

Lecture 3: The Data Access Tier (1)

Olivier Liechti
AMT

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Agenda

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

13h00 - 13h15	15'	Lecture Feedback & discussion about “session” lab
13h15- 14h15	60'	Lecture The DAO pattern & JDBC
14h15 - 15h15	45'	Setup (instructions) Scripting setup of a Glassfish domain + first tests with JDBC
<i>15h00 - 15h30</i>		Break
15h30 - 15h45	15'	Lab introduction Implement the DAO pattern with JDBC.
15h45 - 18h00	135'	Lab Implement the DAO pattern with JDBC.



Last week...

2 key concepts

- **managed components & dependency injection**



We have seen that the application server is **managing the lifecycle** of servlets, EJBs, etc.

In particular, it is the app server that **creates the instances**, not the developer!



We have also seen that with **annotations**, the developer can **declare** that a component **depends** on another one (e.g. a servlet depends on an EJB).

When the container creates the annotated component, it **injects** a reference into the variable.

This is an **alternative** to doing a JNDI **lookup**.

These 2 concepts also apply in the context of Data Access

- A **data source** is also a **managed resource**.
- Components (servlets and EJBs) can have **dependencies on data sources**. The app server can inject these dependencies into the components.

We also talked about...

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **resource pooling**



The EJB container is managing a pool of EJB instances. This provides a way to throttle client requests

On one hand, we can handle several clients **concurrently** (in a thread-safe way)

On the other hand, we can limit the number of clients served in parallel (others are **queued**).

By adjusting the size of the EJB pool, the administrator can balance the contention on shared resources with the queuing time.

In addition, by creating the instances in advance (they are in a “ready to serve” state in the pool), we get a **performance gain**.

Pooling also applies in the context of Data Access

- A **data source** is a shared resource (multiple clients interact with a DB concurrently).
- Without any control, the data source could be **overloaded** (running out of RAM, CPU, sockets, etc.).
- Using a **pool of DB connections** and queuing clients is a way to regulate the system.
- Establishing a connection with a remote RDBMS is time consuming. So by having “**ready to use**” connections in the pool has a significant and positive **performance** impact.



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.

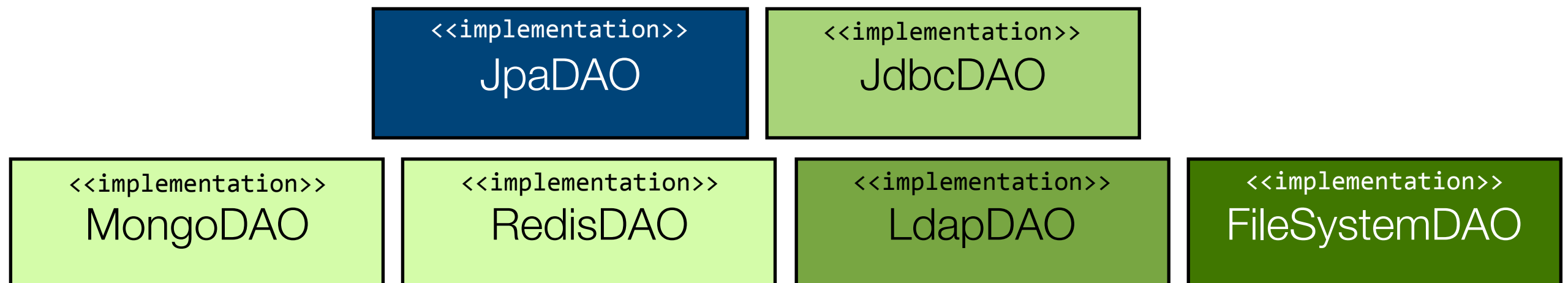
(*) This is true only to some extent... you cannot completely forget about it, for instance for performance reasons

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

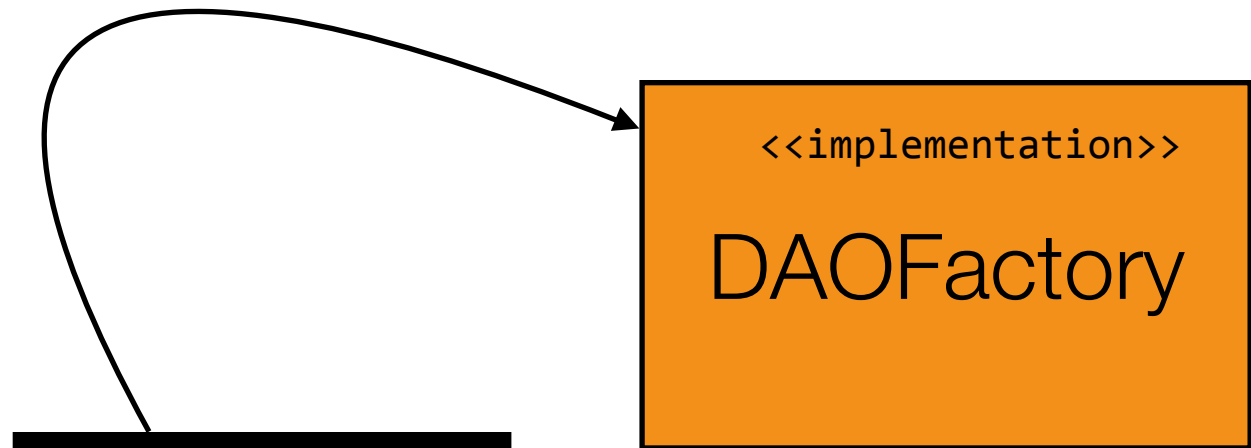


The **Business Service** uses the
DAO interface to interact with a
particular DAO implementation

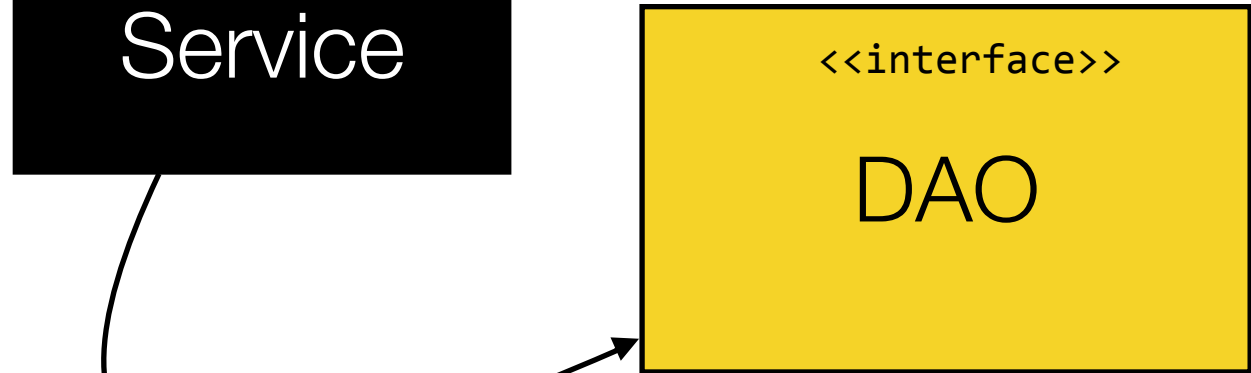
DAO implementations handle
interactions with specific data stores



Give me a DAO implementation!



DAO getDAO();



Do a CRUD operation for me!

long create(T object);
delete(long id);
update(T object);
findById(long);
findAll();
findByXXX(Object k);
findByYYY(Object k);

<<implementation>>
JdbcDAO

<<implementation>>
JpaDAO

<<implementation>>
MongoDAO

<<implementation>>
RedisDAO

<<implementation>>
LdapDAO

<<implementation>>
FileSystemDAO

MySQL

PostgreSQL

Oracle

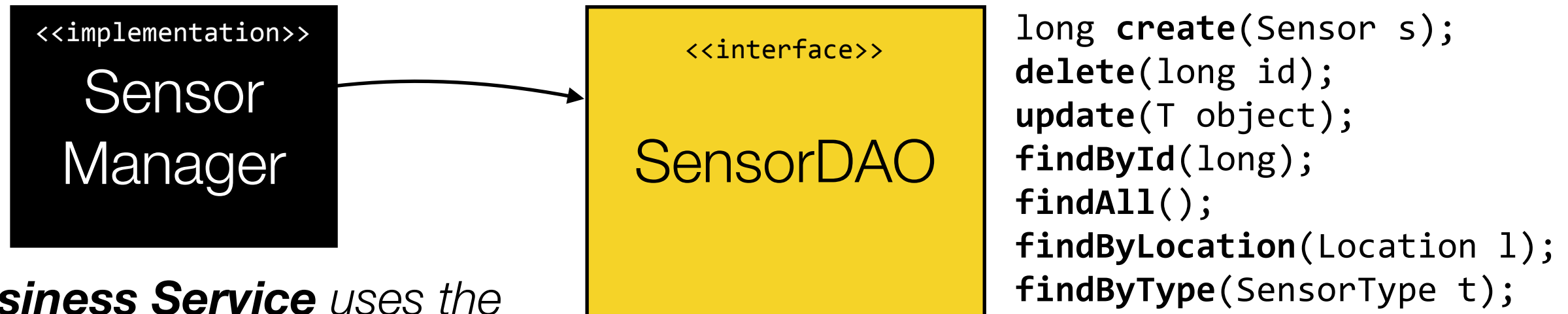
MongoDB

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

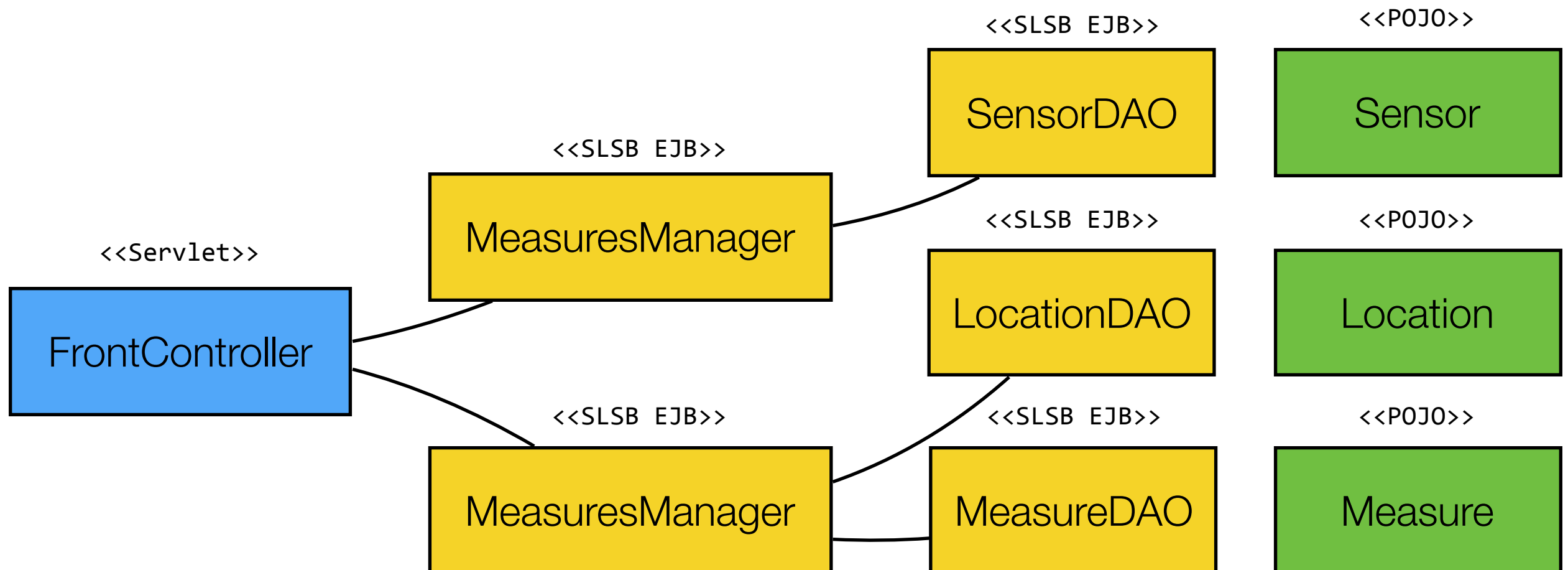
DAO implementations handle
interactions with specific data stores





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:



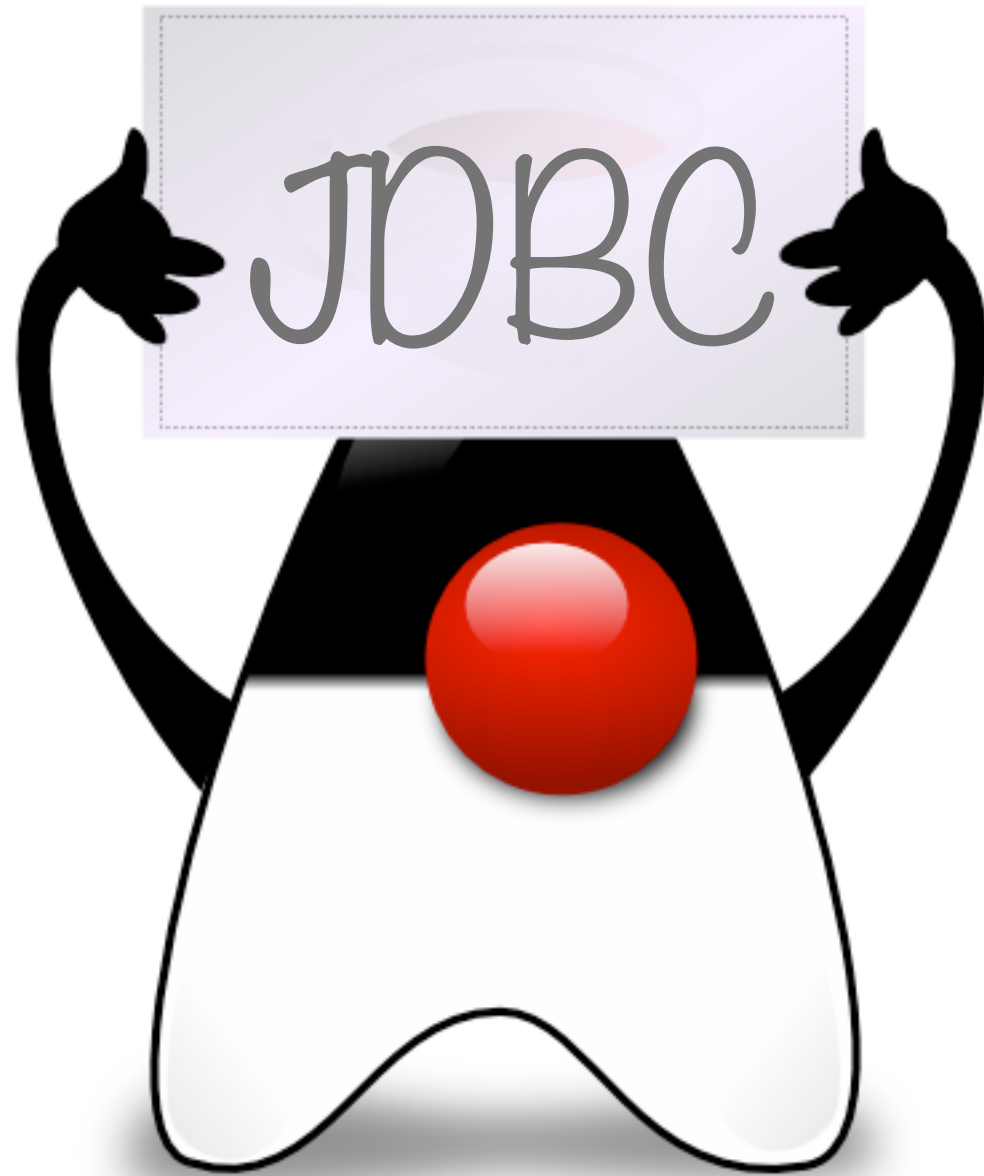
The **name**, **beanName** and **mappedName** annotation attributes have different purposes.

```
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){}
}
```



Java DataBase Connectivity



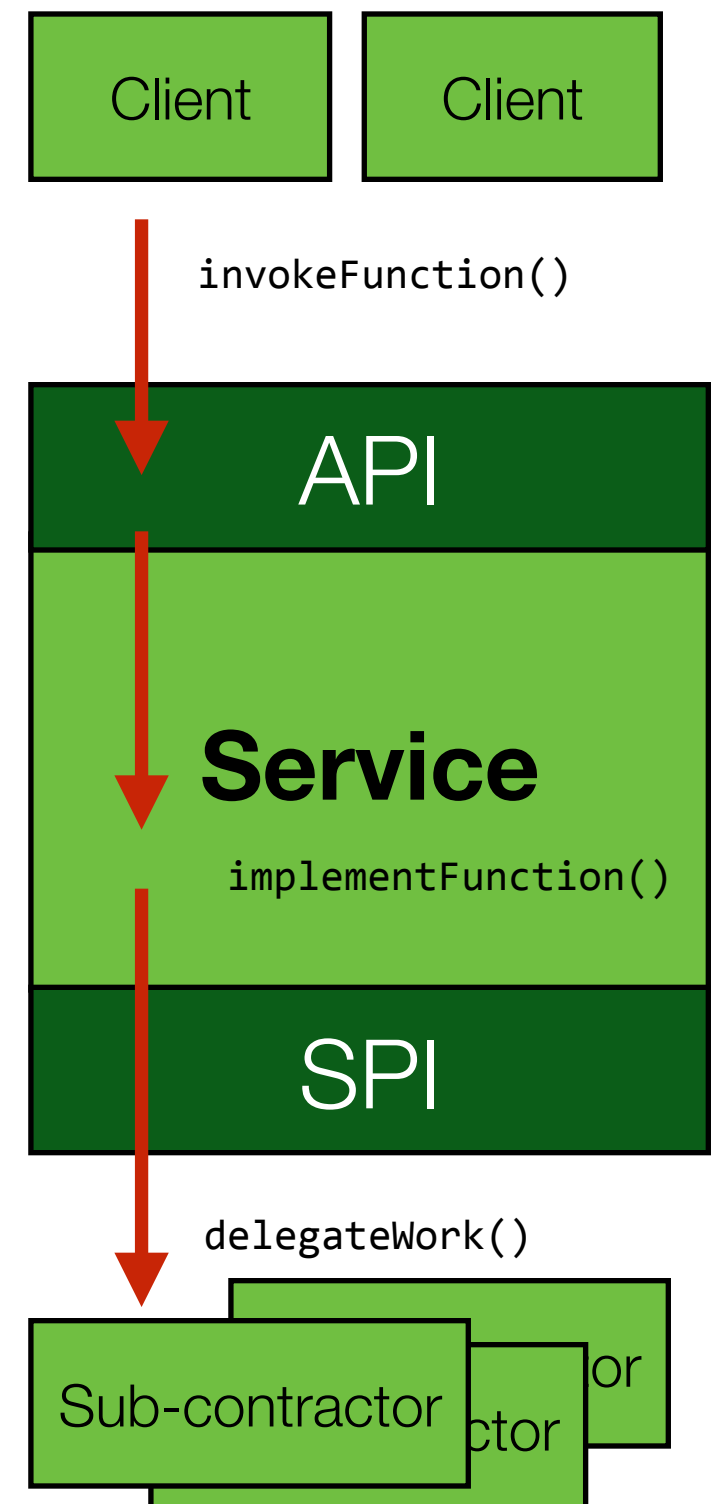
What is **JDBC**?

- 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 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.

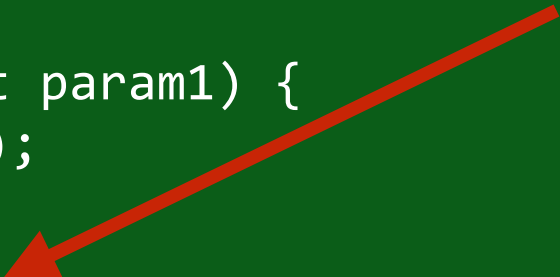




What is the difference between an **API** and a **SPI**?

```
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(); 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); doSomethingNotExposedInAPI();  
    }  
  
    public void registerServiceProvider(ServiceSPI provider) {  
        this.provider = provider  
    }  
}
```

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



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



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:
 - `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**.
 - `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 **SPI**
implementations

Used by **API**
clients



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 SomeDriver());  
    }  
    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

Browser window: 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/__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

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/__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

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	akAUKLJdf!!882_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



Setup



Let's **automate** the creation and configuration of our Glassfish development domain

- Until today, we have used the **default domain (domain1)**, created automagically at installation time.
- Some of you have already had issues with **corrupted domains**. They have used either **Netbeans** or the **asadmin** command line tool to **delete** the domain and **recreate it**.
- During the semester, we will **increasingly automate the build and deployment process** for our Java EE applications. We will start today.



1. We need a **database!** A **clean** database!

- Let's assume that we have installed MySQL on our development machine.
- We want to **create a database** for our application.
- We also want to **create a technical user**, with credentials, to establish communications between our application and MySQL.
- We have to make sure that the technical user has the **permissions** to work with our database.
- We do NOT want to do that manually. **We want to write a script that does that, in an automated and repeatable way.**



2. What do I need to do in **MySQL**?

When you write your script (for instance with **bash**), you will want to use **variables**. This is much better than repeating database or user names throughout the script.

```
DB_NAME=AMTDatabase  
DB_TECHNICAL_USER=AMTEchnicalUser  
DB_TECHNICAL_USER_PASSWORD=dUke!1400$
```

In our current setup, we want to **start with a clean, fresh database**. So let's get rid of the previous one (if it exists), before creating a new one.

```
DROP DATABASE $DB_NAME;  
CREATE DATABASE $DB_NAME;
```



2. What do I need to do in **MySQL**?

Once our database is created, let's **create a technical user**. MySQL is making a difference between a user who accesses the server from the same machine (localhost) and the same user who accesses the server from a remote machine.

```
DROP USER '$DB_TECHNICAL_USER'@'localhost';  
DROP USER '$DB_TECHNICAL_USER'@'%';
```

```
CREATE USER '$DB_TECHNICAL_USER'@'localhost' IDENTIFIED BY  
'$DB_TECHNICAL_USER_PASSWORD';
```

```
CREATE USER '$DB_TECHNICAL_USER'@'%' IDENTIFIED BY '$DB_TECHNICAL_USER_PASSWORD';
```

Finally, let's give **permissions** to the technical user to do whatever he wants we our new database.

```
GRANT ALL PRIVILEGES ON $DB_NAME.* TO '$DB_TECHNICAL_USER'@'localhost';  
GRANT ALL PRIVILEGES ON $DB_NAME.* TO '$DB_TECHNICAL_USER'@'%';
```



3. What do I need to do in **Glassfish**?

The first thing is to (re)create a fresh domain, from scratch. But since we may already have one, we first need to stop and delete it. Let's use the **asadmin** command for that.

```
DOMAIN_NAME=domainAMT
```

```
asadmin stop-domain $DOMAIN_NAME
```

```
asadmin delete-domain $DOMAIN_NAME
```

```
asadmin create-domain --nopassword=true $DOMAIN_NAME
```

The applications deployed in our domain will want to connect to our MySQL database. For that purpose, we need to **copy the MySQL jdbc driver** into the **lib folder**, under our new **domain folder**.

```
cp mysql-connector-java-5.1.33-bin.jar ../domains/$DOMAIN_NAME/lib
```



3. What do I need to do in **Glassfish**?

Now that we are done with the basic domain setup, we can **start** it.

```
asadmin start-domain $DOMAIN_NAME
```

The last thing that we need for now, is to add a **jdbc connection pool** and a **jdbc resource** in our domain. Of course, **asadmin** provides a way to do that.

```
asadmin create-jdbc-connection-pool \  
  --restype=javax.sql.XADataSource \  
  --datasourceclassname=com.mysql.jdbc.jdbc2.optional.MysqlXADataSource \  
  --property User=$DB_TECHNICAL_USER:Password=  
$DB_TECHNICAL_USER_PASSWORD:serverName=localhost:portNumber=3306:databaseName=  
$DB_NAME $JDBC_CONNECTION_POOL_NAME
```

```
./asadmin create-jdbc-resource --connectionpoolid $JDBC_CONNECTION_POOL_NAME  
$JDBC_JNDI_NAME
```

We can even use asadmin to **ping** our database and validate our setup.

```
asadmin ping-connection-pool $JDBC_CONNECTION_POOL_NAME
```



4. What if I want to **feed the DB**?

In some cases, it's interesting to already **create tables and rows** in the script (makes testing easier during development):

```
USE $DB_NAME;
CREATE TABLE `sensors` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `description` tinytext NOT NULL,
  `type` tinytext NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

INSERT INTO `sensors` (`id`, `description`, `type`) VALUES (NULL, 'ROOM_1', 'TEMPERATURE');
INSERT INTO `sensors` (`id`, `description`, `type`) VALUES (NULL, 'ROOM_2', 'TEMPERATURE');
INSERT INTO `sensors` (`id`, `description`, `type`) VALUES (NULL, 'ROOM_31', 'TEMPERATURE');
INSERT INTO `sensors` (`id`, `description`, `type`) VALUES (NULL, 'CAR_12', 'SPEED');
INSERT INTO `sensors` (`id`, `description`, `type`) VALUES (NULL, 'CAR_99', 'SPEED');
```




To **validate** your script, you should...

- Run your script several times in a row, without any error.
- Check the state of the domain in the **web console**:

The screenshot displays the JBoss Web Console interface. On the left, a tree view under 'JDBC' shows 'JDBC Resources' and 'JDBC Connection Pools'. A red arrow points from the 'JDBC Resources' folder to the 'AMTDatabase_pool' entry under 'JDBC Connection Pools'. A blue arrow points from the 'AMTDatabase_pool' entry to the 'Additional Properties' tab on the right.

At the top right, the 'JNDI Name' is 'jdbc/AMTDatabase' and the 'Pool Name' is 'AMTDatabase_pool'. Below this, a link says 'Use the [JDBC Connection Pools](#) page to create new pools'.

The 'Edit JDBC Connection Pool Properties' page has three tabs: 'General', 'Advanced', and 'Additional Properties'. The 'Additional Properties' tab is active, showing a table of properties for the 'AMTDatabase_pool'.

Select	Name	Value
<input type="checkbox"/>	Password	dUke!1400\$
<input type="checkbox"/>	portNumber	3306
<input type="checkbox"/>	databaseName	AMTDatabase
<input type="checkbox"/>	serverName	localhost
<input type="checkbox"/>	User	AMTTechnicalUser

At the bottom left, a 'Ping Succeeded' message is shown with a green checkmark. Below it, the 'Edit JDBC Connection Pool' page is visible, showing a 'Load Defaults' button and a 'Ping' button.



Lab



What have built so far?

- We have implemented a **Front Controller servlet**, which was delegating some of its work to a **Stateless Session Bean** (business logic) and to a **JSP** (presentation logic).
- Last week, we have implemented an **in-memory data store**. Our **MeasuresManager EJB** was using our **DataManager EJB**.
- In other words, we already implemented a simple version of the DAO pattern:
 - It was not possible to **update** or **delete** records.
 - It was only possible to get the **full list of records** (no other finders)
 - There was no persistence.



What do we want to build this week?

- We want to be able to **manage sensors** and to store the related data in a **MySQL database**.
- It should be possible to **create, update, delete** and **retrieve** sensors (all sensors, one sensor by id, a list of sensors by type, a list of sensors by description).
- We want to apply the **DAO pattern** and use **JDBC** for this purpose.



What do we want to build this week?

- **In the Web UI:**

- At the minimum we want to have a page that lists all sensor data in a page.
- It would be nice to be able to filter (by id, by type, by description).
- It would be nice to be able to create, edit and delete from the

- **Using Postman:**

- We want to be able to **read, create, update** and **delete** sensors.



In which order should we build things?

- **Start from the back-end:**

- Implement the **Sensor** class (POJO) . At the minimum, a sensor has a unique id, a textual description and a textual type.
- Define the **SensorDAO interface** (as a @Local interface), with the CRUD operations. Add the finder methods as described in the previous slide.
- Implement a **SensorJdbcDAO Stateless Session Bean** that implements the previous interface. Use JDBC to implement the methods.
- Invoke the methods from a servlet and use the debugger to validate the behavior of your implementation.



In which order should we build things?

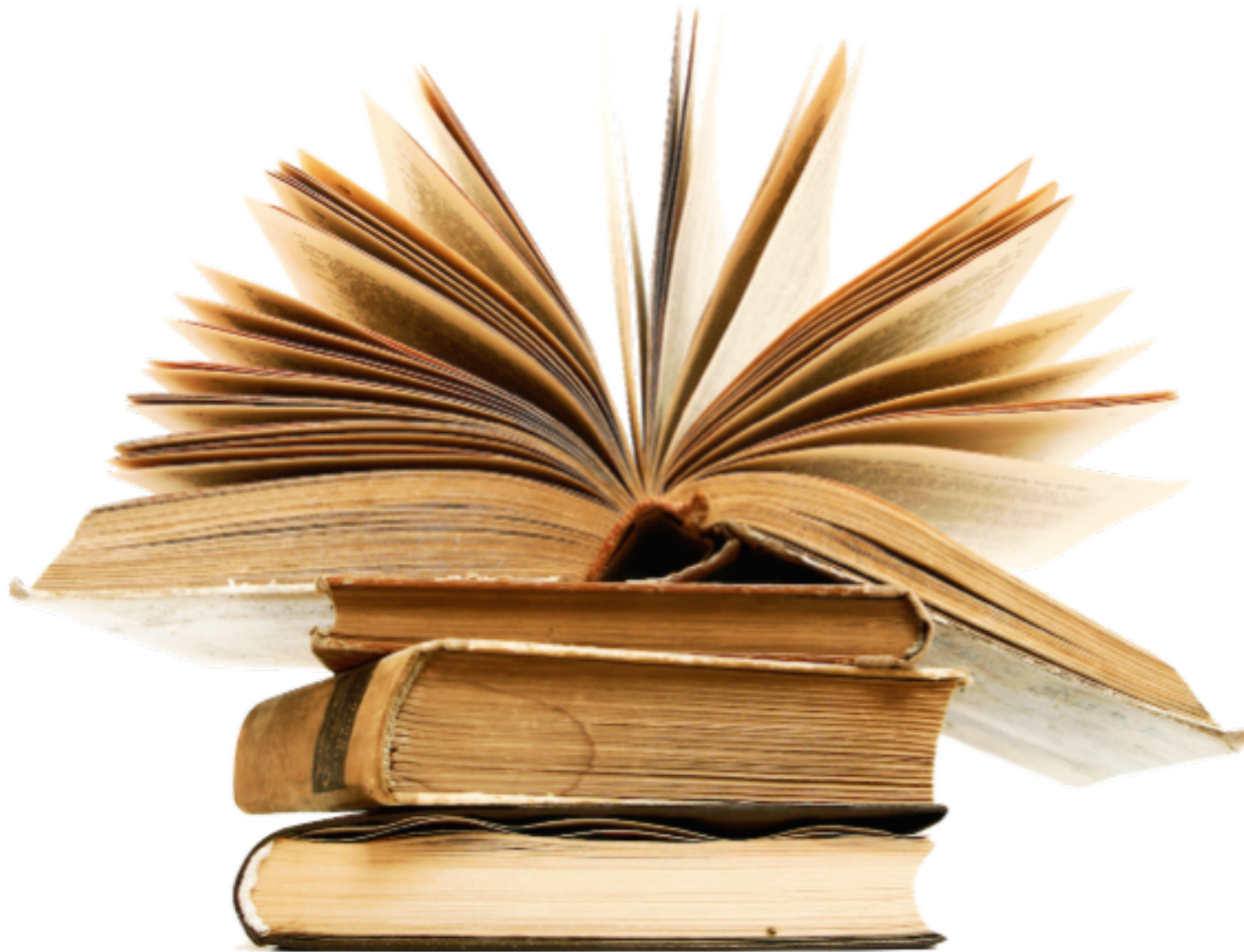
- **Move towards the front-end:**

- Implement a (new) **Front Controller servlet** to handle requests related to sensor operations. You may call it SensorFCServlet.
- You have different ways to structure your code, but one possibility is:
 - to define “something” in the request that defines which CRUD operation is invoked. That “something” could be a query string parameter (e.g. `http://sensors?action=delete`)
 - to define “something” in the request that defines what is the **target** of the operation (e.g. `http://sensors?action=delete&id=334`)
 - to define “something” in the request that defines parameters to the operation (e.g. `http://sensors?action=find&by=sensorType&value=temperature`)
 - to send sensor data as the **payload** of a **POST** request for create and update operations (e.g. `http://sensors?action=update&id=334`).



In which order should we build things?

- **Finish with the front end:**
 - Test all of your operations with Postman.
 - Implement the JSP pages to list and ideally to create, update and delete sensors from the web UI.



References

MUST READ for the Tests

- **Selected sections from the JDBC tutorial**
 - <http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html>
 - <http://docs.oracle.com/javase/tutorial/jdbc/basics/processingstatements.html>
 - <http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html>
 - <http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>
- **Selected sections from the Java EE 7 tutorial**
 - <http://docs.oracle.com/javaee/7/tutorial/doc/resource-creation002.htm#BNCJJ>



What is the difference between a **JDBC resource** and a **JDBC connection pool**?



What are **two methods for adding a jdbc connection pool** to a Glassfish domain?



What is one reason for using a JDBC **Statement** rather than a JDBC **PreparedStatement**?



What is one reason for using a JDBC **PreparedStatement** rather than a JDBC **Statement**?



In JDBC, how does the **DriverManager** class know which drivers are available? How does the registration process happen?



What is the issue when **two Stateless Session Beans** implement the same local interface?



With JDBC, it is possible to obtain **metadata** about returned result sets. Why is that useful?



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



An HTTP client sends a request to find information stored in the database (e.g. a sensor, by giving its unique id). Draw a **sequence diagram** that illustrates the end-to-end process.



What is **connection pooling** and why is it important?