

04 - Persistence tier (1)

The DAO pattern & JDBC

AMT 2019

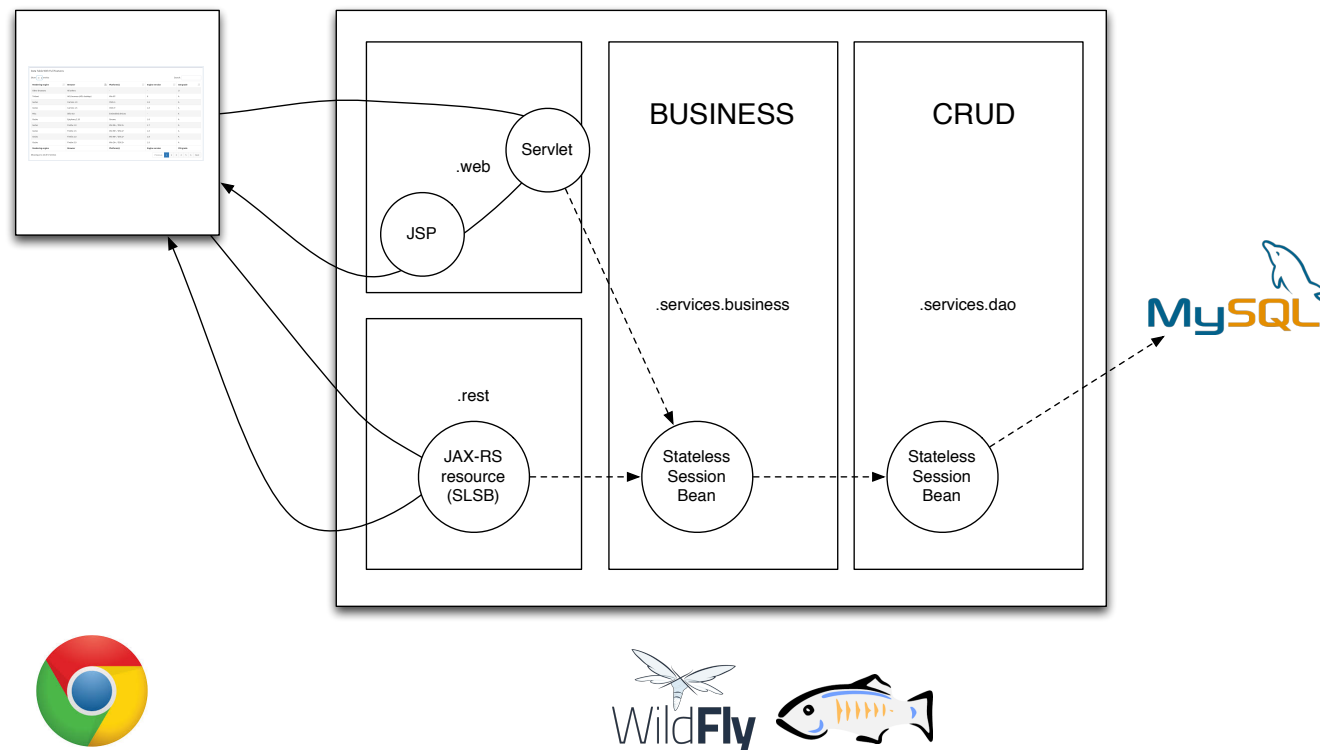
Olivier Liechti

The Persistence Tier (DAO pattern, JDBC, intro to reflection)

Break

Live coding: multi-tiered app (notes)

Live coding: multi-tiered app (notes)



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

17	 A small thumbnail image showing a diagram of a database connection architecture with various components and arrows.	Bootcamp 4.1: Intro aux webcasts "tiers d'accès aux données avec JDBC" by oliechti	4:07
18	 A small thumbnail image showing a diagram of a Docker Compose topology with services and their interconnections.	Bootcamp 4.2: ajout de mysql et phpmyadmin dans notre topologie docker-compose by oliechti	10:53
19	 A small thumbnail image showing a diagram of a Glassfish configuration setup with various modules and settings.	Bootcamp 4.3: configuration de Glassfish by oliechti	17:20
20	 A small thumbnail image showing a diagram of a Docker data source configuration with containers and data flows.	Bootcamp 4.4: configuration de la data source dans Docker by oliechti	8:00
21	 A small thumbnail image showing a diagram of a Wildfly web interface configuration with various parameters and settings.	Bootcamp 4.5: configuration de Wildfly via l'interface web by oliechti	7:16
22	 A small thumbnail image showing a diagram of a Wildfly Docker configuration with containers and their dependencies.	Bootcamp 4.6: configuration de Wildfly via Docker by oliechti	24:30
23	 A small thumbnail image showing a diagram of a Glassfish code and test environment with various components and workflows.	Bootcamp 4.7: écriture du code et test dans Glassfish by oliechti	8:17



The DAO Design Pattern

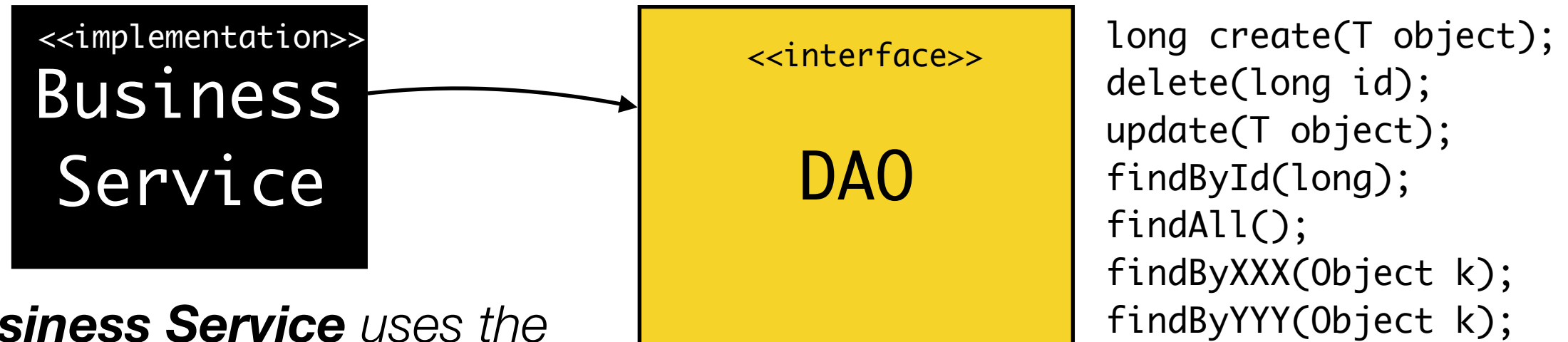


What is the **DAO design pattern** and what are its benefits?

- 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.

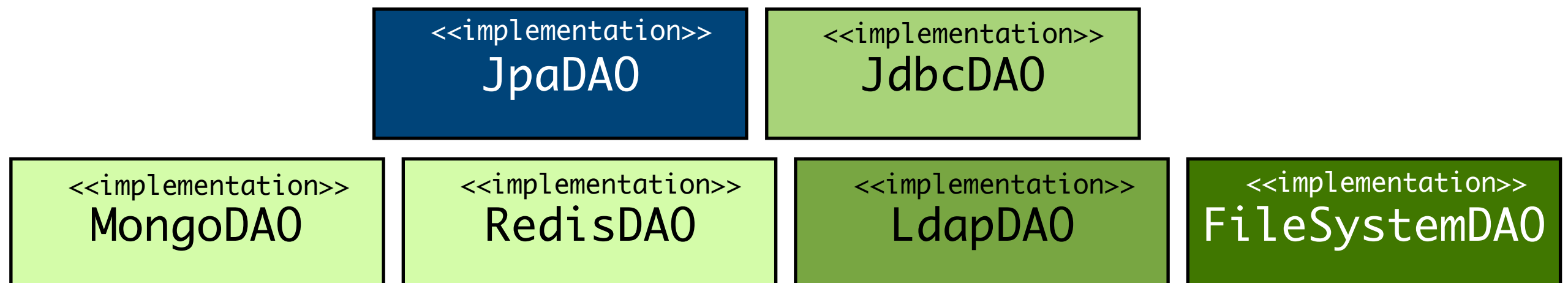
(*) This is true only to some extent... you cannot completely forget about it, for instance for performance reasons

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

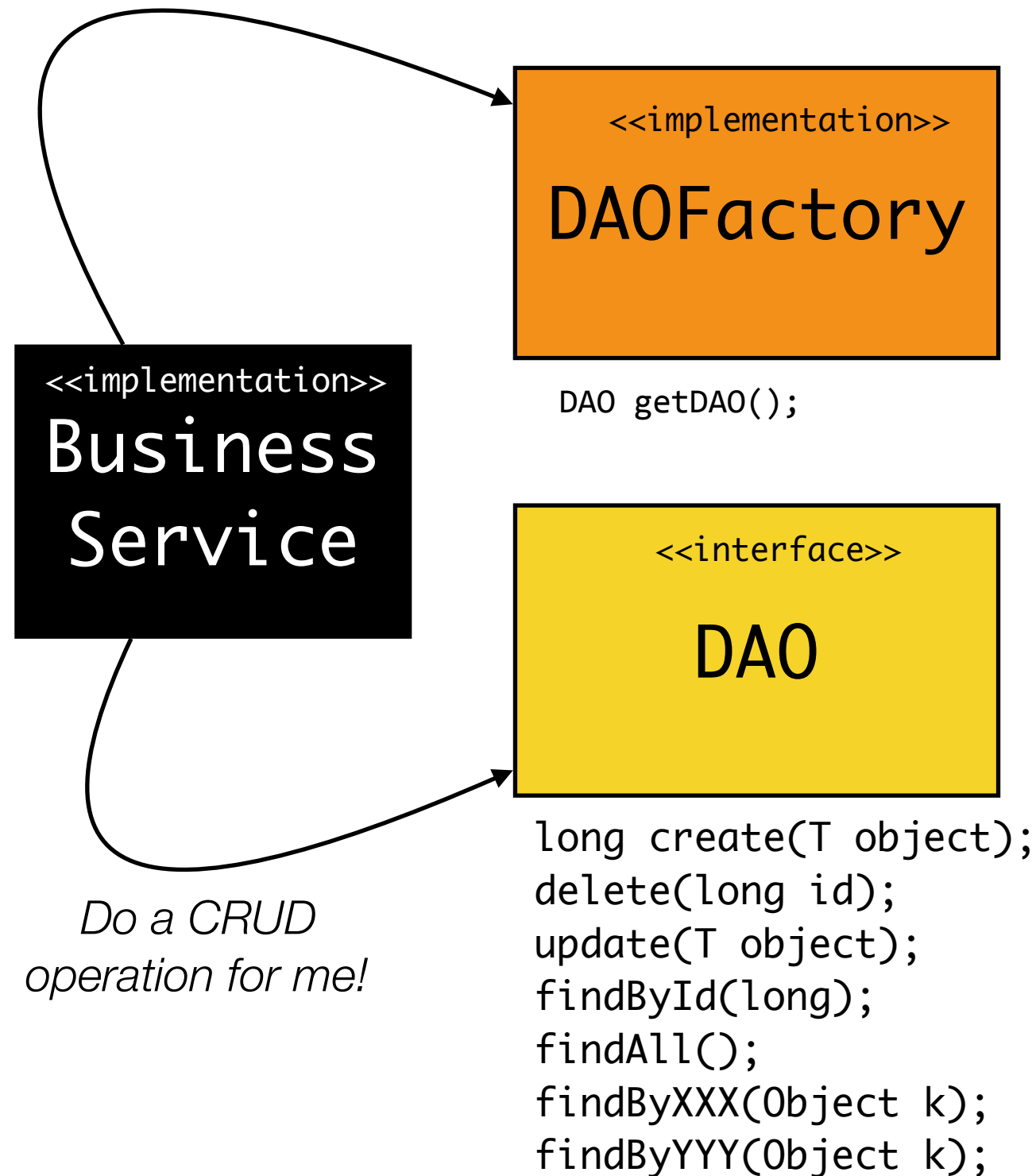


The **Business Service** uses the
DAO interface to interact with a
particular DAO implementation

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

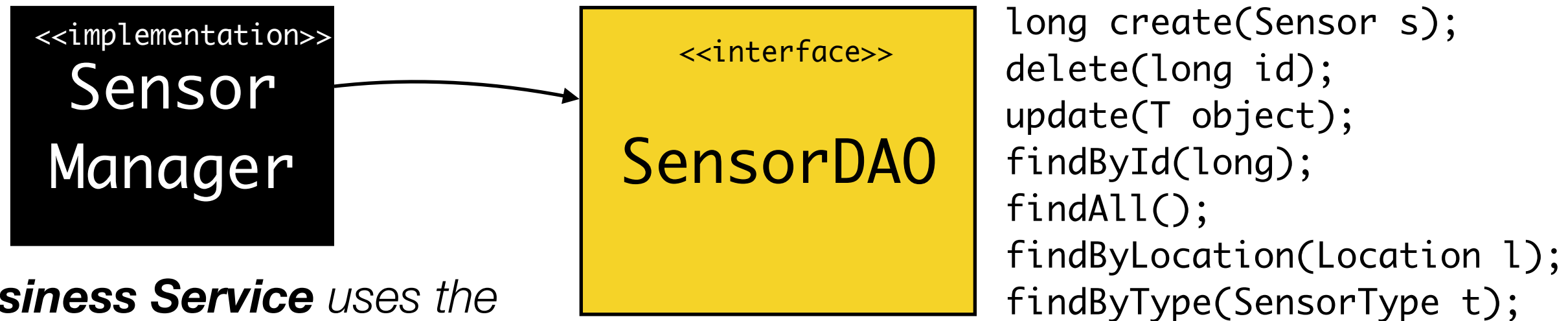
MongoDB

redis

LDAP server

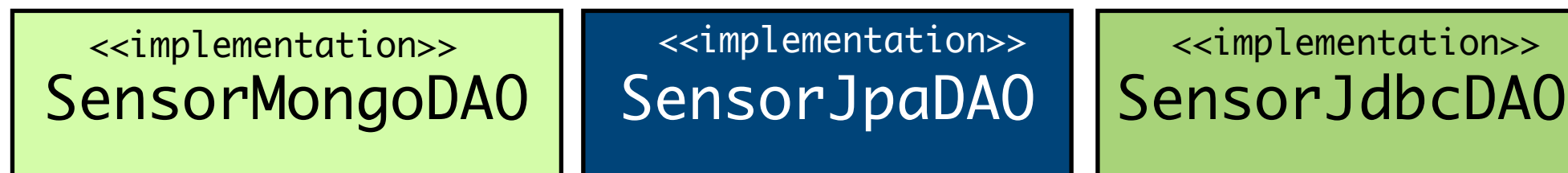
File System

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



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

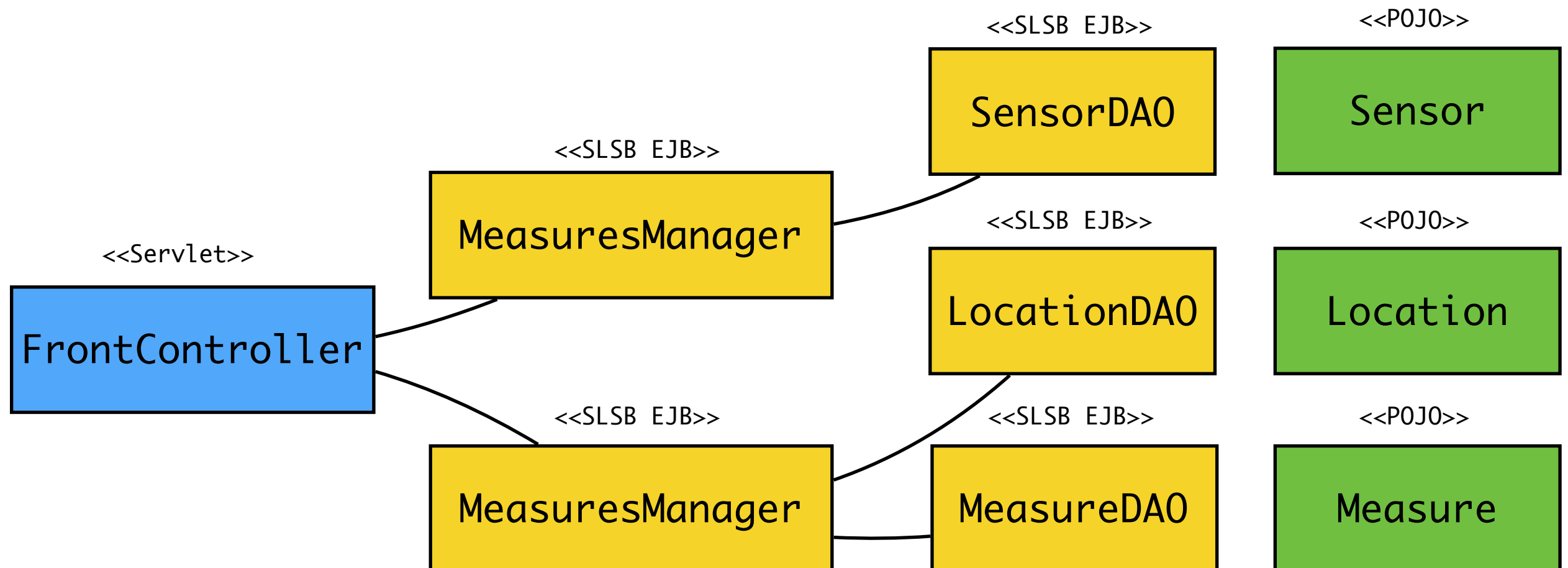
DAO implementations handle
interactions with specific data stores





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.

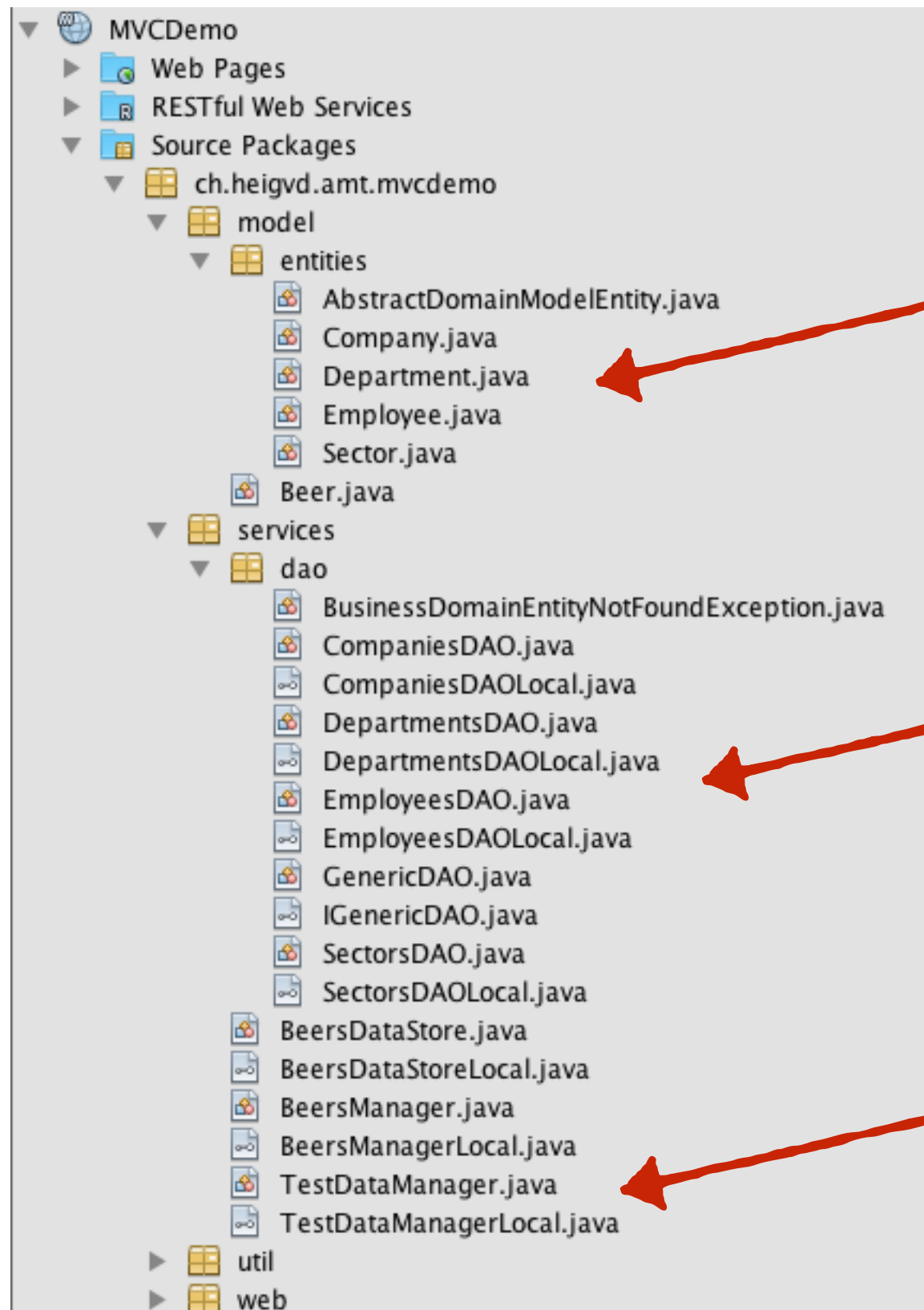
```
public class MyServlet extends HttpServlet {  
    @EJB(beanName="SensorJdbcDAO")  
    SensorDAOLocal sensorDAO;  
}
```

```
@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){}  
}
```

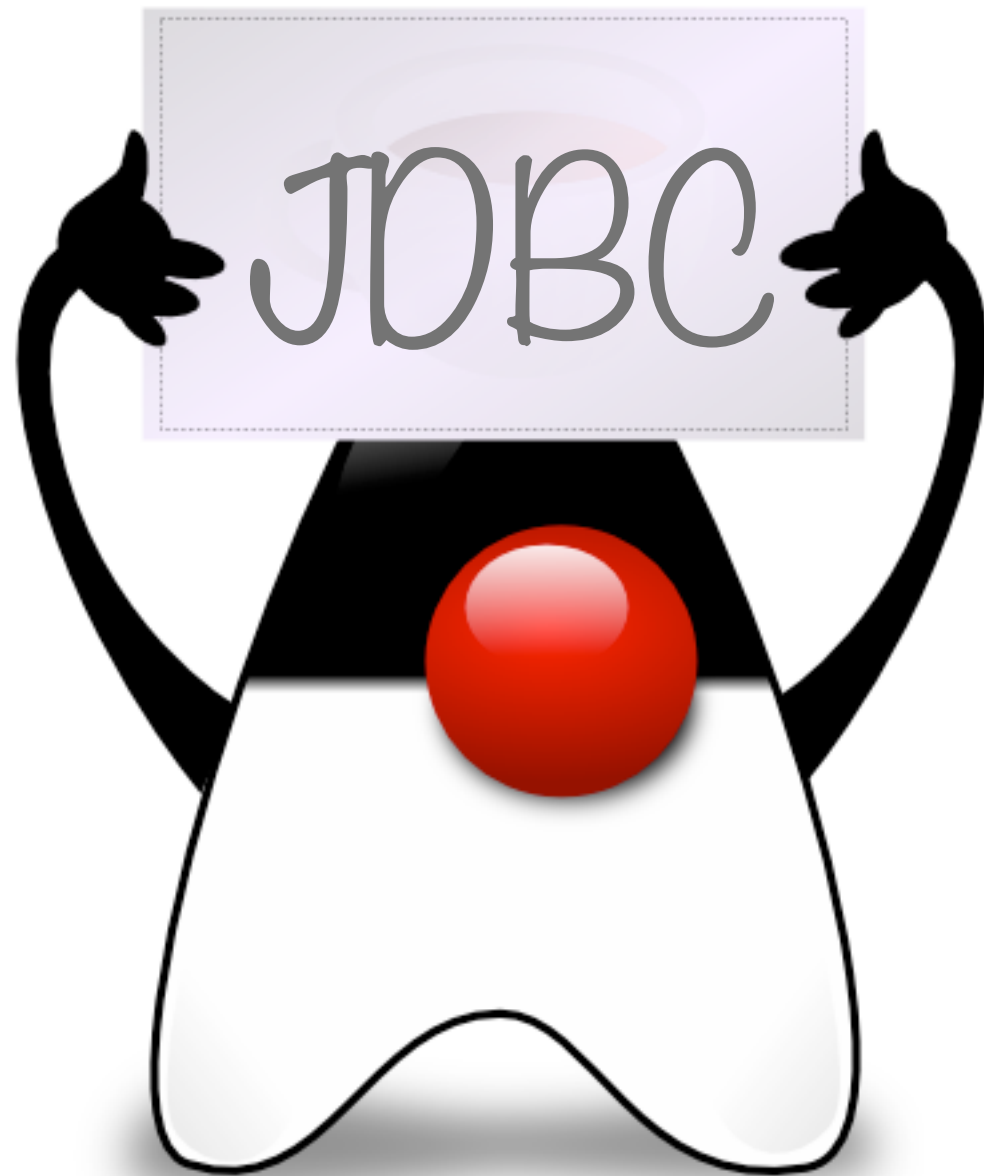
DAO in the MVCDemo Project



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)

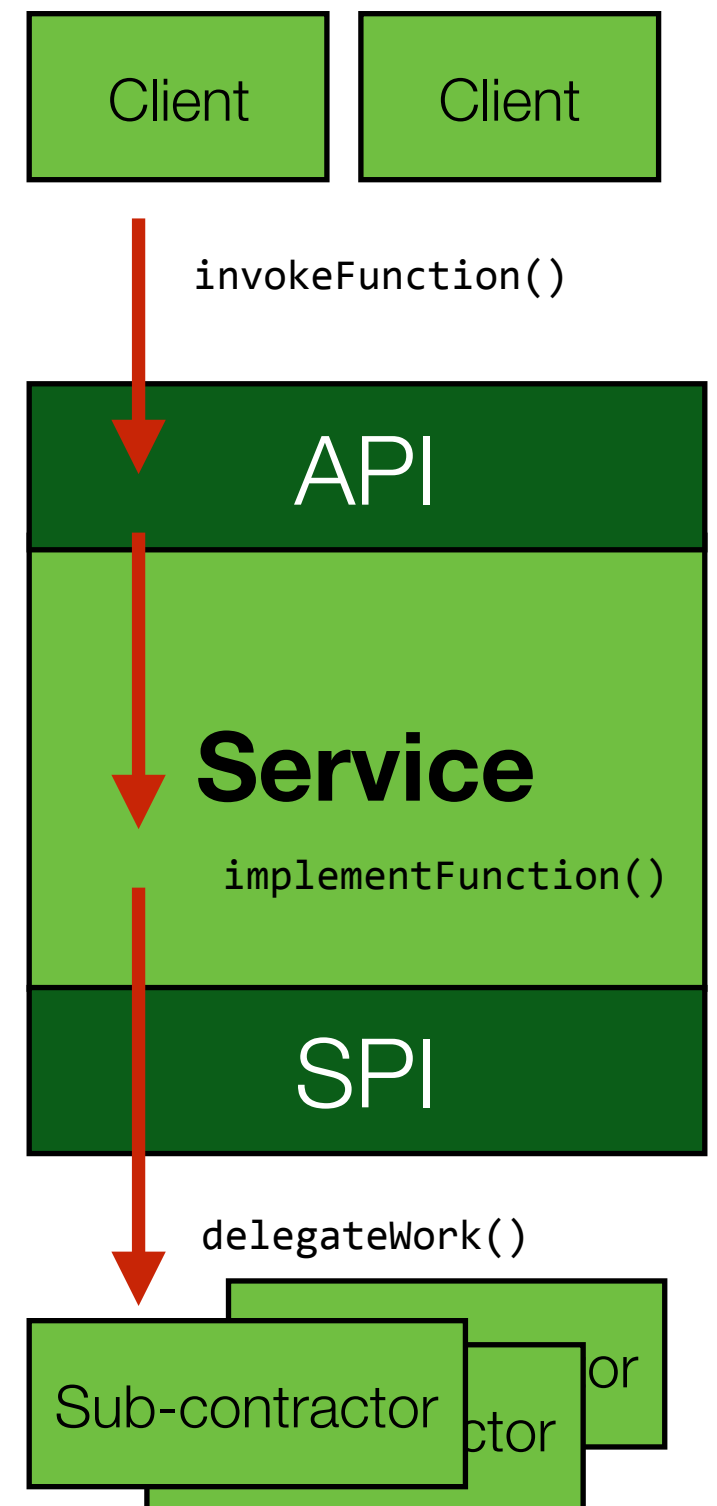
JDBC MySQL driver

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/___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	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

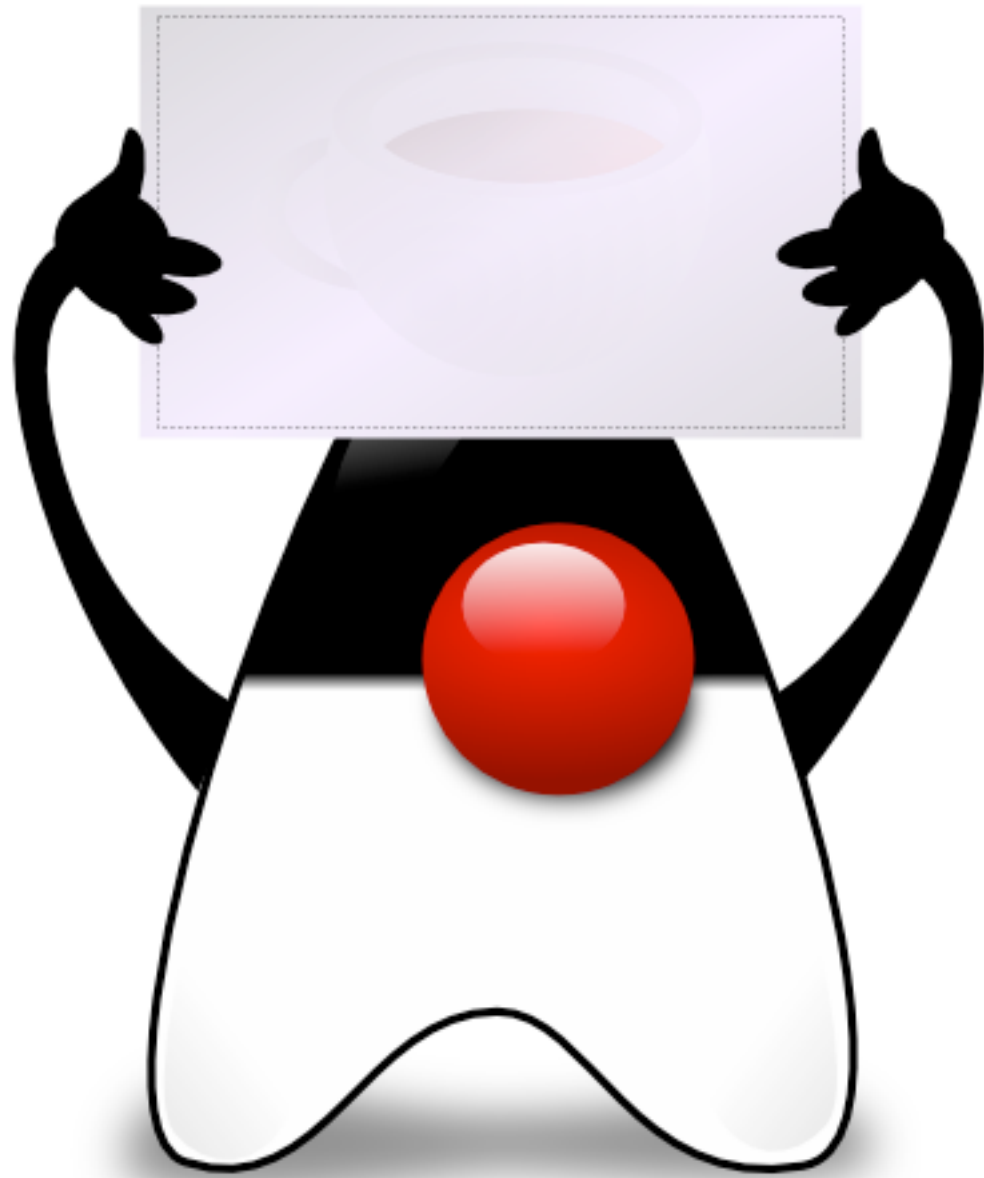
get a connection from the pool

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 without copy-paste



JDBC is pretty straightforward, but... isn't it **verbose and repetitive**?

```
@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;
    }
}
```

When I implement the UserDAO, the RoleDAO, the LocationDAO, will I need to **repeat** all the code around those statements (**boilerplate**)?

Will I need to manually replace the **table and column names** in each DAO?

And when I **maintain** my application, what happens when a new property is added? Do I have to **update my DAO**?


```
@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);
            }

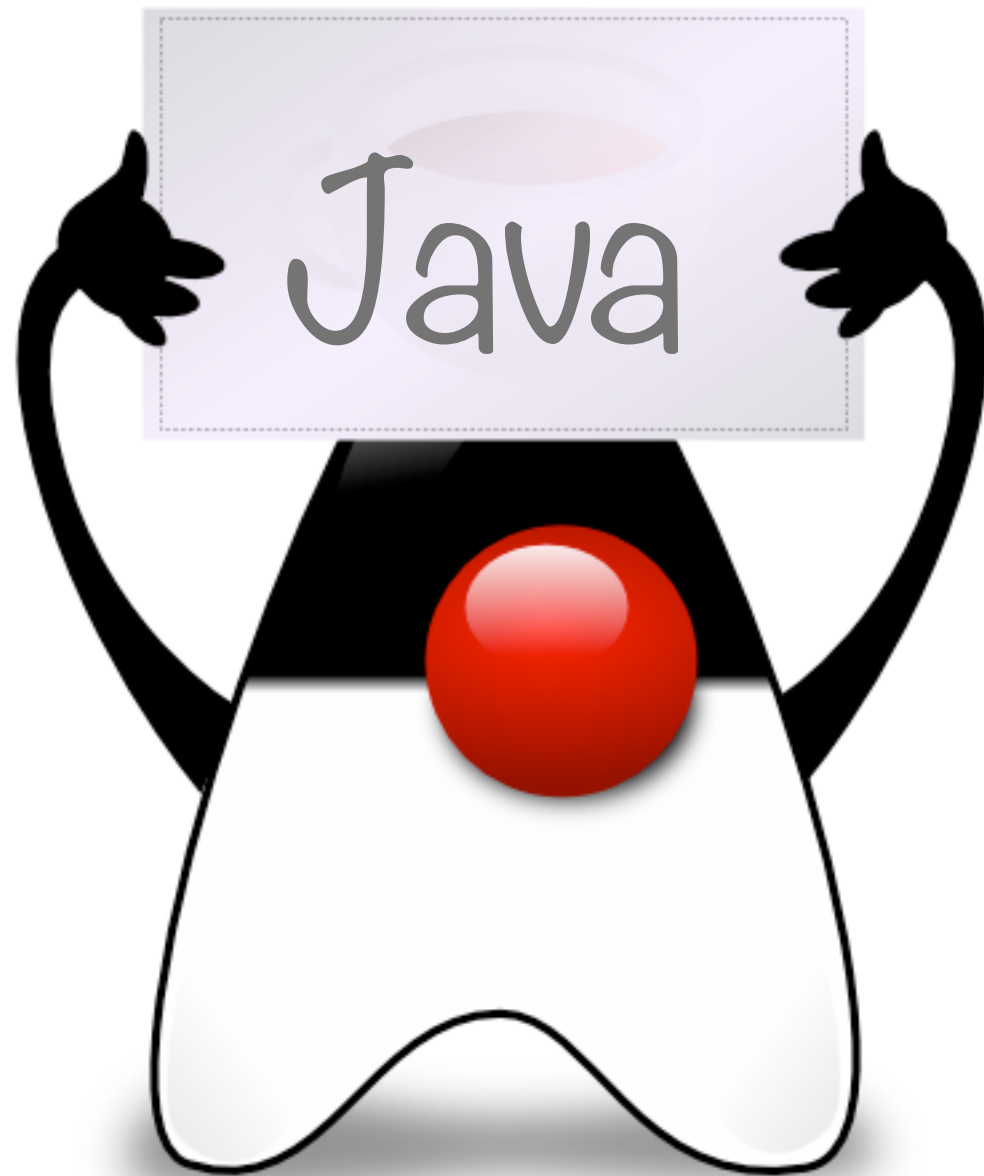
            ps.close();
            con.close();
        } catch (SQLException ex) {
            Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return result;
    }
}
```

Reflection sounds cool. Can't we use it to deal with JDBC in **more generic** ways?

JDBC gives me **metadata** about the DB schema.

Reflection gives me ways to dynamically find and **invoke methods** on Java objects.

Can we combine these features to make this code better?



Java Reflection & JavaBeans



What is the Java **Reflection** API?

- Reflection is a mechanism, through which a program can **inspect and manipulate** its structure and behavior **at runtime**.
- In Java, this means that a program can **get information about classes, their fields, their methods, etc.**
- In Java, this also means that a program can **create instances of classes dynamically** (based on their names, as in the example of JDBC drivers), **invoke methods**, etc.

`java.lang.Class`

`java.lang.reflect.Method`

`java.lang.reflect.Field`



Can you give me an example of **reflective code**?

- We can **load class definitions** and **create instances**, without hard-coding class names into Java identifiers:

```
Class dynamicManagerClass = Class.forName("ch.heigvd.amt.reflection.services.SensorsManager");  
Object dynamicManager = dynamicManagerClass.newInstance();
```

- For a class, we can **get the list of methods** and **their signature**:

```
Method[] methods = dynamicManagerClass.getMethods();  
  
for (Method method : methods) {  
    LOG.log(Level.INFO, "Method name: " + method.getName());  
  
    Parameter[] parameters = method.getParameters();  
    for (Parameter p : parameters) {  
        LOG.log(Level.INFO, "p.getName()+ ":" + p.getType().getCanonicalName());  
    }  
}
```

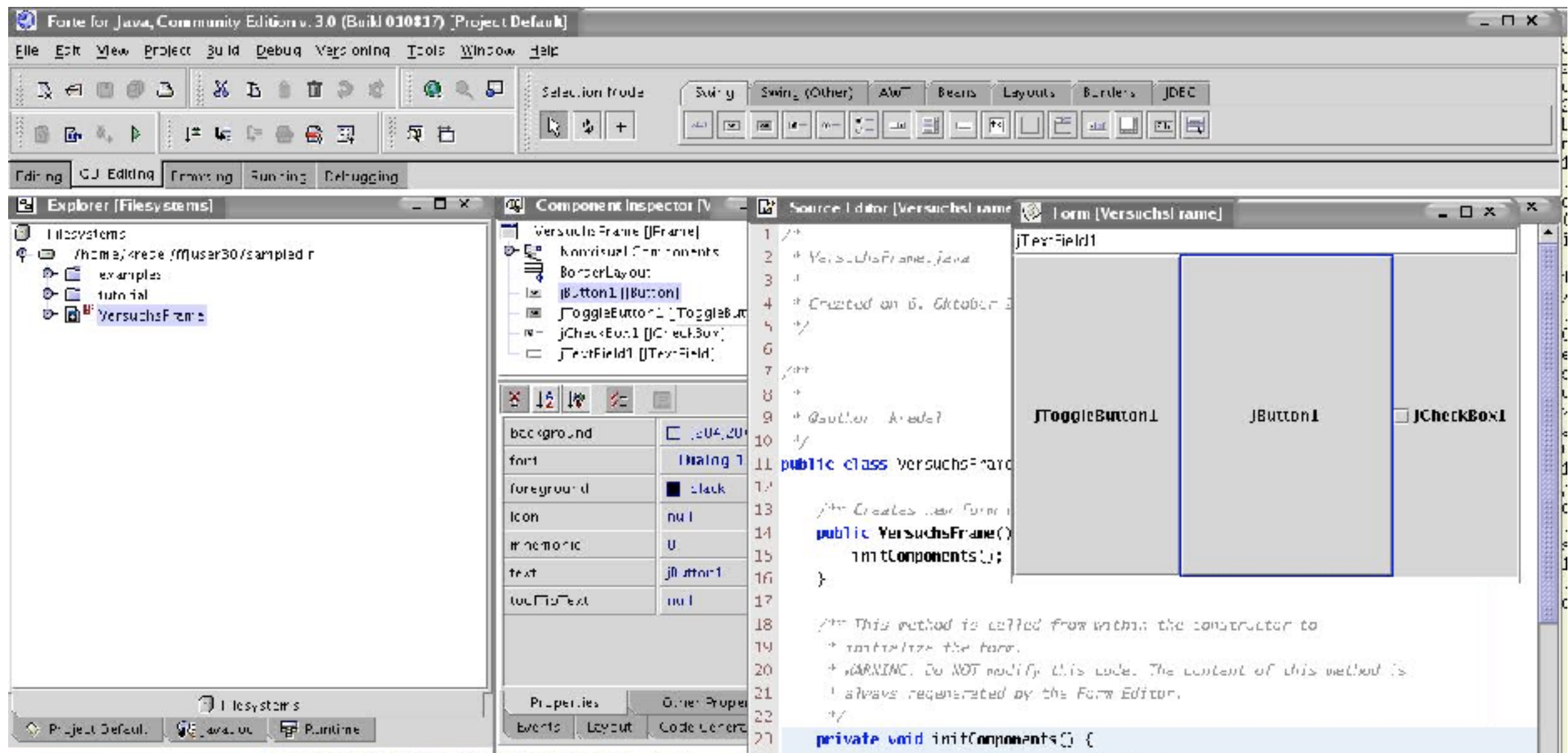
- We can dynamically **invoke a method** on an object:

```
Method method = dynamicManagerClass.getMethod("generateSensors", int.class, String.class);  
Object result = method.invoke(dynamicManager, 5, "hello");
```



What are **JavaBeans**?

- First of all, JavaBeans are **NOT** Enterprise Java Beans.
- The JavaBeans specification was proposed a very long time ago (1997) to enable the creation of **reusable components in Java**.
- One of the first use cases was to support the creation of **WYSIWYG development tools**. The programmer could drag and drop a GUI widget from a palette onto a window and edit its properties in a visual editor (think Visual Basic for Java).
- In this scenario, the GUI widgets would be packaged as JavaBeans by **third-party vendors**. The development tool would recognize them as such and would **dynamically extend the palette** of available components.



Forte for Java (aka Netbeans grand-father)



What are **JavaBeans**?

- Since then, JavaBeans have become **pervasive** in the Java Platform and are **used in many other scenarios**.
- This is particularly true in the Java EE Platform. Actually, **you have already implemented** JavaBeans without realizing it.
- While there are other aspects in the specification, the key elements are **coding conventions** that JavaBeans creators should respect:
 1. A JavaBean should have a **public no-args constructor**.
 2. A JavaBean should expose its properties via **getter** and **setter methods** with **well-defined names**.
 3. A JavaBean should be **serializable**.


```
public class Customer implements Serializable {
```

```
    public Customer() {}
```

```
    private String firstName;  
    private String lastName;  
    private boolean goodCustomer;
```

```
    public String getFirstName() {  
        return firstName;  
    }
```

```
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }
```

```
    public String getLastName() {  
        return lastName;  
    }
```

```
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }
```

```
    public boolean isGoodCustomer() {  
        return goodCustomer;  
    }
```

```
    public void setGoodCustomer(boolean goodCustomer) {  
        this.goodCustomer = goodCustomer;  
    }
```

```
}
```

There is a **specific convention** for writing
getter methods for
boolean properties.

A red arrow originates from the text 'boolean properties' and points to the 'isGoodCustomer()' method in the code block above.



What are **JavaBeans**?

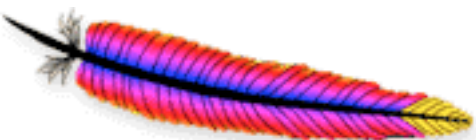
- These **coding and naming conventions** make it easier to **benefit from reflection** in **Java frameworks**:
 1. The framework can use the **public no-args constructor** to **create instances** with `Class.newInstance()`.
 2. The framework can **easily find out which methods it should call** (via reflection), based on a textual name. For instance, when a JSP page includes the string `${sensor.type}`, the runtime knows that it must invoke a method named "get" + "Type".
 3. The **state of a JavaBean** can travel over the wire (for instance when it moves from a remote EJB container to a web container).



What should I be know if I plan **to implement a framework** with JavaBeans?

- With the naming conventions defined in the JavaBeans specification, combined with Java reflection, **you can do pretty much everything yourself.**
- Have a look at the `java.beans` package and at the `Introspector` class. You will have easy access to properties, getters and setters.
- You should be aware of the **Apache Commons BeanUtils** library that will make your life easier.

*“The Java language provides **Reflection** and **Introspection** APIs (see the `java.lang.reflect` and `java.beans` packages in the JDK Javadocs). However, **these APIs can be quite complex** to understand and utilize. The BeanUtils component provides **easy-to-use wrappers** around these capabilities.”*





Back to the original question... How can I use reflection to **make my JDBC code generic**?

```
@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;
    }
}
```

Reflection sounds cool. Can't we use it to deal with JDBC in **more generic** ways?

JDBC gives me **metadata** about the DB schema.

Reflection gives me ways to dynamically find and **invoke methods** on Java objects.

Can we combine these features to make this code better?



Back to the original question... How can I use reflection to **make my JDBC code generic**?

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

Object-Relational Mapping in this example:

Table name = Class name + "s"

Column name = property name

```
String entityName = "Sensor";
String className = "ch.heigvd.amt.lab1.model." + entityName;
String tableName = entityName + "s";
PreparedStatement ps = con.prepareStatement("SELECT * FROM " + tableName);
ResultSet rs = ps.executeQuery();
Class entityClass = Class.forName(className);
PropertyDescriptor[] properties =
Introspector.getBeanInfo(entityClass).getPropertyDescriptors();

while (rs.next()) {
    Object entity;
    entity = entityClass.newInstance();
    for (PropertyDescriptor property : properties) {
        Method method = property.getWriteMethod();
        String columnName = property.getName();
        try {
            method.invoke(entity, rs.getObject(columnName));
        } catch (SQLException e) {
            LOG.warning("Could not retrieve value for property " + property.getName()
                + " in result set. " + e.getMessage());
        }
    }
    result.add(entity);
}
```

Class names, property names, table names and column names do not have to be hard-coded.

What we need is a **mapping**. We can either rely on **conventions** or define it **explicitly**.

These mechanisms are used by
people who build
Object Relational Mapping (ORM)
frameworks.

We will we look at one of them:
JPA.