

# 04 - Persistence tier (1)

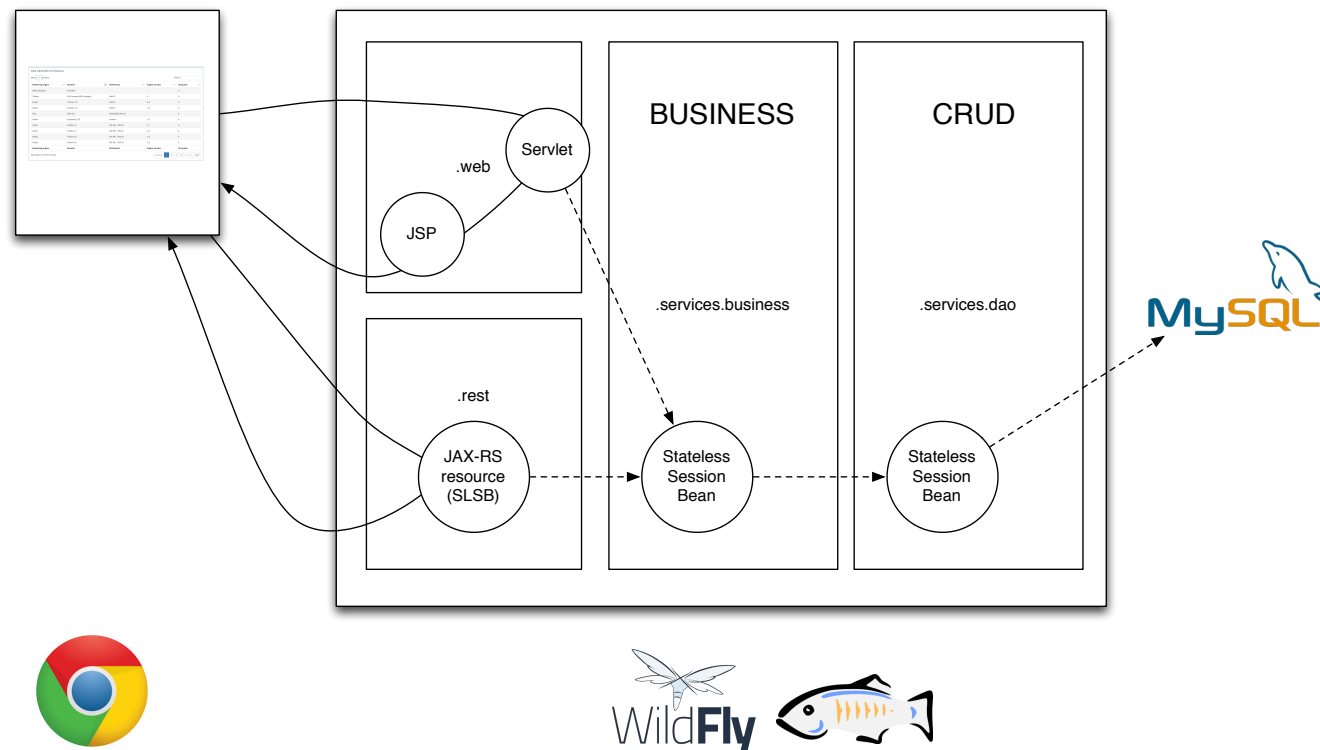
---

The DAO / Repository pattern & JDBC

**AMT 2020**

**Olivier Liechti**

# Webcasts (**legacy**). Use this if you want to configure Wildfly, Payara, etc.



## Tasks

### 1. Prepare the environment

- 1.1. add a MySQL image to our Docker topology
- 1.2. add a PhpMyAdmin image to our Docker topology
- 1.3. insert sample data into our database

### 2. Configure Glassfish (manually)

- 2.1. Install MySQL driver
- 2.2. Configure connection pool and data source

### 3. Configure Glassfish (Docker)

- 3.1. Install MySQL driver
- 3.2. Configure connection pool and data source

### 4. Configure Wildfly (manually)

- 4.1. Install MySQL driver
- 4.2. Configure data source

### 5. Configure Wildfly (Docker)

- 5.1. Install MySQL driver
- 5.2. Configure data source








### 6. Implement a Data Access Object (DAO)

- 6.1. Create a new Stateless Session Bean (SLSB)
- 6.2. Inject the data source into the SLSB
- 6.3. Use JDBC to send SQL queries to the DB



Why are data source useful in Java EE?  
What is JDBC and what is its relationship with Java EE?  
What is a DAO?

# Webcasts (**legacy**). Use this if you want to configure Wildfly, Payara, etc.

|    |   |  |       |
|----|---|--|-------|
| 17 |    | <b>Bootcamp 4.1: Intro aux webcasts "tiers d'accès aux données avec JDBC"</b><br>by oliechti         | 4:07  |
| 18 |    | <b>Bootcamp 4.2: ajout de mysql et phpmyadmin dans notre topologie docker-compose</b><br>by oliechti | 10:53 |
| 19 |   | <b>Bootcamp 4.3: configuration de Glassfish</b><br>by oliechti                                       | 17:20 |
| 20 |  | <b>Bootcamp 4.4: configuration de la data source dans Docker</b><br>by oliechti                      | 8:00  |
| 21 |  | <b>Bootcamp 4.5: configuration de Wildfly via l'interface web</b><br>by oliechti                     | 7:16  |
| 22 |  | <b>Bootcamp 4.6: configuration de Wildfly via Docker</b><br>by oliechti                              | 24:30 |
| 23 |  | <b>Bootcamp 4.7: écriture du code et test dans Glassfish</b><br>by oliechti                          | 8:17  |



# The DAO Design Pattern

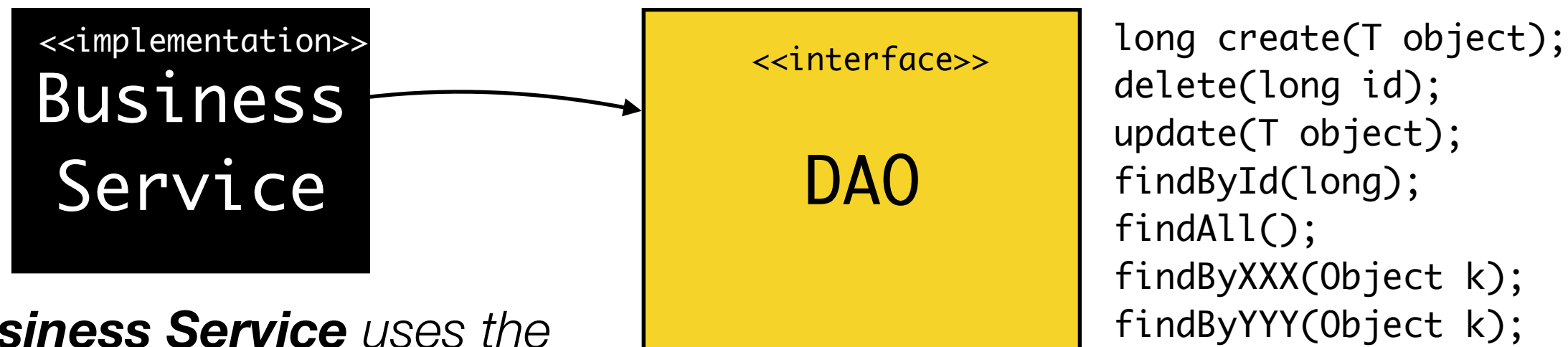


What is the **DAO design pattern** and what are its benefits?

- Most applications manipulate data that is stored in one or more **data stores**.
- There are **different ways** to implement a data store. Think about specific RDMS, NoSQL DBs, LDAP servers, file systems, etc.
- When you implement business logic, you would like to create code that is **independent** from a particular data store implementation (\*).
- In other words, you want to **reduce coupling** between your business service and your data store implementation.
- When you apply the **Data Access Object** design pattern, you create an abstraction layer to achieve this goal.

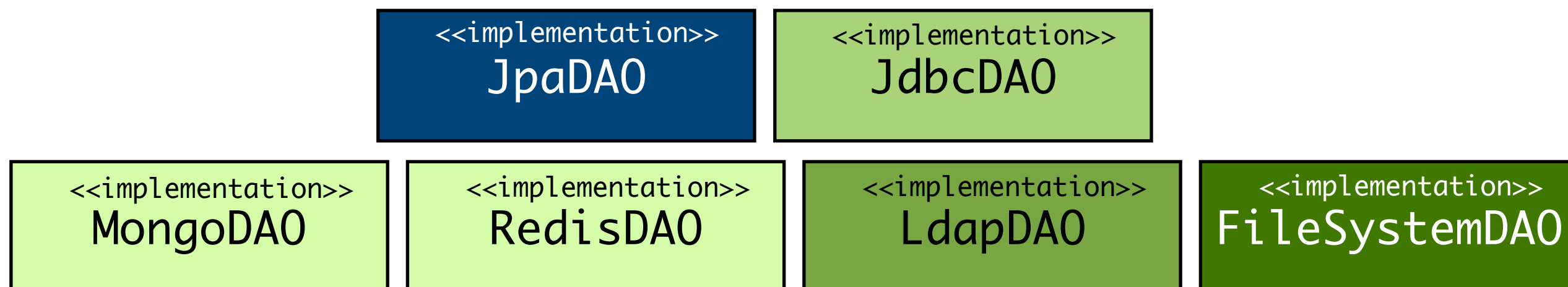
(\*) This is true only to some extent... you cannot completely forget about it, for instance for performance reasons

The **DAO interface** defines  
generic **CRUD** operations  
and **finder** methods

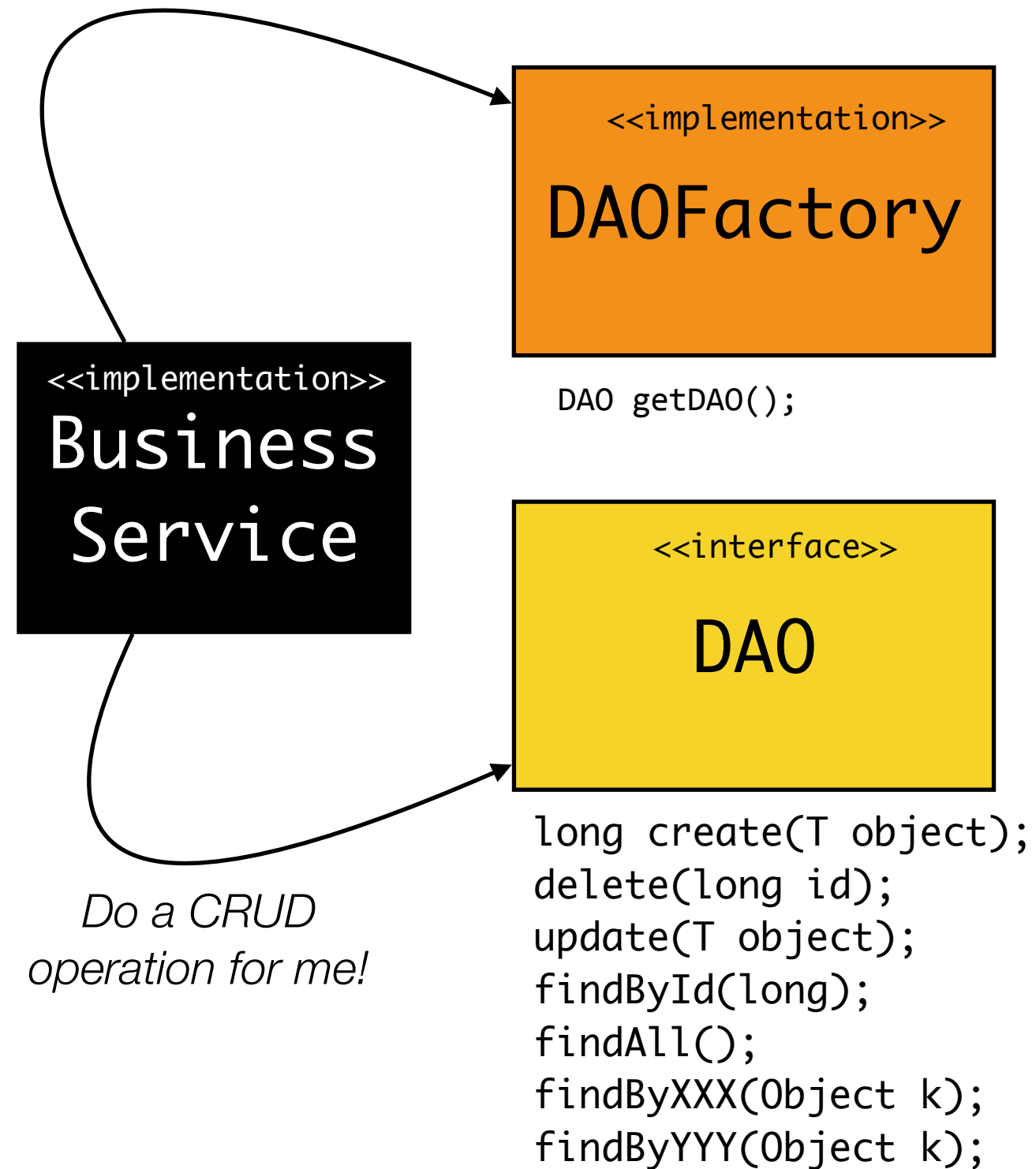


The **Business Service** uses the  
DAO interface to interact with a  
particular DAO implementation

**DAO implementations** handle  
interactions with specific data stores



*Give me a DAO implementation!*



`<<implementation>>`  
**JdbcDAO**

`<<implementation>>`  
**JpaDAO**

`<<implementation>>`  
**MongoDAO**

`<<implementation>>`  
**RedisDAO**

`<<implementation>>`  
**LdapDAO**

`<<implementation>>`  
**FileSystemDAO**

heig-vd

MySQL

PostgreSQL

Oracle

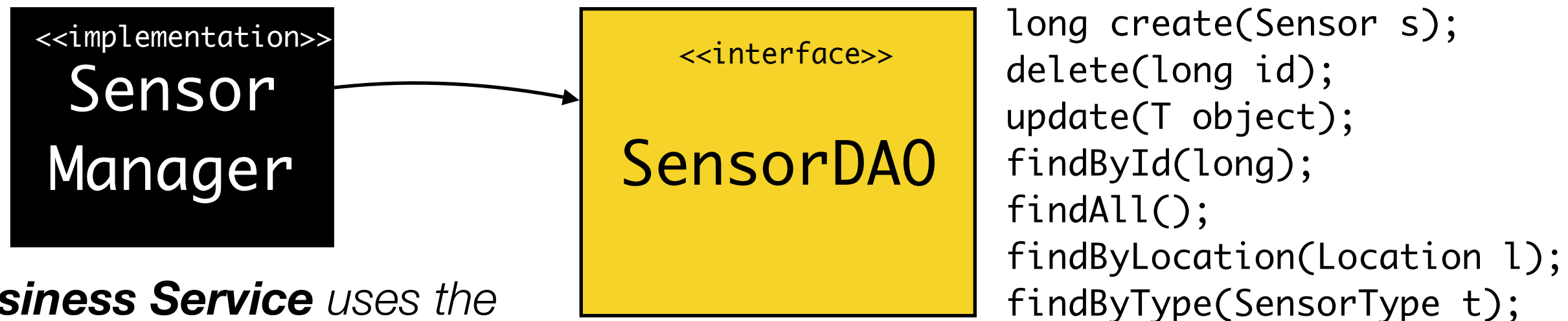
MongoDB

redis

LDAP server

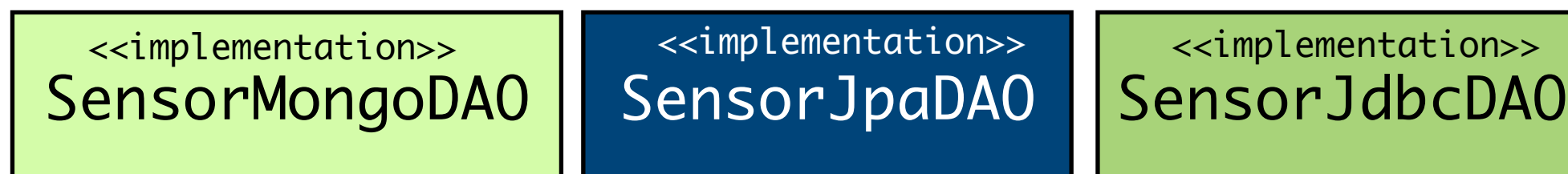
File System

The **DAO interface** defines  
generic **CRUD** operations  
and **finder** methods



The **Business Service** uses the  
DAO interface to interact with a  
particular DAO implementation

**DAO implementations** handle  
interactions with specific data stores

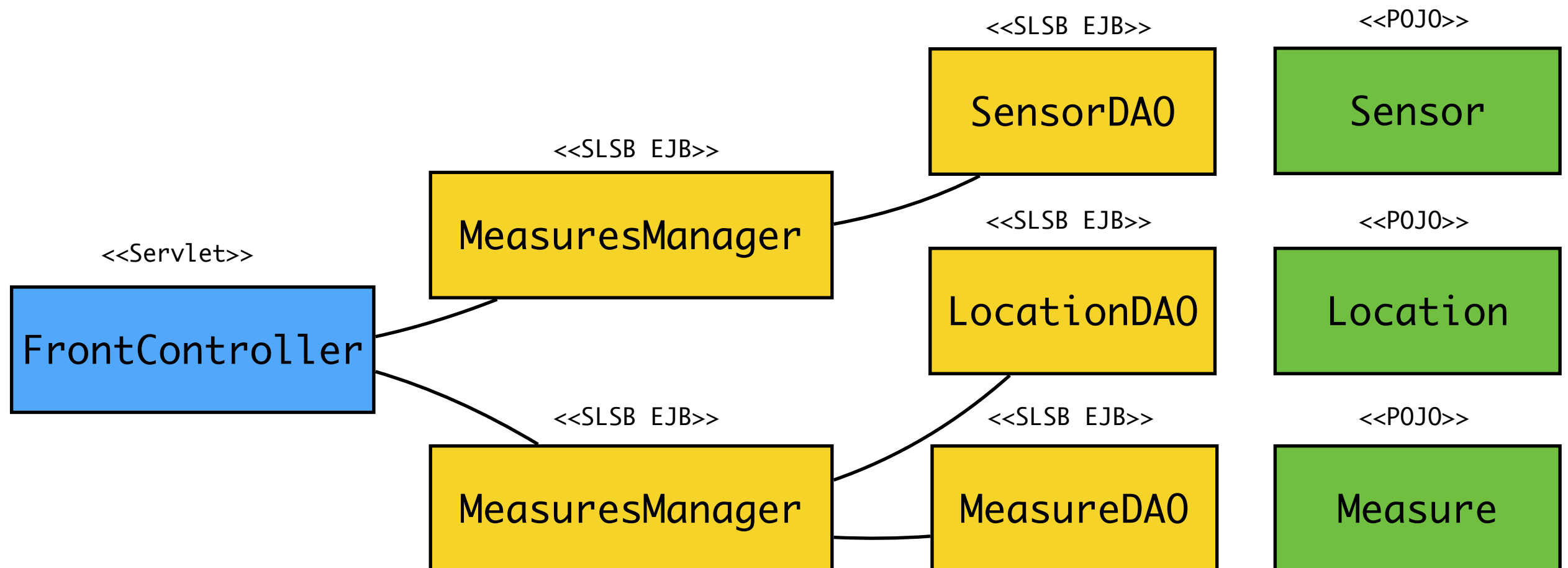






# How do I implement the DAO pattern with Java EE technologies?

- There are **different ways to do it**. Some frameworks (e.g. Spring) do that in the web tier (with POJOs).
- If you use **EJBs**, then your architecture is going to look like this:





Is it possible to have **two EJB classes** that implement the **same interface**?

- In the examples so far (and in most cases in practice), we have always created **one local interface** and **one stateless session bean class**.
- If we define the **DAO interface as a local interface** and implement two stateless session beans (JdbcDAO and JpaDAO), then we have an issue:

The **container is unable to resolve this dependency**, because there is more than one implementation. Which one should it choose?

```
public class MyServlet extends HttpServlet
{
    @EJB
    SensorDAOLocal sensorDAO;
}
```

```
@Local
public interface SensorDAOLocal {
    public long insert(Sensor sensor);
}
```

```
@Stateless
public class SensorJdbcDAO {
    implements SensorDAOLocal
    public long insert(Sensor sensor){}
}
```

```
@Stateless
public class SensorJpaDAO {
    implements SensorDAOLocal
    public long insert(Sensor sensor){}
}
```



Is it possible to have **two EJB classes** that implement the **same interface**?

- We can help the container by giving additional information in the annotation.
- If we define the **DAO interface as a local interface** and implement two stateless session beans (JdbcDAO and JpaDAO), then we have an issue:



The **name**, **beanName** and **mappedName** annotation attributes have different purposes.

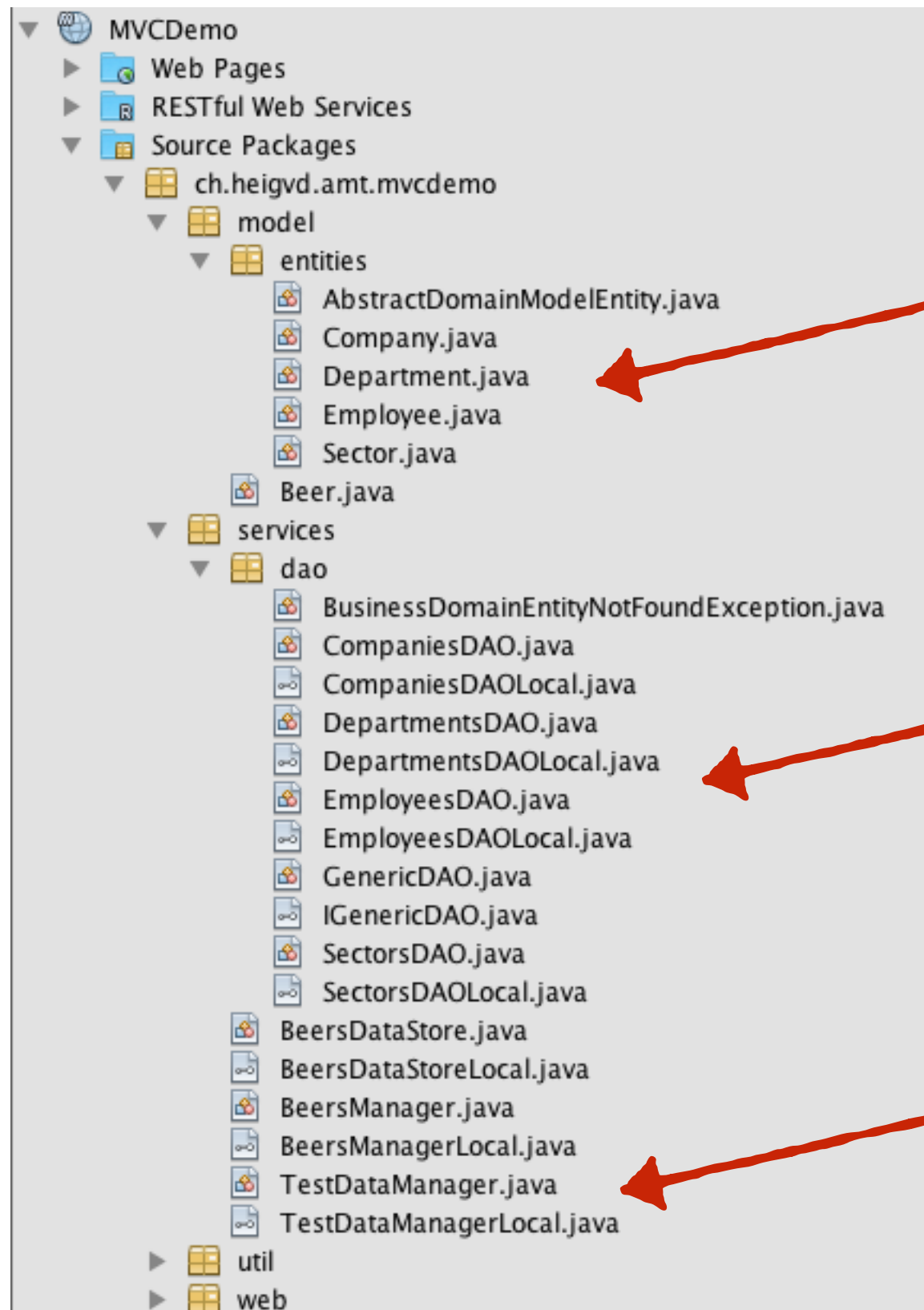
```
public class MyServlet extends HttpServlet {  
    @EJB(beanName="SensorJdbcDAO")  
    SensorDAOLocal sensorDAO;  
}
```

```
@Local  
public interface SensorDAOLocal {  
    public long insert(Sensor sensor);  
}
```

```
@Stateless  
public class SensorJdbcDAO {  
    implements SensorDAOLocal  
    public long insert(Sensor sensor){}  
}
```

```
@Stateless  
public class SensorJpaDAO {  
    implements SensorDAOLocal  
    public long insert(Sensor sensor){}  
}
```

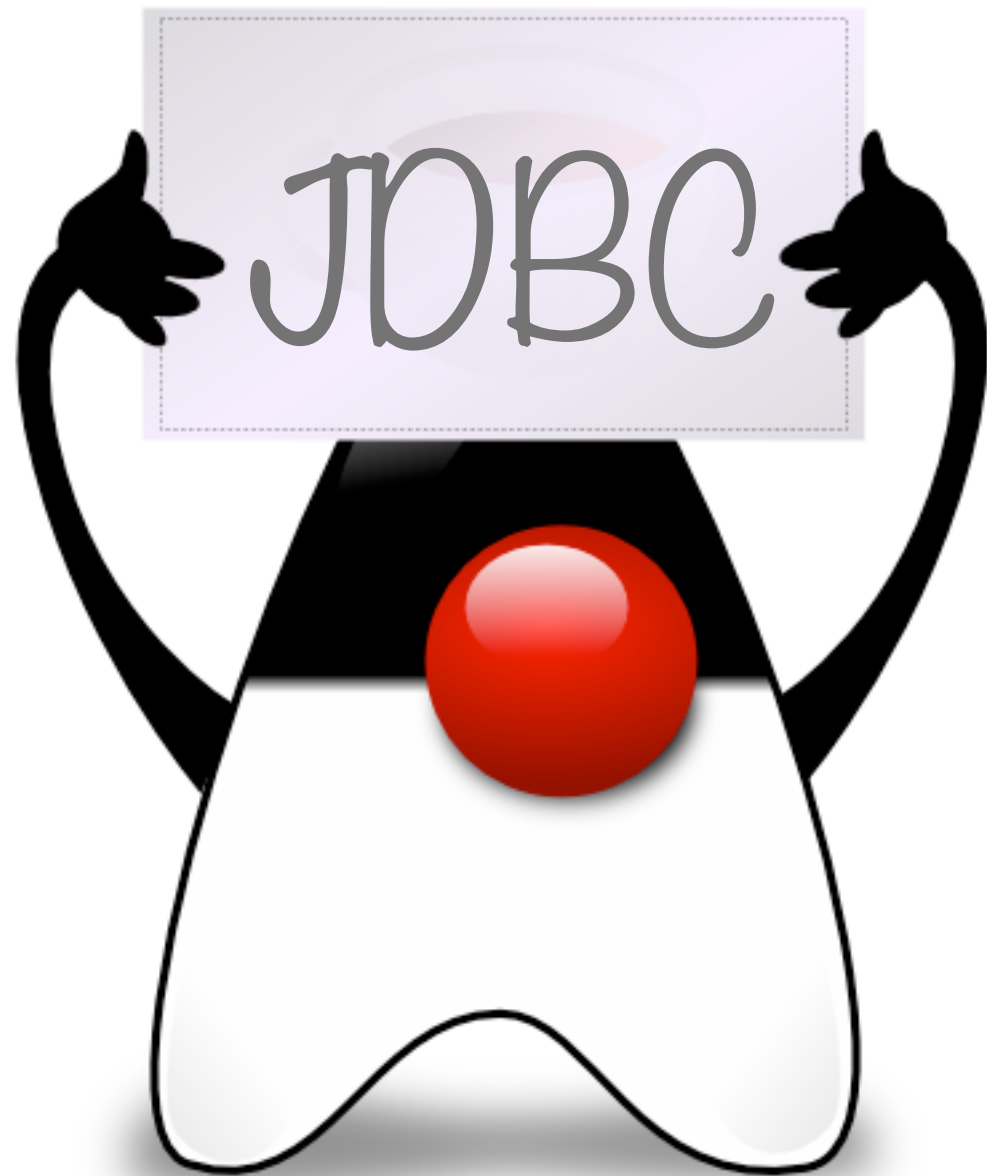
# DAO in the (**legacy**) MVCDemo Project



These are the model classes, which can be used with different DAO implementations

These are the DAOs. We only have one implementation, based on JPA.

We use the DAOs to generate test data



# Java DataBase Connectivity



## What is **JDBC**?

- The **Java DataBase Connectivity** is a specification that defines how applications can interact with **relational database** management systems in a **standard way**.
- Its goal is to **create an abstraction layer** between applications and specific RDBMS (MySQL, Oracle, PostgreSQL, DB2, etc.).
- Through this abstraction layer, applications can **submit SQL queries to read, insert, update and delete** records in tables.
- Applications can also get **metadata** about the relational schema (table names, column names, etc.).



# What does it look like?

```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

    @Resource(lookup = "jdbc/AMTDatabase")
    private DataSource dataSource;

    public List<Sensor> findAll() {
        List<Sensor> result = new LinkedList<>();
        try {
            Connection con = dataSource.getConnection();

            PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
            ResultSet rs = ps.executeQuery();

            while (rs.next()) {
                Sensor sensor = new Sensor();
                sensor.setId(rs.getLong("ID"));
                sensor.setDescription(rs.getString("DESCRIPTION"));
                sensor.setType(rs.getString("TYPE"));
                result.add(sensor);
            }

            ps.close();
            con.close();

        } catch (SQLException ex) {
            Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return result;
    }
}
```

**dependency injection**

**get a connection from the pool**

**create and submit a SQL query**

**scroll through the tabular result set**

**get data from the result set**

**return the connection to the pool**



# What is **JDBC**?

## **JDBC API**

java[x].sql.\* interfaces

## **JDBC Service** (provided by JRE)

java[x].sql.\* classes

## **JDBC SPI** (extends JDBC API)

## **JDBC MySQL driver**

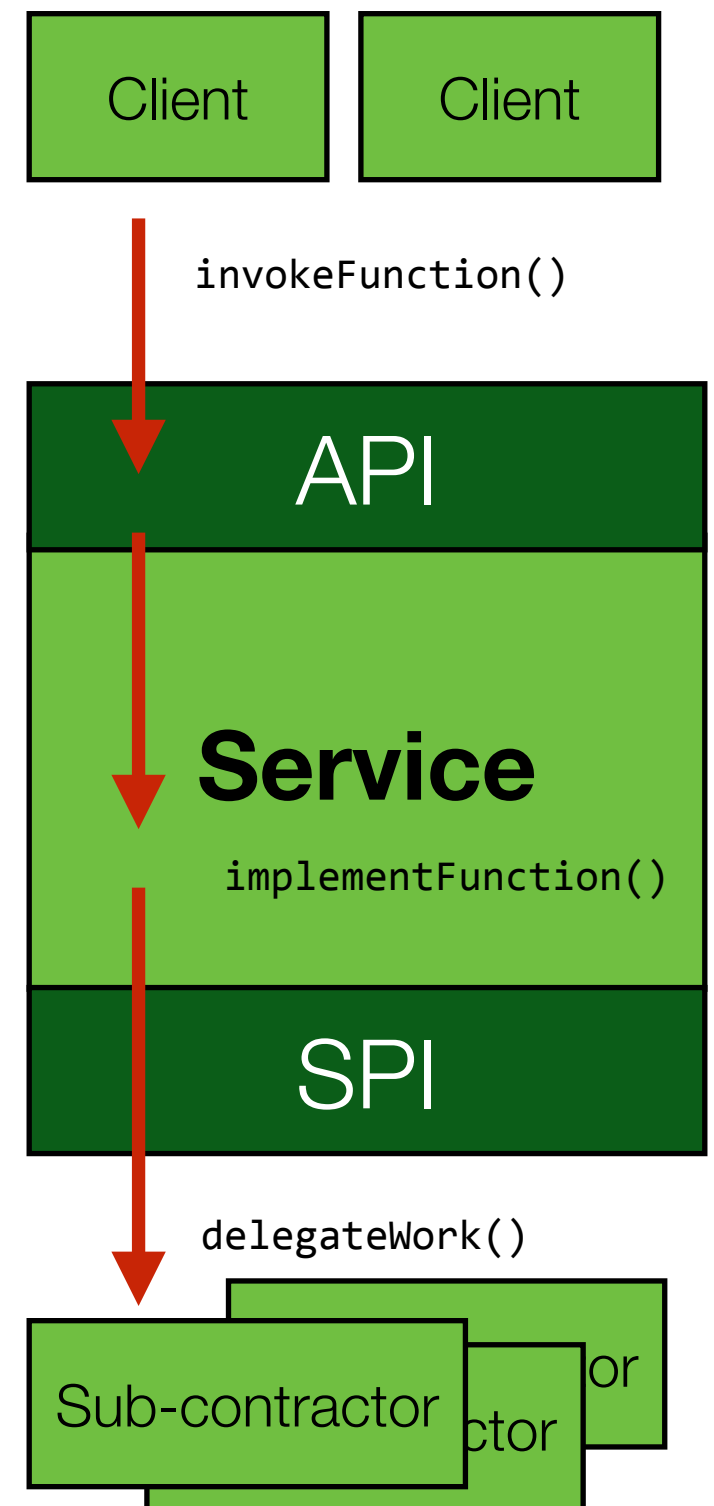
implements java[x].sql.\* interfaces





What is the difference between an **API** and a **SPI**?

- An **Application Programming Interface** (API) is a **contract** between a client and a service.
- It **defines what the client can request** from the service.
- A **Service Provider API** (SPI) is a contract between a service and its **subcontractors** (components to which it delegates some of the work).
- It **defines what the subcontractors need to do** in order to receive work from the service.





What is the difference between an **API** and a **SPI**?

```
public interface ServiceAPI {  
    public void invokeFunction1();  
    public String invokeFunction2(Object param1);  
}
```

```
public class Service implements ServiceAPI {  
    private ServiceSPI provider;  
  
    public void invokeFunction1() { provider.delegateWork(null); }  
  
    public String invokeFunction2(Object param1) {  
        doSomething(); provider.delegateOtherWork();  
    }  
  
    public void registerServiceProvider(ServiceSPI provider) {  
        this.provider = provider;  
    }  
}
```

```
public interface ServiceSPI {  
    public void delegateWork(String[] params);  
    public void delegateOtherWork();  
    public void doSomething();  
}
```



**In some cases**, the SPI is an **extension** of the API.

```
public interface ServiceAPI {  
    public void invokeFunction1();  
    public String invokeFunction2(Object param1);  
}
```

```
public class Service implements ServiceAPI {  
    private ServiceSPI provider;  
  
    public void invokeFunction1() { provider.invokeFunction1(); }  
  
    public String invokeFunction2(Object param1) {  
        provider.invokeFunction2(param1); provider.doSomethingNotExposedInAPI();  
    }  
  
    public void registerServiceProvider(ServiceSPI provider) {  
        this.provider = provider;  
    }  
}
```

```
public interface ServiceSPI extends ServiceAPI {  
    public void doSomethingNotExposedInAPI();  
}
```



How is it possible to **obtain a reference** to a JDBC service provider (driver)?

- At some point, the application wants to **obtain a reference to a specific provider**, so that it can invoke JDBC functions.
- The method depends on the Java environment. You do not the same thing if you are in a **Java SE** or **Java EE** environment.

## Java SE

`java.sql.DriverManager`

Think “**explicit** class loading and connection URLs”

## Java EE

`java.sql.DataSource`

Think “**managed** resources and “dependency injection”



How do I **obtain a reference** to a JDBC service provider in **Java SE**?

- In Java SE, the **DriverManager** class addresses this need:
  - It is used by clients who use the API.
  - It is also used by drivers who implement the SPI.
- Think of it as a **broker**, or a **registry**, who puts clients and service providers in relation.
- As a client, I am **explicitly** loading JDBC drivers (1 or more).
- As a client, I am **explicitly** telling with which database I want to interact (via a URL). The URL is used both to find a proper driver and to establish a connection (e.g. hostname, port, etc.).



How do I **obtain a reference** to a JDBC service provider in **Java SE**?

- From the specifications: “Key `DriverManager` methods include: **1. A service provider registers itself in the directory.**

- `registerDriver` — this method **adds a driver to the set of available drivers** and is invoked implicitly when the driver is loaded. The `registerDriver` method is typically called by the static initializer provided **by each driver**.

Used by **SPI** implementations

- `getConnection` — the method the **JDBC client** invokes to establish a connection. The invocation includes a JDBC URL, which the `DriverManager` passes to each driver in its list until it finds one whose `Driver.connect` method recognizes the URL. That driver returns a `Connection` object to the `DriverManager`, which in turn passes it to the application.”

Used by **API** clients

**2. A client looks for a service provider in the directory.**



How do I **obtain a reference** to a JDBC service provider in **Java SE**?

## Client

```
Class.forName("ch.heigdb.HeigDbDriver");  
DriverManager.getConnection("jdbc:heigdb://localhost:2205");
```

## JDBC Service (provided by JRE)

```
java.sql.DriverManager  
registerDriver(Driver driver)  
Connection getConnection(String url)
```

## JDBC HeigDB driver

```
public class HeigDbDriver implements java.sql.Driver {  
  
    static {  
        DriverManager.registerDriver(new HeigDBDriver());  
    }  
    public boolean acceptsURL(String url) {};  
    public Connection connect(String url, Properties p) {};  
}
```

1

Load a class

3

"Find a SPI provider that will connect me to this DB"

2

"I am an SPI provider"

4

"Can you connect me with this DB?"

5

"Connect me with this DB"



How do I **obtain a reference** to a JDBC service provider in **Java EE**?

- In Java EE, the **DataSource** interface is used for managing DB connections.
  - It is used by **application components** (servlets, EJBs, etc.) to obtain a connection to a database.
  - It is also used by **system administrators**, who define the **mapping** between a logical data source name and a concrete database system (by configuration).
- As a developer, I am only using a logical name and I know that it will be bound to a specific system at runtime (but I don't care which...).
- As a developer, I obtain a DataSource either by doing a **JNDI lookup** or via **dependency injection** (with annotations).





How do I **obtain a reference** to a JDBC service provider in **Java EE**?

## Client

```
Context ctx = new InitialContext();  
DataSource ds = (DataSource)ctx.lookup("jdbc/theAppDatabase");
```

**OR**

3 @Resource(lookup="jdbc/theAppDatabase")  
DataSource ds;

4 ds.getConnection();

## JDBC Service (provided by Java EE)

`java.sql.DataSource`

mysql-connector-java-5.1.33.jar

1



Install a **driver** (.jar file)  
in the app server (/lib/)

2



Create a (logical) data  
source...

3



... and map it to a (physical)  
connection pool

Home About... Help

User: anonymous | Domain: domainAMT | Server: localhost

GlassFish™ Server Open Source Edition

Tree

- server (Admin Server)
  - Clusters
  - Standalone Instances
  - Nodes
  - Applications
  - Lifecycle Modules
  - Monitoring Data
  - Resources
    - Concurrent Resources
    - Connectors
    - JDBC
      - JDBC Resources
        - jdbc/\_\_TimerPool
        - jdbc/\_\_default
        - jdbc/myDataSource
        - jdbc/sample
      - JDBC Connection Pools
        - DerbyPool
        - SamplePool
        - \_\_TimerPool
        - mysql\_mysql\_rootPool
      - JMS Resources
      - JNDI
      - JavaMail Sessions
      - Resource Adapter Configs
    - Configurations

### Edit JDBC Resource

Edit an existing JDBC data source.

Load Defaults

Save Cancel

JNDI Name: jdbc/myDataSource

Pool Name: mysql\_mysql\_rootPool

Use the [JDBC Connection Pools](#) page to create new pools

Deployment Order: 100

Specifies the loading order of the resource at server startup. Lower numbers are loaded first.

Description:

Status: ☒ Enabled

#### Additional Properties (0)

Add Property Delete Properties

| Select          | Name | Value | Description |
|-----------------|------|-------|-------------|
| No items found. |      |       |             |

A **JDBC Resource** simply defines a **mapping** between a **logical name** (used in the code) and a **concrete DB connection pool** (hooked to a “physical” DB).

localhost:4848/common/index.jsf

Home About... Help

User: anonymous | Domain: domainAMT | Server: localhost

GlassFish™ Server Open Source Edition

Tree

- server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
    - JDBC Resources
      - JDBC Connection Pools
        - mysql\_mysql\_rootPool

General Advanced Additional Properties

### Edit JDBC Connection Pool Properties

Modify properties of an existing JDBC connection pool.

Pool Name: mysql\_mysql\_rootPool

Save Cancel

Additional Properties (7)

Add Property Delete Properties

| Select                   | Name         | Value   | Description |
|--------------------------|--------------|---|-------------|
| <input type="checkbox"/> | URL          | jdbc:mysql://localhost:3306/mysql?zeroDateTin |             |
| <input type="checkbox"/> | driverClass  | com.mysql.jdbc.Driver                         |             |
| <input type="checkbox"/> | Password     | akAUKLJdfI1882_2                              |             |
| <input type="checkbox"/> | portNumber   | 3306  |             |
| <input type="checkbox"/> | databaseName | mysql   |             |
| <input type="checkbox"/> | User         | technicalAccount                              |             |
| <input type="checkbox"/> | serverName   | localhost                                     |             |

The **connection pool** is configured with “physical” database attributes (host, port, credentials, etc.).



What are some of the key JDBC interfaces and classes?

DriverManager

DataSource

XADataSource

Connection

PreparedStatement

ResultSet

ResultSetMetaData

- **DriverManager** and **DataSource** variations provide a means to obtain a **Connection**.
- **XADataSource** is used for distributed transactions.
- Once you have a **Connection**, you can submit SQL queries to the database.
- The most common way to do that is to create a **PreparedStatement** (rather than a **Statement**, which is useful for DDL commands).
- The response is either a number (number of rows modified by an UPDATE or DELETE query), or a **ResultSet** (which is a tabular data set).
- **ResultSetMetadata** is a way to obtain information about the returned data set (column names, etc.).



# How do I use these classes in my code?

```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

    @Resource(lookup = "jdbc/AMTDatabase")
    private DataSource dataSource;

    public List<Sensor> findAll() {
        List<Sensor> result = new LinkedList<>();
        try {
            Connection con = dataSource.getConnection();

            PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
            ResultSet rs = ps.executeQuery();

            while (rs.next()) {
                Sensor sensor = new Sensor();
                sensor.setId(rs.getLong("ID"));
                sensor.setDescription(rs.getString("DESCRIPTION"));
                sensor.setType(rs.getString("TYPE"));
                result.add(sensor);
            }

            ps.close();
            con.close();

        } catch (SQLException ex) {
            Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return result;
    }
}
```

**dependency injection**

**get a connection from the pool**

**create and submit a SQL query**

**scroll through the tabular result set**

**get data from the result set**

**return the connection to the pool**





# JDBC with Open Liberty

# TL;DR: "repository" is another name for "DAO"

Long story: start with <https://martinfowler.com/eaCatalog/repository.html>  
<https://www.baeldung.com/java-dao-vs-repository>, and dig into DDD

The screenshot shows an IDE with a project structure on the left and code in the center. The project structure includes a 'person' package, a 'question' package, a 'services' package, a 'vote' package, an 'Answer' class, a 'UI' class, an 'infrastructure' package, a 'persistence' package, a 'jdbc' package, a 'jpa' package, a 'memory' package, a 'liberty' package, a 'config' package, a 'mysql-connector-java-8.0.12.jar' file, a 'server.xml' file, a 'resources' package, a 'webapp' package, a 'test' package, a 'target' package, a 'demo.iml' file, a 'pom.xml' file, a 'readme.md' file, and 'External Libraries'. The 'jdbc' package is highlighted with a red circle, and it contains 'JdbcPersonRepository', 'JdbcQuestionRepository', and 'JdbcVoteRepository'. The 'JdbcPersonRepository' class is selected, and its code is displayed in the center. The code includes annotations for application scope and naming, a constructor that takes a 'DataSource', and an override for the 'findById' method. The 'findById' method uses the 'DataSource' to execute a SQL query to find a person by ID.

```
31  
32 <!-- Declare the jar files for MySQL access through JDBC. -->  
33 <library id="MySQLLib">  
34   <fileset dir="${server.config.dir}" includes="*.jar"/>  
35 </library>  
36  
37 <dataSource jndiName="jdbc/StackOverflowDS" >  
38   <jdbcDriver libraryRef="MySQLLib"/>  
39   <properties databaseName="AMT_STACKOVERFLOW"  
40     serverName="localhost" portNumber="9906"  
41     user="root" password="s3cret"/>  
42 </dataSource>  
43 <!-- https://localhost:9443/ibm/api/validation/dataSource/dataSource%5Bdefault-0%5D -->  
44  
45  
server > webApplication  
JdbcPersonRepository.java  
17  
18 @ApplicationScoped  
19 @Named("JdbcPersonRepository")  
20 public class JdbcPersonRepository implements PersonRepository {  
21  
22   @Resource(lookup = "jdbc/StackOverflowDS")  
23   DataSource dataSource;  
24  
25   public JdbcPersonRepository() {  
26   }  
27  
28   public JdbcPersonRepository(DataSource dataSource) { this.dataSource = dataSource; }  
29  
30  
31  
32 @Override  
33 public Optional<Person> findById(PersonId id) {  
34   try {  
35     PreparedStatement statement = dataSource.getConnection().prepareStatement( sql: "SELECT email,  
36     statement.setString( parameterIndex: 1, id.asString());  
37     ResultSet rs = statement.executeQuery();
```

# Install a driver and define a data source

When you download the MySQL driver, pay attention to the version that you get. Remember that version when you dig into detailed documentation.

The screenshot displays an IDE with two main panels. The left panel shows a project structure with a 'lib' directory containing 'mysql-connector-java-8.0.12.jar' and 'server.xml'. The right panel shows the 'server.xml' file with a data source configuration for 'StackOverflowDS' using the 'MySQLLib' library. Below this, the 'JdbcPersonRepository.java' file is shown, implementing the 'PersonRepository' interface and using the 'StackOverflowDS' data source to find people by ID.

```
<!-- Declare the jar files for MySQL access through JDBC. -->
<library id="MySQLLib">
  <fileset dir="${server.config.dir}" includes="*.jar"/>
</library>

<dataSource jndiName="jdbc/StackOverflowDS" >
  <jdbcDriver libraryRef="MySQLLib"/>
  <properties databaseName="AMT_STACKOVERFLOW"
    serverName="localhost" portNumber="9906"
    user="root" password="s3cret"/>
</dataSource>
<!-- https://localhost:9443/ibm/api/validation/dataSource/dataSource%5Bdefault-0%5D -->
```

```
@ApplicationScoped
@Named("JdbcPersonRepository")
public class JdbcPersonRepository implements PersonRepository {

  @Resource(lookup = "jdbc/StackOverflowDS")
  DataSource dataSource;

  public JdbcPersonRepository() {
  }

  public JdbcPersonRepository(DataSource dataSource) { this.dataSource = dataSource; }

  @Override
  public Optional<Person> findById(PersonId id) {
    try {
      PreparedStatement statement = dataSource.getConnection().prepareStatement("SELECT email,
      statement.setString(1, id.asString());
      ResultSet rs = statement.executeQuery();
```



# Define a data source

Ouch... credentials hard-coded in a the code base. See <https://openliberty.io/docs/20.0.0.10/reference/config/server-configuration-overview.html> and look for environment variables

The screenshot shows the project structure of an application. The 'server' directory contains 'webApplication' and 'server.xml'. The 'webApplication' directory contains 'JdbcPersonRepository.java'. The 'server.xml' file is highlighted in the 'server' directory. The 'JdbcPersonRepository.java' file is highlighted in the 'webApplication' directory. The 'JdbcPersonRepository' class is highlighted in the 'Structure' view at the bottom.

The screenshot shows the configuration of a data source in the `server.xml` file and the implementation of the `JdbcPersonRepository` class in `JdbcPersonRepository.java`.

**server.xml configuration:**

```
<!-- Declare the jar files for MySQL access through JDBC. -->
<library id="MySqlLib">
  <fileset dir="{server.config.dir}" includes="*.jar"/>
</library>

<dataSource jndiName="jdbc/StackOverflowDS" >
  <jdbcDriver libraryRef="MySqlLib"/>
  <properties databaseName="AMT_STACKOVERFLOW"
    serverName="localhost" portNumber="9906"
    user="root" password="s3cret"/>
</dataSource>

<!-- https://localhost:9443/ibm/api/validation/dataSource/dataSource%5Bdefault-0%5D -->
```

**JdbcPersonRepository.java implementation:**

```
@ApplicationScoped
@Named("JdbcPersonRepository")
public class JdbcPersonRepository implements PersonRepository {

  @Resource(lookup = "jdbc/StackOverflowDS")
  DataSource dataSource;

  public JdbcPersonRepository() {
  }

  public JdbcPersonRepository(DataSource dataSource) { this.dataSource = dataSource; }

  @Override
  public Optional<Person> findById(PersonId id) {
    try {
      PreparedStatement statement = dataSource.getConnection().prepareStatement( sql: "SELECT email,
      statement.setString( parameterIndex: 1, id.asString());
      ResultSet rs = statement.executeQuery();
```

# Define a data source

This assumes that you have a MySQL accessible via the loopback interface and port 9906 (instead of 3306). In this setup, it runs in a Docker topology.

heig-vd

HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
www.heig-vd.ch

The screenshot displays an IDE with three main panels:

- Left Panel (Project Structure):** Shows a project hierarchy with folders like `person`, `question`, `services`, `vote`, `Answer`, `Id`, `infrastructure`, `persistence`, `jpa`, `memory`, `liberty`, `config`, `resources`, `webapp`, `test`, and `target`. The `server.xml` file is highlighted in the `config` folder.
- Top Right Panel (XML Configuration):** Displays the `server.xml` file. A red circle highlights the `<dataSource>` element, which is configured with the following properties:

```
<!-- Declare the jar files for MySQL access through JDBC. -->
<library id="MySqlLib">
  <fileset dir="${server.config.dir}" includes="*.jar"/>
</library>

<dataSource jndiName="jdbc/StackOverflowDS" >
  <jdbcDriver libraryRef="MySqlLib"/>
  <properties databaseName="AMT_STACKOVERFLOW"
    serverName="localhost" portNumber="9906"
    user="root" password="s3cret"/>
</dataSource>

<!-- https://localhost:9443/ibm/api/validation/dataSource/dataSource%5Bdefault-0%5D -->
```
- Bottom Right Panel (Java Code):** Shows the `JdbcPersonRepository.java` file. The code defines a class that implements `PersonRepository` and uses the data source defined in the XML. A red arrow points from the `dataSource` property in the XML to the `dataSource` field in the Java class.

```
@ApplicationScoped
@Named("JdbcPersonRepository")
public class JdbcPersonRepository implements PersonRepository {

  @Resource(lookup = "jdbc/StackOverflowDS")
  DataSource dataSource;

  public JdbcPersonRepository() {
  }

  public JdbcPersonRepository(DataSource dataSource) { this.dataSource = dataSource; }

  @Override
  public Optional<Person> findById(PersonId id) {
    try {
      PreparedStatement statement = dataSource.getConnection().prepareStatement("SELECT email,
      statement.setString(parameterIndex: 1, id.asString());
      ResultSet rs = statement.executeQuery();
```

# Inject the data source in your code

@Resource works in a similar way as @EJB last week: the container injects a reference to the connection pool into the dataSource variable

The screenshot displays an IDE with three main panels. The left panel shows a project structure with folders like 'person', 'question', 'services', 'vote', 'infrastructure', 'persistence', 'jpa', 'memory', 'liberty', 'config', 'resources', 'webapp', 'test', and 'target'. The 'target' folder contains 'demo.iml', 'pom.xml', and 'readme.md'. The middle panel shows an XML configuration file 'server.xml' with the following content:

```
<!-- Declare the jar files for MySQL access through JDBC. -->
<library id="MySQLLib">
  <fileset dir="${server.config.dir}" includes="*.jar"/>
</library>

<dataSource jndiName="jdbc/StackOverflowDS" >
  <jdbcDriver libraryRef="MySQLLib"/>
  <properties databaseName="AMT_STACKOVERFLOW"
    serverName="localhost" portNumber="9906"
    user="root" password="s3cret"/>
</dataSource>
<!-- https://localhost:9443/ibm/api/validation/dataSource/dataSource%5Bdefault-0%5D -->
```

The right panel shows the Java code for 'JdbcPersonRepository.java'. The code is annotated with '@ApplicationScoped' and '@Named("JdbcPersonRepository")'. A red circle highlights the '@Resource(lookup = "jdbc/StackOverflowDS")' annotation and the 'DataSource dataSource;' variable. The code also includes a constructor and a 'findById' method that uses the injected 'dataSource' to execute a query.

```
@ApplicationScoped
@Named("JdbcPersonRepository")
@Resource(lookup = "jdbc/StackOverflowDS")
DataSource dataSource;

public JdbcPersonRepository() {
}

public JdbcPersonRepository(DataSource dataSource) { this.dataSource = dataSource; }

@Override
public Optional<Person> findById(PersonId id) {
  try {
    PreparedStatement statement = dataSource.getConnection().prepareStatement("SELECT email,
    statement.setString(parameterIndex: 1, id.asString());
    ResultSet rs = statement.executeQuery();
```



# Use the data source

Warning: you need to call **close** on the dataSource (it will actually not be closed, but rather returned into the pool). So use a variable instead of this half-baked code.

The screenshot displays an IDE with three main panels:

- Left Panel (Project Structure):** Shows a project hierarchy. The 'persistence' folder is expanded, showing 'jpa' and 'memory' subfolders. The 'liberty' folder is also expanded, showing 'config' (containing 'mysql-connector-java-8.0.12.jar' and 'server.xml') and 'resources'.
- Top Right Panel (server.xml):** Contains XML configuration for a data source. A comment indicates the purpose: `<!-- Declare the jar files for MySQL access through JDBC. -->`. The configuration includes a `<library id="MySQLLib">` block with a `<fileset>` for MySQL JARs, and a `<dataSource>` block with JNDI name 'jdbc/StackOverflowDS', driver 'MySQLLib', and database properties (databaseName, serverName, portNumber, user, password). A URL comment is also present.
- Bottom Panel (JdbcPersonRepository.java):** Shows the implementation of the `PersonRepository` interface. The class is annotated with `@ApplicationScoped` and `@Named("JdbcPersonRepository")`. It has a `@Resource` field for `DataSource dataSource`. The constructor `public JdbcPersonRepository(DataSource dataSource)` initializes `this.dataSource = dataSource;`. The `findById` method is annotated with `@Override` and uses `dataSource.getConnection().prepareStatement` to execute a SQL query: `SELECT email,`.

# Side note: dependency injection with CDI 2.0

We have seen @Singleton and @Stateless to define managed components (EJB). @ApplicationScoped does the same thing, with a more general API provided by Java EE: CDI. @Inject is used instead of @EJB.

The screenshot displays an IDE with three main views:

- Project Structure (Left):** Shows a project with packages like `person`, `question`, `services`, `vote`, `Answer`, `Id`, `infrastructure`, `persistence`, `jpa`, `memory`, `ui`, `liberty`, `config`, `resources`, `webapp`, `test`, and `target`. The `liberty` folder contains `mysql-connector-java-8.0.12.jar` and `server.xml`.
- server.xml (Middle):** Contains XML configuration for a data source:

```
<!-- Declare the jar files for MySQL access through JDBC. -->
<library id="MySQLLib">
  <fileset dir="{server.config.dir}" includes="*.jar"/>
</library>

<dataSource jndiName="jdbc/StackOverflowDS" >
  <jdbcDriver libraryRef="MySQLLib"/>
  <properties databaseName="AMT_STACKOVERFLOW"
    serverName="localhost" portNumber="3306"
    user="root" pas
</dataSource>
<!-- https://localhost:9443/ibm
```
- Java Code (Right):** Shows two files:
  - `JdbcPersonRepository.java`:

```
@ApplicationScoped
@Named("JdbcPersonRepository")
public class JdbcPersonRepository {

    @Resource(lookup = "jdbc/StackOverflowDS")
    DataSource dataSource;

    public JdbcPersonRepository() {}

    public JdbcPersonRepository(DataSource dataSource) { this.dataSource = dataSource; }

    @Override
    public Optional<Person> findById(PersonId id) {
        try {
            PreparedStatement statement = dataSource.getConnection().prepareStatement("SELECT email,");
            statement.setString(1, id.asString());
            ResultSet rs = statement.executeQuery();
```
  - `ServiceRegistry.java`:

```
@ApplicationScoped
public class ServiceRegistry {

    @Inject @Named("InMemoryPersonRepository")
    PersonRepository personRepository;

    @Inject @Named("InMemoryQuestionRepository")
    QuestionRepository questionRepository;

    @Inject @Named("InMemoryVoteRepository")
    VoteRepository voteRepository;
```