoreLegacyConnection() {
       legacySystem = new LegacyConnection();
   }
}
```

6.2.3. Stateful Bean Lifecycle

Stateful session beans have an additional state in their lifecycle: the passive state.



The states *Does not exist* and *Method Ready* are the same for all the kinds of EJBs, as explained in the previous topic. The diagram contains a new state: *Passive*. After some time out, the EJB container needs to recover memory or wants to be able to recover in case of crash: it decides to passivate the bean. Any @PrePassivate method is called, then the bean is serialized to secondary storage.

If/when the client of the stateful bean needs to call a business method again, the EJB containers activates the bean: deserialize and call @PostActivate method(s).

6.2.4. Timeout

Stateful session beans are coupled with a specific client. It means that the bean must exist as long as the client has not disapeared or has not called a @Remove method. It may happen that the client is badly programmed and does not call a @Remove method or crashes. To prevent the corresponding bean to stay forever in the container (active or passive), a timeout mechanism removes these beans after a long inactivity. The delay can be specified by the @SatefulTimeout annotation. A clients that tries to use a bean that disappeared after a timeout, will get an exception. The example below sets a time-out of 30 minutes.

Source

```
@Stateful
@StatefulTimeout(unit = TimeUnit.MINUTES, value = 30)
public class LoanAdvisorBean {
    ...
}
```

6.2.5. Sequence Diagram

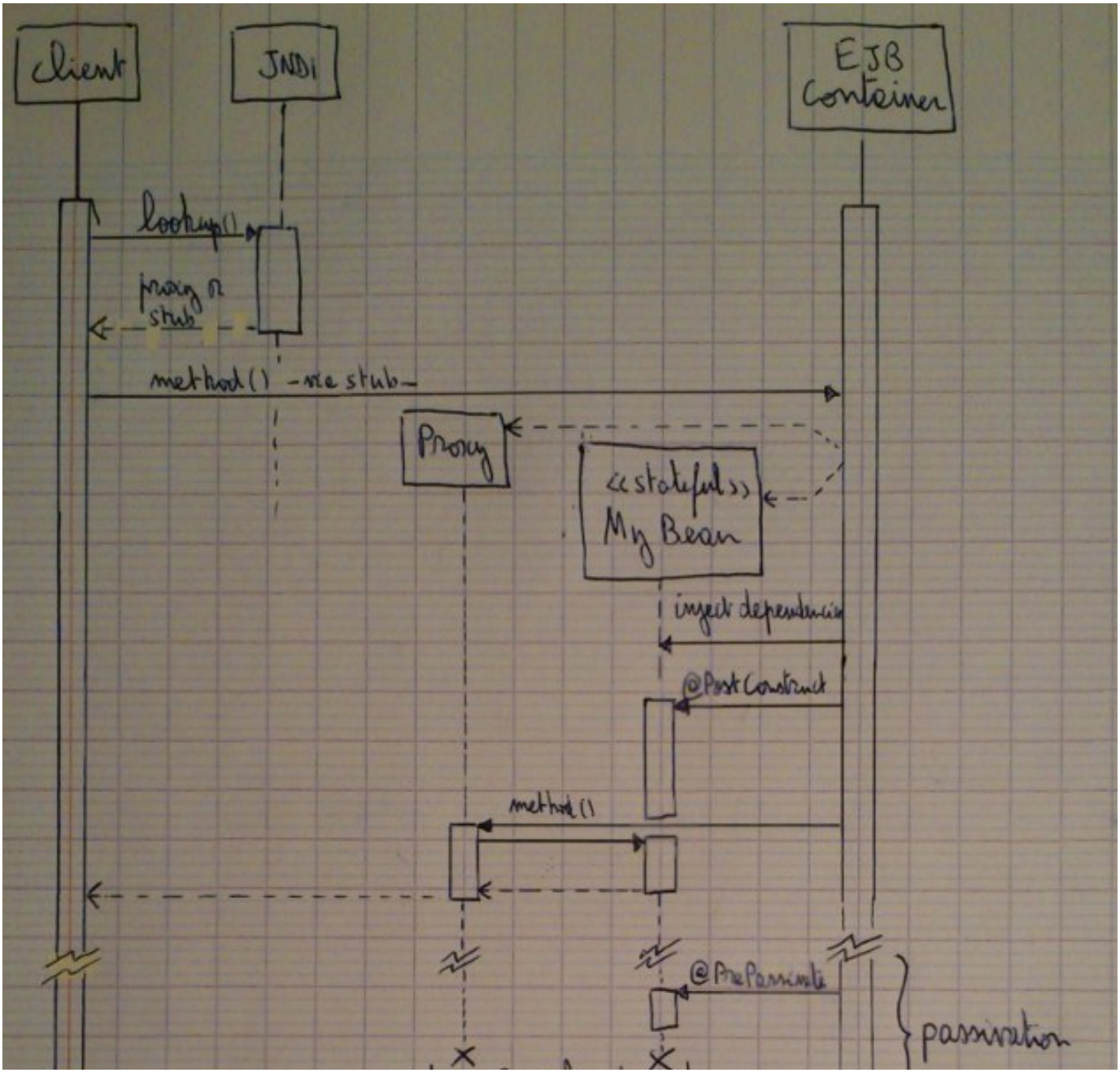
The sequence diagram below tells the following story.

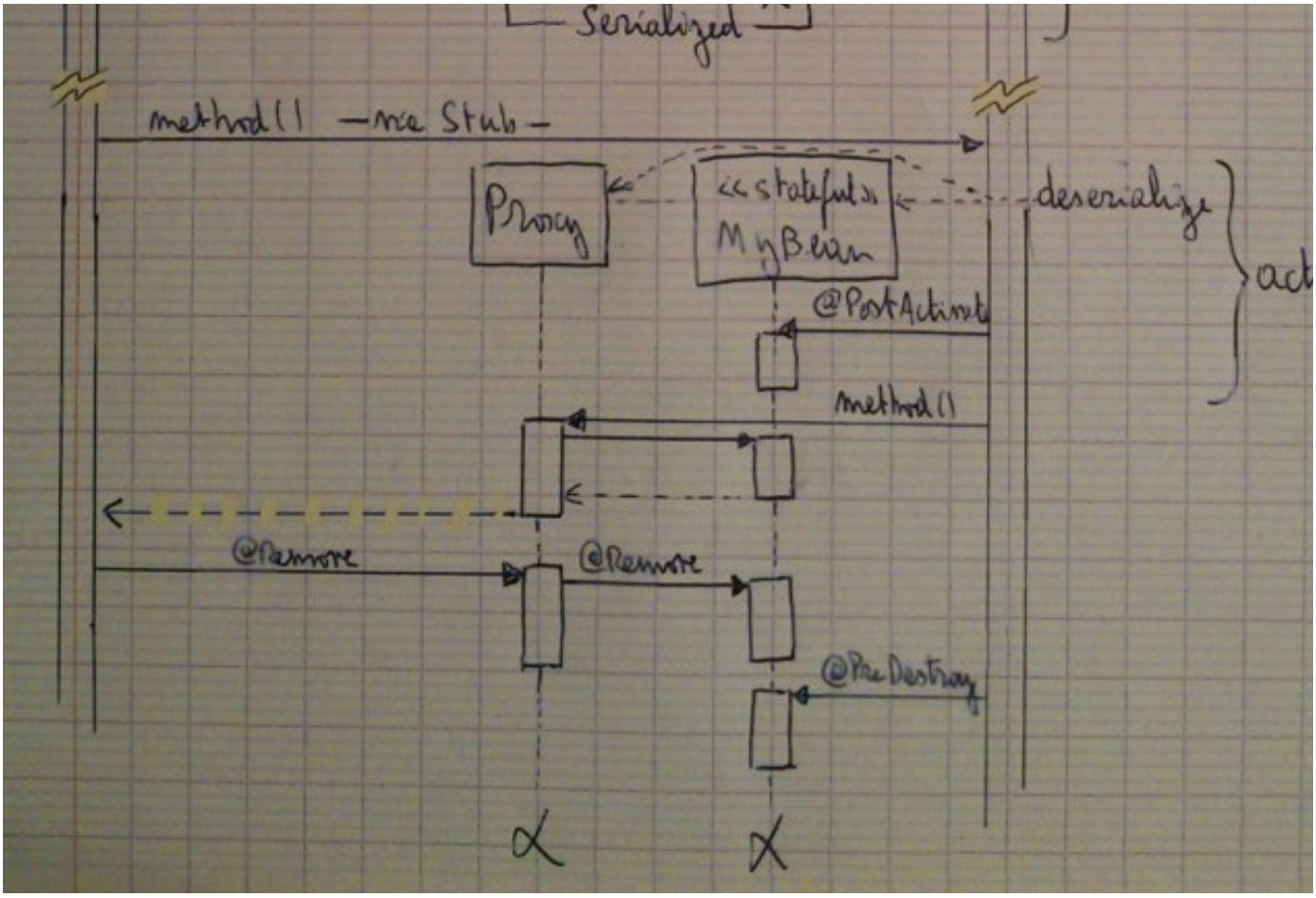
A client needs an instance of our stateful session bean. It gets a stub (it is a remote client) from JNDI. Then it calls a business method. The container, creates a new instance of the bean and its proxy for this client. It injects the dependencies and calls any @PostConstruct method of the bean. Then the business is called via the proxy.

Much later, the bean has not been used and the container is short on resources. It decides to passivate the bean. Any @PrePassivate method is called and the bean is serialized to secondary storage and can be garbage collected.

Some time later, the clients calls a business method again on the bean. The container needs to activate the bean back in memory in order to make that call possible. Along the way the bean is deserialized and any @PostActivate method is called.

When the client does not need the bean anymore, it calls a method annotated with @Remove. The containers figures that out and calls any @PreDestroy method before making the bean reasy for garbage collection.





6.2.6. Activity: Make a Bean Time Out

You are reponsible of maintaining an EJB bases application that uses too many stateful session beans (because of bad design probably). You think that some clients forget to call the @Remove method. It consumes much memory and you want to introduce some time-out. Before doing that, you make some test in a small application. In this activity, you create a bean with a short timeout and try to reach it after the delay.

1. Start from a working hello world application with a singleton session bean.
2. Make the bean stateful.
3. Annotate the bean to specify a 10 seconds timeout.
4. Add a method annotated @PrePassivate. It prints "Passivating" at the console.
5. Add a method annotated @PreActivate. It prints "Activating" at the console.
6. Test your application. It should work well.
7. Change the client to introduce a 20 seconds delay (*Thread.sleep(20*1000)*) after the business method call. Call the business method a second time after the delay.
8. Test your application. It should not work, the client should get an exception because the bean timed out before the second call.

Note that the "Passivating" and "Activating" messages will probably not appear at the console because your EJB container is under no memory pressure and the delays are very short in this activity. Therefore it will probably not try to passivate the bean.

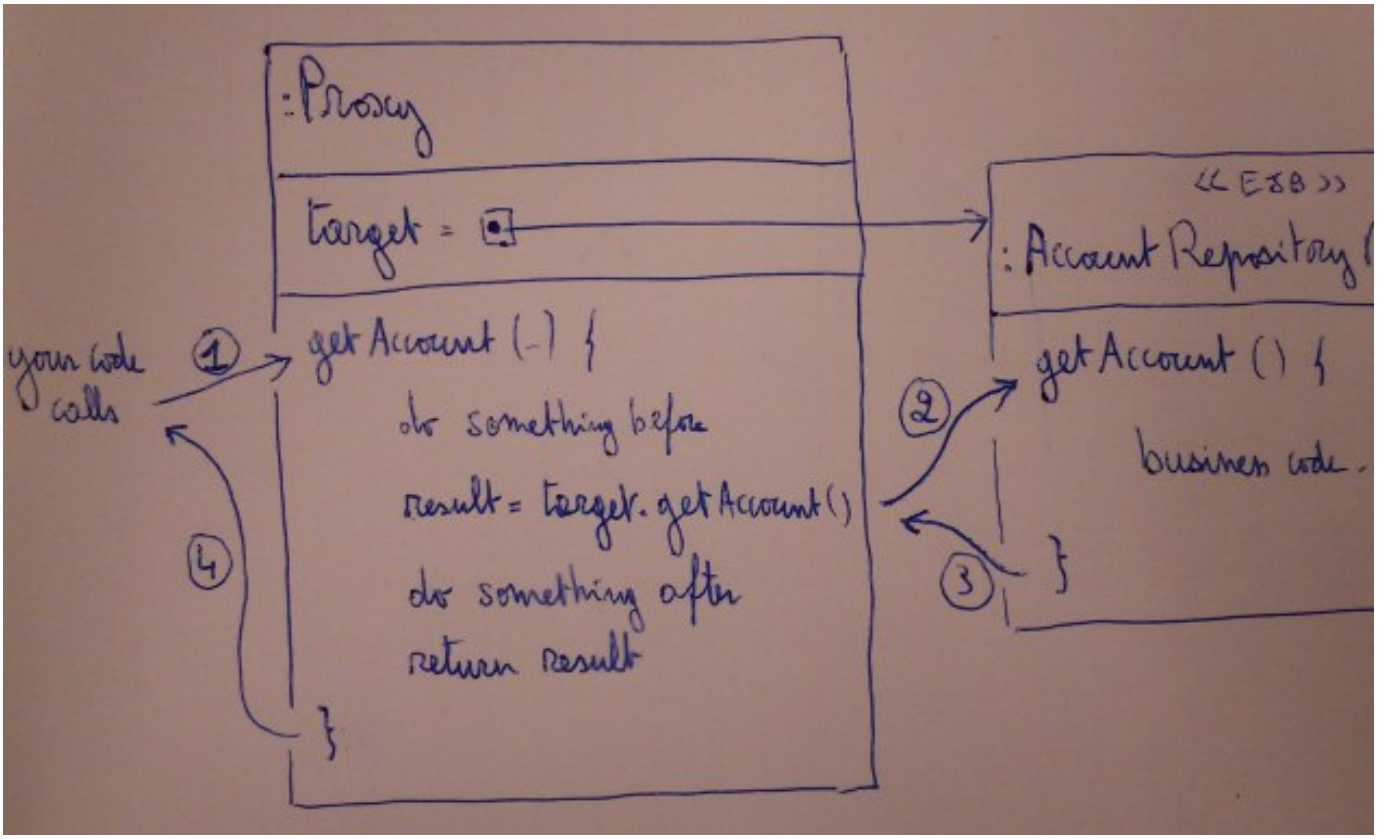
6.3. Proxies

The EJB container provides services to your beans, such as transaction management and security, or custom services that you program yourself as we will see in the next topic.

In this topic, you will see how the EJB container creates and uses proxy classes to enable these services. Every bean has a proxy. Everytime you get a reference to a bean through a lookup or dependency injection, in fact you have a reference to its proxy.

6.3.1. Proxy Class

A proxy is a class between your client code and a target bean that has the same public method signatures. It mainly forwards any call to your target class. Maybe it will perform a little action right before or after calling your target class method. Typical "little actions" include transaction start, commit or rollback; authorization checking; logging.



In the diagram above, the EJB container has created a proxy class (has generated Java code) with the same method signatures as AccountRepositoryBean.

6.3.2. Proxy References

Within the EJB container, when a bean gets a reference to another bean, in fact it gets a reference to the proxy. You can get this reference through a JNDI lookup or through dependency injection. For instance, you want a reference to the AccountRepositoryBean.

Source

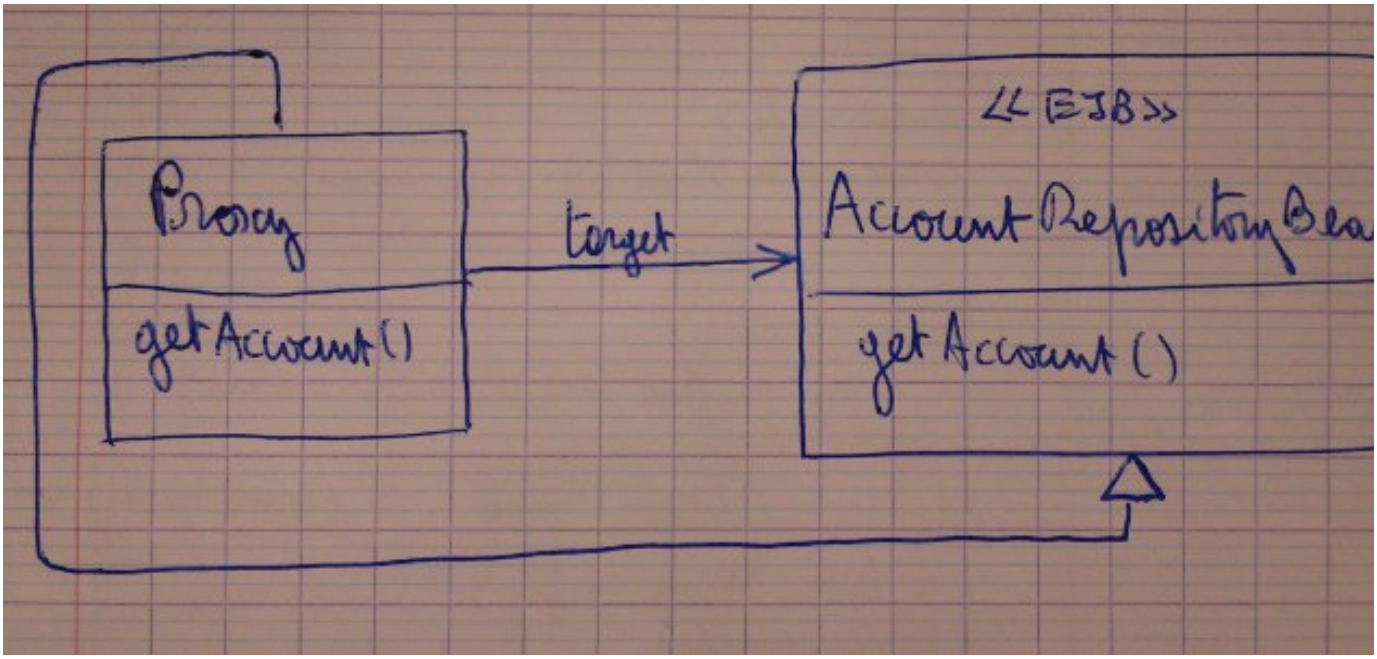
```
AccountRepositoryBean accountRepository = (AccountRepositoryBean) ctx.lookup("ejb/
```

Source

```
@EJB
AccountRepositoryBean accountRepository;
```

In both cases, you put the reference in a variable of type AccountRepositoryBean, which must be your bean class or your local interface. It means that the proxy needs to extend your bean class or implement your local interface, else you would have an InvalidClassCastException.

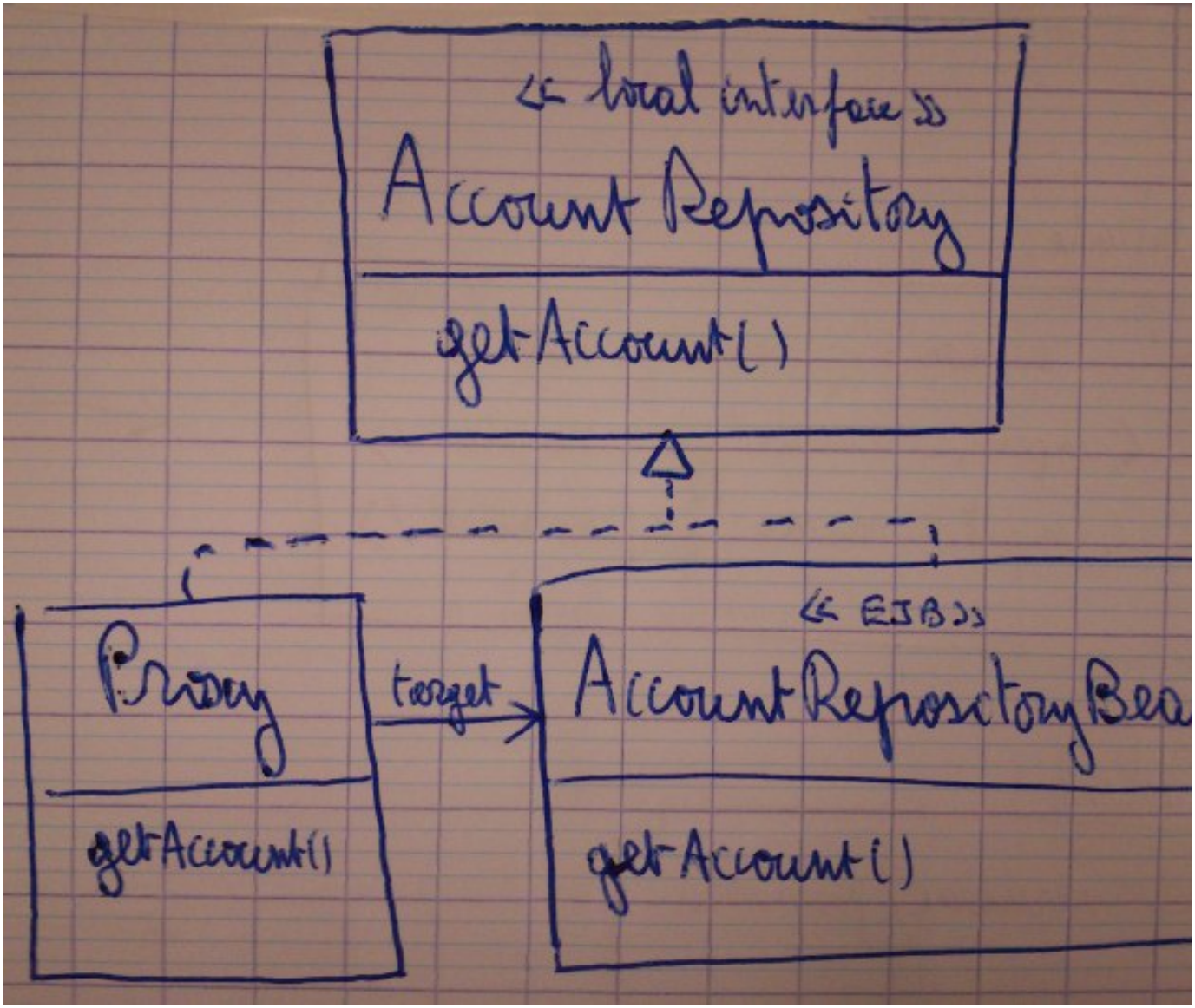
In this class diagram, there is no local interface and the proxy inherits from the bean class. The variables in the code above (DI and lookup) are of the bean type (AccountRepositoryBean).



In the code and class diagram below, there is a local interface and the proxy extends that interface.

Source

```
@EJB
AccountRepositoryLocal accountRepository;
```



6.3.3. Method Chaining

In the method chaining programming style, a method returns the instance of its class to enable another method call from the result.

The following code uses a calculator class to display 47 at the console. It chains the call to the methods add(), multiply, and sub().

Source

```
Calculator calc = new Calculator();
calc.add(5).multiply(10).sub(3);
System.out.println(calc.getResult());
```

This is possible because each of these methods returns its Calculator instance.

Source

```
public class Claculator {
   private double result;

   public double getResult() {
```

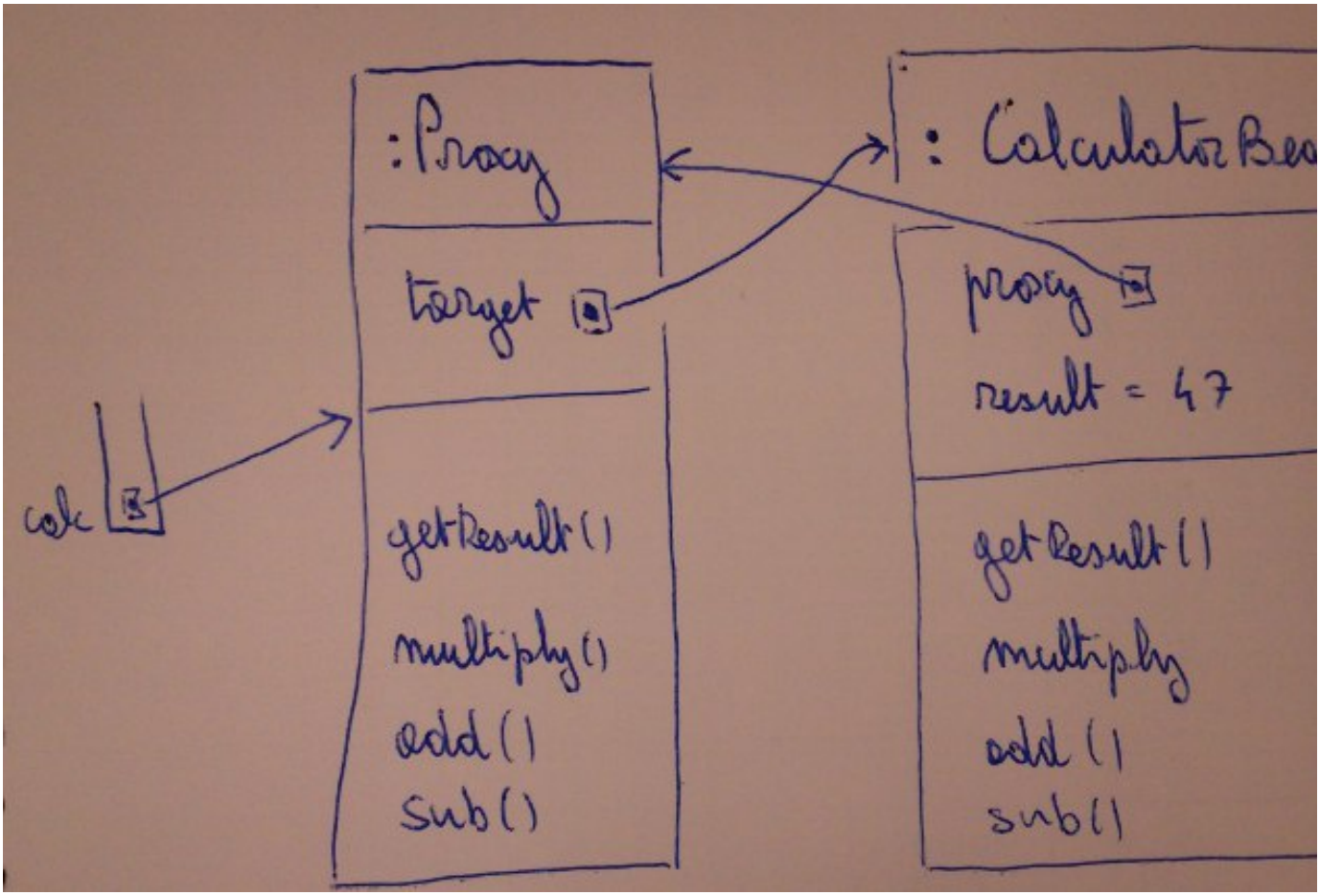
```
        return result;
    }

    public Calculator multiply(double op) {
        result *= op;
        return this;
    }

    public Calculator add(double op) {
        result += op;
        return this;
    }

    public Calculator sub(double op) {
        result -= op;
        return this;
    }
}
```

If you do method chaining with an EJB and return *this*, you return a direct instance to the bean and not to the proxy. The next method call would not go through the proxy. Bypassing the proxy would violate the EJB specification (and prevent some transversal services such as transactions to be started by the proxy).

Because of that, you should not return *this*, but a reference to the proxy. The easiest way to get this proxy is to ask the EJB container to inject it.

Source

```
@Statelful
public class ClaculatorBean {
    private double result;

    @EJB private CalculatorBean proxy;

    public double getResult() {
        return result;
    }

    public ClaculatorBean multiply(double op) {
        result *= op;
        return proxy;
    }

    public ClaculatorBean add(double op) {
        result += op;
        return proxy;
    }

    public ClaculatorBean sub(double op) {
        result -= op;
        return proxy;
    }
}
```



6.4. Interceptors

Interceptors are methods which intercept a business or lifecycle method call, and have the opportunity to perform operations before and after that call.

They enable to isolate cross-cutting concerns into a separated class, and make this code being called for many beans by the web container.

Interceptors are typically used for logging, security or transactions.

6.4.1. AOP

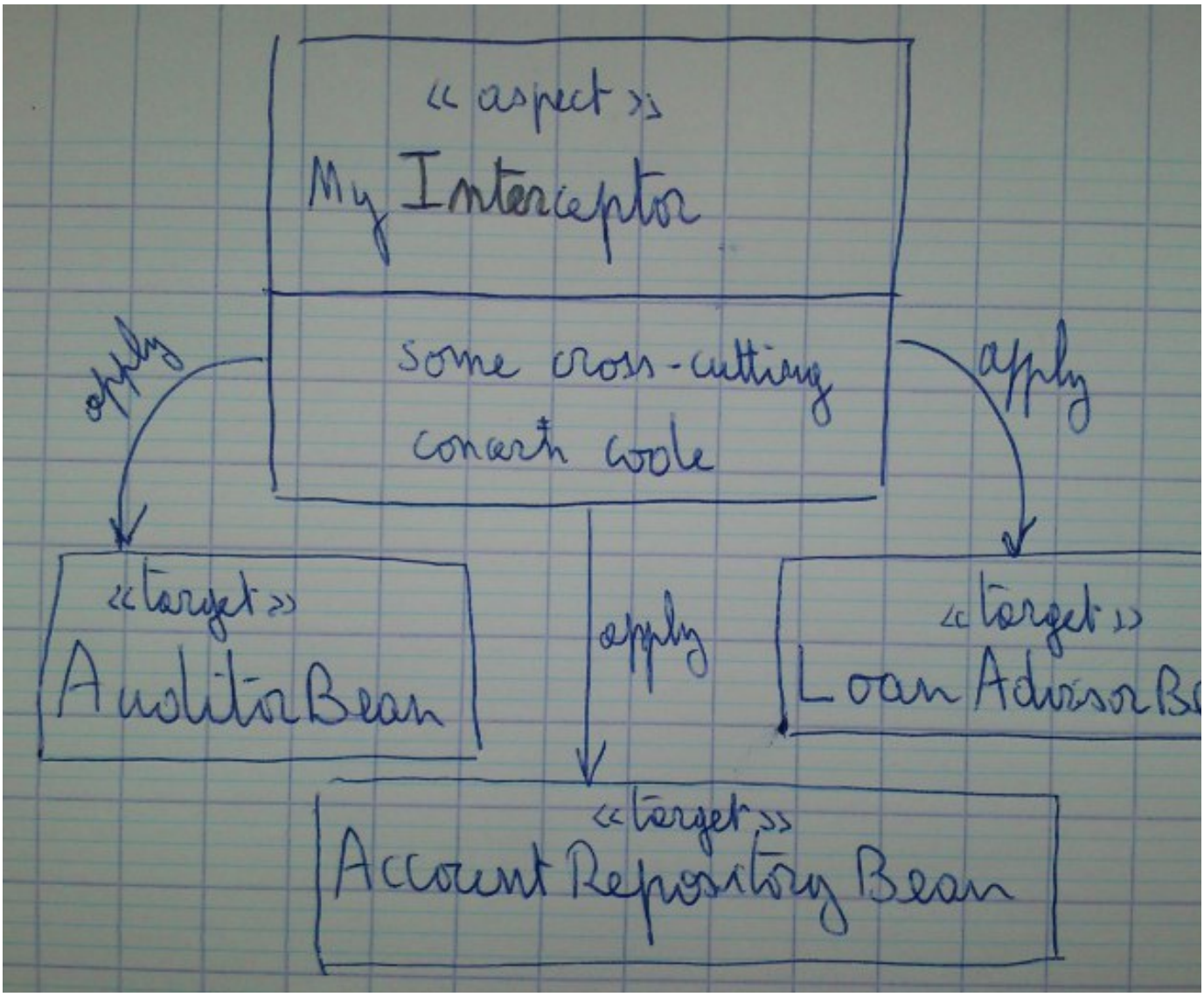
Before we dive into the world of interceptors let's have a quick look at the AOP (Aspect Oriented Programming). What is AOP? We'll take couple of simple examples first:

Let's say that you want to write a code to log some statements for your business method executions. Or you want to have transactions to be used while some common business methods are executing. Essentially you want to intervene some code (method) execution with some other code.

In the logging example above you would add statements in every method where you want the the statements to be logged. What this brings you is that whenever you want to change the content/behavior of your logging you ended up with modifying all the source files which contain the log statements. AOP is to help you on this by separating the logging logic/behavior (in this escenario) in to a seperate module without duplication and then apply the logic/behavior in to various parts of the application code declaratively. In this way you would have to change only one (or few) place(s) for the logging logic/behavior instead all the source files they are applied to. This is applied to transaction

example as well where you can omit the transaction behavior in the business logic (which make the code highly cohesive with the code only focissed on the business logic) and then apply the transaction declaratively where ever the business methods are used.

More formally, AOP is a programming paradigm (achieved with the help of tools such as compilers) that enables the separation of cross-cutting concenrns with the goal to increase modularity.



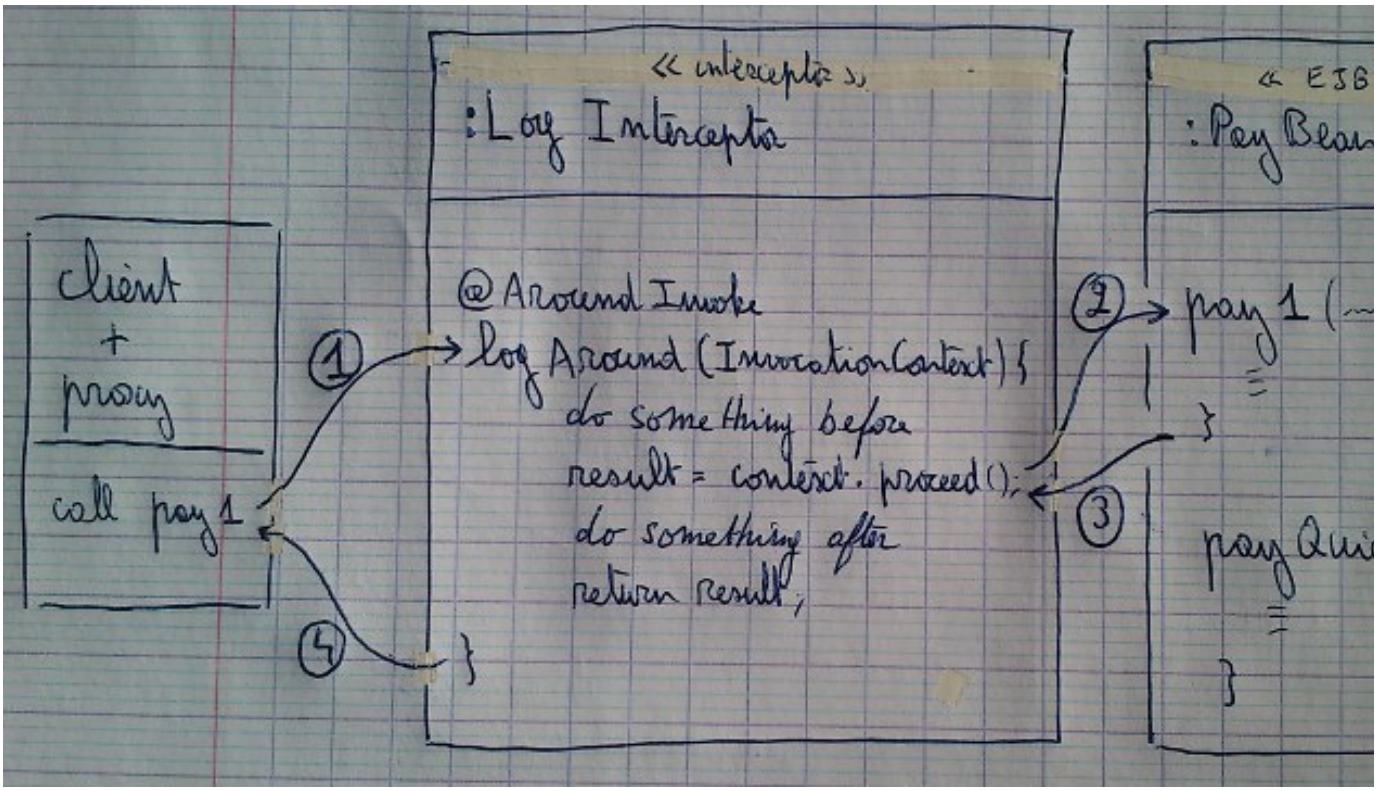
AOP is a common idiom used in application development heavily these days. The core concept behind this is that the common duplicated code across different layers of the application which are not directly related to the core business logic can be separated in to independent module(s) and then can be applied to any other application code (where applicable) as necessary. It essentially modularize the crosscutting concerns (i.e.: logging, auditing, transactions, security etc...).

EJB interceptors are "AOP in EJB". They provide a way to handle crosscutting concerns for EJBs. They can be applied to both Session Beans & Message Driven beans. In AOP the interception is triggered at different points (known as point cuts) of code execution. There are various point cuts available in AOP such as at the begining of a method, at the end of a method etc... EJB interceptors provide around invoke advice (advice is a additional code you want to apply to your code) interception. That is they are triggered at the begining of a method execution and are around when the invoked business method returns.

6.4.2. Creating an Interceptor

Let's first illustrate the concept with an example. We will detail its various parts through this topic. The following *LogInterceptor* class do logging. This very classical example writes a text at the console before any bean method is called and after that method call.

In the diagram below, you see two objects. The interceptor on the left, and a bean on the right. PayBean is very simple for the sake of the example: its methods return a string. Everytime a method of PayBean is called, the interceptor intercepts the call and executes its method first (*logAround* in this example). Somewhere during its execution, the interceptor will call PayBean's method.



The body of this *logAround* method is simple: it prints a text, it let the bean's method being called (context.proceed()), and it prints another text. The @AroundInvoke annotation marks that method as an interceptor method: it tells the EJB container that this specific method contains the cross-cutting concern code.

Source

```
public class LogInterceptor {

   @AroundInvoke
   public Object logAround(InvocationContext context) throws Exception {
       System.out.println("Executed: " + context.getMethod().getName());
       Object result = context.proceed();
       System.out.println("Results is: " + result);
       return result;
   }

}
```

Interceptors need to be applied on beans to be used. In the example below, we apply our interceptor to *PayBean*, thanks to the @Interceptors annotation.

Source

```
@Singleton @ConcurrencyManagement(BEAN)
@Interceptors(LogInterceptor.class)
public class PayBean {

   public String pay1(int amount) {
       return "Paid: " + amount;
   }


   public String payQuick(int amount) {
       return "Paid quicly: " + amount;
   }

}
```

If you call the *pay1()* method on *PayBean* and pass 100 to it, the interceptor will report it to the standard output:

Source

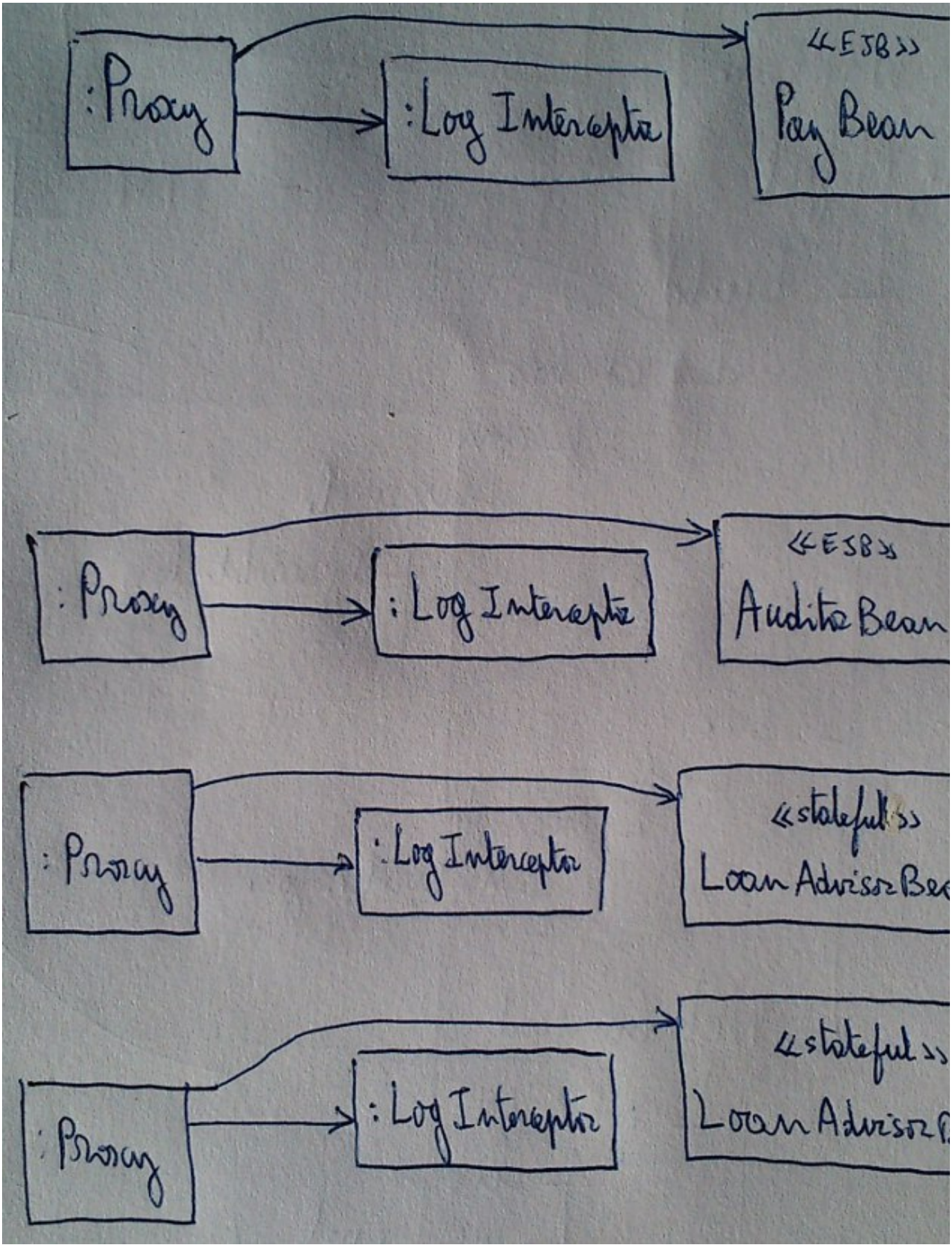
```
Executed: pay1

Result is: Paid: 100
```

6.4.3. Interceptor life cycle

Interceptors just like other managed beans have a lifecycle. The lifecycle of an interceptor instance is the same as that of the target bean instance which the interceptor is associated with. When the target bean instance is created any interceptor(s) instance(s) associated with it will be created. The interceptor instances are destroyed when the target instance is destroyed. In case of an interceptor associated with a statefull session bean interceptor instance is passivated/activated along with the related session bean.

The rule for Interceptor life cycle is that only a single instance of an interceptor is created per target bean instance.

The diagram below contains two singleton beans (PayBean and AuditorBean) and two instanced of the stateful LoanAdvisorBean, on the right. Each bean instance has its own proxy and LogInterceptor instance. The proxy calls the interceptor. The interceptor calls InterceptorContext.proceed(). That method calls the bean's method. The proxy keeps a reference to its interceptor and bean instances.



6.4.4. InvocationContext interface

The InvocationContext interface is one of the key components of EJB Intercpetors. It provides various methods to access metadata which the intereptor methods are interested in.

Below is the interface definition:

Source

```
public interface InvocationContext{
public Object getTarget();
public Object getTimer();
public Method getMethod();
public Object[] getParameters();
public void setParameters(Object[] params);
public java.util.Map<String, Object> getContextData();
public Object proceed() throws Exception;
}
```

InvocationContext instance can be used to pass data between interceptors applied for a given target class. This is possible due to the same instance of the InvocationContext is used for a given target class method or lifecycle event interception. The getContextData() method returns a Map of name-value pairs where you can store/retrive your data that you want to pass between the intercptors.

The getParameters() method returns the parameters in an Object array passed to the method being intercepted by this interceptor. The setParameters() method can be used to change the values of the parameters passsed to the method. This can be very handy where an intercpetor can analyze the parameters passed in and then modify them according to a defined businees rule. The setParameters() method modifies the parameters used to invoke the target method of the bean instance. This however cannot be used as a mechanism to change the method invocation incorrectly in the target instance (by changing the types or number of the parameters), by doing so results in a IllegalArgumentException.

The getTimer() method is applicable to timeout method interceptors which returns the associated timer instance. For other types like normal method interceptors, lifecycle callback method interceptors this method simply returns null. The getMethod() returns the method of the target bean which the interceptor was invoked, for life cycle callback interceptor methods this returns null.

Interceptor methods must invoke the proceed() method which causes the next interceptor method in chain or the target method of the bean instance (if it is the last interceptor method) to be invoked. proceed() is the most important method of InterceptorContext, and in most cases, the only one you need to know/call.

6.4.5. Kinds of Interceptors

Any class can be an interceptor. It does not need to extends/implement a specific class/interface. In fact, the class is not really an interceptor, but some methods of a class may be annotated as interceptor methods. There are 3 groups of interceptor methods:

1. Business methods: @AroundInvoke
2. Timer methods: @AroundTimeout
3. Lifecycle methods: @PostConstruct or @PreDestroy

Our LogInterceptor.logAround() example is from the first group: business method. It is the most common.

@AroundTimeout is used for timer beans which is an asynchronous kind of bean that is covered in a later topic.

These two first kind of interceptor methods must return an Object (the object returned by the intercepted bean call) and throw Exception in their signature.

In the third case, an interceptor method can be annotated with @PostConstruct or @PreDestroy. It will be called for any post-constructor or pre-destroy call of all the beans its applied on. This last kind of interceptor returns void and throws no exception in its signature.

6.4.6. Applying an Interceptor With Annotations

So far, in the example, you have used the @Interceptors annotation to apply an interceptor to a bean.

Source

```
@Singleton @ConcurrencyManagement(BEAN)
@Interceptors(LogInterceptor.class)
public class PayBean {
    ...
}
```

But this is impractical because we must annotate all the beans that need to have the interceptor applied. There could be many.

6.4.7. Applying an Interceptor With XML

Using the deployment descriptor is a more common/useful way to apply interceptors to beans in large applications. In the deployment descriptor fragment below, the `<interceptor-binding>` element enables you to define a pattern for the name of the target bean. In this case "*" has been used to designate all the beans of your application.

Source

```
<assembly-descriptor>
   <!-- Log interceptor that will apply to all methods for all beans in deployment
   <interceptor-binding>
       <ejb-name>*</ejb-name>
       <interceptor-class>com.example.LogInterceptor</interceptor-class>
   </interceptor-binding>
   ...
</assembly-descriptor>
```

In most cases, you write interceptors to be applied to many classes. Else you would just add the code in the target class and would not use AOP and the interceptors mechanism. Applying interceptors to many target classes is easier with patterns in the XML deployment descriptor (than with the @Interceptor annotation in each target class).

6.4.8. Activity: Execution Meter

The banking application you are working on is slow. To help the team spotting the slowest methods, you have decided to write an interceptor that measures and display the time spent by an EJB method for its execution.

1. Start from any working EJB project, such as the HelloWorld project.
2. Create an intercetor class named ExecutionMeterInterceptor.
3. In this interceptor, create a *measureTime()* interceptor method with the appropriate signature and annotation.
4. Fill that method with the following code:
    1. You first instantiate Date (new Date()) to get the current date/time.
    2. You proceed with the bean method call.
    3. You instantate Date again to get the time after the bean method execution.
    4. The Date class has a getTime() method that returns miliseconds. By substracting the miliseconds of both dates, you get the duration in miliseconds between the two dates. At the end of the method, display that duration to the console.
5. Create the deployment descriptor file if there is none in your project. Add the neccesarry information to your deployement descriptor to apply that interceptor to all the beans of your application.
6. Start the server and test your code by calling any bean. You should see the call duration on the server's console printed by the interceptor.

7. Transactions

In the previous lesson, we have seen how the EJB container creates proxies to provide transaversal services. A major service is the the transactions control. Transactions help your business application to keep external data not corrupted by concurrent access. That data usually resides on relational databases. Sometimes, message queues are involved. In this lesson, you will see how the EJB container acts as a transaction manager to make multiple actors (such as two DB and one MOM) involved in a single transaction.

7.1. Transaction Principles

XXXXXXXXX I'm not inspired for an intro, please propose one, Chennai guys ;-)

7.1.1. Needing Transactions

In more than 99% of the cases, EJB transactions implies transactions within relational databases (RDBs). The following example reviews the basic of DB concurrency.

Two money transfer transactions need to happen:

- Bob must give $20 to Alice.
- Charlie must give $5 to Alice.

Before these transactions, Alice starts with $1000 on her account.

Each transaction separately has the following sequence:

Source

```
Local variable 1 = Read Alice balance from DB. (= $1000)
Local variable 1 = Local variable + $20   (= $1020)
Write Local variable 1 as new balance for Alice in DB  (= $1020)
... Idem to remove $20 from Bob.
```

Source

```
 Local variable 2 = Read Alice balance from DB.  (= $1020)
 Local variable 2 = Local variable + $5   (= $1025)
 Write Local variable 2 as new balance for Alice in DB  (= $1025)
 ... Idem to remove $5 from Charlie.
```

At the end, Alice has $1025 on her account. That code is easy to transcribe in Java and is fine taken independently.

If we run these sequenses concurrently the result may not be the same, as in this example:

Source

```
Local variable 1 = Read Alice balance from DB. (= $1000)
Local variable 1 = Local variable + $20   (= $1020)

 Local variable 2 = Read Alice balance from DB. (= $1000)
 Local variable 2 = Local variable 2 + $5   (= $1005)
 Write Local variable 2 as new balance for Alice in DB  (= $1005)
 ... Idem to remove $5 from Charlie.

Write Local variable 1 as new balance for Alice in DB  (= $1020)
... Idem to remove $20 from Bob.
```

At the end, we write the value of *Local variable 1* as the balance of Alice who has $1020 (instead of $1025) on her account while $20 has been taken from Bob and $5 has been taken from Charlie. $5 have been lost for everybody in the confusion.

The aim of transactions if to prevent such a data loss situation to happen. The EJB container will help you to control when transaction starts and end. Please note that transactions will prevent these problems to happen to your DB in the DB. They will not protect you from concurrency problems on data in memory when no DB is envolved.

7.1.2. Unit of Work

A major mechanism in DBs to enable transactions is the record locking. It is possible to read a value and specify that no other transaction should read that value until the first transaction has released the lock. Such a lock is hold during a unit of work which is one transaction unit. The boundaries of a unit of work are defined with a begin transaction command and a commit command. Any SQL instruction between the begin transaction and the commit is part of the transaction.

Locking and units of work would be integrated in our example that way:

Source

```
Begin transaction 1:
Local variable 1 = Read Alice balance from DB and lock it. (= $1000)
Local variable 1 = Local variable + $20   (= $1020)

 Begin transaction 2:
```

```
 Local variable 2 = Read Alice balance from DB and try to lock it.
     ----> DB makes transaction 2 wait for record locked by transaction 1.

(back in transaction 1)
Write Local variable 1 as new balance for Alice in DB  (=$1020)
... Idem to remove $20 from Bob.
commit transaction 1 (and release the lock on Alice).

 (back in transaction 2)
 Local variable 2 = Read Alice balance from DB and lock it. (= $1020)
 Local variable 2 = Local variable 2 + $5   (= $1025)
 Write Local variable 2 as new balance for Alice in DB  (= $1025)
 ... Idem to remove $5 from Charlie.
 commit transaction 2 (and release the lock on Alice)
```

Our scenario has been saved by the fact that the transaction 2 has been placed in wait mode until transaction 1 finished. These two transactions have been isolated from each other thanks to the record locking mechanism.

A transaction that should be "cancelled" is *rolled back* instead of being committed. Once committed it's not cancellable anymore.

7.1.3. ACID Properties

Transactions must enforce ACID properties (Atomic, Consistent, Isolated, Durable).

Atomic	Atomicity guarantees that many operations are bundled together and appear as one contiguous unit of work (all or nothing). In our account example, it would ensure, that removing $20 from an account and adding $20 to another account in the same transaction, is either completely done, or not done at all. Atomicity guarantees that it will never be half done, even in case of severe hardware failure. Having it half done would be, for example, removing $20 form an account but not having had the time to add $20 to the other account.
Consistent	Consistency guarantees that a transaction will leave the system's state to be consistent after a transaction completes. In our account example, it means that the amount of an account in the DB will never be null if we defined a *not null* constraint on that column.
Isolated	Isolation protects concurrently executing transactions from seeing each other's incomplete results. In our account example, it means that the transaction adding $20 to an account, will not interfer with the simultaneous transaction adding $5 to the same account (the same record). There are different levels of interraction as we will see later.
Durable	Durability guarantees that updates to managed resources, such as database records, survive failures. Recoverable resources keep a transactional log for this purpose. In our example, it means that if any hardware fails right after the $20 transaction is committed for a total amount of $1020, then the amount must be $1020 when the system restarts (it may eventually implies to write some data from logs). It would indeed not be acceptable that the user got a message just before the

	failure, that the $20 transaction is successfully made, if the system could lose that transaction.

Most relational DB product have these ACID properties. They are enforced by the DB, not by the EJB server.

7.1.4. Isolation Levels

As we have seen, concurrency control is achieved with record locking on RDB. But locking may lead to deadlocks, meaning that two thread (transaction) are waiting for each other to release a lock. They would wait forever (until some time out resulting in an exception). Because of that, DBs have different isolation levels. The higher isolation level insures fewer problems but weaker performances and more deadlocks. A compromise needs to be done between transactions isolation and performance.

Weaker isolation levels introduce one or more of the following problems:

- **Dirty read problem**: A dirty read occurs when your application reads data from a database that has not been committed to permanent storage yet (and that may be rolled back later). With a low isolation level, the scenario would enable treansaction 2 to read the value of $1020 (after transaction 1 provided the new value but) before transaction 1 commits. If transaction 1 does not commit but rollsback, it will place $1000 back in the DB, but later, transaction 2 will place $1025 instead of $1005.
- **Unrepeatable read problem**: occur when a component reads some data from a database, but upon rereading the data, the data has been changed. This can arise when another concurrently executing transaction modifies the data being read. In our scenario transaction 2 requests to read the balance for update, telling the DB that it plans to update the value. Let's suppose that we have a third transaction that reads the value, but not for update. It reads the value for displaying it or for some computation and does not plan to write a new balance back on that record. That transaction would start before transaction 1 and 2. It would read $1000. After transactions 1 and 2 are finished, the third transaction would read the value again. If it reads $1025, it could not read the same record twice and obtain the same value.
- **Phantom read problem**: A phantom is a new set of data that magically appears in a database between two database read operations. Our scenario could not make a phantom read happen, we need another scenario. A transaction queries the DB to have all the accounts open these last 24h. Then, transaction 2 adds the account of Bob. Finally, transaction 1 reads again all the accounts open these last 24h (it may seem useless to read if twice but technically it could happen). If that second read within transaction 1 contains Bob's account, then we have a phantom read.

The following table lists the possible isolation levels and which isolation problems they produce:

READ_UNCOMMITTED	Yes	Yes	Yes
READ_COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Not all DBs support all isolation levels. The isolation level is usually set through the JDBC API with the method java.sql.Connection.SetTransactionIsolation(). But when the EJB Container manages transactions for you (as we'll see later in this lesson), there is no way to specify isolation levels in the deployment descriptor. You need to either use resource manager APIs (such as JDBC), or rely on your container's tools or database's tools to specify isolation. This lack in the EJB specification may change as the vendor community finds a consensus.

Most business applications require at least the READ_COMMITTED level. JPA requires at least READ_COMMITTED.

7.1.5. Transactions and Threads

The EJB container controls the thread creation, and the specification explicitely forbids you to manually create new threads. Each transaction is associated to a an execution thread. Two threads never share the same transaction.

Usually a transaction is short lived. For example, a client calls a method of the remote interface of a singleton bean (as AuditorBean) which calls many other beans and finally returns. Most transactions would begin and end within that timeframe. They are short lived. In the example of a 4 tier web application, the code in the web container may call an EJB. In most web applications, a transaction would not be longer than an HTTP request/response cycle. Programmers try to keep transactions short to reduce how long records are locked and to reduce the performance and deadlocks problems.

While it is avoided in most cases, it is technically possible to have long lived transactions that last a longer time, through extended persistence contexts that are out of scope of this course.

In the EJB container, transactions are managed by the *Transaction Manager* which communicates with the DB. Each thread has a javax.transaction.UserTransaction instance to hold the transaction state for that thread.

7.1.6. Transactional Services

The transaction manager can manage different kind of external transactional servers. Databases are one example. Messaging Severs (MOMs) are another example: a message send in a transaction would not really be sent until that transaction has been committed. JCA (Java Connector Architecture) enables the coupling of external transactional servers to the transaction manager.

Some external services are not transactional. This is the case of most SMTP (Simple Mail Transfer Protocol) e-mail servers. If your application sends an e-mail during a transaction it is sent immediately. If your transaction rolls back, the e-mail cannot be "cancelled".

The transaction manager can include multiple transactional servers into the same transaction. For example, you may update a record in an Oracle DB, insert a record in DB/2, and send a message with MQ-Series, inside the same transaction. If sending the message fails, the transaction manager will roll back the transaction and both DB changes will be rolled back. Under the covers, this is managed by the *two phase commit* protocol where the transaction managers acts as a director synchronizing transactional actors that do not know each other.

7.1.7. Activity: Quiz

Your business application makes use of DB transactions. At all cost, you want to avoid any *dirty read* problem, but you do not care about uprepeatable reads and phantom reads.

What isolation level do you require?

( ) READ_UNCOMMITTED

( ) READ_COMMITTED

( ) REPEATABLE_READ

( ) SERIALIZABLE

Solution: READ_COMMITTED.

Your application uses an Oracle DB, a MySQL DB and no MOM. If relational DBs manage transactions for decades, why would you need an EJB server to manage the transactions?

( ) for transactions involving both DBs.

( ) since you use no MOM, you don't need the EJB container to manage your transactions.

( ) because your version of MySQL does not support transactions

Explanation: The EJB container will provides you multiple helps for managing transactions. A main help is the declarative transaction demarcation programming model covered in the next topic. Another main point of help is the transaction monitor enabling transactions over multiple DBs and MOMs. If a single DB such as Oracle can manage transactions for its data, it will help you to include MySQL in a transaction. That transaction monitor helps you as soon as you have multiple transactional services involved in a transactions, are they DBc or MOMs. It's not because you have no MOM that the transaction monitor would be useless. If a DB, such as an old version of MySQL, does not support transaction, then the transaction monitor will not help you to make that fact change.

Solution: for transactions involving both DBs

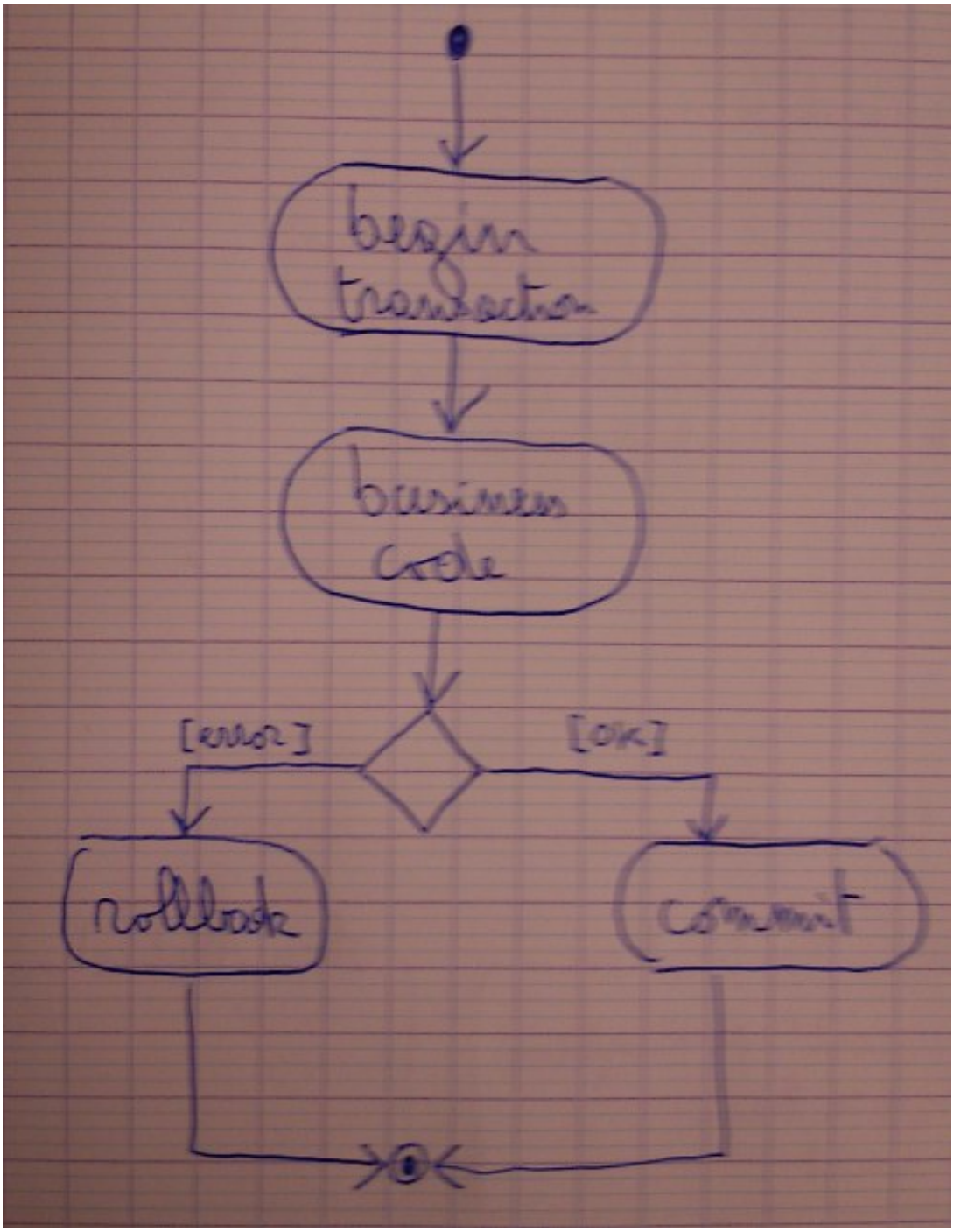
7.2. Declarative Transactions

Enterprise Java Beans were designed to support transactions in a very natural way, in fact, if you do not specify anything in your EJB's they will be transactional. That's because a lot of defaults are assumed in order make EJB's simple and robust. From now on we're going to talk about how to work transactions on Enterprise Java Beans declaratively.

Declarative transaction demarcation (opposed to programmatic transaction demarcation, next topic) is the easiest and preferred way to control transactions in an EJB application.

7.2.1. Transaction Control

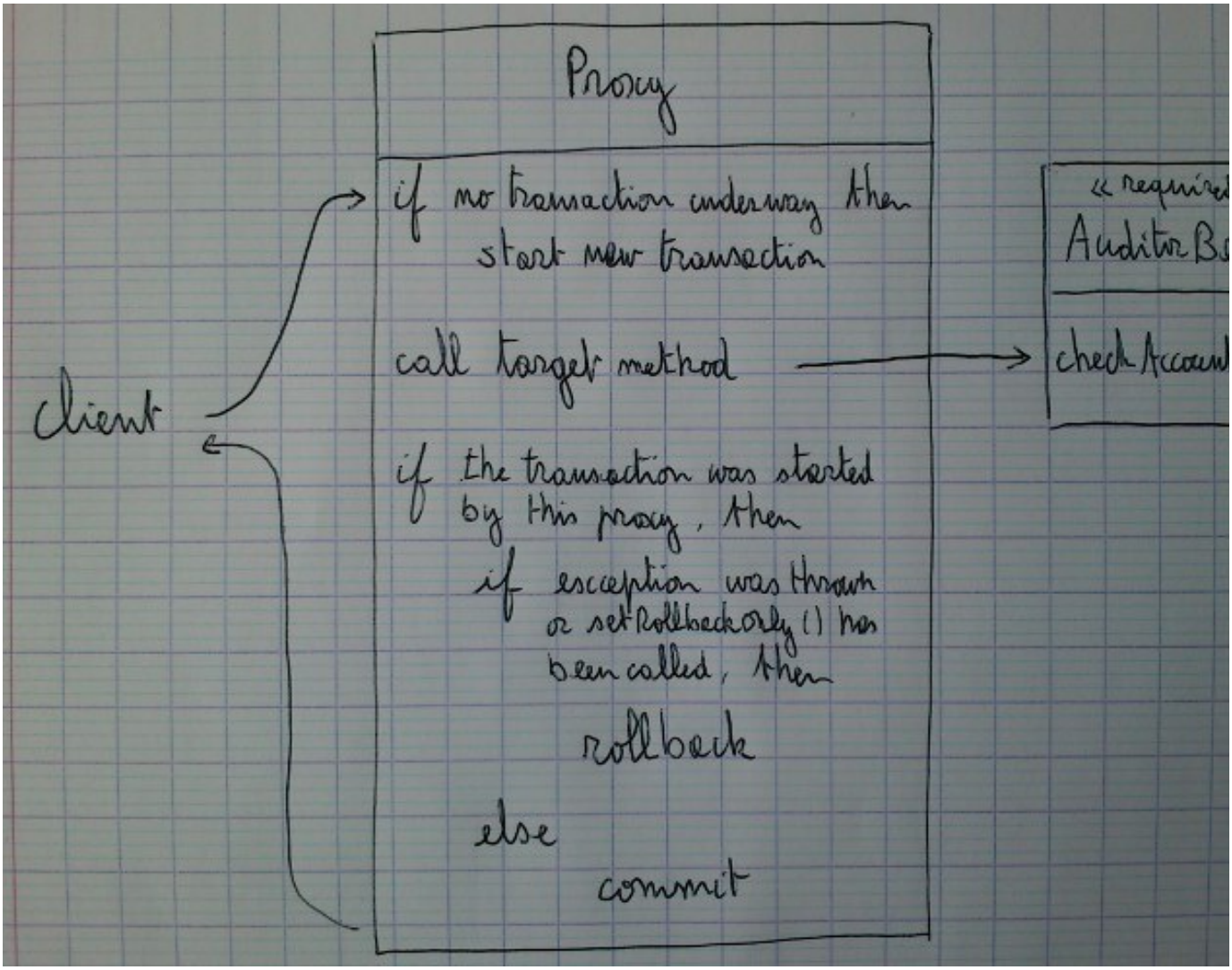
This activity diagram shows a high level view of the behavior of a transaction:



With BMT (Bean Managed Transactions), you explicitely call methods to begin, rollback or commit a transaction as you will see in the next topic. With CMT (Container Managed Transactions), the EJB container creates that code for you and you do not see it.

7.2.2. Proxies Controlling Transactions

With CMT, the transaction control code is in the proxy in front of your bean. If you bean business method needs to run inside a transaction, the proxy (created by the EJB container) will begin the transaction before forwarding the call to your business method, and commit that transaction when your method returns.



In EJB 3.1 all methods in session beans are transactional by default. Until you change the transaction setting, all methods of session beans are intercepted by the EJB container and declarative transaction with propagation *Required* is applied to them.

With BMT, the proxy will not interfer with transaction control.

7.2.3. Annotation Specifying Transaction

As you may imagine, transactions are propagated altogether with the thread of execution. But it can be finished if a exception is thrown, it also can be suspended depending on the transaction attribute specified. As shown in the example below, the @TransactionAttribute annotation enables you to change the transaction attribute of a method.

Source

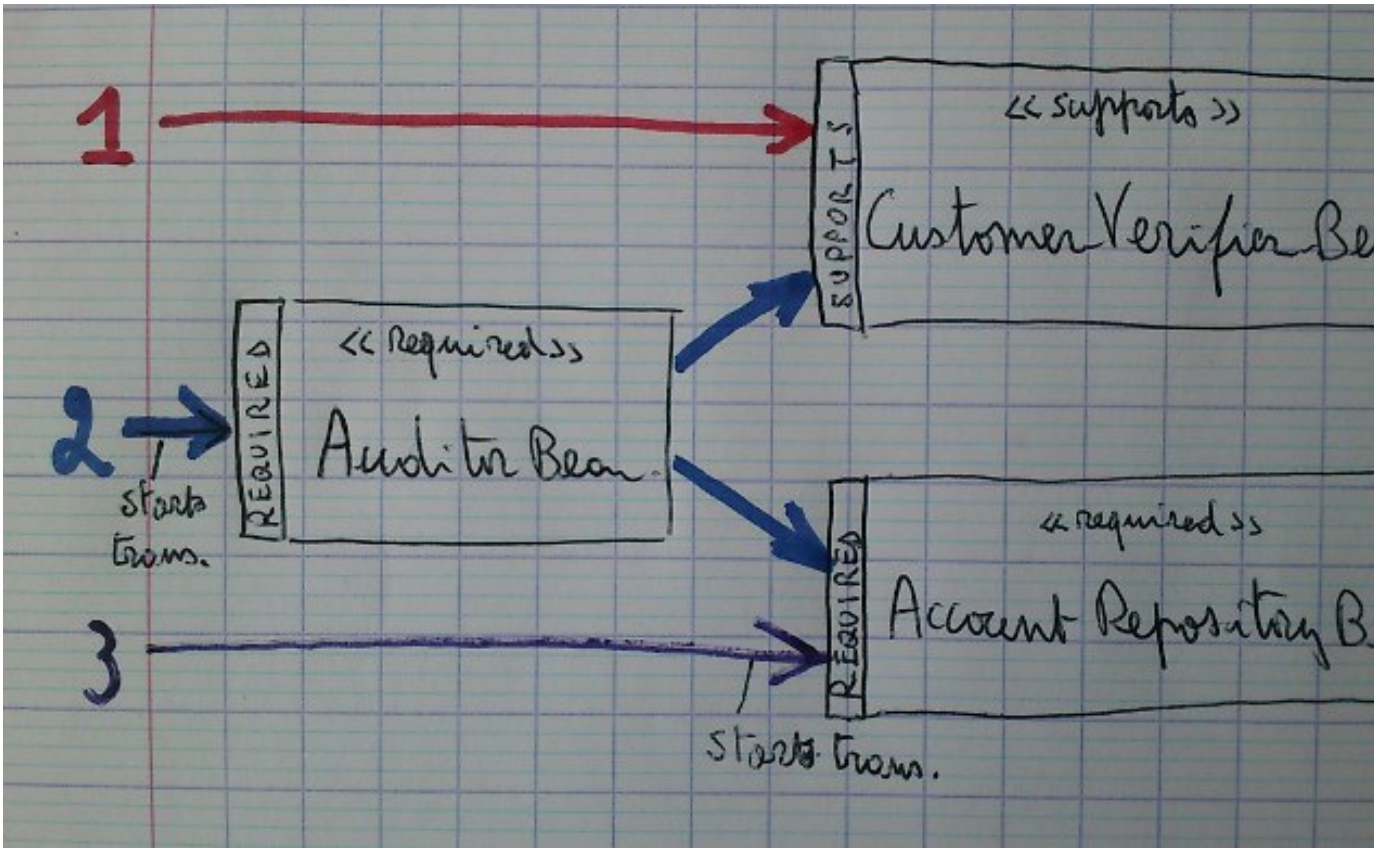
```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
class AuditorBean implements Auditor {
    ...

    @TransactionAttribute(TransactionAttributeType.NEVER)
    Account checkAccountStatus(String userName) {
        ...
    }
}
```

In the example above, we specify that no transaction should be started (by the proxy) when the checkAccountStatus() method is called. That annotation can be specified at the bean level to cover all its methods. Transaction attributes can also be specified through the deployment descriptor.

The EJB container will associate an instance of javax.transaction.UserTransaction to the thread and the proxies will use that object to begin, commit and rollback transactions.

7.2.4. Transaction Propagation

A transaction is bound to a thread of execution. With CMT, the container decides when to start/stop the transaction for the current thread, according to the settings of the beans it goes through. To visualize how transactions are propagated from one bean call to another, we should see a graph of beans:



Let's say that CustomerVerifierBean supports transactions. It is annoated with @TransactionAttribute(TransactionAttributeType.SUPPORTS). It means that if there is a transaction underway on the current thread which is going to execute its code, then it's fine. Else it is fine too, and no transaction needs to start because of CustomerVerifierBean.

AuditorBean requires a transaction. It is annotated with @TransactionAttribute(TransactionAttributeType.REQUIRED). It could be not annotaded since REQUIRED is the default. It means that if there is a transaction underway on the current thread which is going to execute its code, then it's fine. Else a new transaction should be created to ensure that AuditorBean's code is always executed within a transaction.

In the example above, the client directly calls CustomerVerifierBean. It is executed within the thread 1 (red arrow) without transaction. Another client calls AuditorBean. It is executed within the thread 2 (blue arrows) and a transaction is started. Thread 2 executes CustomerVerifierBean (called by AuditorBean) within that transaction. In this example CustomerVerifierBean is executed with no transaction or within a transaction according to the execution path.

AuditorBean also calls a method of AccountRepositoryBean which is in REQUIRED mode. When called from thread 2, it starts no transaction because it runs inside the transaction previously started by AuditorBean. If a third thread (purple line) with no transaction underway calls AccountRepositorybean directly, then AccountRepositoryBean would start a new transaction, which will be committed when the thread returns from its method.

This scenario is easy with declarative (annotation) transaction demarcation. Imagine that, on the contrary, you have to write the code, at the beginning of each method, to figure out if a transaction is already underway or if a new should be started. It is complex boilerplate code when many beans are collaborating.

7.2.5. TransactionAttributeType

The possible values for TransactionAttributeType are:

- **MANDATORY**: A client that calls a method marked with Mandatory is required to have a transaction context. If a client without a transaction calls a method marked with this attribute will receive an javax.ejb.EJBTransactionRequired. Use this value if you do not want the bean to create a new transaction, but if the bean should only be called by other beans having already started a transaction.
- **REQUIRED**: A method marked with Required will join an existent propagated transaction, if no transaction was propagated then the bean will create a new one, this new transaction will live and propagate altogether this method stack, it can finish with an exception or in the end of the method that created it. This is also the default if no annotation is used.
- **REQUIRES_NEW**: A method marked with RequiresNew will always create a new transaction. If the caller method have a transaction, it will be suspended until the end of the execution of the called method and the methods called by it. If the caller doesn't have a transaction, a new one will be created and then live for the execution of the called method, until it finishes or some exception is thrown.
- **SUPPORTS**: A method marked with Supports will execute in a transactional context if it's caller have a transaction associated, if the caller does not have a transaction it will also execute, but without a transaction.
- **NOT_SUPPORTED**: A method marked with NotSupported in the case of its caller having a transaction, the transaction will be suspended until it finishes, exceptionally or normally. A caller without a transaction will execute normally without a transaction.
- **NEVER**: A method marked with the Never attribute will not accept any transactional client to access it. If a client with a transaction associated calls a method with this attribute an javax.ejb.EJBException will be thrown. This attribute is used when no transaction is required and you want to ensure that your bean increase it's performance and availability (transaction is a heavy-weight feature and should be avoided if not really needed in order to increase performance and availability). Programmers use NEVER when they perform non transactional (no rollbackable) operation such as sending an e-mail, and they think that calling this bean within a transaction (implicitely assuming that the bean can participate) is a programming mistake from another programmer.

SUPPORTS and REQUIRED are the most common transaction attributes in business applications.

The only possible values for message driven beans are REQUIRED and NOT_SUPPORTED.

7.2.6. Rolling Back

You want to rollback the current transaction when you figure out that it is not possible to continue executing the code gracefully. It typically happens when your defensive code figures out an unexpected condition (detected a bug) as in the example below:

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
class AccountRepositoryBean {
   ...

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   Account getAccount(String userName) {
```

```
        if (userName == null || userName.trim().equals("")) {
      // Ooops, BUG! BUG! BUG!
  throw new IllegalArgumentExceptiont(
             "Bug: Programmers should never call this method "+
             "with a null or blank userName");
        }
        ... // Do the normal code to retreive the account.
    }
}
```

As you can see, if we figure out that the parameter of the getAccount() method is not valid, we are not going to remain silent and continue executing the code as if everything was fine. We detected a bug from the caller which is supposed to give a valid userName. We suspect that continuing executing the current thread could have undesired results (as invalid data in the DB). We prefer to "stop" the current thread by throwing a runtime exception.

Another way to rollback a transaction is to use the setRollbackOnly() method on the EJBContext object. You may want to use setRollbackOnly() (in the rare case) when you want to roolback but you also want your code to continue executing some process, such as sending an e-mail to a user to tell that the application could not complete the business operation he expected to be completed. Programmers perfer throwing a runtime exception over calling setRollbackOnly() because they usually don't need their code to continue executing when they decide to rollback.

7.2.7. Stateful Session Beans

A stateful session bean has some conversational state (values of its attributes) that needs to be rollbacked to previous values, together with a transaction rollback. Consider the stateful CalculatorBean example of the stateful session bean topic:

Source

```
@Stateful
public class CalculatorBean {
   int result;

   public int add(int op) {
       result += op;
 if (result > 99) {
     throw new RuntimeException("Too many digits... please rollback");
 }
       return result;
   }
   public void clear() {
       result = 0;
   }
}
```

The add() method has been modified to throw an exception if the result is above 99 (because our weak calculator cannot display 3 digits numbers). As you know, this will rollback any on-going transaction. We made no DB calls in CalculatorBean, but maybe the client (using CalculatorBean) did. That client EJB did start the transaction, made uncommitted DB changes, used CalculatorBean (which throws and exception). The EJB container will rollback the DB changes. The EJB container will **not** rollback the state of our stateless session bean, but it will help your bean to do it itself. Please note that the exception has been thrown after the result attribute has been incremented. The value before the increment should be restored.

In the code below, we have added two call-back methods, properly annotated. The afterBegin() method is called

when the transaction begins, to let you save any significant converstational attribute. Here we use another attribute to remember the result when the transaction begins. The afterCompletion() method is called when the transaction commits or rollbacks. If the boolean parameter indicates that we face a rollback, then we restore in result the value that we did save.

Source

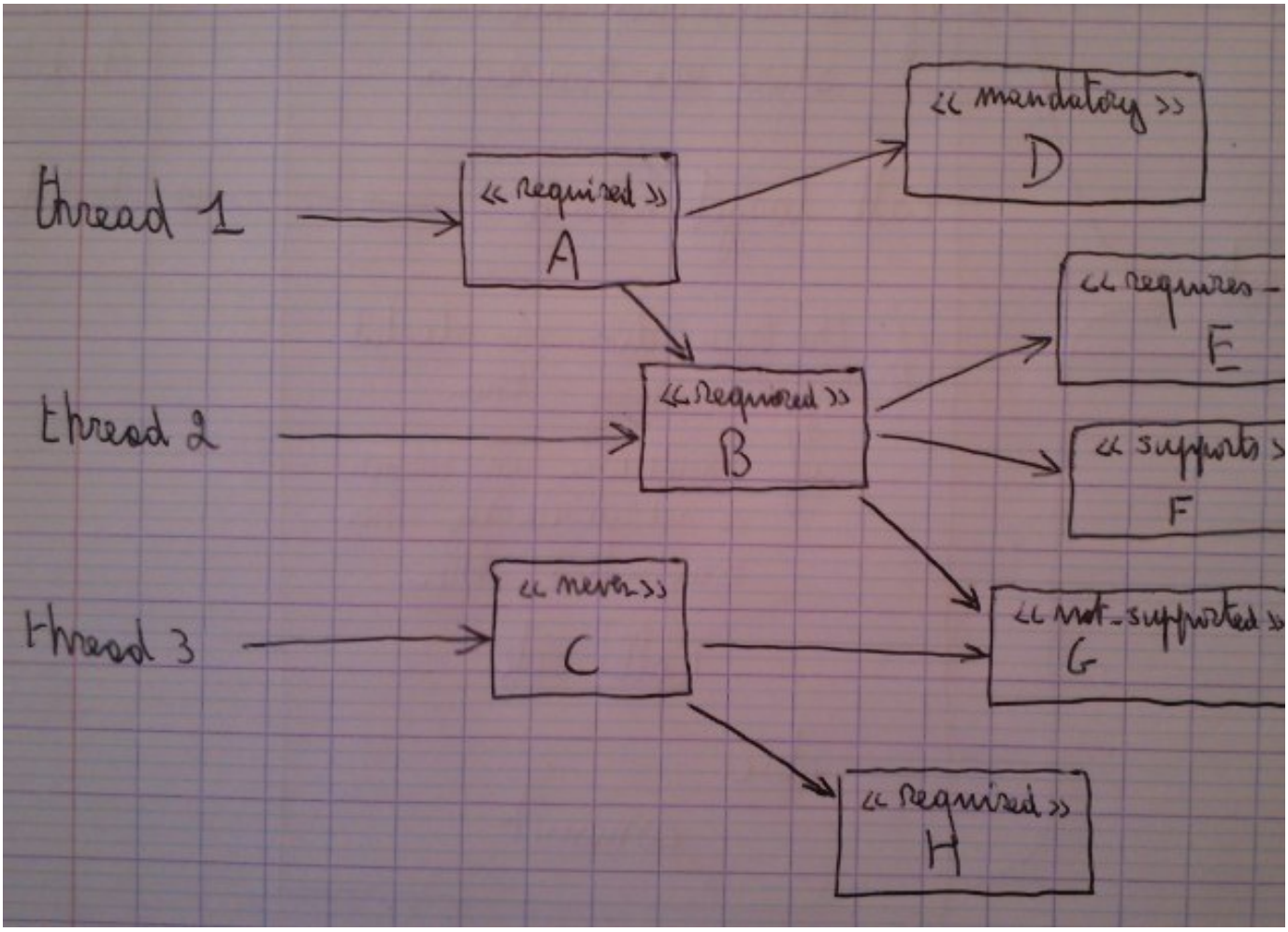
```
@Stateful
public class CalculatorBean {
   int result;
int resultWhenTransactionBegins;

   public int add(int op) {
       result += op;
 if (result > 99) {
     throw new RuntimeException("Too many digits... please rollback");
 }
       return result;
   }
   public void clear() {
       result = 0;
   }

   @AfterBegin
   public void afterBegin() {
       resultWhenTransactionBegins = result;  // Save current value
   }
   @AfterCompletion
   public void afterCompletion(boolean commit) {
       if (commit == false) { // Rollback
           result = resultWhenTransactionBegins;  // Restore prior value
       }
   }
}
```

A third annotation @BeforeComplete enables you to write a method called just before the container commit to give your bean a last opportunity to roll it back (with a RuntimeException, for example).

Alternatively, to using annotations, your stateful session bean can implement the SessionSynchronization interface that defines the three methods.

7.2.8. Activity: Understanding Propagation

You are presented the following bean diagram of your application. According to the transaction attributes, which of the following statements are true? There are exactly two true statements below:



[ ] During the thread 1 execution, the bean A, B and F run in the same transaction.

[ ] During the thread 2 execution, the transaction is suspended.

[ ] During thread 2 execution, an exception thrown because bean G NON_SUPPORTED is called by bean B REQUIRED.

[ ] During the thread 3 execution, an exception is raised by the container because a bean with NEVER (bean C) cannot call a bean with REQUIRED (bean H).

Explanation:

Choice 1: A starts the transaction, which is reused by B. F does not care about transaction and will join the existing one. It will not be the case of E which will start a new transaction, but this is not covered by this choice.

Choice 2: B starts a transaction which will be suspended twice. First during the execution of E. The main transaction is suspended during the new transaction made by E executes. Second during the execution of G which does not support transactions.

Choice 3: A call from REQUIRED to NON_SUPPORTED triggers no exception, but the thread is suspended in bean G.

Choice 4: A bean with NEVER cannot be called with a transaction on-going. But it can itself call transactional beans as a REQUIRED one. The transaction will be active during the execution of H.

Solution: First and second choices are correct, the two others are wrong.

7.3. Programatic Transactions

Instead of using annotations or the deployment descriptor to identify the transactional beans to the EJB container, it is possible to call methods on UserTransaction explicitely from your code to control the units of work. That kind of transaction control is rarely preferred to the declarative transaction demarcation with annotations, but it may help you to manage some obscure corner cases.

In this topic, you will disable the container transaction management for your bean, inject a UserTransaction instance, and call commit() and rollback() on it.

7.3.1. Switching to BMT

By default, the beans use CMT (Container Managed Transaction). To use BMT (Bean Managed Transaction), you need to annotate your bean with @TransactionManagement(BEAN).

Source

```
@TransactionManagement(BEAN)
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
class AccountRepositoryBean {
    ...
}
```

7.3.2. Obtaining UserTransaction

With dependency injection, you can obtain a reference to UserTransaction very easily. In the code below, the container detects the @Resource annotation and knows that UserTransaction is a special object bound to the current thread by the container, and will provide a reference to it.

Source

```
@TransactionManagement(BEAN)
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
class AccountRepositoryBean {

    @Resource
    UserTransaction userTransaction;

    ...

    Account getAccount(String userName) {
        try {
            userTransaction.begin();

            ... // Do the normal code to retreive the account.

            userTransaction.commit();

        catch(RuntimeException rte) {
            userTransaction.rollback();
            throw rte;
        }
    }
}
```

The getAccount() method controls the transaction. It's template is very typical and most BMT work the same way. It starts it and performs some business operation, that may involve calling other beans. Then it commits the transaction unless a RuntimeException happens. In that case it rolls back the transaction and let the exception continue its path.

7.3.3. Using UserTransaction

The UserTransaction class provides the following methods:

- **begin()**: Starts a new transaction.
- **commit()**: Commits the current transaction. If the current transaction has already been rollbacked, the commit() method throws a RollbackException.
- **setRollbackOnly()**: Marks the current transaction as to be rollbacked as soon as possible by the container. As you have seen in the previous topic, you use this method with container managed transaction.
- **getStatus()**: Returns an int with the status of the transaction. javax.transaction.Status defines constants returned by this method, including STATUS_ACTIVE, STATUS_COMMITTED, STATUS_MARKED_ROLLBACK, STATUS_NO_TRANSACTION, STATUS_ROLLEDBACK.

As for CMT, these methods only affect the current thread, since the transaction is bound to the thread with BMT as well.

Note that for stateful session beans, it is forbidden to start a transaction in a method, and commit/rollback it in another conversational method. The begin and end of a transaction need to happen in the same method.

8. Scheduled and Asynchronous Processing

In business application, it is usual to have batch jobs to execute regularly. It can be, for example, statistical data collection and computation that cannot be done in real-time. I it also usually some cleanup task to delete or archive old data from the DB.

The Timer Service enables EJBs to be executed as specific times. It is possible for all the beans except stateful session beans. The timing can be configured through programming or configuration.

Timers enable you to execute code in another thread. Asynchronous session beans also, but immediately, with no delay. Optionnally asynchronous beans enable the new thread to return some result to the initial thread.

8.1. Declarative Timer

Declarative timers are defined in the EJB container when the application is deployed. They are suitable when the programmer (or the deployer) knows in advance when the task must run. It is the case, for example, if a bean responsible to paying the employees must be executed every month the first day of the month. With a declarative timer, you would annotate the method of that bean with the @Schedule annotation to specify that the EJB container must automatically start it once a month. Another example is a batch job running everyday at midnight.

Declarative timers are kind of triggers in the web container that will call a bean's method. They are configured with the @Schedule annotation on the method or with the `<timer>`/`<schedule>` deployment descriptor elements to specify the timer/date/frequency of execution.

8.1.1. Example of Declarative Timer

These timers are easily configured with the @Schedule annotation, which takes some time definition as parameter. In the example below, we annotate the collectStats() method to be run at 4AM everyday, when the activity on the system is (supposed to be) low.

Source

```
@Singleton  @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean {

    ...

    @Schedule(hour="4")
    public void collectStats() {
        ... // business logic to collect statistics once a day.
    }

}
```

A job that needs to be executed the first day of each month at midnight, would be annotated as such:

8.1.2. @Schedule Annotation

The @Sechedule annotation has many attributes but they are very intuitive:

Attribute	Values	Default Value
second	0 to 59	0
minute	0 to 49	0
hour	0 to 23	0
dayOfMonth	1 to 31	*
month	1 to 12 or "Jan" to "Dec"	*
dayOfWeek	0 to 6 or "Sun" to "Sat"	*
year	4 digits year (such as "2012")	*
expression	cron scheduling expression	

Source

```
@Schedule(second="0", minute="0", hour="0", dayOfMonth="1", month="*", year="*")
```

The following expression is a shortened (cron-style) version with the same meaning:

Source

```
@Schedule(expression="0 0 0 1 * * *")
```

8.2. Programmed Timer

With programmed timer, your Java code registers the bean to the timerService by calling the createTimer() method. This way of doing enables your program to decide at run-time when the task must be executed by the timer service. For example, it may make sense that a confirmation mail is sent to a customer one hour after some reply have been received from the stock. You cannot say that the mail should be sent every day at 10AM. It depends when you will get replies from the stock, which you cannot predict when you program the application.

8.2.1. Mechanism

Programmed timers work with the TimerService interface obtained from EJBContext.getTimerService().

The programmer calls one of the TimerService.create...Timer() method to specify when to execute the timer.

On the same bean having created the timer, a method should be annotated with @Timeout, which will be the method executed asynchronously by the timer.

8.2.2. Example of Programmed Timer

Source

```
@Singleton  @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class StockManagerBean {

   @Resource
   SessionContext sessionContext;

   public void receiveReplyFromStock(Long customerId) {
       ... // Some business logic.
       TimerService timerService = sessionContext.getTimerService();
       timerService.createTimer(60*60*1000,  // 1 hour
     customerId);  // Store info to be retreived by sendMail
   }

   @Timeout      // Called 1h receiveReplyFromStock()
   public void sendMail(Timer timer) {
       Long customerId = (Long)timer.getInfo();
       ... // send mail to that customer.
   }

}
```

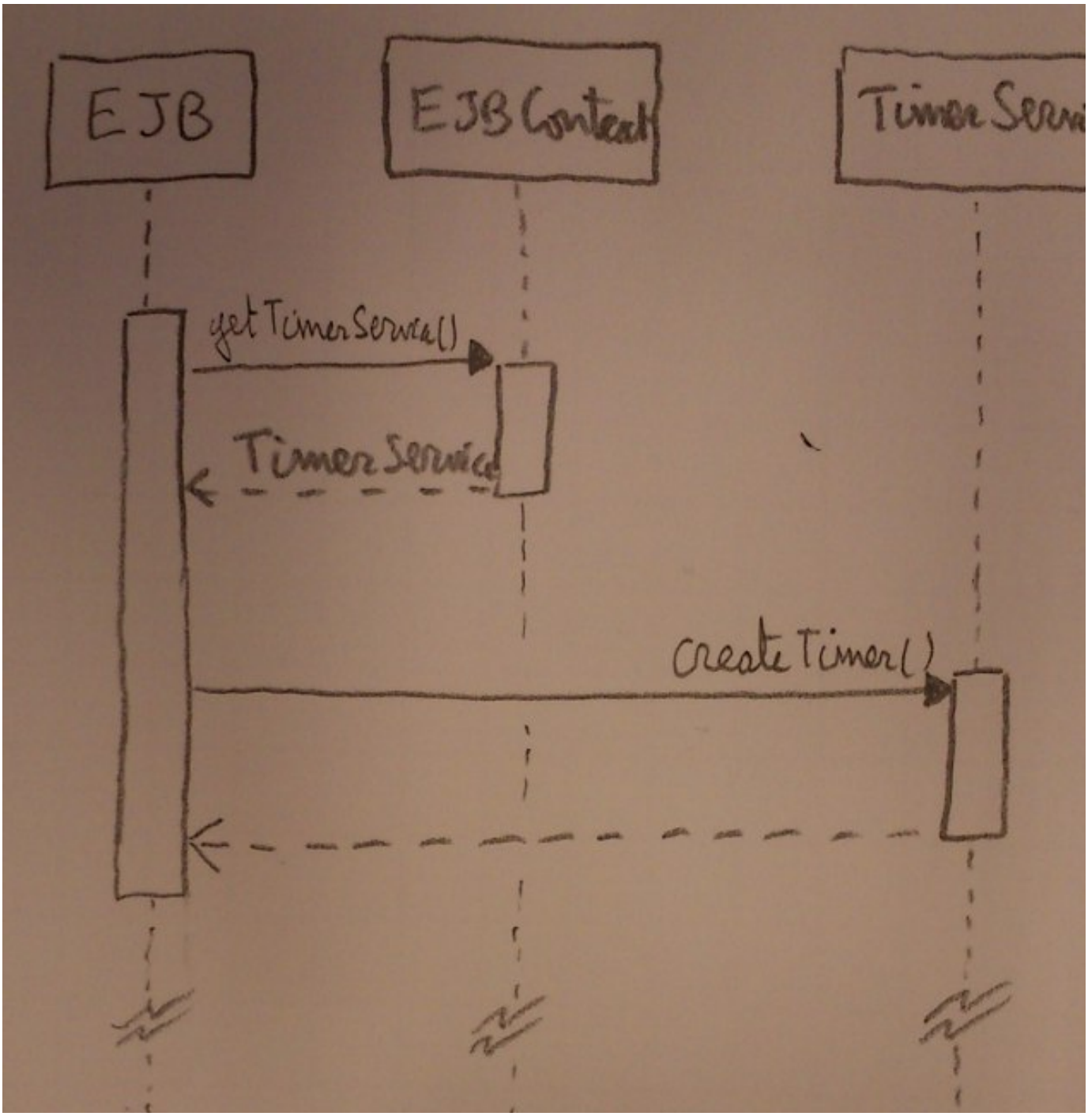
In the code above, the receiveReplyFromStock() method is called by the application, probably after some user interraction confirming that the ordered item is in stock. It retreives an instance of the javax.ejb.TimerService interface through the EJBContext object (injected with @Resource). Then it creates a timer, specifying a delay and some application data (customerId) needed by the delayed method.

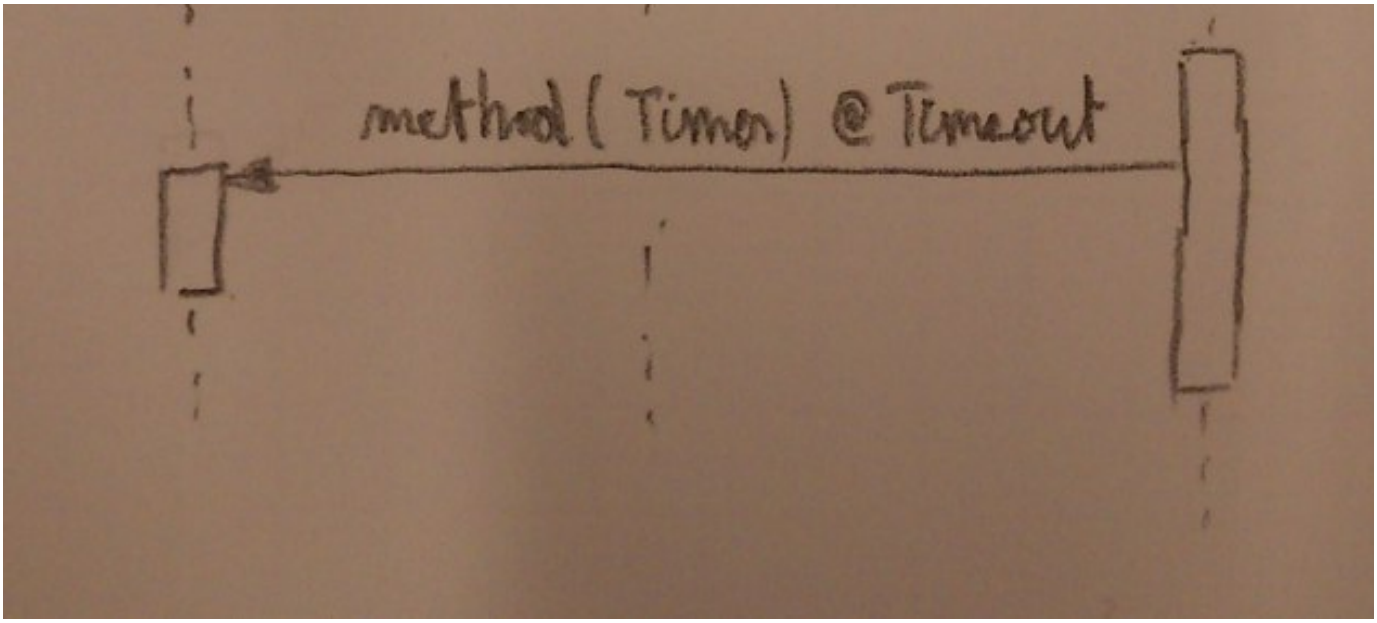
One hour later, TimerService identifies which method to call thanks to the @Timeout annotation on sendMail(). That method can either have no parameter, either take a javax.ejb.Timer instance. In our case, we need that instance to get the customerId that was passed at the timer's creation, in order to know to who to send the mail.

TimerService provides multiple create method, to create timers that execute once or repetitively. As you have seen in the sample, you can optionnaly provide some data that can be recovered by the @Timeout method, as long as that data is serializable. Indeed, it is not kept in the heap only because the server could crash and restart before the @Timeout method is called.

8.2.3. Sequence Diagram

In the sequence diagram below, you retreive the execution story of the example above. The bean method receiveReplyFromStock() executes and





8.2.4. TimerService and Timer

The TimerService interface provides various timer creation methods, some based on a specific date, some on an interval. It also provides the getTimers() method that returns a collection of all the active Timer instances.

The Timer interface (returned by the TimerService create...() methods), encapsulates the information about a defined timer. It contains the following methods for which the JavaDoc is below:

- **cancel()**: Cause the timer and all its associated expiration notifications to be cancelled.
- **TimerHandle getHandle()**: Get a serializable handle to the timer.
- **java.io.Serializable getInfo()**: Get the information associated with the timer at the time of creation.
- **java.util.Date getNextTimeout()**: Get the point in time at which the next timer expiration is scheduled to occur.
- **ScheduleExpression getSchedule()**: Get the schedule expression corresponding to this timer.
- **long getTimeRemaining()**: Get the number of milliseconds that will elapse before the next scheduled timer expiration.
- **boolean isCalendarTimer()**: Return whether this timer is a calendar-based timer.
- **boolean isPersistent()**: Return whether this timer has persistent semantics.

8.2.5. Activity: TimerService

When a customer confirms his interest for a loan, he gets an e-mail with details immediately. The marketing department would like that 30 minutes later, the customer gets a second mail with some promotion for optional insurances about the loan.

In this activity, you create a singleton session bean bean LoanServiceBean, with a method to confirm a loan that uses a TimerService to start another method 30 minutes later to send the mail. You will simulate sending e-mails by displaying some text at the server's console.

1. Start from any working EJB project, such as the HelloWorld project.
2. Create a singleton bean LoanServiceBean.
3. In that bean, create the confirmLoan(Long contractNumber) method to confirm the loan. That method first prints a text "loan confirmed" at the console. Then it creates a timer that starts in 30 minutes. For the test, make it 10 seconds. That timer gets the contract number as parameter.
4. Still in LoanServiceBean, create a method sendMarketingMail() that takes the Timer as parameter. It is annotated with @Timeout. It prints the contract number at the console.

5. Make LoanServiceBean.confirmLoan() a remote method and call it from the client program.
6. Test your code. The contract number should display 10 seconds after the "loan confirmed" print out.

8.3. Asynchronous Session Beans

Sometime, you need an EJB to start the execution of long operation and during that execution you would like to continue your processing without waiting for the result of the long operation. In a Java SE application you would start a new thread the acheiving that. But the EJB specification forbids any manual thread creation by your code within the EJB container.

Asynchronous methods, annotated with @Asynchronous, enables you to start such long operations without having to wait for their completions.

8.3.1. Asynchronous Methods

Since the EJB 3.1, you can call methods asynchronously. The @Asynchronous method can be placed at the class level or on a single method level, either from the bean class or its interface. Placing the annotation at the class level makes all the methods asynchronous.

A method called asynchronously means that another thread is created (by the container) for executing that method.

When you start an asynchronous method you either run it in fire-and-forget mode, either in retrieve-result-later mode. In the first case, you expect no result, and the method signature returns void. In the second case, you expect to retreive a result later and the method signature returns `Future<?>`

8.3.2. Asynchronous Methods Returning No Result

The AccountancyBean below has a method that takes several minutes to complete, it closes the fiscal year. It is annotated with @Asynchronous.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountancyBean {

    ...

    @Asynchronous
    public void closeCurrentYear() {
        ... // Long operation to finalize the accountancy.
    }

}
```

The code below, from any other bean, calls closeCurrentYear():

Source

```
accountancyBean.closeCurrentYear();
System.out.println("Closing started");
```

The println will not wait for closeCurrentYear() to return. The call to closeCurrentYear() method returns immediately and the println executes while AccountancyBean.closerCurrentYear() executes in another method.

8.3.3. Asynchronus Methods Returning Future

Your asynchronous method may have some result to return. But the caller method cannot get the result immediately as the call will return before the asynchronous method has finished executing. The EJB specification has a mechanism for the proxy to return you a handle to the future result, of type java.util.concurrent.Future.

When the asynchronous method will be finished, then Future.isDone() will be true, and the client could call Future.get() to get the result returned by the asynchonous method. The first code example below is using an asynchronous method defined in the second code example further. In the first code below, some client calls AccountancyBean.countBadAccounts(). The proxy of AccountBean returns an instance of Future which does not contain the result yet. Calling future.isDon() immediately after the call returns false because the asynchronous method has not finished executing yet and the result is not ready. When Future.isDone() returns true, then the result is ready and can be obtained through Future.get().

Source

```
Future future = accountancyBean.countBadAccounts();

while (!future.isDone()){ // While result is not ready
   Thread.sleep(1000); // Simulate doing something
}

Integer result = (Integer)future.get();  // Get the result.
```

This way of doing is useful if the client has something else interesting to do while waiting for the result, else there is no point making the call asynchronous: the thread of the client could have done the long operation itself. A typical "interesting" action for the client thread is to tell the end-user (through the UI) that the operation is progressing with some regular visual feedback.

The code below contains the asynchonous method countBadAccount() that returns an Integer (encapsulated in a Future). At the end, it returns the int in a new AsyncResult instance. AsyncResult implements Future. The client already has another instance of Future (created by the proxy). AsyncResult will automatically fill the instance of the client with the result and mark it as done.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountancyBean {

   ...

   @Asynchronous
   public Future<Integer> countBadAccounts() {
       int n;
       ... // Long operation to compute n;
       return new AsyncResult<Integer>(n);
   }
```

```
}
```

8.3.4. Activity: Remote Asynchronous Method

You have been asked to implement an AccountancyBean to count bad accounts, as shown above. It must have an asynchronous method. Your job is to test the asynchronicity, another developer will develop the business logic later to count bad accounts. You been given the estimation that the business logic will probably last around 5 seconds.

1. Start form any working EJB project, such as the HelloWorld project or the first banking application with AuditorBean.
2. Create the AccountancyBean as singleton.
3. Add a remote interface Accountancy to it, with a countBadAccounts() method returning a Future.
4. Make AccountancyBean implement that interface and method. The method waits for 5 seconds (use Thread.sleep(5000)) and returns always the same number. Annotate that method as asynchronous.
5. In the Java SE client main method, lookup that session bean and call the countBadAccount() method.
6. After having called that method enter in a loop until the result is ready. In the loop, wait for 1 second and display "Still computing ...". When the result is ready, get out of the loop and display the result.
7. Test your code.

9. Security

It is possible to secure a Java EE application with your own code. It is also possible to rely on security features provided by the EJB container. It provides both programmatic and declarative security to manage users and roles. With declarative security, the container will perform the verifications for you, according to the annotations you have put on your classes. With programmatic security, you will write that code yourself, using the APIs from the EJB container.

In this lesson, you will see the different authorization methods. You will see, for example, how to make a method executable only by managers with declarative authorization. You will also see how to use programmatic authorization to return only the accounts that the current user may see.

9.1. Authentication

Authentication is the process of figuring out who is accessing your application, and to prove it usually with a password. In this topic, we define the notions of user, group and role. Then we briefly see how users can authenticate (provide their name and password). Finally, you will see how to use the @RunAs annotation to force the usage of a specific role.

9.1.1. Realm, Users, Groups, and Roles

The Java EE application server (web and EJB containers) uses the notion of user, user group, and user role to define the security context of the application. A user is any individual who accesses a web application. You must define the accessibility of the application to users. You may do this by defining user privileges and authentication mechanisms such as a user name and password. A group is a set of users who have similar access rights in common. A role is a specific function that a user or group can perform in the web application.

In our banking example, the possible users, groups, and roles are:

- Users: Alice, Bob, John.
- Groups: Administrators
- Roles: Manager (can access all accounts), Cashier (can make financial transactions).

It is common to hesitate between creating a group or a role. Sometimes, the difference between these two notions is less. Technically, a group is global for the application server. Maybe this application server is hosting multiple EJB applications. A role is local to an EJB application. For example, the bank has two web applications: one for consumers and one for administrators. The following would be only valid within the administrator EJB application and would be roles: manager and cashier. John, for example, has the *manager* role. John has access to both applications. In the consumer ejb application, the notion of *manager* does not exist and your code from the consumer EJB application would not see that role. But if John is part of a group (such a s *Administrators*), the fact is visible from both EJB applications.

A realm is a "database" of usernames and passwords. It identifies valid users and the list of roles associated with each valid user. A realm concerns one or multiple Java EE applications.

9.1.2. Web Container Authentication

Often, the main client of an EJB container is the code running in a web container in the same application server. In the web container, the servlet specification provides some authentication mechanisms to get the user and password of the user (which are out of scope of this course). Once the user is identified in the web container, it can be automatically propagated and reused in the EJB container.

9.1.3. Programmatic Authentication

If you have a remote client that has its own login mechanism and want to propagate the credentials (user identity) to the EJB container when doing remote calls, you need to use programmatic authentication. Unfortunately, that mechanism has not been standardized and is specific to every EJB container product. It is not part of the EJB specification and not covered in this course.

9.1.4. @RunAs

You can hardcode the role on which behalve a specific EJB should run. This is, for example, useful for a @Scheduled bean that starts spontaneously. A scheduled method is started by no user, but by the EJB container. Therefore no user is attached to the thread and it may be convenient to use @RunAs to associate a role to the thread for passing authrization check points in beans called from the scheduled method.

In the code below, the scheduled collectStats() method will run as an manager (a user having the role "manager"). It will call the method isMoneyDirty() that can only be called by managers because of the @AllowRoles annotation as you will see in the next topic. Because our collectStats has the @RunAs("manager") annotation, we are sure that the current thread has the "manager" role and the call to isDirtyMoney() will be authorized by the EJB container.

Source

```
@Singleton  @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AuditorBean {

    ...
```

```
    @Schedule(hour="4")
    @RunAs("manager")
    public void collectStats() {
        ... // business logic to collect statistics once a day.
 boolean b = isMoneyDirty(account);
 ...
    }

@AllowRoles("manager")
    public boolean isMoneyDirty(Account account) {
        ... // business logic to figure out if money on account is dirty
    }

}
```

9.2. Authorization

Authorization is figuring out if the user logged in can do some action or access some resource. You can specify it through annotations with declarative authorization which is more convenient and used most of the time, it is enough.

Sometimes, you need to test corner cases, you need to write Java code and use programmatic authorization. For example, a method could return more or fewer data according to the role of user requesting that data. As another example, some users may only access the application during busiess hours.

9.2.1. Where to Authorize

Authorization is always at least done at the UI level. Indeed, you would not propose a link or show a screen/page that the user is not authorized to see. In case of breach in the UI security, you may want to put other security check points inside your business logic. For example, if the method AccountRepositoryBean.deleteAccount() is supposed to never be called, but by an administrator of the application, then you may want to verify the role of the current user just before executing that method and throw an exception if it is not an administrator. If a user finds a breach in the UI security (i.e. URL hacking), it will be stopped at the EJB AccountRepositoryBean.deleteAccount() method level.

You have two ways to verify the role of a user before executing a method:

- declarative authorization,
- programmatic authorization.

9.2.2. Declarative Authorization

Declarative authorization is the easiest and most used compared to programmatic authorization. You annotate your methods to specify the roles that are allowed to execute them, and the proxy will perform the verification.

In the exampble below, you specify that the "manager" role can execute the deleteAccount() method. A good practice would be to put the "manager" string litteral in a constant static variable.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountRepositoryBean {

    ...

    @RolesAllowed("manager")
    public void deleteAccount(long accountNumber) {
        ... // Deletion of the account from the DB.
    }

}
```

If a thread associated to a user not having the "manager" role tries to execute that method, the proxy will throw a javax.ejb.EJBAccessException.

You can specify multiple roles in the annotation, as for example a manager and a cashier:
`@RolesAllowed({"manager", "cashier"})`

The @PermitAll and @DenyAll annotations enables you to give access to everybody or nobody. These 3 annotations can be defined at the bean class level to cover all the methods.

The @DenyAll annotation will make your method useless and if you have access to the code, you better delete or comment the method out. If you don't have access to the source code because you got the bean from a third party, you may want to deny all access to a method through the deployment description as an alternative to altering the source code.

In the example below, we declare AdminBean. Then, in the `<assembly-descriptor>` element, the `<exclude-list>` element enables us to define methods to be excluded, here the hardDeleteProduct() method. It would have had the same effect as using the @Denyall annotation on the method.

Source

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1">
 <enterprise-beans>
   <session>
     <ejb-name>AdminBean</ejb-name>
   </session>
 </enterprise-beans>
 <assembly-descriptor>
   <exclude-list>
     <method>
       <ejb-name>AdminBean</ejb-name>
       <method-name>hardDeleteProduct</method-name>
     </method>
   </exclude-list>
 </assembly-descriptor>
</ejb-jar>
```

9.2.3. Programmatic Authorization

Usually, declarative authorization with annotation is enough. In some more rare cases, you need finer control. For example, you may want to authorize a specific role to execute a method only during business hours. You cannot configure that with annotations, you need some API to know if the current user has a specific role.

The EJBContext object provide two methods that will help you figure out on who's behalf the code is being executed.

- **isCallerInRole()**: It takes a role name as parameter and returns true if the user associated to the execution context has that role.
- **getCallerPrincipal()**: It returns the javax.security.Principal object of the user associated to the execution context. That object is made by the container after authentication and reused for each call of that user. It has a getName() method to find the user name. The Principal class is part of the JAAS API (Java Authorization and Authentication Services) from Java SE.

The code below uses isCallerInRole() to verify if the current user has the "manager" role, because only admins are authorized to delete protected accounts.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountRepositoryBean {

   @Resource
   SessionContext sessionContext;

   ...

   public void deleteAccount(long accountNumber) {
       List<Long> protextedAccountNumbers = ...
       if (protectedAccountNumber.contains(accountNumber)) {
           if (! sessionContext.isCallerInRole("manager") {
               throw new RuntimeException("Only managers "+
                           may delete protected accounts");
           }
       }
       ... // Deletion of the account from the DB.
   }

}
```

Another example would for a method to return more data if the executing user has a specific role. In AccountRepositoryBean, for example, it could be a method named getAccountsThatCurrentUserMaySee() and that would restrict the list of returned account to those managed by the current user. The code below shows such a method. After the name of the user has been retreived, it uses it as a parameter to restrict the results of a JPA query.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class AccountRepositoryBean {

   @Resource
   SessionContext sessionContext;

   ...

   public List<Account> getAccountsThatCurrentUserMaySee() {
       Principal currentUser = sessionContext.getCallerPrincipal();
       String useName = currentUser.getName();
       return entityManager.createQuery("select a from Account a "+
```

```
                    "where a.manager = :managerName")
            .setParameter("managerName", userName)
            .getResultList();
    }

}
```

9.2.4. Activity

What annotation can be used on a method to enable the users having the role "cashier" to execute it? Check all the correct answers.

[ ] no annotation

[ ] @Authorize("cashier")

[ ] @Authorize({"cashier"})

[ ] @RolesAllowed("cashier")

[ ] @AllowAll

[ ] @DenyAll

Explanation: A method having no annotation (on the method and via the class) will by default authorize all roles. The @Authorize and @AlloAll annotations do not exist (at least in the EJB specification). @PermitAll exists but is not in the list of choices for this question. @DenyAll prevent any role to execute the method.

Solution: no annotation, @RolesAllowed("cashier")

A method enables you to check if a user has a given role. What is the name of this method and in which class is it defined? Select the correct answer.

( ) EJBContext.isCallerInRole(String)

( ) EJBContext.getCallerRoles()

( ) EJBContext.getCallerPrincipal()

( ) Principal.isCallerInRole(String)

( ) Principal.getCallerRoles()

( ) Principal.getCallerPrincipal()

Explanation: The method getCallerRoles() does not exist. The other methods are defined in EJBContext and its descendants such as SessionContext, not in Principal.

Solution: first choice.

10. Best Practices

Note for Chennai: I let you add add an intro.

10.1. Exception Handling

So far, we did not pay much attention to exceptions. In this topic, we'll see which exception can be thrown for which circumstances, and how the client and container should react.

The exception mechanism is a core part of Java. It enables applications to manage exceptional conditions gracefully, including bugs, DB connection problems and incorrect method parameter values. The EJB container will

help you to handle exceptions thrown by you or other frameworks used by your application. The EJB container will pay special attention to continuing or terminating transactions according to the exception involved.

10.1.1. Checked and Unchecked Exceptions

Java SE defines two kind of exceptions with different behaviors: check exception and uncheck exceptions.

Unchecked exception extend the RuntimeException class. All other exceptions are checked exception and extend the Exception class (but not RuntimeException). Checked exceptions foces the programmer to add them in the method signature (throws clause) of the method that could throw it. The other method that calls the method throwing an exception, needs to either do the same, either put the call in a try/catch block. Therefore, you use checked exception only when you want to force the calling code to handle the exception.

Usually, most of your exceptions are unchecked exceptions, because nothing can be done. The thread needs to be aborted, or the exception should go through all the application layers up to the UI (to display an error message).

Some older Java API, such as JDBC, throw checked exceptions (such as SQLExcpetion) for unrecoverable errors. You need to nest them in a RuntimeException as in the example below (EJBException extends RuntimeException).

Source

```
try {
   ... // JDBC call such as connection.createStatement(...)
} catch (SQLException sqle) {
   throw new EJBException("Impossible to load the invoices.", sqle);
}
```

10.1.2. Influencing Transactions with Exceptions

The EJB container will react differently with a checked or unchecked exception regarding transaction control. By default, it:

- Continues the transaction when an checked exception is thrown.
- Rollbacks the transaction when an unchecked exception or a RemoteException (which is checked) is thrown.

You can change that behavior with the @ApplicationException annotation and corresponding `<application-`

exception> deployment descriptor element. In the example below, we define a business exception. As it directly extends Exception, it is checked and by default, would not cause a transaction rollback. The @ApplicationException annotation changes it to make the EJB container rollback the transaction when a method returns because of that exception.

Source

```
@ApplicationException(rollback=true)
public class InvalidInvoiceException extends Exception {
   private Long invoiceNumber;
   public InvalidInvoiceException(Long invoiceNumberParam, String text) {
       super(text);
       invoiceNumber = invoiceNumberParam;
   }
}
```

The method below throws that exception (if the creation date of the invoice is null). That method is defined in an EJB.

Source

```
public boolean isInvoiceLate(Invoice invoice) throws InvalidInvoiceException {
   if (invoice.getCreationDate() == null) {
       throw new InvalidInvoiceException(invoice.getNumber(),
           "Bug: Creation date should not be null at this point.");
}

....
}
```

10.1.3. Stateful Session Beans

When a bean throws an unchecked exception or RemoteException, the EJB container will destroy that bean instance. It has no much consequences for stateless session beans and message driven beans that are pooled and have other reusable instances. Singleton session beans are re-instantiated. But stateful session beans are more problematic to that regard becasue they hold some conversational state and the client probably expected to reuse that specific instance to continue the conversation. If, after an unchecked (or remote) exception, the client a statefull session beans tries to use the bean again, it gets a NoSuchEJBException.

10.1.4. Exception Consequences

When an exception happens it has the following consequences:

- If a transaction is on-going, then it is automatically rolled-back.
- If the client having called the exception-throwing method did start the transaction, it gets an unchecked EJBTransactionRolledBackException.
- If that client did not start the transaction, it gets and EJBExcetpion.
- The bean instance having thrown the exception is discarded.
- Unchecked and RemoteException are logged by the server.

10.2. Java EE Design Patterns

At the beginning of the previous decade, the Java EE platform was lower level than now and not as covenient to use. Because of that, the need for giving architecture guidelines to developers was especially strong. The notion of *J2EE Design Patterns* emerged. It is a catalogue of design patterns adapted to Java EE applications. A design pattern is a best practice solution to a commonly recurring problem on which we put a name.

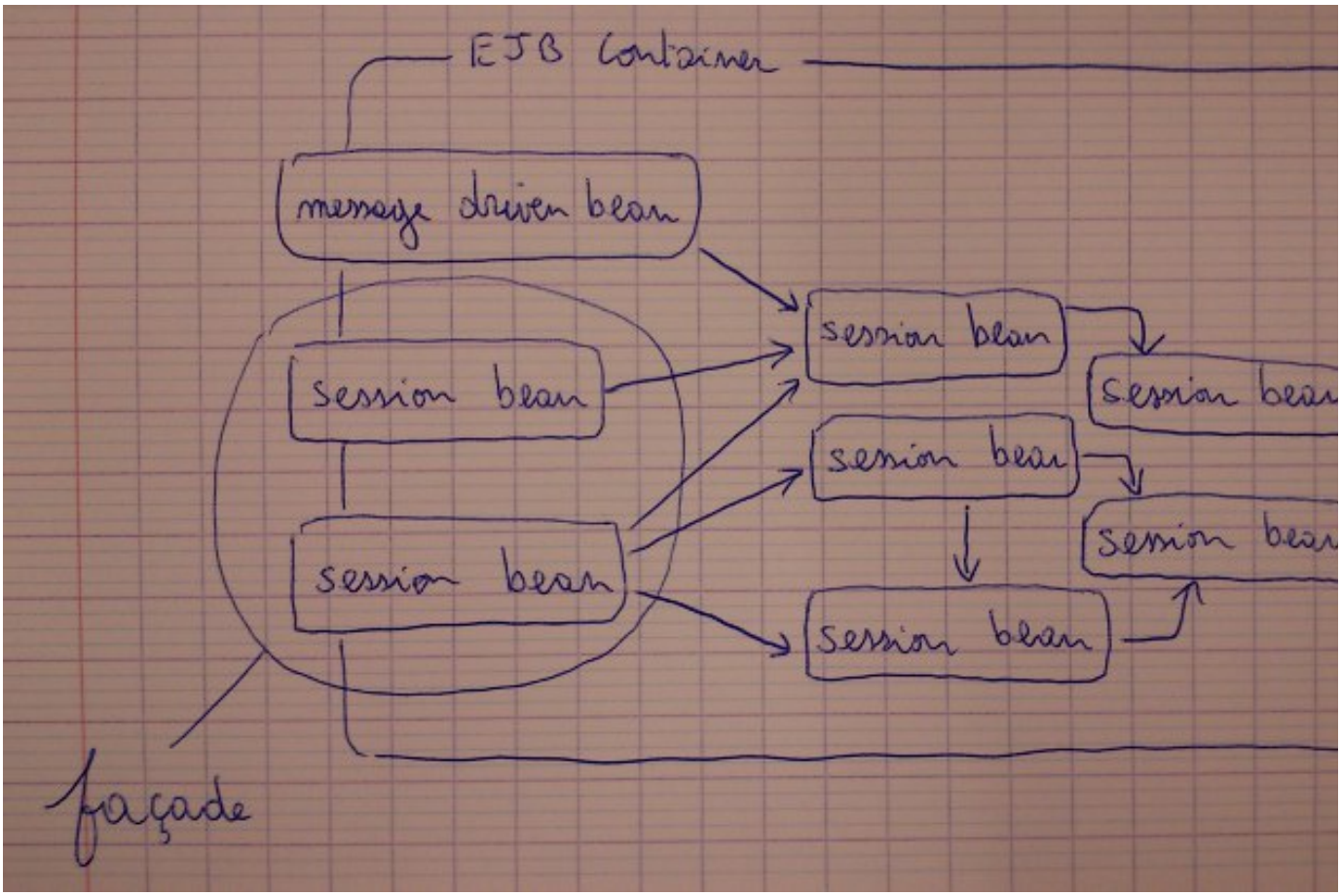
These design patterns are known by many Java EE developers and using their name facilitates architectural meetings and documentation.

Many (if not most) of these design patterns did compensate the architectural holes and mistakes of the Java EE platform itself. Since Java EE v5, they are much less meaningful. When they don't make sense anymore, we don't explain them extensively but explain what other mechanism/technology replaces them. The original J2EE design patterns are devided into two groups: web container and EJB container. In this topic we only talk about the EJB one.

10.2.1. Session Facade

A session facade is a set of session beans with a remote interface which serves as communication gateway between the clients and the other behind that facade. The API of session beans in the facade is coarse grained to minimize the amound of remote method calls.

It's some kind of API for the client to use your EJB application.



In the diagram above, the 3 session beans having a remote interface are part of the session facade. The message driven bean is not not part of the session facade because it is not a session bean.

10.2.2. Service Locator

A service locator is a utility class doing JNDI lookups to find other EJBs or resources (such as a JMS queue). This code is not needed anymore since its usage is replaced by dependency injection. For example, if AuditorBean needs a reference to AccountRepositoryBean. It could create an InitialContext and call the loopkup() method on it. Instead of doing that, it is much easier to inject AccountRepositoryBean into AuditorBean with the @EJB annotation.

10.2.3. Transfer Object

A transfer object, or *transfer data object*, is a class encapsulating data transferred between EJBs and their remote clients. It is for example the Account, Invoice, User and Product classes. In the first EJB specifications, entity beans were heavyweight objects that could not leave the EJB container. Nowadays, since Java EE 5 (EJB v3.0) they are replaced by JPA (Java Persistency API). In JPA, the entity beans can be detached and leave the EJB container to be transferred to the client. There is no need to create additionnal transfer classes.

Another pattern, the *Transfer Object Assembler*, is not used anymore because it is heavily based on the notion of transfer objecT. In short, the transfer object assembler was reponsible of instantiating and filling transfer objects.

10.2.4. Composite Entity

Before Java EE 5, JPA and Hibernate, EJB entity beans were heavy technical objects. Because they were heavy programmers tried to limit their number and made coarse-grained entities. A coarse-grained entity contains more data than a fine grained. For example putting more in a Customer entity, including it's address makes it coarse grained, compared to having the address in a separate entity Address, which would be fine grained. A composite entity did model a set of interrelated persistent objects rather than representing them as indivisual fine-graned entity beans.

But now, JPA enables us to have fine grained entities and this pattens is no best practice anymore.

10.2.5. Data Access Object

The DAO (Data Access Object) is one of the best known Java EE design patterns, at such a point that most application using a DB have classes which name end by "Dao" (such as "AccountDao"). An alternative and more modern name is "Repository" ("AccountRepository").

The DAO class contains the DB access logic for persisting and retreiving entities. In business applications, it is typical to create one DAO per entity. For example, if you need an *Invoice* entity, you will create an *InvoiceDao* class for grouping the code to persist/query invoices.

Source

```
@Singleton @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class InvoiceDao {
   @PersistenceContext
EntityMagnager em;

   public void create(Invoice invoice) {
       em.persist(invoice);
   }
```

```
    public Invoice find(long id) {
        em.find(Invoice.class, id);
    }

    public List<Invoice> findUnpaidInvoices(Date since) {
        return (List<Invoice>)em.createQuery(
                "select i from Invoice i where i.paid=false")
            .getResultList();
    }

    ...
}
```

Thanks to JPA, the code for doing simple operations (find/persist) is minimal. With JPA, the DAOs mostly contain queries. Programmers do one method per query, such as *findUnpaidInvoices(Date since)* or *getAccountsForCustomer(Customer)*.

In an EJB container, a DAO is a session beans, usually a singleton.

10.2.6. Service Activator

A service activator reveives asynchronous client requests and messages. They were the way activate business services asynchronously.

Nowadays, message driven beans ans Session beans asynchronous methods are used for that purpose and service activators are not needed at all anymore.

11. Chaining singleton beans

Sometimes one singleton has to be initialized before the others. For example beans from the business layers (such as AuditorBean) may need to have a DAO (Data Access Object layer) such as AccountRepositoryBean, to be initialized first. It is rarely needed to instruct the EJB container explicitely to instantiate a bean before another. To indicate that given singleton requires some other singleton beans to be initialized before it, we can use the @DependsOn annotation. The latter annotation takes list of singleton beans' names that should be initialized before the annotated class.

For example to indicate that AccountRepositoryBean should be initialized before the AuditorBean use the following construction:

Source

```
@Singleton
public class AccountRepositoryBean {
 // irrelevant code ommited
}

@Singleton
@DependsOn("AccountRepositoryBean")
public class AuditorBean {
 // irrelevant code ommited
}
```

We can also specify more than single dependency:

Source

```
@Singleton
@DependsOn("CacheProviderBean", "AccountRepositoryBean")
public class AuditorBean {
 // irrelevant code ommited
}
```

Please note that dependencies located in different JAR files need to be prefixed with the name of the containing JAR archive. For example if AccountRepositoryBean is located in the **data.jar** and AuditorBean in **business.jar** then dependency of DAO could look like the following code:

Source

```
@Singleton
public class AccountRepositoryBean {
 // irrelevant code ommited
}

@Singleton
@DependsOn("data.jar#AccountRepositoryBean")
public class AuditorBean {
 // irrelevant code ommited
}
```

Please note however that hardcoding jar name in the class metadata (annotations) is very bad practice and breaks the maintainability of the code. The jar name is related to the deployment-level configuration and could change. After that change the old (and invalid) name of the jar file will remain in the bean class.

12. @AccessTimeout