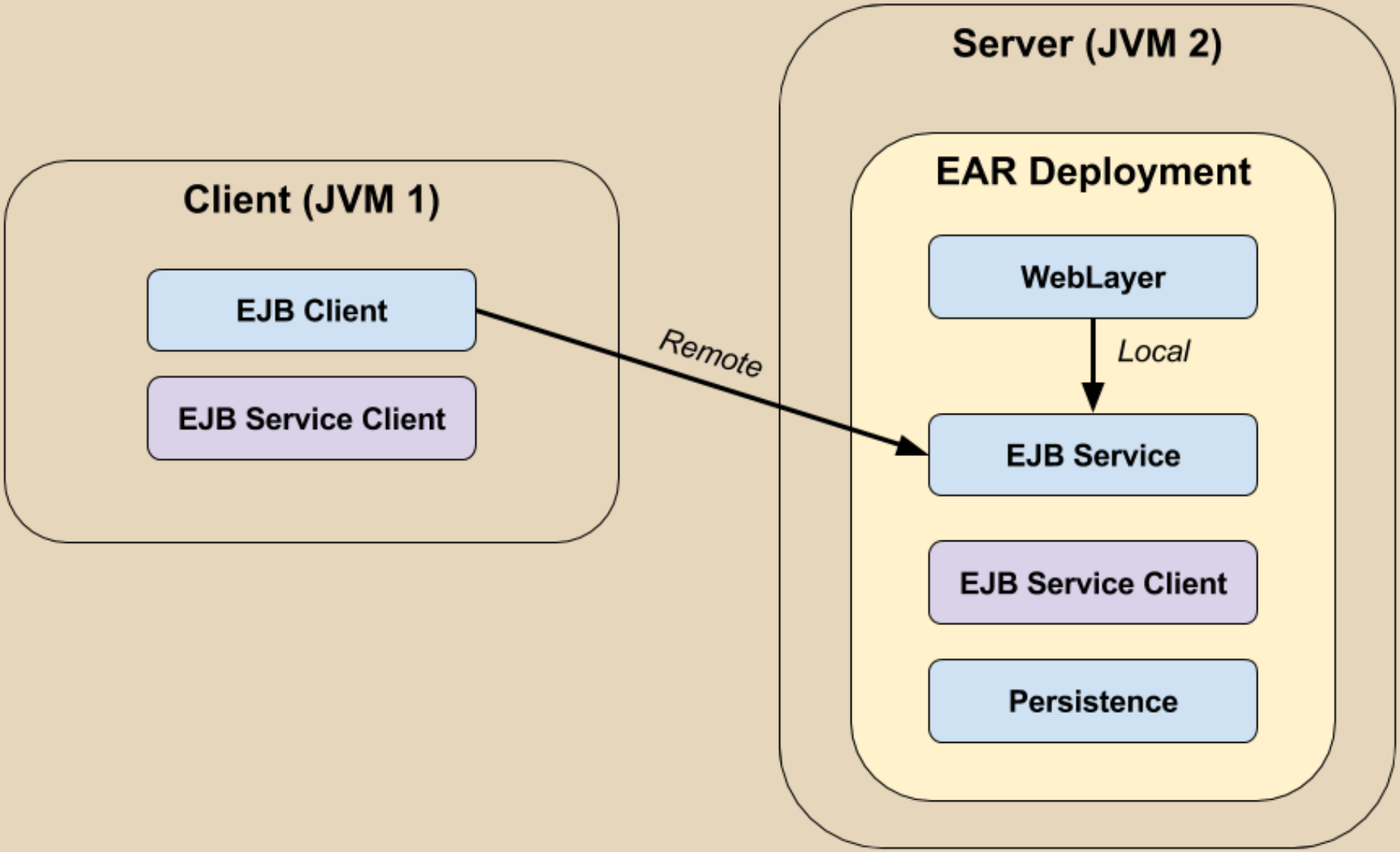
DiskStore #gradle

Remote EJB, JNDI, Dependency management, Service client, Context root, InitialContext, MyBatis 3

Óbuda University, Java Enterprise Edition
John von Neumann Faculty of Informatics
Lab 4

Dávid Bedők
2018-01-17
v1.1

Local vs Remote EJB



Prerequisites

What do we need to call an EJB service remotely?

- We have to know the location of the *server* (ip address/host, port)
- We have to know the type of the *server*, possibly some configuration environment (application server, jndi rules, ...)
- We have to know the meta/descriptor data of the *enterprise* application (application name, ejb module name, ...)
- We have to know the signature of the remote *service* and we have to load the classes which are used (serializable arguments, return values and exceptions, and all other related types)

Because of the strong source code dependency (**EJB Service Client**) on the client side, most of the time the client and the server side development are located at the same team/company (the source code of the client is part of the product). To achive this we need to **consider the packaging** of the enterprise application.

JNDI

Java Naming and Directory Interface

- With JNDI the distributed application can access and get services and registered resources in an abstract and resource-independent way.
- It maintains a set of objects tipically in a directory/tree structure
- It ensures the item registration, searching and retrieval
- It is able to notify a client if an item was modified (event sending)
- Usually the key is a `String` instance but it can be anything which implements the `Name` interface
- Usage:
  - Set the configuration parameters (jndi.properties or Hashtable<String, String>)
    - The `host` name and the `port` of the server and the communication `protocol` have to be set
    - We have to set the *full qualified* name of the `InitialContextFactory` class (this class has to be loaded to the *classpath*)
    - Other, e.g.: authentication or server specific configurations
  - Creating `InitialContext` through the server specific factory
  - With the `lookup( [JNDI name] )` method of the `javax.naming.Context` instance we can get the resources what we are looking for

JNDI usage

Remote: the configuration needs to be provided (it could be via `jndi.properties` file as well)

```
1 Hashtable<String, String> jndiProperties = new Hashtable<>();
2 jndiProperties.put("java.naming.factory.initial",
      "org.jboss.naming.remote.client.InitialContextFactory");
3 jndiProperties.put("java.naming.provider.url",
      "remote://localhost:4447");
4 Context context = new InitialContext(jndiProperties);
5 context.lookup("...");
```

These values are JBoss 6.4 specific, the referred `InitialContextFactory` class needs to be on the *classpath*. We can use constants instead of the `String` keys:
`javax.naming.Context.INITIAL_CONTEXT_FACTORY`
`javax.naming.Context.PROVIDER_URL`

Local: the configuration is provided by the container (the approriate `jndi.properties` file has already loaded from the claspath)

```
1 Context context = new InitialContext();
2 context.lookup("...");
```

JNDI name

[context]/[application-name]/[module-name]/[bean-name]![full-qualified-interface-name]

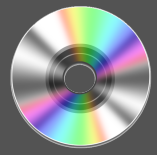
- **context**
  - JavaEE standard
    - `java:comp`: accessible for the given component (inside *ejb*)
    - `java:module`: accessible for the given module (inside *ejb module*)
    - `java:app`: accessible for the application (inside *ear*)
    - `java:global`: accessible for the given application server domain (inside *standalone domain*)
  - JBoss specific
    - `java:jboss/exported`: accessible outside the container
- **application-name**
  - It can be configured by the *deployment descriptor* of the *ear* (`application.xml`).
- **module-name**
  - It can be configured by the *deployment descriptor* of the *EJB module* (`ejb-jar.xml`).
- **bean-name**
  - It can be configured by the `name` value of the `@Stateless( name = "[BEAN_NAME]")` annotation.

Technical details of the prerequisites

- The **client library** of the server which helps us the communicate to the specific JNDI tree of the application server (protocols, server specific configuration settings, ...). (**JBoss client library / WebLogic (full)client jar**).
- The used **EJB implementation** of the application server (the client code will run without an *enterprise container*). The Java EE 6.0 API will not be enough, we have to add (the entire) implementation of the JBoss/WebLogic's Java EE API into the *classpath*.
- The related types will transport across the network during the remote communication, so we have to care about the **serialization/deserialization** of these types (most time we just need to implement the `Serializable` interface in Java and add the `transient` keyword to the non-serializable fields).

DiskStore

CRUD operations above disks



Task: create a **disk store** application which is very similar to the previous 'BookStore'. Implement EJB services to handle CRUD operations.

- The database schema may be the same as the schema in the 'BookStore' project.
- The weblayer may be very similar than the web pages in the 'BookStore' project.
- Do not use JPA in the persistence layer. Use a simpler but *type-safe* ORM solution instead with the help of **MyBatis 3** 3rd party library.
- We will able to remotely retrieve the data of a disk by its unique reference (**remote ejb call**).

Project structure

Subprojects, modules

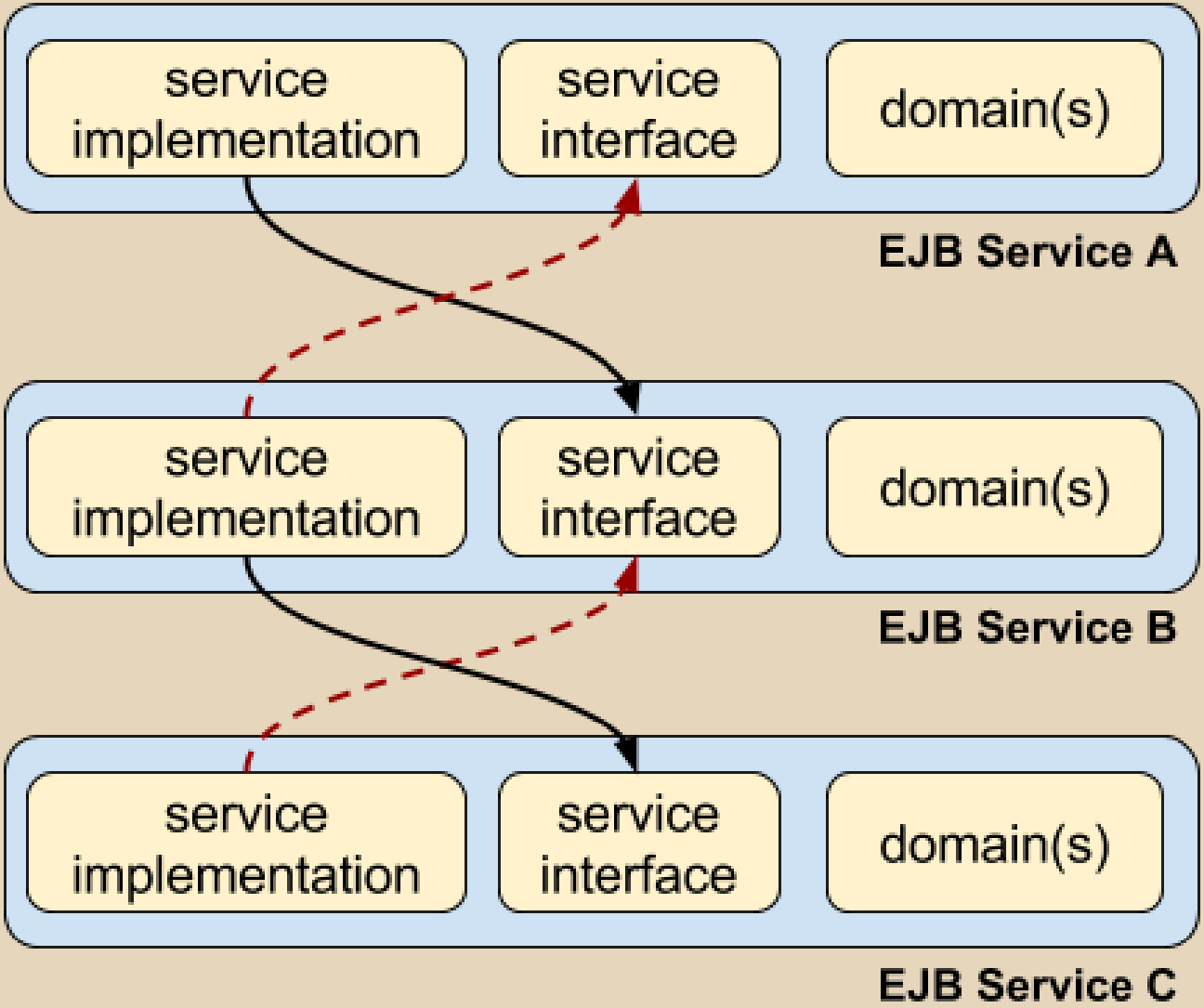
- **diskstore** (root project)
  - **ds-weblayer** (EAR web module)
    - `DiskPingServlet`
    - Servlets and JSPs according to MVC pattern
  - **ds-ejbservice** (EAR ejb module)
    - Implementation of the business methods
    - Local EJB interface
  - **ds-persistence** (EAR ejb module)
    - ORM layer (MyBatis 3)
  - **ds-ejbserviceclient** (EAR library)
    - Serializable domain classes (stubs), exceptions
    - Remote EJB interface
  - **ds-client** (EJB client application)

Part of the EAR: project + project

Part of the EJB client application: project + project

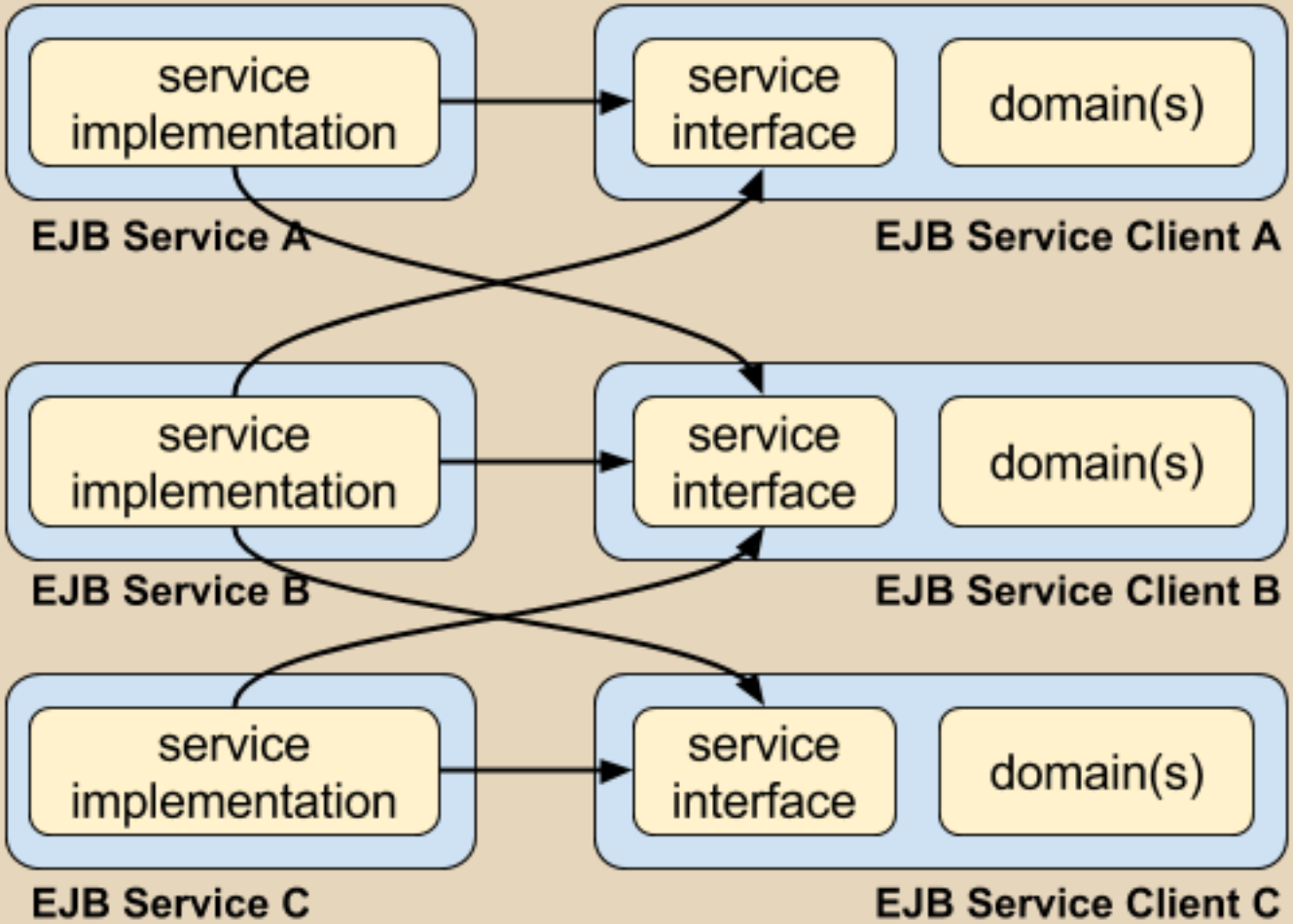
Circle dependency among EJB Modules

Not working solution



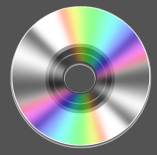
Circle dependency among EJB Modules

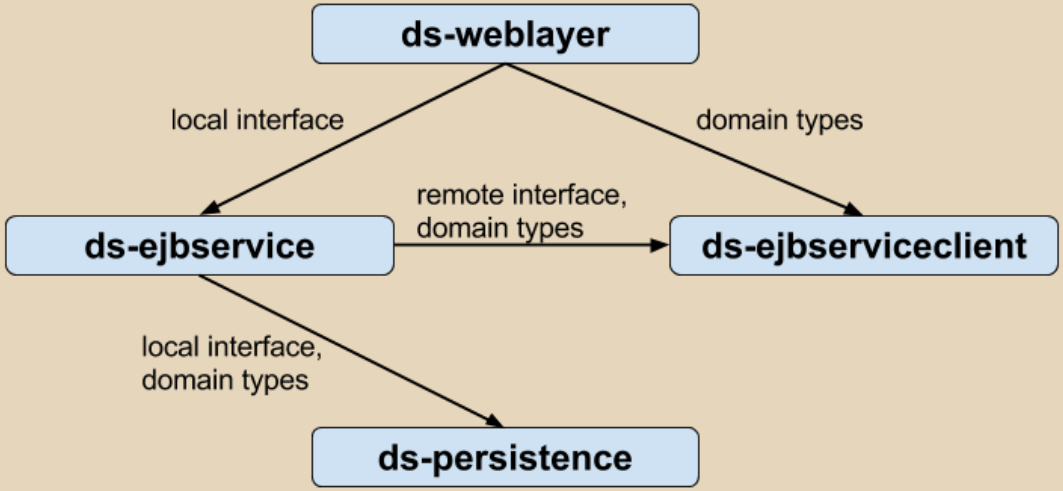
Working solution (eliminate circles)



Dependencies amoung modules

DiskStore project





Disassembling the `ds-ejbservice` into two parts (itself and `ds-ejbserviceclient` projects) was important because of the remote EJB call. In general this 'trick' is good to avoid dependency circles. (It will not cause failure in practice if we put the entire `ds-ejbservice` project's artifact into the client classpath, but it would be a serious principle mistake.) The `@Local` EJBs remain in the `ds-ejbservice` module (localy we can perform the entire CRUD operations, but remotely we just have a query possibility). The `ds-weblayer` project will depend from the `ds-ejbservice` beside the `ds-ejbserviceclient` project. The business operations of the `ds-ejbservice` will use the persistence layer, like in the *BookStore* project, so the dependency chain will be the same in that side.

Databse schema

disk and diskcategory

```
1  CREATE TABLE diskcategory (
2    diskcategory_id INTEGER NOT NULL,
3    diskcategory_name CHARACTER VARYING(100) NOT NULL,
4    CONSTRAINT PK_DISKCATEGORY_ID PRIMARY KEY (diskcategory_id)
5  );
6
7  CREATE TABLE disk (
8    disk_id SERIAL NOT NULL,
9    disk_reference CHARACTER VARYING(100) NOT NULL,
10   disk_author CHARACTER VARYING(100) NOT NULL,
11   disk_title CHARACTER VARYING(100) NOT NULL,
12   disk_diskcategory_id INTEGER NOT NULL,
13   disk_price REAL NOT NULL,
14   disk_number_of_songs INTEGER NOT NULL,
15   CONSTRAINT PK_DISK_ID PRIMARY KEY (disk_id),
16   CONSTRAINT FK_DISK_DISKCATEGORY FOREIGN KEY (disk_diskcategory_id)
17     REFERENCES diskcategory (diskcategory_id) MATCH SIMPLE ON UPDATE RESTRICT ON DELETE
           RESTRICT
18 );
```

create-schema.sql

EJB Service client

Types which are important both the client and the server sides

- Domain classes
  - `DiskStub`
  - `DiskCategoryStub`
    - the `Enum` superclass `implements Serializable`, no additional task to do
- Exception class
  - `ServiceException`
    - the `Exception` superclass `implements Serializable`, no additional task to do
- Remote interface
  - `DiskFacadeRemote`

EJB Service client

Build script

```
1 jar { archiveName 'ds-ejbservice-client.jar' }
2
3 dependencies {
4   compile group: 'org.jboss.spec', name: 'jboss-javaee-6.0', version:
        '3.0.3.Final'
5 }
```

build.gradle

At *Compile time* we only need the Java EE 6.0 API or implementation because of the `@Remote` annotation, but at *Run time* we need to load other parts of the EJB specification on the **EJB client** side.

Remote interface

Retrive the Disk data by unique reference

```java
package hu.qwaevisz.diskstore.ejbserviceclient;

import javax.ejb.Remote;

import hu.qwaevisz.diskstore.ejbserviceclient.domain.DiskStub;
import hu.qwaevisz.diskstore.ejbserviceclie

@Remote
public interface DiskFacadeRemote {

  String BEAN_NAME = "DiskStoreService";

  DiskStub getDisk(String reference) throws ServiceException;

}
```

The `BEAN_NAME` is a public constant which is part of the JNDI name of the EJB service. As the Remote interface is public we usually store that value here.

`DiskFacadeRemote.java`

The substantial difference is the usage of the `@Remote` annotation instead of the `@Local`. It is allow to use these at the same interface but in that case pay attention to the future changes (especially when you create new *business methods*).

Domain class

Disk data

The main difference compared to the previous *BookStore* project's `BookStub` entity that here we implement the `Serializable` interface (we need this to send an instance through the network).

```
1  package hu.qwaevisz.diskstore.ejbserviceclient.domain;
2
3  import java.io.Serializable;
4
5  public class DiskStub implements Serializable {
6
7    private String reference;
8    private String author;
9    private String title;
10   private DiskCategoryStub category;
11   private double price;
12   private int numberOfSongs;
13
14   [..]
15 }
```

DiskStub.java

EAR customization

Some small but more important customization steps are worth checking out during EAR assembling.

- using 3rd party libraries in EAR (In case of *Gradle* there is am `earlib` `dependency configuration`. In case of *Maven* there is a `jar type` `dependency` which we can use.) These jars are not ejb/web modules of the EAR, but resources/libraries which were not loaded by the started server domain (because only this EAR would like to use them)).
  - EJB Service client (non **transitive** dependency!)
  - MyBatis 3
- Setting the Java EE version information (`version`), display name (`displayName`) and reference name (`applicationName`) of the EAR. The latter one is part of the JNDI name of the services (default value is the file name of the `ear`).
- Setting the **context root** properties of the EAR's webmodules/webapplications
  - We have to use the *context root* to refer to the webapplication's `Servlet` classes. The *context root* is especially public information. It can be configured via the `application.xml`, the default value is the file name of the `war` inside the `ear` (it depends on the used build system).

EAR Deployment descriptor

```
1  <?xml version="1.0"?>
2  <application xmlns="http://java.sun.com/xml/ns/javaee"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/application_6.xsd"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       version="6">
3    <application-name>diskstoreapp</application-name>
4    <display-name>Disk Store Application</display-name>
5    <module>
6      <web>
7        <web-uri>ds-weblayer.war</web-uri>
8        <context-root>diskstore</context-ro
9      </web>
10   </module>
11   <module>
12     <ejb>ds-persistence.jar</ejb>
13   </module>
14   <module>
15     <ejb>ds-ejbservice.jar</ejb>
16   </module>
17   <library-directory>library</library-directory>
18 </application>
```

In case of *Gradle* the version of the projects are *empty String*, but in *Maven* these are tipically are marked with the literal like e.g.: `1.0`. In the latter case the name of the `jar` files will contain the version number.

Let the reference name of the application be **diskstoreapp**, and the *context root* of the `ds-weblayer` web-module be **diskstore**. We will store the 3rd party libraries into the **library** directory.

application.xml

EAR customization

Build script

```
1  [..]
2  ext {
3    [..]
4    mybatisVersion = '3.3.1'
5    mybatiscdiVersion = '1.0.0-beta3'
6    webapplicationArtifact = 'ds-weblayer.war'
7  }
8  [..]
9  ear {
10   deploymentDescriptor {
11     version = "6"
12     applicationName = "diskstoreapp"
13     displayName = "Disk Store Application"
14     libraryDirectory = "library"
15     webModule( webapplicationArtifact, 'diskstore' )
16   }
17 }
18 [..]
19 dependencies {
20   deploy project('ds-persistence')
21   deploy project('ds-ejbservice')
22   deploy project(path: 'ds-weblayer', configuration: 'archives')
23   earlib ( project('ds-ejbserviceclient') ) {
24     transitive = false
25   }
26   earlib group: 'org.mybatis', name: 'mybatis', version: mybatisVersion
27   earlib group: 'org.mybatis', name: 'mybatis-cdi', version: mybatiscdiVersion
28 }
```

The `ds-weblayer` subproject will use the `webapplicationArtifact` variable too.

The default value of the `version` is 6, and `lib` in case of the `libraryDirectory`.

We can not add the `ds-ejbserviceclient` as a **transitive** dependency of the `EAR`, because in that case the entire JBoss Java EE 6.0 implementation will be part of the *deployment*.

`build.gradle`

EJB module deployment descriptor

ds-ejbservice subproject

src | main | resources | META-INF | ejb-jar.xml

```
1 <ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
          http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
3     version="3.2">
4     <module-name>dsservicemodule</module-name>
5 </ejb-jar>
```

ejb-jar.xml

Bean name configuration

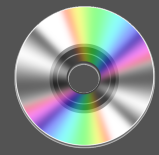
ds-ejbservice subproject

```
1 package hu.qwaevisz.diskstore.ejbservice.facade;
2 [..]
3 @Stateless(mappedName = "ejb/diskFacade", name = DiskFacadeRemote.BEAN_NAME)
4 public class DiskFacadeImpl implements DiskFacade, DiskFacadeRemote {
5
6   [..]
7
8 }
```

DiskFacadeImpl.java

DiskStore remote service

Assembling JNDI name



JNDI name

[context]/[application-name]/[module-name]/[bean-name]![full-qualified-interface-name]

DiskStore remote JNDI name

context: **java:jboss/exported/**
application-name: **diskstoreapp/**
module-name: **dsservicemodule/**
bean-name: **DiskStoreService!**
full-qualified-interface-name: **hu.qwaevisz.diskstore.ejbserviceclient.DiskFacadeRemote**

Deployment

diskstore.ear

[JBOSS_HOME] | standalone | log | server.log

```
18:08:08,665 INFO
[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC
service thread 1-6) JNDI bindings for session bean named DiskStoreService in deployment
unit subdeployment "ds-ejbservice.jar" of deployment "diskstore.ear" are as follows:

  java:global/diskstoreapp/dsservicemodule/DiskStoreService!hu.qwaevisz.diskstore.ejbservic
  java:app/dsservicemodule/DiskStoreService!hu.qwaevisz.diskstore.ejbservice.facade.DiskFac
  java:module/DiskStoreService!hu.qwaevisz.diskstore.ejbservice.facade.DiskFacade
  java:global/diskstoreapp/dsservicemodule/DiskStoreService!hu.qwaevisz.diskstore.ejbservic
  java:app/dsservicemodule/DiskStoreService!hu.qwaevisz.diskstore.ejbserviceclient.DiskFaca
  java:module/DiskStoreService!hu.qwaevisz.diskstore.ejbserviceclient.DiskFacadeRemote
  java:jboss/exported/diskstoreapp/dsservicemodule/DiskStoreService!hu.qwaevisz.diskstore.e
```

server.log

ds-weblayer testing:

http://localhost:8080/diskstore/DiskPing

http://localhost:8080/diskstore/DiskList

EJB Client

Build script

```
1  jar { archiveName 'ds-client.jar' }
2
3  repositories {
4    flatDir { dirs 'lib' }
5  }
6
7  dependencies {
8    compile project(':ds-ejbserviceclient')
9    compile name: 'jboss-client', ext: 'jar'
10 }
```

build.gradle

The **jboss-client.jar** which contains the JBoss specific communication helper classes (naming, protocols) is located on the `[JBOSS-HOME]\bin\client\jboss-client.jar` path. It would be a more elegant solution of we download this jar from a *JBoss repository* via an *artifact uri*. For demonstration purposes we are going to load this jar from a local directory (the **lib** directory is located at the root of the *EJB client* project). *Gradle* will search the dependencies in the registered **flatDir** repository as well.

EJB Client

Pseudocode

- `InitialContext` creations
  - Programmatically solution, filling up a `Hashtable<String, String>` or
  - use a `jndi.properties` file and put it in the *Classpath*
  - In both cases we need a JBoss specific key as well:
    - **Key**: `jboss.naming.client.ejb.context`
    - **Value**: `true`
- Lookup the DiskService (`DiskFacadeRemote` proxy) from the JNDI
  - with the `context.lookup([JNDI name])` method
- Get the Disk (`DiskStub` instance)
  - with the `diskFacadeRemote.getDisk([disk reference])` method

Creating Initial Context

Programmatically solution

```
1  package hu.qwaevisz.diskstore.client.context;
2  [..]
3  public class ProgrammedContextFactory implements ContextFactory {
4
5    private static final String JBOSS_NAMING_CLIENT_EJB_CONTEXT_KEY
         = "jboss.naming.client.ejb.context";
6
7    @Override
8    public Context getContext() throws NamingException {
9      final Hashtable<String, String> jndiProperties = new
           Hashtable<String, String>();
10     jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
           "org.jboss.naming.remote.client.InitialContextFactory");
11     jndiProperties.put(Context.PROVIDER_URL,
           "remote://localhost:4447");
12     jndiProperties.put(JBOSS_NAMING_CLIENT_EJB_CONTEXT_KEY,
           "true");
13     return new InitialContext(jndiProperties);
14   }
15 }
```

ProgrammedContextFactory.java

Creating Initial Context

jndi.properties file based configuration

src | main | resources | jndi.properties

```
1 java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
2 java.naming.provider.url=remote://localhost:4447
3 jboss.naming.client.ejb.context=true
```

jndi.properties

```
1 package hu.qwaevisz.diskstore.client.context;
2 [..]
3 public class JndiPropertiesContextFactory implements
      ContextFactory {
4
5   @Override
6   public Context getContext() throws NamingException {
7     return new InitialContext();
8   }
9 }
```

JndiPropertiesContextFactory.java

JNDI lookup

```
1  package hu.qwaevisz.diskstore.client;
2  [..]
3  public class DiskClient {
4
5    private ContextFactory contextFactory;
6
7    public DiskClient(ContextFactory contextFactory) {
8      this.contextFactory = contextFactory;
9    }
10
11   public DiskStub getDisk(final String reference) {
12     DiskStub disk = null;
13     try {
14       final DiskFacadeRemote facade =
               this.getDiskService(this.contextFactory.getContext());
15       disk = facade.getDisk(reference);
16       LOGGER.info(disk);
17     } catch (final ServiceException e) {
18       LOGGER.error(e, e);
19     } catch (final NamingException e) {
20       LOGGER.error(e, e);
21     }
22     return disk;
23   }
24
25   private DiskFacadeRemote getDiskService(Context context) throws NamingException {
26     return (DiskFacadeRemote) context.lookup("[JNDI-NAME]");
27   }
28 }
```

diskstoreapp/dsservicemodule/DiskStoreService!
hu.qwaevisz.diskstore.ejbserviceclient.DiskFacadeRemote

Logging in the client side

Log4J

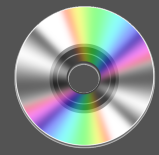
src | main | resources | log4j.xml

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
3  <log4j:configuration debug="true"
4    xmlns:log4j='http://jakarta.apache.org/log4j/'>
5
6    <appender name="console"
         class="org.apache.log4j.ConsoleAppender">
7        <layout class="org.apache.log4j.PatternLayout">
8      <param name="ConversionPattern"
9        value="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n" />
10       </layout>
11   </appender>
12
13   <root>
14     <level value="DEBUG" />
15     <appender-ref ref="console" />
16   </root>
17
18 </log4j:configuration>
```

log4j.xml

Testing the client application

Running



Run the application from the JVM of your favourite IDE.

```
1 2017-11-08 16:48:53 DEBUG EJBClientContext:720 -
      org.jboss.ejb.client.RandomDeploymentNodeSelector@37374a5e
      deployment node selector selected qwaevisz-mac node for
      appname=diskstoreapp,modulename=dsservicemodule,distinctname=
2 2017-11-08 16:48:53 INFO  DiskClient:28 - DiskStub
      [reference=WAM124, author=Mozart, title=Requiem Mass in D
      minor, category=ROCK, price=2850.0, numberOfSongs=4]
3
4 DiskStub [reference=WAM124, author=Mozart, title=Requiem Mass in
      D minor, category=ROCK, price=2850.0, numberOfSongs=4]
5
6 2017-11-08 16:48:53 DEBUG RemoteNamingStoreV1:263 - Channel end
      notification received, closing channel Channel ID c1d5c67d
      (outbound) of Remoting connection 5ed31789 to
      localhost/127.0.0.1:4447
7 2017-11-08 16:48:53 INFO  remoting:445 - EJBCLIENT000016: Channel
      Channel ID d1b9a950 (outbound) of Remoting connection
      5ed31789 to localhost/127.0.0.1:4447 can no longer process
      messages
```

Alternative ORM solutions

Is there anything usable outside the JPA?

We got to know the JPA in a very basic level ("Hello World" of JPA) during the *BookStore* project. We are going to move forward in that way later, because this it is inevitable in enterprise environment. But we have to know that JPA has a pretty big learning path, and a special characteristic that you can use working "prototypes" without deep theoretical knowledge (e.g.: lack of knowledge about ANSI SQL, ORM and JPA). Because of that relatively many projects run into a critical issue: the number of generated queries will be extremly high and the operations of the *entity manager* won't be optimum, and the JPA will be the *bottleneck* of the system.

Alternative ORM solutions

type-safe, native SQL, JavaEE compatibility

The *DiskStore* project is presenting an alternative ORM solution which **supports the Java EE integration** and ensures **type-safe** behavior above JDBC. Only ANSI SQL knowledge is required to use it, the learning path is short and easy to understand. It ignores the rich automatisms and the resulting problems. On the other hand we have to use **native SQL** queries, the library is *non vender independent* and after a time it becomes unconfortable (lots of *boilerplate* source codes) compared to the JPA.

Without ORM we are able to use database operations alone with the JDBC library in Java EE environment, but it never be the intent (to maintain a non-type safe codebase is almost impossible).

http://www.mybatis.org/mybatis-3/
Version: **3.4.6-SNAPSHOT** (2017-08-20)
Artifact URI: `org.mybatis:mybatis:3.4.6-SNAPSHOT`
The persistence configuration file is a `config.xml` file, which responsibility is similar than the `persistence.xml` of JPA. This xml document might refer several `mapper.xml` files where we can find native queries. Most of the XML based configuration elements can be replaced by annotations.

For the Java EE integration we need an additional **MyBatis CDI** dependency as well
http://www.mybatis.org/cdi/
Version: **1.0.2** (2017-10-13)
Artifact URI: `org.mybatis:mybatis-cdi:1.0.2`
The CDI is part of the Java EE, we will learn about that later. We just pay attention to the existence of the `beans.xml` file on the *classpath*.

```
<transactionManager type="MANAGED">
  <property name="closeConnection" value="true" />
</transactionManager>
```

Values of the TransactionManager type:

- **JDBC**: use directly with the commit/rollback possibilities of the JDBC
- **MANAGED**: MyBatis will not do anything with the transactions (will not be any automatical commit/rollback operations)

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</dataSource>
```

Values of the Datasource type:

- **UNPOOLED**: every single time it will open/close the connection
- **POOLED**: effective solution in case of webapplications
- **JNDI**: the datasource will be provided by a JavaEE container (we will use this)

ds-persistence | src | main | resources | mybatis-config.xml

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-config.dtd">
3  <configuration>
4    <typeAliases>
5      <typeAlias type="hu.qwaevisz.diskstore.persistence.entity.Disk" alias="Disk" />
6      <package name="hu.qwaevisz.diskstore.persistence.entity"/>
7    </typeAliases>
8    <typeHandlers>
9      <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
           javaType="hu.qwaevisz.diskstore.persistence.entity.trunk.DiskCategory"/>
10   </typeHandlers>
11   <environments default="development">
12     <environment id="development">
13       <transactionManager type="MANAGED">
14         <property name="closeConnection" value="true" />
15       </transactionManager>
16       <dataSource type="JNDI">
17         <property name="data_source" value="java:jboss/datasources/diskstoreds" />
18       </dataSource>
19     </environment>
20   </environments>
21   <mappers>
22     <mapper resource="hu/qwaevisz/diskstore/persistence/mapper/DiskMapper.xml" />
23   </mappers>
24 </configuration>
```

mybatis-config.xml

- **Type alias**: inside the `mapper.xml` you may use shorter names instead of the *full qualified* names if you set an `alias`. You can use annotation as well for the same purpose `@Alias`.
- **Type handler**: if the conversion is not unequivocal (Java $\rightarrow$ SQL), you have to write a *type handler*. If you would like to use the *ordinal* value of an *enum* there is a predefined *type handler* (`org.apache.ibatis.type.EnumOrdinalTypeHandler`), we will use it in the example.
- **Environment**: the configuration of the transaction handling and the datasource.
- **Mapper**: list of configuration files which contain the native SQL queries and/or *resultMaps*. Here you can add *mapper classes* as well if you use annotation based configuration.

Disk entity

```java
package hu.qwaevisz.diskstore.persistence.entity;
[..]
@Alias("Disk")
public class Disk {

  private Integer id;
  private String reference;
  private String author;
  private String title;
  private DiskCategory category;
  private Double price;
  private Integer numberOfSongs;

  public Disk() {
    [..]
  }

  [..]
}
```

Disk.java

Mapper

The listed `mapper.xml` files in the configuration contain native SQL queries and *resultMaps* under the given namespace. Generating *boilerplate* source codes from the `mapper.xml` files are one of the main task of **MyBatis 3** library (it results type-safe behavior, because we do not need to maintain these implementations).

`ds-persistence | src | main | resources | hu | qwaevisz | diskstore | persistence | mapper | DiskMapper.xml`

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3  <mapper namespace="hu.qwaevisz.diskstore.persistence.mapper.DiskMapper">
4
5    <resultMap [..]>[..]</resultMap>
6    <select [..]>[..]</select>
7    <insert [..]>[..]</insert>
8    <update [..]>[..]</update>
9    <delete [..]>[..]</delete>
10
11 </mapper>
```

DiskMapper.xml

Mapper file - count

core input, core output

```
1 <select id="count" parameterType="String" resultType="int">
2   SELECT COUNT(1)
3   FROM disk
4   WHERE disk_reference = #{reference}
5 </select>
```

DiskMapper.xml

With that declaration we define a `count` method which has a `String` input and an `int` return value.

Mapper file - readByReference

core input, entity output

```
1  <select id="readByReference" parameterType="String"
       resultType="Disk">
2    SELECT
3      disk_id AS id,
4      disk_reference AS reference,
5      disk_author AS author,
6      disk_title AS title,
7      disk_diskcategory_id AS category,
8      disk_price AS price,
9      disk_number_of_songs AS numberOfSongs
10   FROM disk
11   WHERE disk_reference = #{reference}
12 </select>
```

With that declaration we define a `readByReference` method which has a `String` input and a `Disk` return value. Because we defined an *alias* in the `config.xml`, we can use the `Disk` alias instead of the *full qualified* type name.
The used **AS** in the query establishes the relationship between the *mutator* (setter) method of the entity (according to Java bean naming rules).

Mapper file - readAll

- input, list of resultMap output

```
1  <resultMap type="Disk" id="DiskResult">
2    <id property="id" column="disk_id" />
3    <result property="reference" column="disk_reference" />
4    <result property="author" column="disk_author" />
5    <result property="title" column="disk_title" />
6    <result property="category" column="disk_diskcategory_id" />
7    <result property="price" column="disk_price" />
8    <result property="numberOfSongs" column="disk_number_of_songs"
         />
9  </resultMap>
10
11 <select id="readAll" resultMap="DiskResult">
12   SELECT * FROM disk
13 </select>
```

In case of a custom (e.g.: multiple tables, aggregation or some custom associations) query we cannot define the result value as an entity (alias). For that issue we can define a `resultMap`. In that example this is only a "hello world" sample, its possibilities are far wider. Obviously the `property` attribute refers to a *mutator* method in the given `type`.

Mapper file - insert

entity input, - output

```
<insert id="create" parameterType="Disk" useGeneratedKeys="true" keyProperty="id">
  INSERT INTO disk (
    disk_reference,
    disk_author,
    disk_title,
    disk_diskcategory_id,
    disk_price,
    disk_number_of_songs
  ) VALUES (
    #{reference},
    #{author},
    #{title},
    #{category},
    #{price},
    #{numberOfSongs}
  )
</insert>
```

With that declaration we define a `create` method which has a `Disk` input and an `int` return value. The various input values have significance, e.g. the `{#author}` will call the `getAuthor()` *accessor* method of the input entity.
In the example a sequence will generate the value of the ID column, we do not include that ID in the native query. We indicate this with the `useGeneratedKeys` attribute.

Mapper file - update és delete

missing CRUD operations

```
<update id="update" parameterType="Disk">
  UPDATE disk SET
    disk_reference = #{reference},
    disk_author = #{author},
    disk_title = #{title},
    disk_diskcategory_id = #{category},
    disk_price = #{price},
    disk_number_of_songs = #{numberOfSongs}
  WHERE disk_id = #{id}
</update>
```

```
<delete id="delete" parameterType="int">
  DELETE FROM disk WHERE disk_id = #{id}
</delete>
```

Mapper interface

The given *abstract* methods of the interface are fit the background `xml` configuration. We will get runtime exception it something is not OK (this is kind of JDBC style and this is not fault tolerant enough, but better than a simple Java based solution).

```
1  package hu.qwaevisz.diskstore.persistence.mapper;
2
3  import java.util.List;
4  import hu.qwaevisz.diskstore.persistence.entity.Disk;
5
6  public interface DiskMapper {
7
8    int count(String reference);
9    int create(Disk disk);
10   Disk readById(Integer id);
11   Disk readByReference(String reference);
12   List<Disk> readAll();
13   int update(Disk disk);
14   int delete(Integer id);
15 }
```

DiskMapper.java

Load MyBatis configuration

You have to load the configuration one time at *runtime*. Here we see a *Provider* class and its method which has a `@Produces` annotation. This two annotatioms are part of the CDI and this is totaly independent of the *MyBatis 3* library. Its purpose that the injection of the `SqlSessionFactory` will be available everywhere (in the CDI context). We will learn about CDI later.

```
1  package hu.qwaevisz.diskstore.persistence.config;
2  [..]
3  import javax.enterprise.context.ApplicationScoped;
4  import javax.enterprise.inject.Produces;
5  import org.apache.ibatis.io.Resources;
6  import org.apache.ibatis.session.SqlSessionFactory;
7  import org.apache.ibatis.session.SqlSessionFactoryBuilder;
8  [..]
9  public class SqlSessionFactoryProvider {
10
11   @Produces
12   @ApplicationScoped
13   public SqlSessionFactory produceFactory() throws IOException {
14     final InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
15     final SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
16     return factory;
17   }
18
19 }
```

SqlSessionFactoryProvider.java

Use MyBatis in JavaEE environment

```
1  package hu.qwaevisz.diskstore.persistence.service;
2  [..]
3  import javax.inject.Inject;
4  import org.mybatis.cdi.Mapper;
5  [..]
6  @Stateless(mappedName = "ejb/diskService")
7  public class DiskServiceImpl implements DiskService {
8
9    @Inject
10   @Mapper
11   private DiskMapper mapper;
12
13   @Override
14   public Disk readByReference(final String reference) throws
         PersistenceServiceException {
15     LOGGER.debug("Read Disk by reference (" + reference + ")");
16     try {
17       return this.mapper.readByReference(reference);
18     } catch (final Exception e) {
19       LOGGER.error(e, e);
20       throw new PersistenceServiceException("Failed to read Disk by reference (" +
             reference + ")! " + e.getMessage(), e);
21     }
22   }
23   [..]
24 }
```

The `@Inject` is the CDI variation of the `@EJB` annotation. The `@Mapper` annotation here is a kind of CDI qualifier, it is defined by the *MyBatis CDI* library.

DiskServiceImpl.java