Let’s master Hibernate!

Michał Żmuda
Piotr Turek

Agenda

1. project creation

   employing maven to include hibernate and spring libraries

2. database connection

   db setup and configuring connection

3. schema generation, model generation

   testing automate code generation and schema generation

4. accessing SessionFactory
5. data access and modification methods

   session methods and developing DAO

6. overview of object states

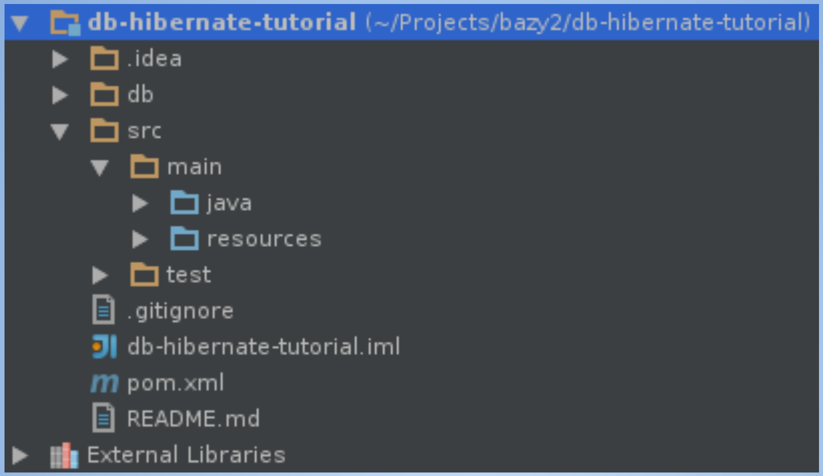
   impact of object states on flow design shown by example

7. fetching types case study
8. association types case study
9. extending model

   testing automate schema update

Creating project

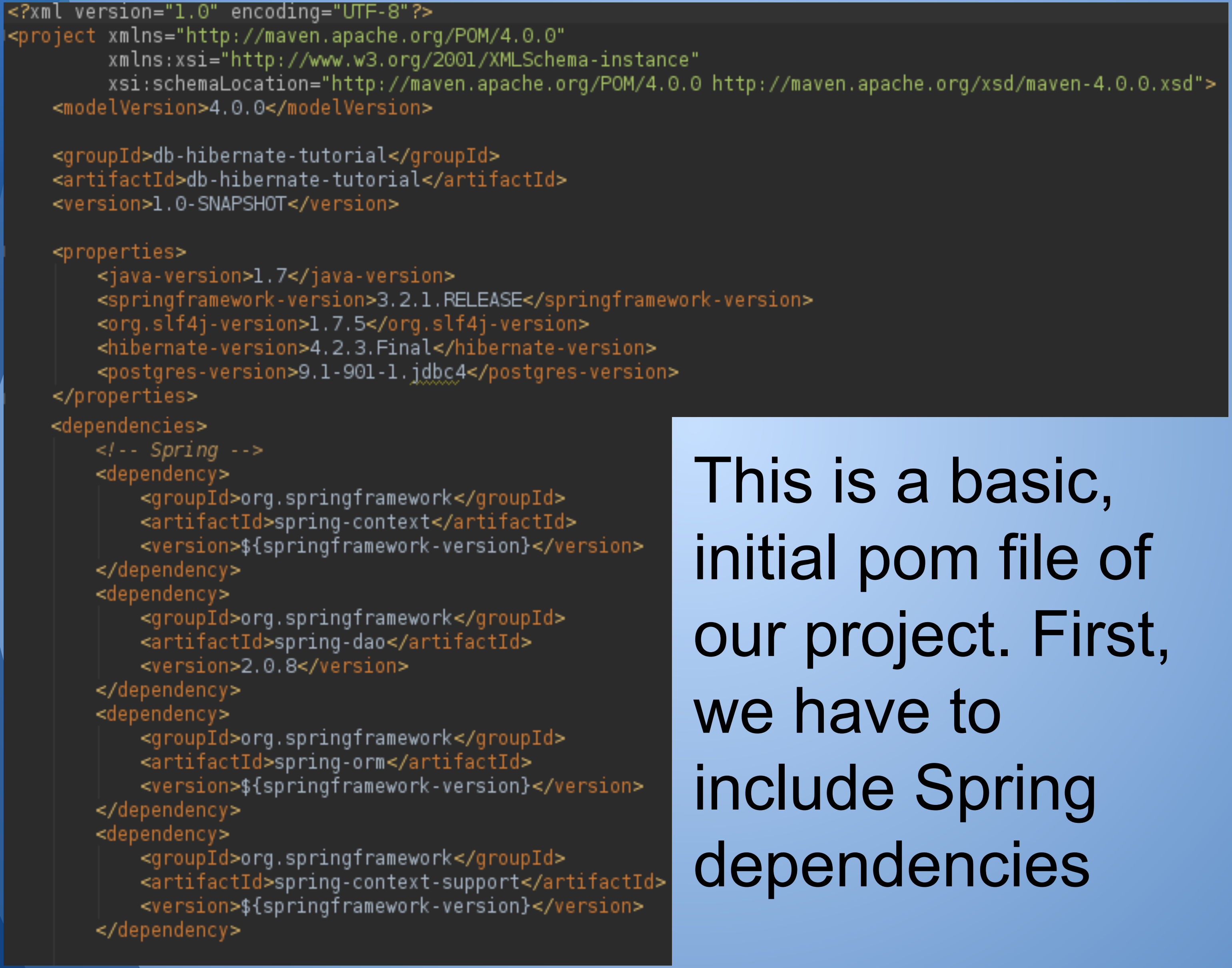
- Project structure



```
▼ db-hibernate-tutorial (~/Projects/bazy2/db-hibernate-tutorial)
    ▶ .idea
    ▶ db
    ▼ src
        ▼ main
            ▶ java
            ▶ resources
        ▶ test
      .gitignore
      db-hibernate-tutorial.iml
      pom.xml
      README.md
▶ External Libraries
```

- standard Maven-based structure
- separate “db” folder for database artifacts
- build definition in pom.xml

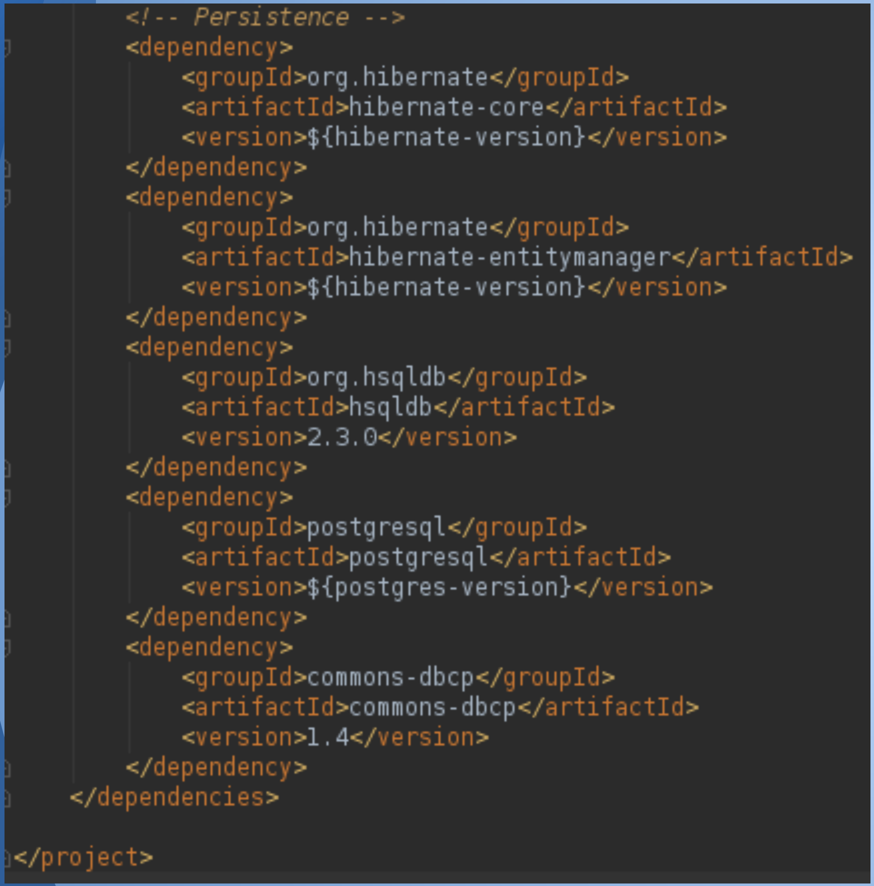
Creating project - pom file



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>db-hibernate-tutorial</groupId>
    <artifactId>db-hibernate-tutorial</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java-version>1.7</java-version>
        <springframework-version>3.2.1.RELEASE</springframework-version>
        <org.slf4j-version>1.7.5</org.slf4j-version>
        <hibernate-version>4.2.3.Final</hibernate-version>
        <postgres-version>9.1-901-1.jdbc4</postgres-version>
    </properties>
    <dependencies>
        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${springframework-version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-dao</artifactId>
            <version>2.0.8</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${springframework-version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${springframework-version}</version>
        </dependency>
```

This is a basic, initial pom file of our project. First, we have to include Spring dependencies

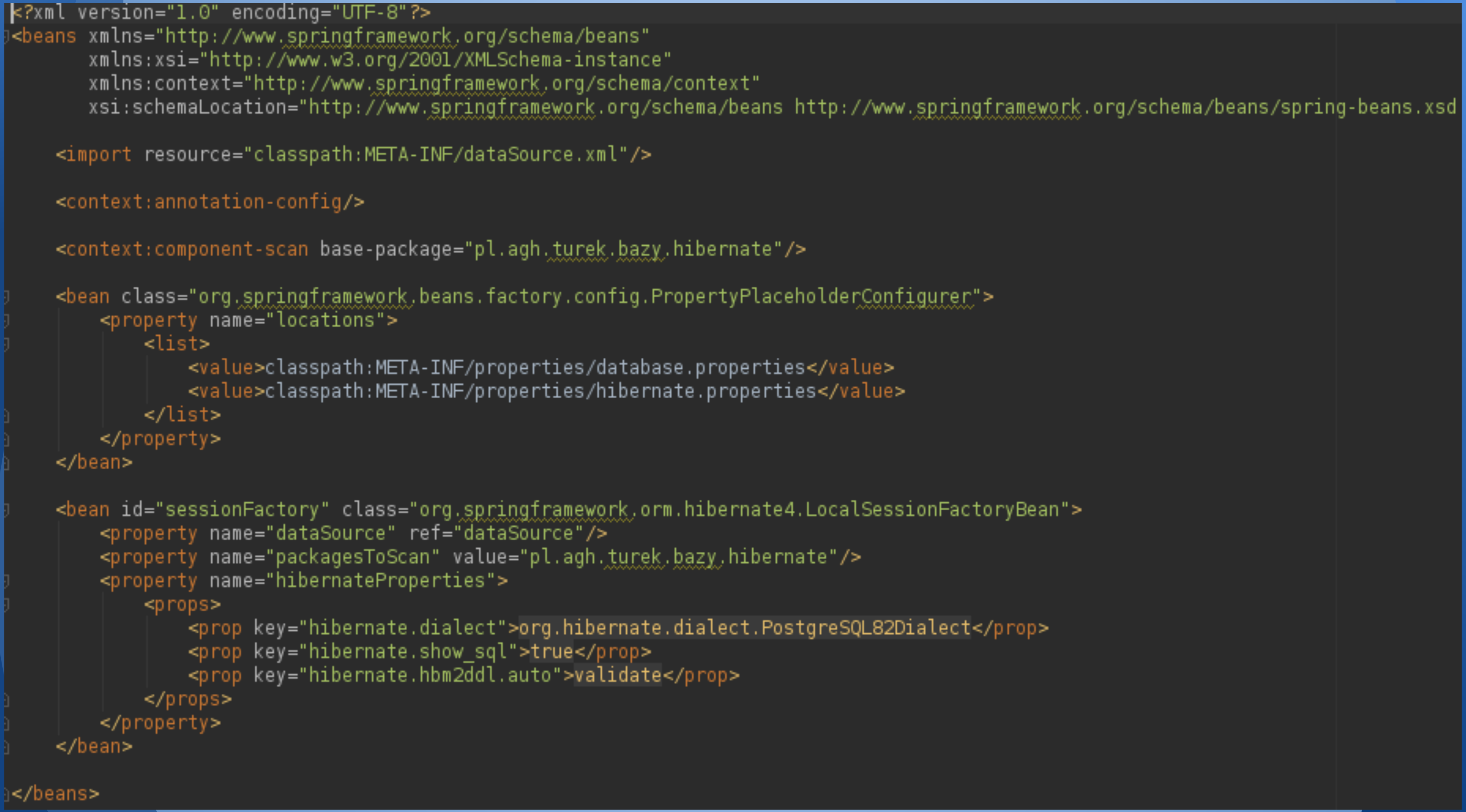
Creating project - pom file (cont.)



```
        <!-- Persistence -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>${hibernate-version}</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>${hibernate-version}</version>
        </dependency>
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.3.0</version>
        </dependency>
        <dependency>
            <groupId>postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>${postgres-version}</version>
        </dependency>
        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
        </dependency>
    </dependencies>

</project>
```

- The following part of pom file defines all the needed persistence dependencies including Hibernate and db connection stuff

Creating project - context definition

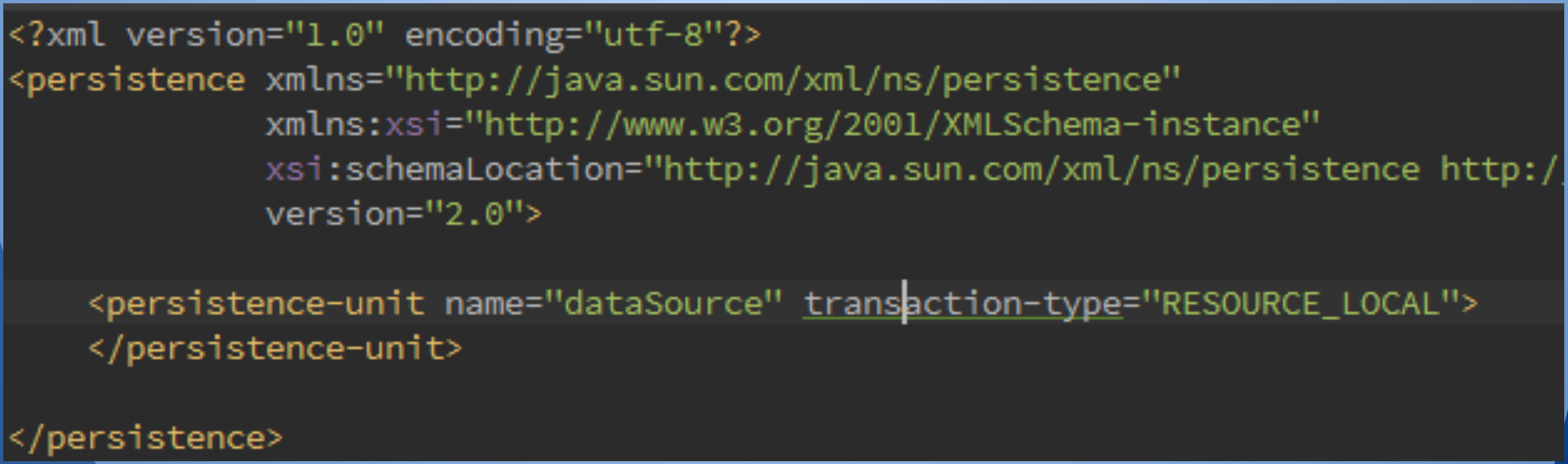


```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

    <import resource="classpath:META-INF/dataSource.xml"/>

    <context:annotation-config/>

    <context:component-scan base-package="pl.agh.turek.bazy.hibernate"/>

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:META-INF/properties/database.properties</value>
                <value>classpath:META-INF/properties/hibernate.properties</value>
            </list>
        </property>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="packagesToScan" value="pl.agh.turek.bazy.hibernate"/>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.PostgreSQL82Dialect</prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.hbm2ddl.auto">validate</prop>
            </props>
        </property>
    </bean>

</beans>
```

We only need to define sessionFactory in a declarative way

Hibernate configuration

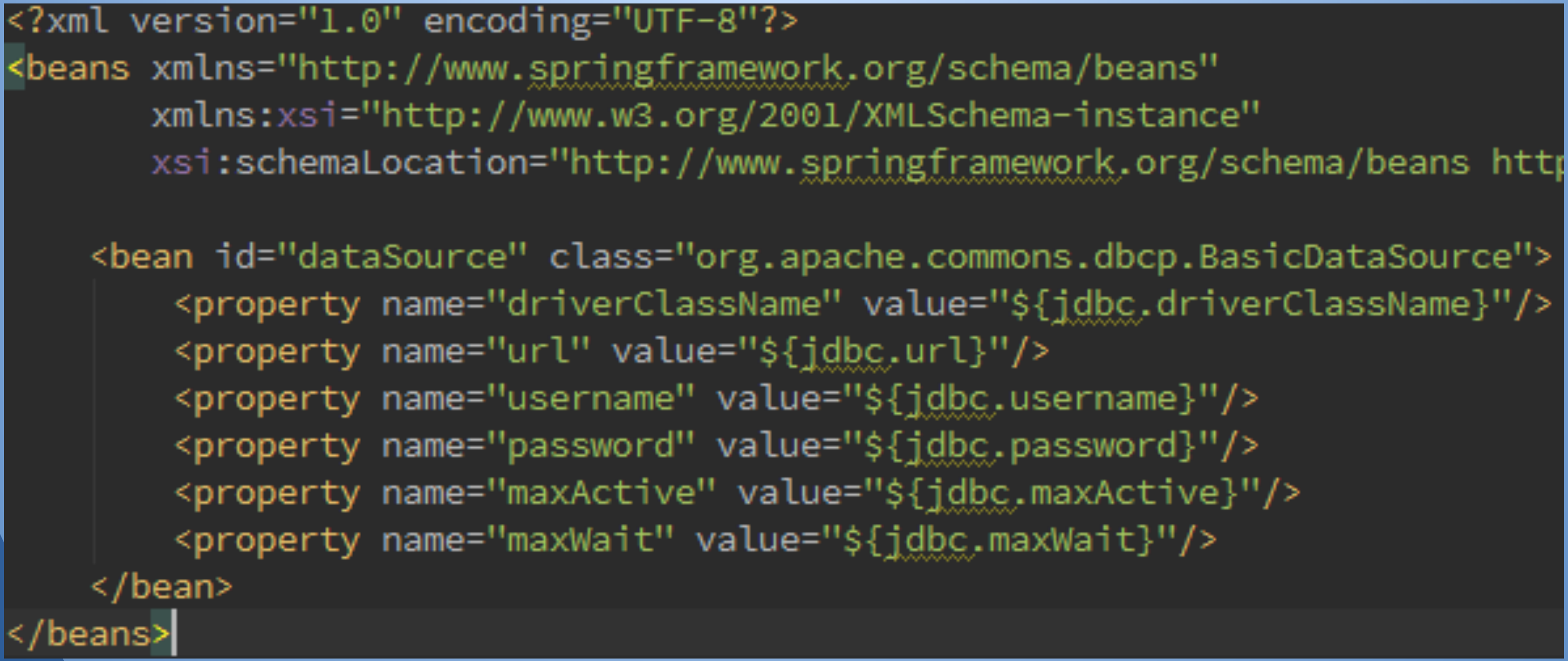
- provide persistence.xml file
  - this is standard JPA (and also Hiberante) configuration element
  - this contains required rules to setum session factory and could contain mapping definition
  - in our configuration we only define that we won’t employ any additional transaction management (for exapmle transactions with multilple distributed databases)



```
<?xml version="1.0" encoding="utf-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http:/
             version="2.0">

    <persistence-unit name="dataSource" transaction-type="RESOURCE_LOCAL">
    </persistence-unit>

</persistence>
```

Hibernate configuration

- provide dataSource.xml file
  - this file defines data sources
  - you may consider data source simply as database
  - in basic project setup we have only one, local data source
  - good practice is to have parameters in properties file (whitch would be described soon)

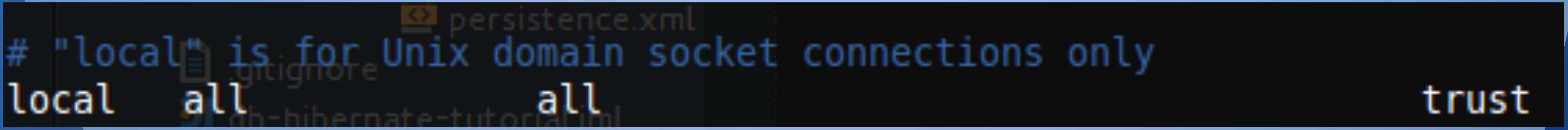


```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
        <property name="maxActive" value="${jdbc.maxActive}"/>
        <property name="maxWait" value="${jdbc.maxWait}"/>
    </bean>
</beans>
```

Database setup

- provide user accounted as system user
- create database and grant permissions
- be sure to have adequate postgresql configuration, for instace you might want to edit pg_hba.conf

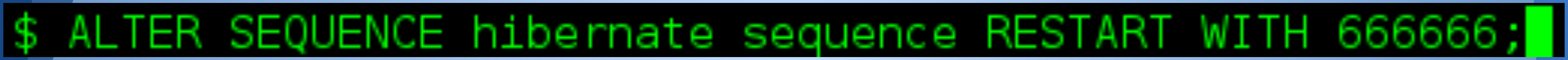
following line may prove useful, but refrain from modifying others (unauthorized acces may occur)



```
# "local" is for Unix domain socket connections only
local   all             all                                     trust
```

Database setup

Prior to using Hibernate on existing database, you need to set properly the hibernate sequence



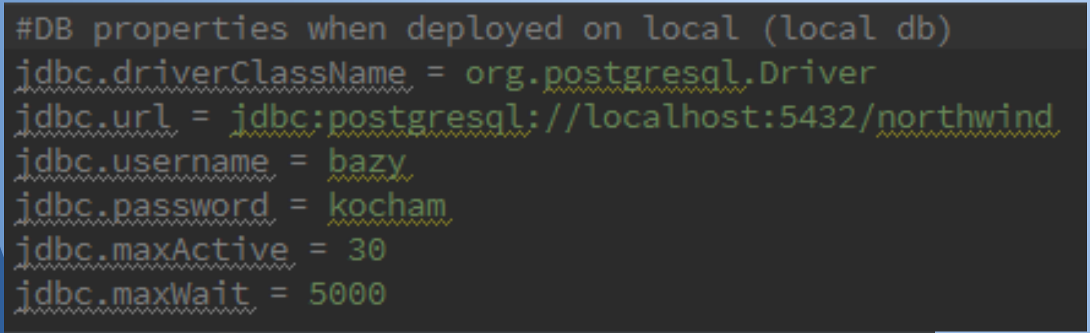
```
$ ALTER SEQUENCE hibernate sequence RESTART WITH 666666;
```

Otherwise there will be ID conflicts and Hibernate will malfunction

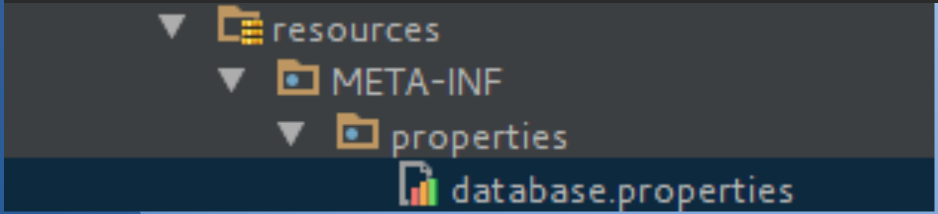
Connection setup

- be sure to provide connection parameters
- good practise is to create for that purpose separate properties file

examplary stucure and values



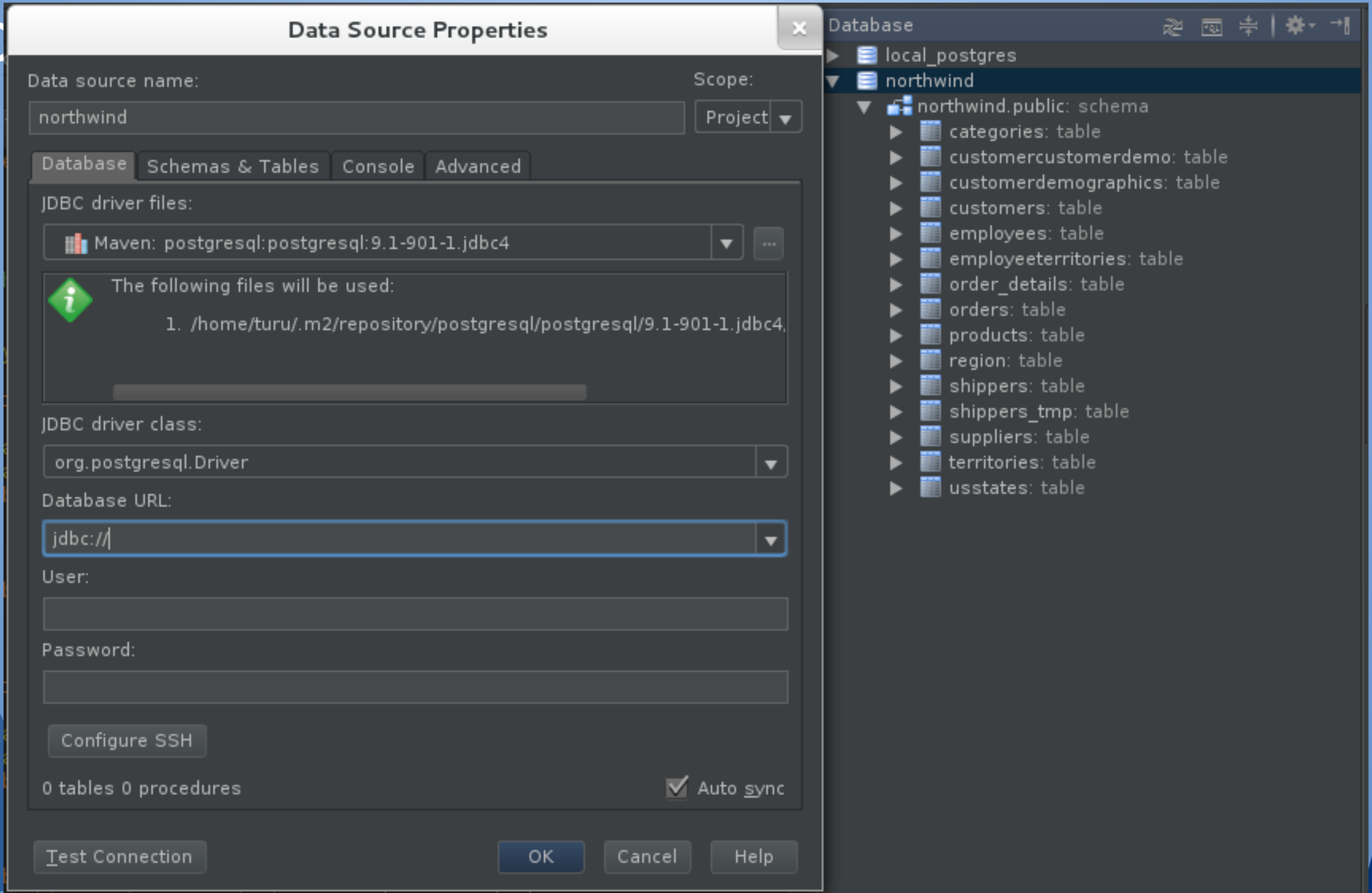
```
#DB properties when deployed on local (local db)
jdbc.driverClassName = org.postgresql.Driver
jdbc.url = jdbc:postgresql://localhost:5432/northwind
jdbc.username = bazy
jdbc.password = kocham
jdbc.maxActive = 30
jdbc.maxWait = 5000
```



```
▼ resources
  ▼ META-INF
    ▼ properties
        database.properties
```

Model & mapping generation

Modern IDEs like IntelliJ provide facilities for generating model & mappings from db sc



Data source name: northwind

Scope: Project

JDBC driver files: Maven: postgresql:postgresql:9.1-901-1.jdbc4

The following files will be used:

1. /home/turu/.m2/repository/postgresql/postgresql/9.1-901-1.jdbc4,

JDBC driver class: org.postgresql.Driver

Database URL: jdbc://

User:

Password:

0 tables 0 procedures

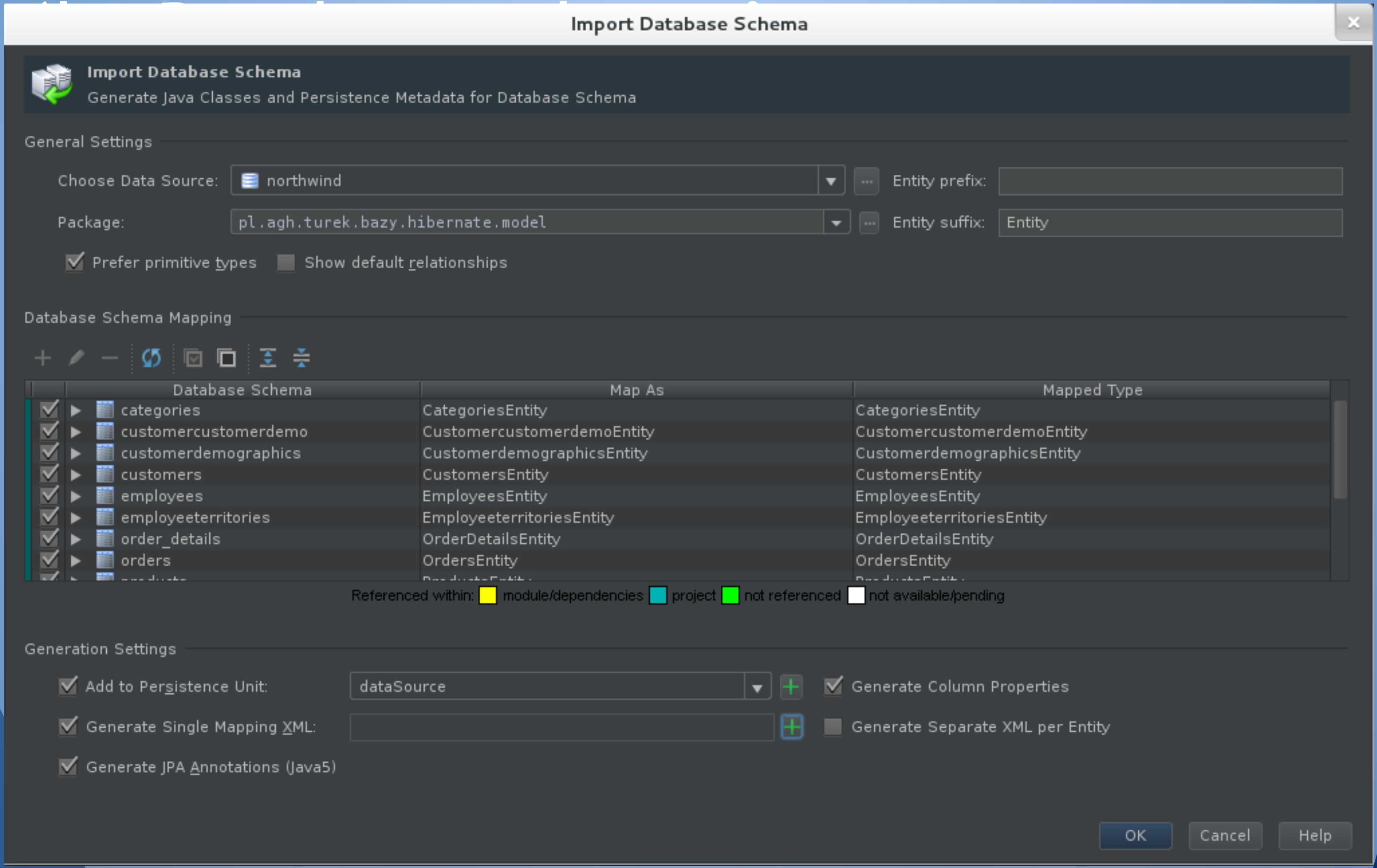
☑ Auto sync

Database

- local_postgres
- northwind
  - northwind.public: schema
    - categories: table
    - customercustomerdemo: table
    - customerdemographics: table
    - customers: table
    - employees: table
    - employeeterritories: table
    - order_details: table
    - orders: table
    - products: table
    - region: table
    - shippers: table
    - shippers_tmp: table
    - suppliers: table
    - territories: table
    - usstates: table

Model & mapping generation

- ctrl + shift + a + ‘generate persistence mappin’



Import Database Schema

Import Database Schema

Generate Java Classes and Persistence Metadata for Database Schema

General Settings

Choose Data Source: northwind

Entity prefix:

Package: pl.agh.turek.bazy.hibernate.model

Entity suffix: Entity

☑ Prefer primitive types □ Show default relationships

Database Schema Mapping

	Database Schema	Map As	Mapped Type
☑	categories	CategoriesEntity	CategoriesEntity
☑	customercustomerdemo	CustomercustomerdemoEntity	CustomercustomerdemoEntity
☑	customerdemographics	CustomerdemographicsEntity	CustomerdemographicsEntity
☑	customers	CustomersEntity	CustomersEntity
☑	employees	EmployeesEntity	EmployeesEntity
☑	employeeterritories	EmployeeterritoriesEntity	EmployeeterritoriesEntity
☑	order_details	OrderDetailsEntity	OrderDetailsEntity
☑	orders	OrdersEntity	OrdersEntity
☑	products	ProductsEntity	ProductsEntity

Referenced within: 🟨 module/dependencies 🟦 project 🟩 not referenced ⬜ not available/pending

Generation Settings

☑ Add to Persistence Unit: dataSource

☑ Generate Column Properties

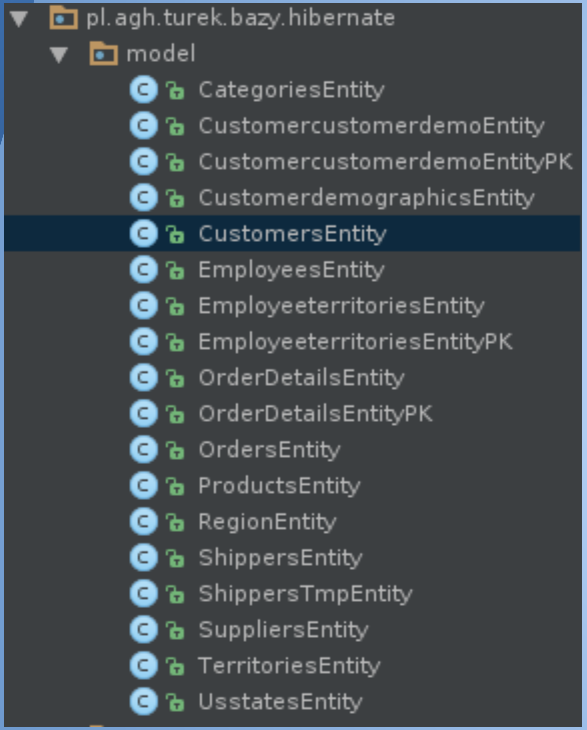
☑ Generate Single Mapping XML:

□ Generate Separate XML per Entity

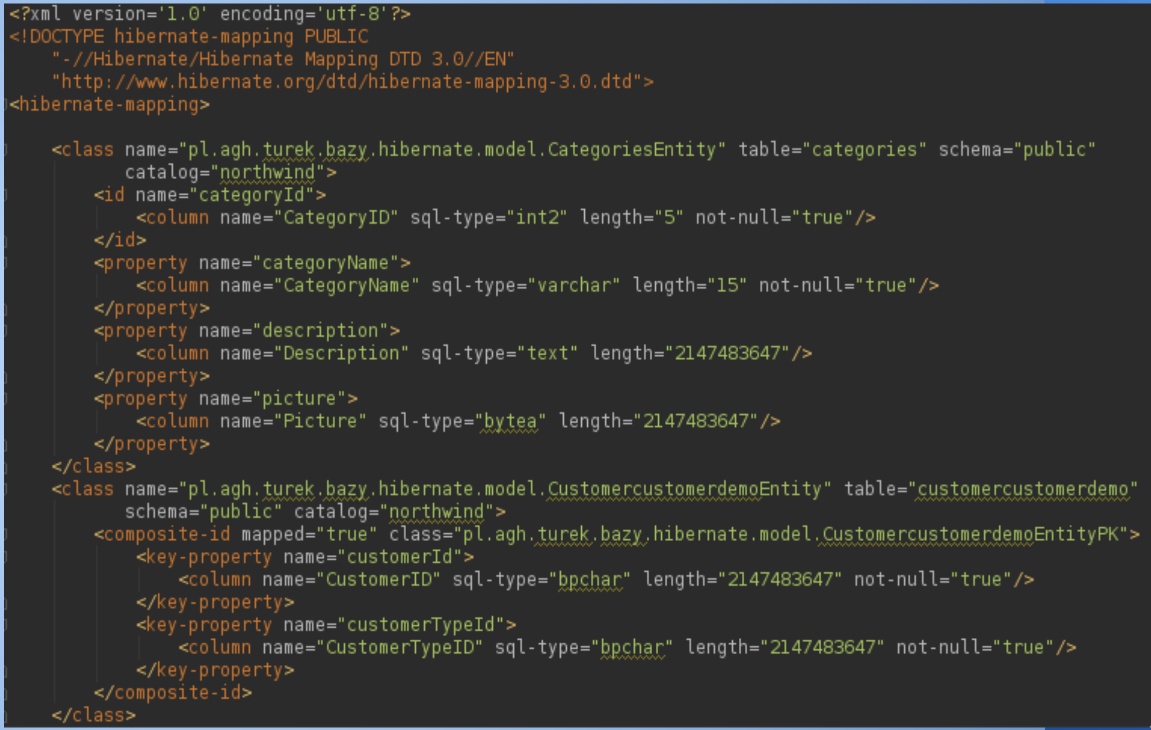
☑ Generate JPA Annotations (Java5)

Model & mapping generation

As a result:



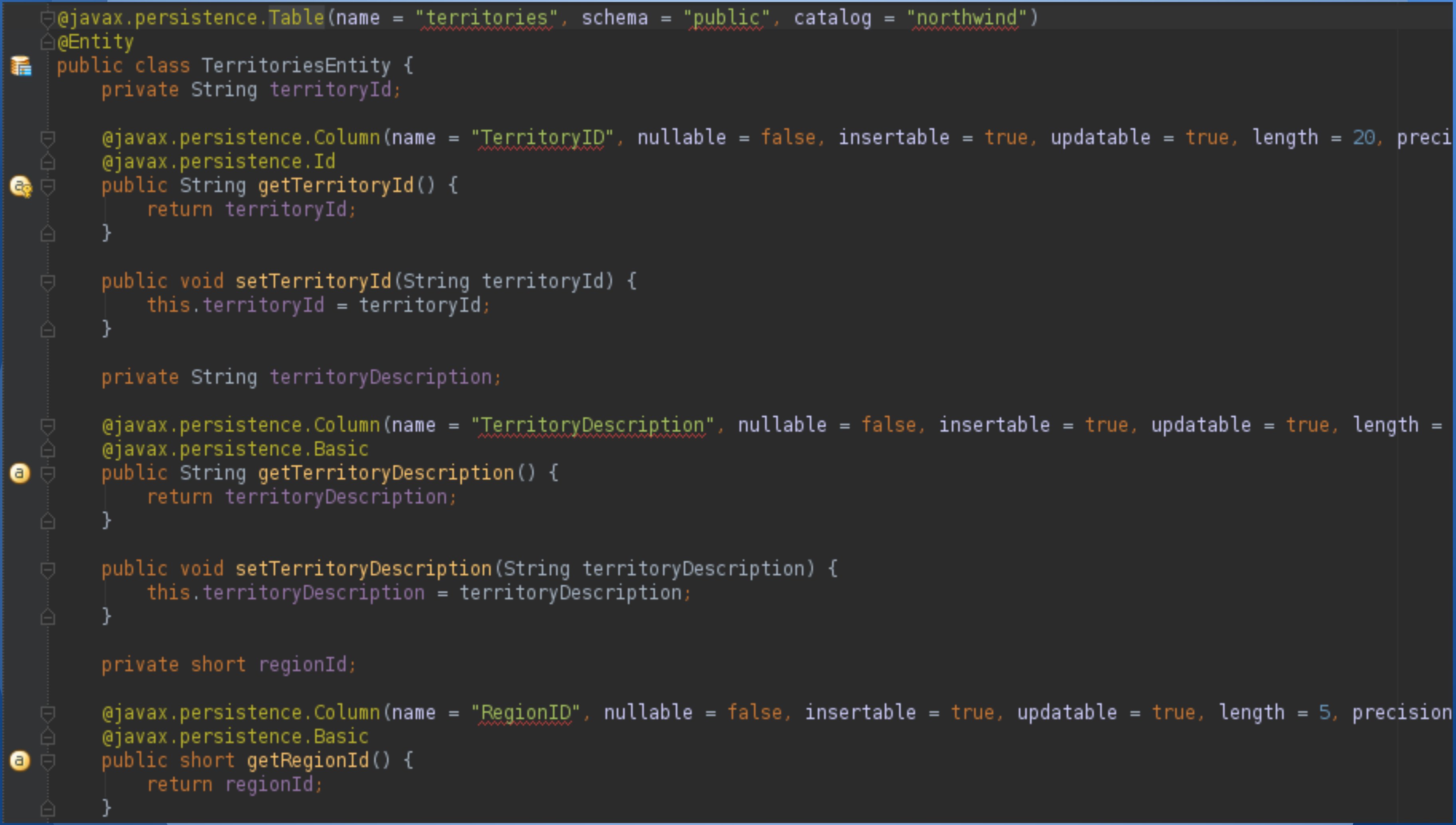
```
▼ pl.agh.turek.bazy.hibernate
    ▼ model
          CategoriesEntity
          CustomercustomerdemoEntity
          CustomercustomerdemoEntityPK
          CustomerdemographicsEntity
          CustomersEntity
          EmployeesEntity
          EmployeeterritoriesEntity
          EmployeeterritoriesEntityPK
          OrderDetailsEntity
          OrderDetailsEntityPK
          OrdersEntity
          ProductsEntity
          RegionEntity
          ShippersEntity
          ShippersTmpEntity
          SuppliersEntity
          TerritoriesEntity
          UsstatesEntity
```



```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>

    <class name="pl.agh.turek.bazy.hibernate.model.CategoriesEntity" table="categories" schema="public"
           catalog="northwind">
        <id name="categoryId">
            <column name="CategoryID" sql-type="int2" length="5" not-null="true"/>
        </id>
        <property name="categoryName">
            <column name="CategoryName" sql-type="varchar" length="15" not-null="true"/>
        </property>
        <property name="description">
            <column name="Description" sql-type="text" length="2147483647"/>
        </property>
        <property name="picture">
            <column name="Picture" sql-type="bytea" length="2147483647"/>
        </property>
    </class>
    <class name="pl.agh.turek.bazy.hibernate.model.CustomercustomerdemoEntity" table="customercustomerdemo"
           schema="public" catalog="northwind">
        <composite-id mapped="true" class="pl.agh.turek.bazy.hibernate.model.CustomercustomerdemoEntityPK">
            <key-property name="customerId">
                <column name="CustomerID" sql-type="bpchar" length="2147483647" not-null="true"/>
            </key-property>
            <key-property name="customerTypeId">
                <column name="CustomerTypeID" sql-type="bpchar" length="2147483647" not-null="true"/>
            </key-property>
        </composite-id>
    </class>
```

Model & mapping generation (cont.)

As a result:

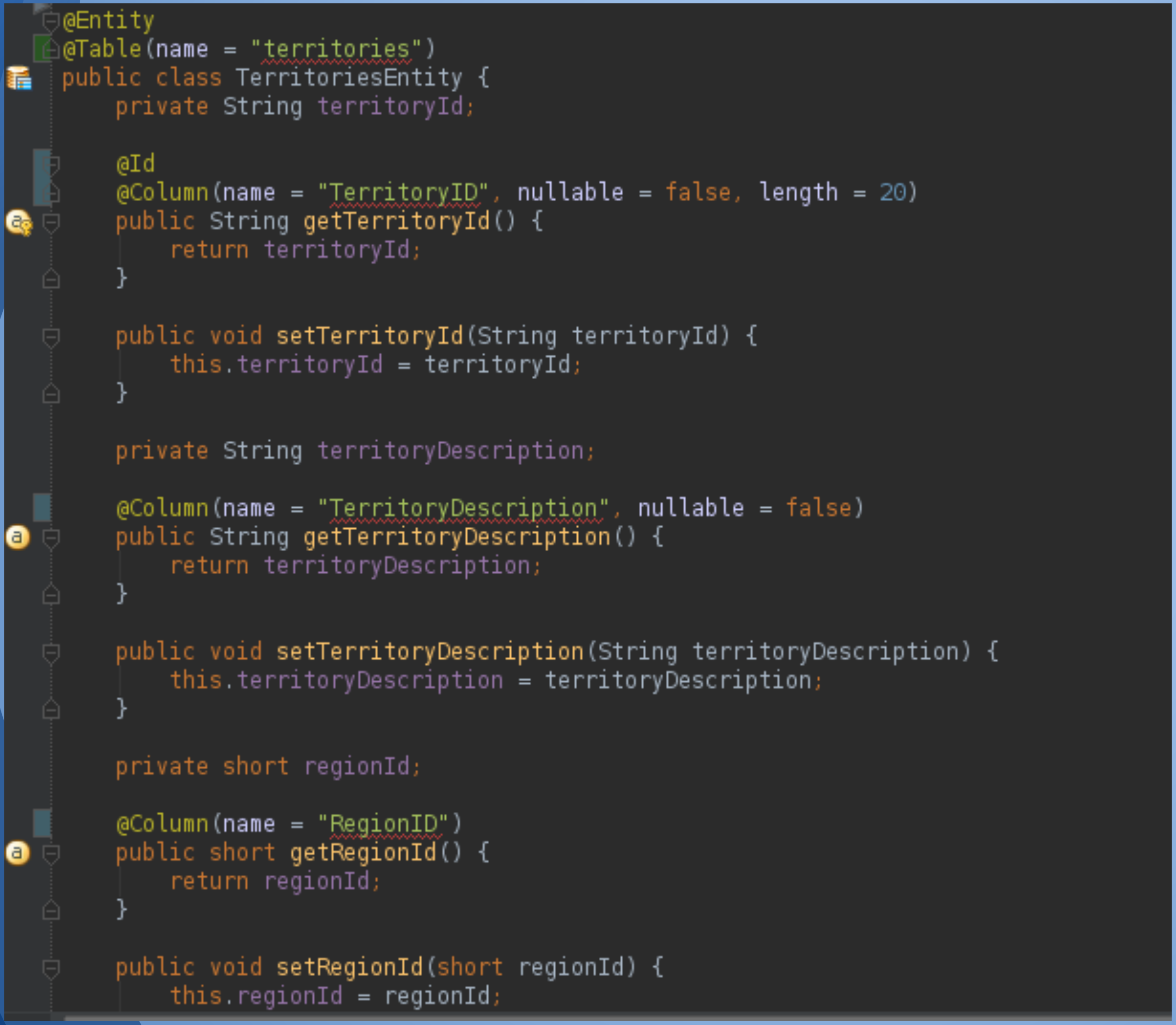


```
@javax.persistence.Table(name = "territories", schema = "public", catalog = "northwind")
@Entity
public class TerritoriesEntity {
    private String territoryId;

    @javax.persistence.Column(name = "TerritoryID", nullable = false, insertable = true, updatable = true, length = 20, preci
    @javax.persistence.Id
    public String getTerritoryId() {
        return territoryId;
    }

    public void setTerritoryId(String territoryId) {
        this.territoryId = territoryId;
    }

    private String territoryDescription;

    @javax.persistence.Column(name = "TerritoryDescription", nullable = false, insertable = true, updatable = true, length =
    @javax.persistence.Basic
    public String getTerritoryDescription() {
        return territoryDescription;
    }

    public void setTerritoryDescription(String territoryDescription) {
        this.territoryDescription = territoryDescription;
    }

    private short regionId;

    @javax.persistence.Column(name = "RegionID", nullable = false, insertable = true, updatable = true, length = 5, precision
    @javax.persistence.Basic
    public short getRegionId() {
        return regionId;
    }
```

Model & mapping generation (cont.)

- There are however a few problems
  - generated mappings are unnecessarily verbose
  - names can be far from “clean code”
  - outright mistakes happen (!)
    - data types
    - names
- … which you need to fix manually

Model & mapping generation (cont.)



```
@Entity
@Table(name = "territories")
public class TerritoriesEntity {
    private String territoryId;

    @Id
    @Column(name = "TerritoryID", nullable = false, length = 20)
    public String getTerritoryId() {
        return territoryId;
    }

    public void setTerritoryId(String territoryId) {
        this.territoryId = territoryId;
    }

    private String territoryDescription;

    @Column(name = "TerritoryDescription", nullable = false)
    public String getTerritoryDescription() {
        return territoryDescription;
    }

    public void setTerritoryDescription(String territoryDescription) {
        this.territoryDescription = territoryDescription;
    }

    private short regionId;

    @Column(name = "RegionID")
    public short getRegionId() {
        return regionId;
    }

    public void setRegionId(short regionId) {
        this.regionId = regionId;
```

Example model class and its mapping after fix

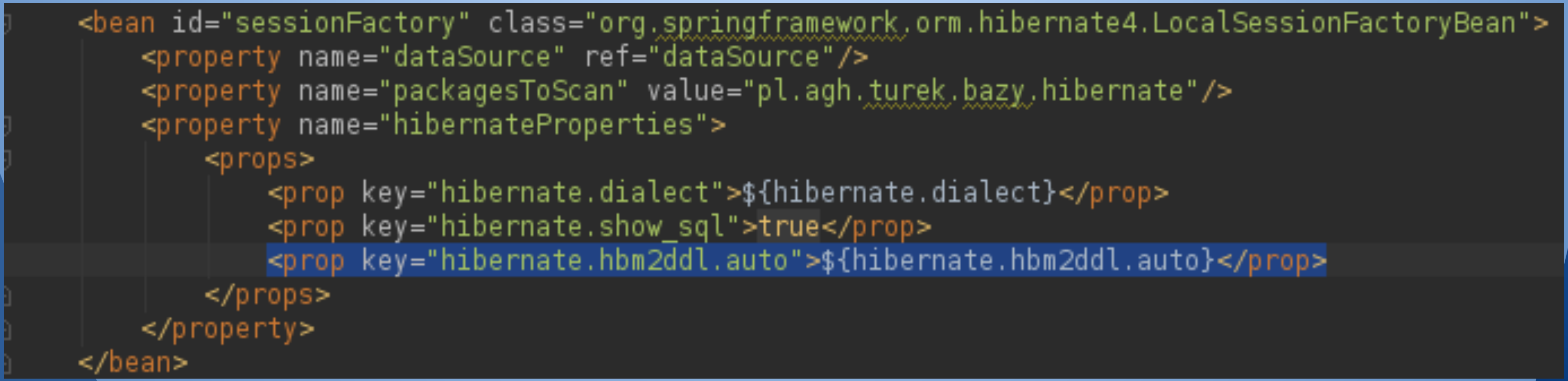
Model & mapping generation (cont.)

Things to consider:

- Generating model from schema is generally considered a bad practice
  - schema should be developed together with model
- However, if we need to port a project to ORM, generated model can be a good basis for development
- Annotation-based mappings are prefered over XML-based. The latter isn’t commonly used nowadays

Schema generation / update

- Hibernate can generate or update your db schema, based on changes to the model.
- This process happens upon creation of SessionFactory
- *hibernate.hbm2ddl.auto* property is used to select desired behaviour



```
<bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan" value="pl.agh.turek.bazy.hibernate"/>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
        </props>
    </property>
</bean>
```

Schema generation / update

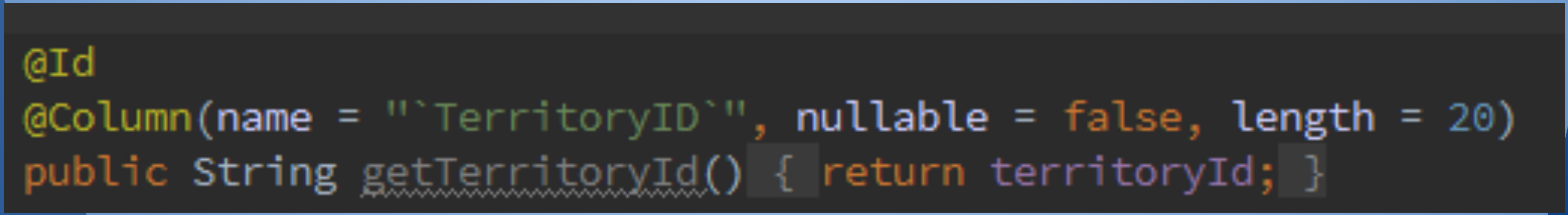
- *hibernate.hbm2ddl.auto* Automatically validates or exports schema to the database when the SessionFactory is created.
- Possible values:
  - ***validate***: validates the schema
  - ***update***: updates the schema.
  - ***create***: creates the schema, destroying previous data.
  - ***create-drop***: drop the schema at the end of the session.

Schema generation / update

- When you want to create schema, simply set the aforementioned property to *create* and run the application
- By default this property should **ALWAYS** be set to ***validate***
- Schema update is considered an experimental feature and should **NEVER** be used on production databases (!)
    - for that use db migration tools such us Liquibase
        - additional advantage is managing and tracking db schema changes across time

Schema generation / update - postgre specific issues

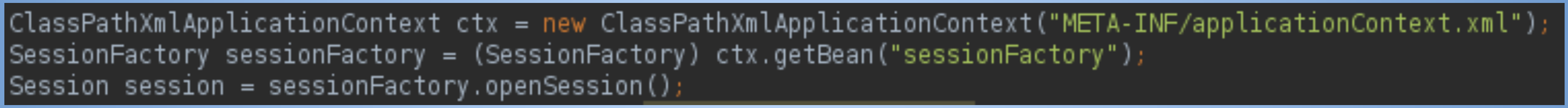
- you may observe that generated schema contains uppercase characters in column or table names
- unless you quote them manually any query will fail due to lowercase naming convention in postgre



```
@Id
@Column(name = "`TerritoryID`", nullable = false, length = 20)
public String getTerritoryId() { return territoryId; }
```

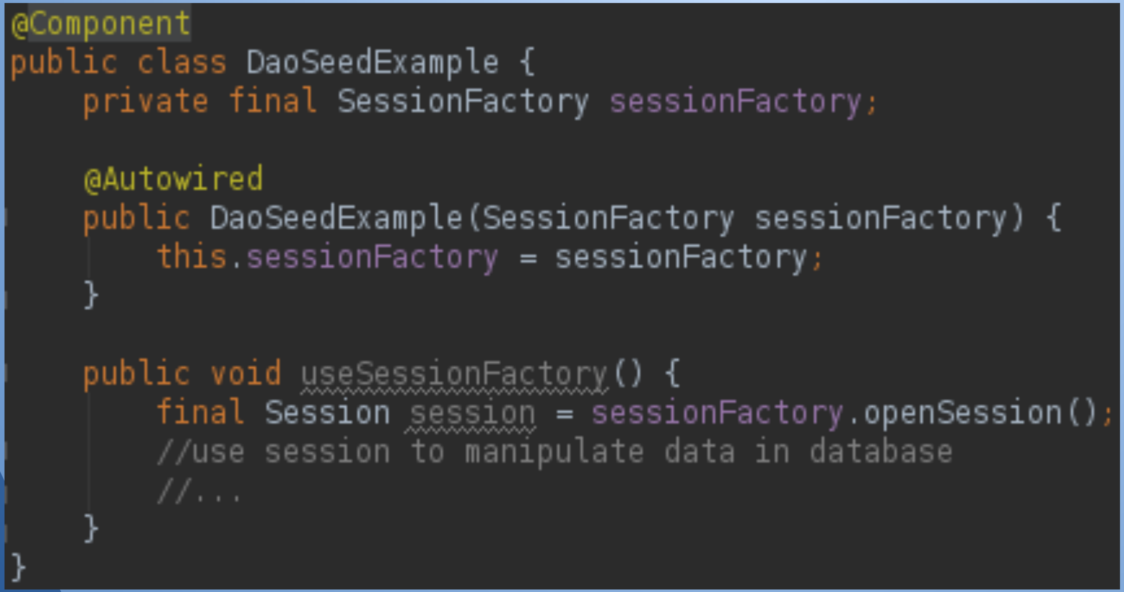
Accessing Session Factory

- Programatically



```java
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("META-INF/applicationContext.xml");
SessionFactory sessionFactory = (SessionFactory) ctx.getBean("sessionFactory");
Session session = sessionFactory.openSession();
```

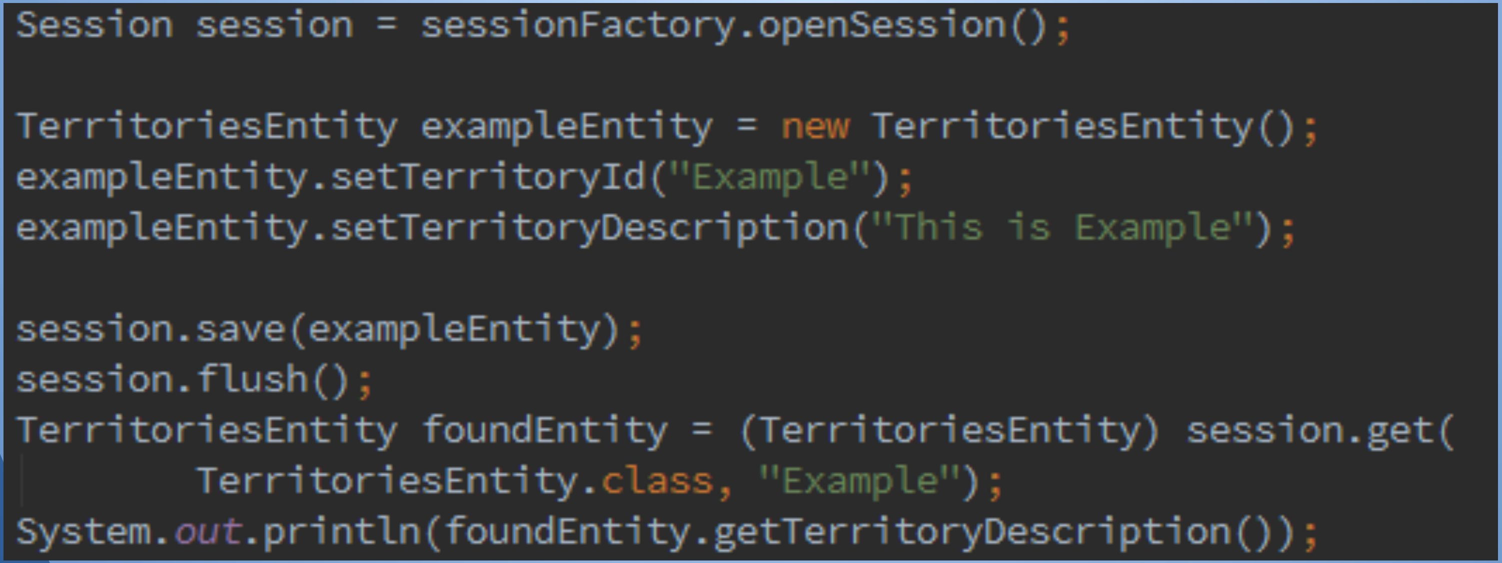
- via Dependency Injection



```java
@Component
public class DaoSeedExample {
    private final SessionFactory sessionFactory;

    @Autowired
    public DaoSeedExample(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public void useSessionFactory() {
        final Session session = sessionFactory.openSession();
        //use session to manipulate data in database
        //...
    }
}
```

Accessing and modifying database - session methods

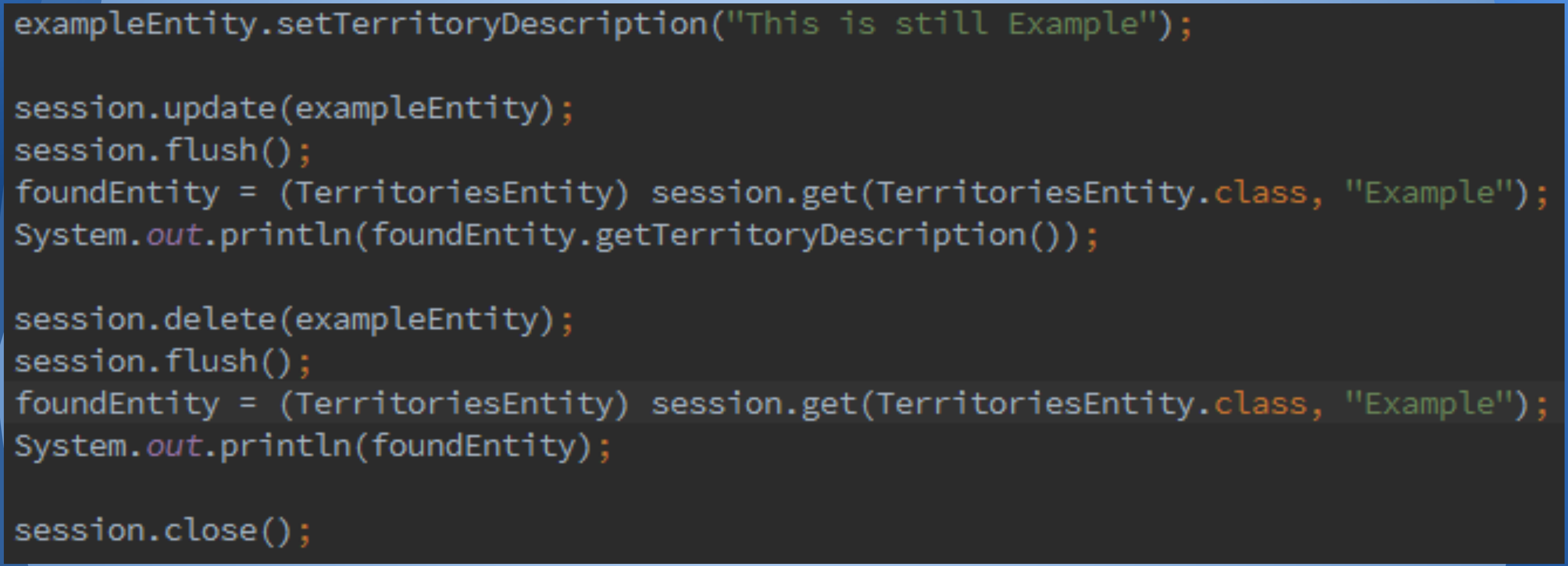
- after you access sesion factory you may manually open it and use it’s methods to modify database
- to learn create or open SessionMethodsCaseRunner and try other session methods



```
Session session = sessionFactory.openSession();

TerritoriesEntity exampleEntity = new TerritoriesEntity();
exampleEntity.setTerritoryId("Example");
exampleEntity.setTerritoryDescription("This is Example");

session.save(exampleEntity);
session.flush();
TerritoriesEntity foundEntity = (TerritoriesEntity) session.get(
        TerritoriesEntity.class, "Example");
System.out.println(foundEntity.getTerritoryDescription());
```

cdn...

Accessing and modifying database - session methods



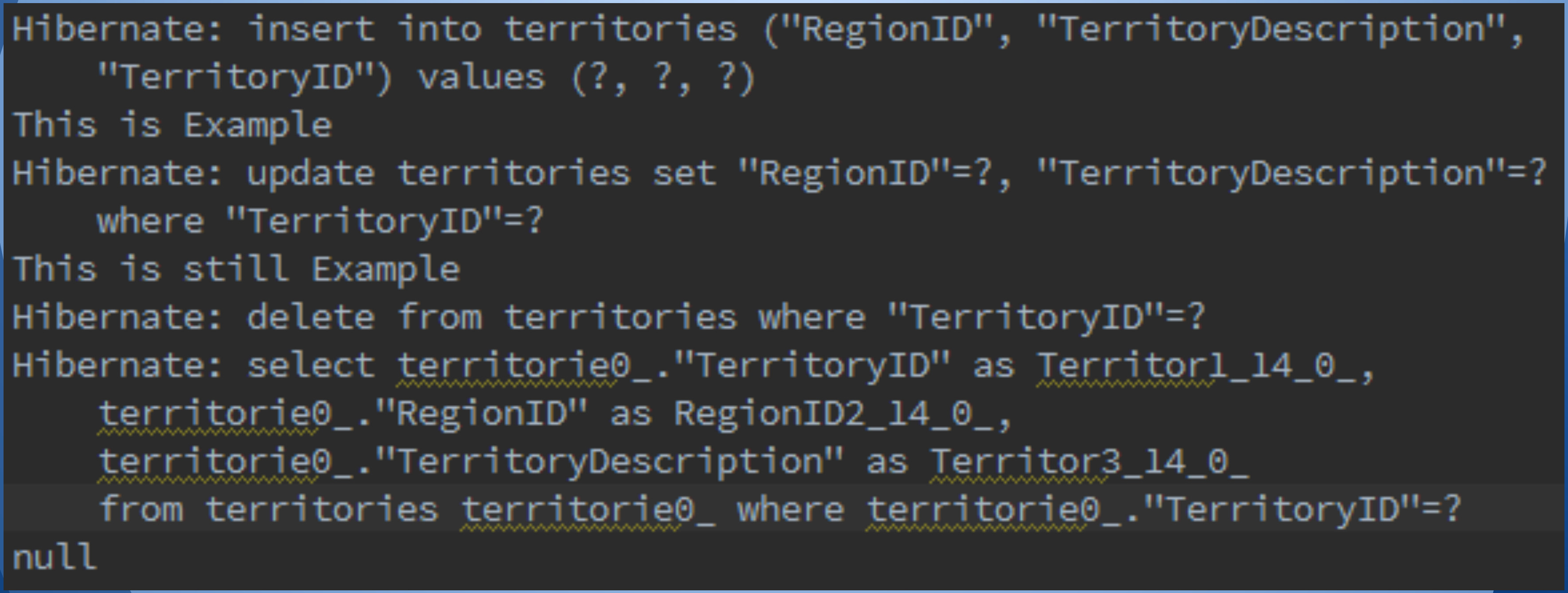
```
exampleEntity.setTerritoryDescription("This is still Example");

session.update(exampleEntity);
session.flush();
foundEntity = (TerritoriesEntity) session.get(TerritoriesEntity.class, "Example");
System.out.println(foundEntity.getTerritoryDescription());

session.delete(exampleEntity);
session.flush();
foundEntity = (TerritoriesEntity) session.get(TerritoriesEntity.class, "Example");
System.out.println(foundEntity);

session.close();
```

questions:

- why you need to flush session before accessing saved data? what happens when you don’t?

Accessing and modifying database - hibernate sql logs and output

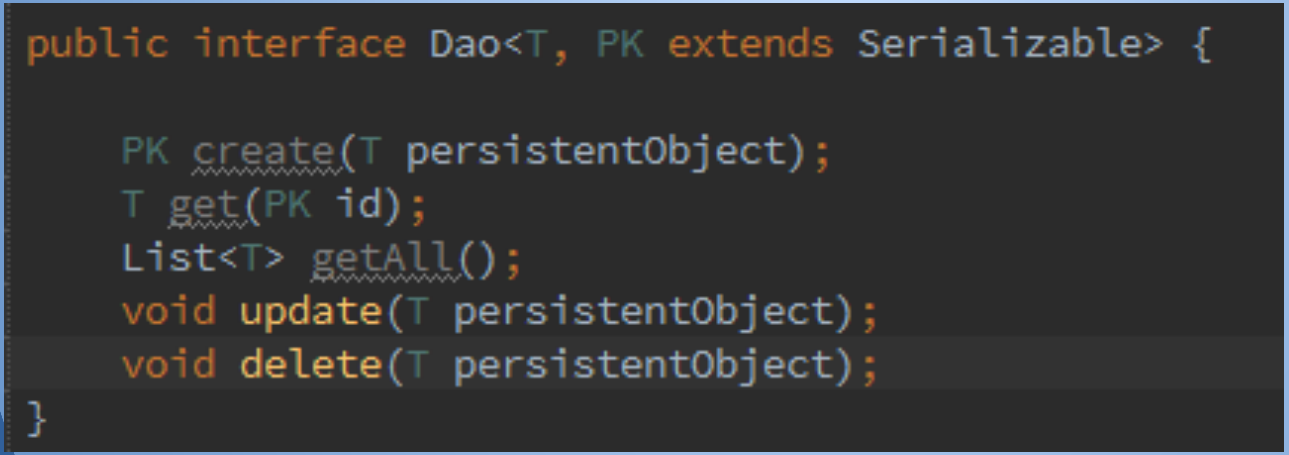
- take a look at queries generated by hibernate, are they as simple as they could?
- do you know what prepared statement is?



```
Hibernate: insert into territories ("RegionID", "TerritoryDescription",
    "TerritoryID") values (?, ?, ?)
This is Example
Hibernate: update territories set "RegionID"=?, "TerritoryDescription"=?
    where "TerritoryID"=?
This is still Example
Hibernate: delete from territories where "TerritoryID"=?
Hibernate: select territorie0_."TerritoryID" as Territor1_14_0_,
    territorie0_."RegionID" as RegionID2_14_0_,
    territorie0_."TerritoryDescription" as Territor3_14_0_
    from territories territorie0_ where territorie0_."TerritoryID"=?
null
```

Accessing and modifying database - Data Access Object

- in order to hide session management and its methods we often create DAO
- very simple generic DAO coul have interace like:

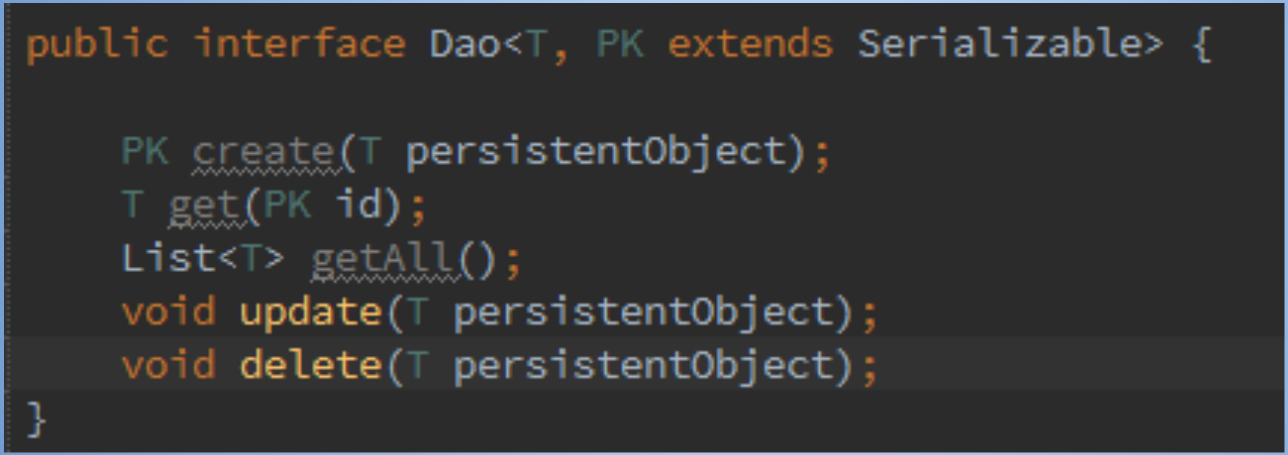


```
public interface Dao<T, PK extends Serializable> {

    PK create(T persistentObject);
    T get(PK id);
    List<T> getAll();
    void update(T persistentObject);
    void delete(T persistentObject);
}
```

- try your luck with implementing this
- remember you don’t have to access session factory in dao methods - treat sessionfactory as given for instance by contructor

Accessing and modifying database - Data Access Object

- in order to hide session management and its methods we often create DAO
- very simple generic DAO coul have interace like:

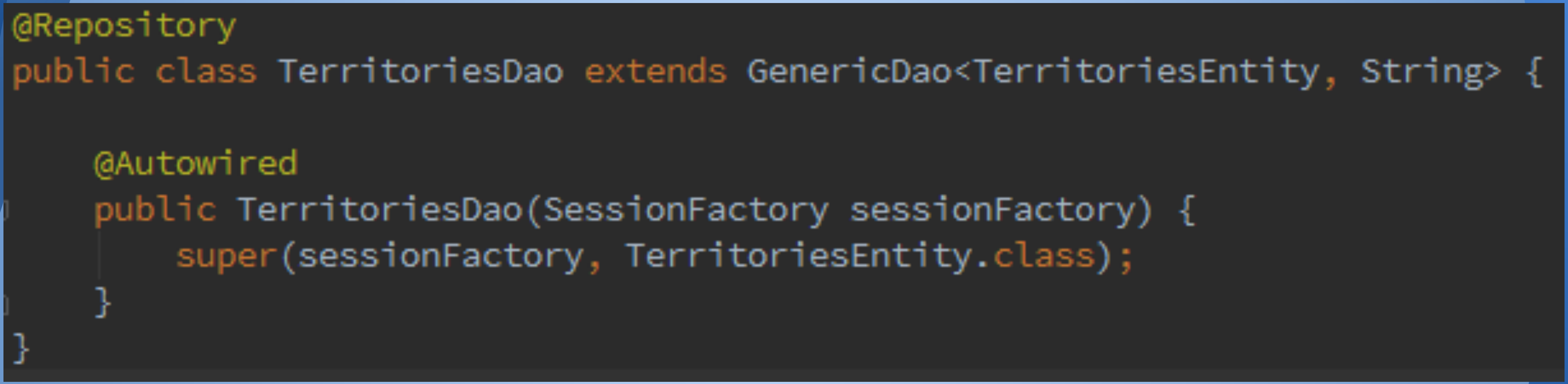


```
public interface Dao<T, PK extends Serializable> {

    PK create(T persistentObject);
    T get(PK id);
    List<T> getAll();
    void update(T persistentObject);
    void delete(T persistentObject);
}
```

- try your luck with implementing this basing on session methods
- remember you don’t have to access session factory in dao methods - treat sessionfactory as given for instance by contructor
- then compare your work with GenericDao from project

Accessing and modifying database - Data Access Object

- DAO are created one per entity, so lets now create examplaty instance

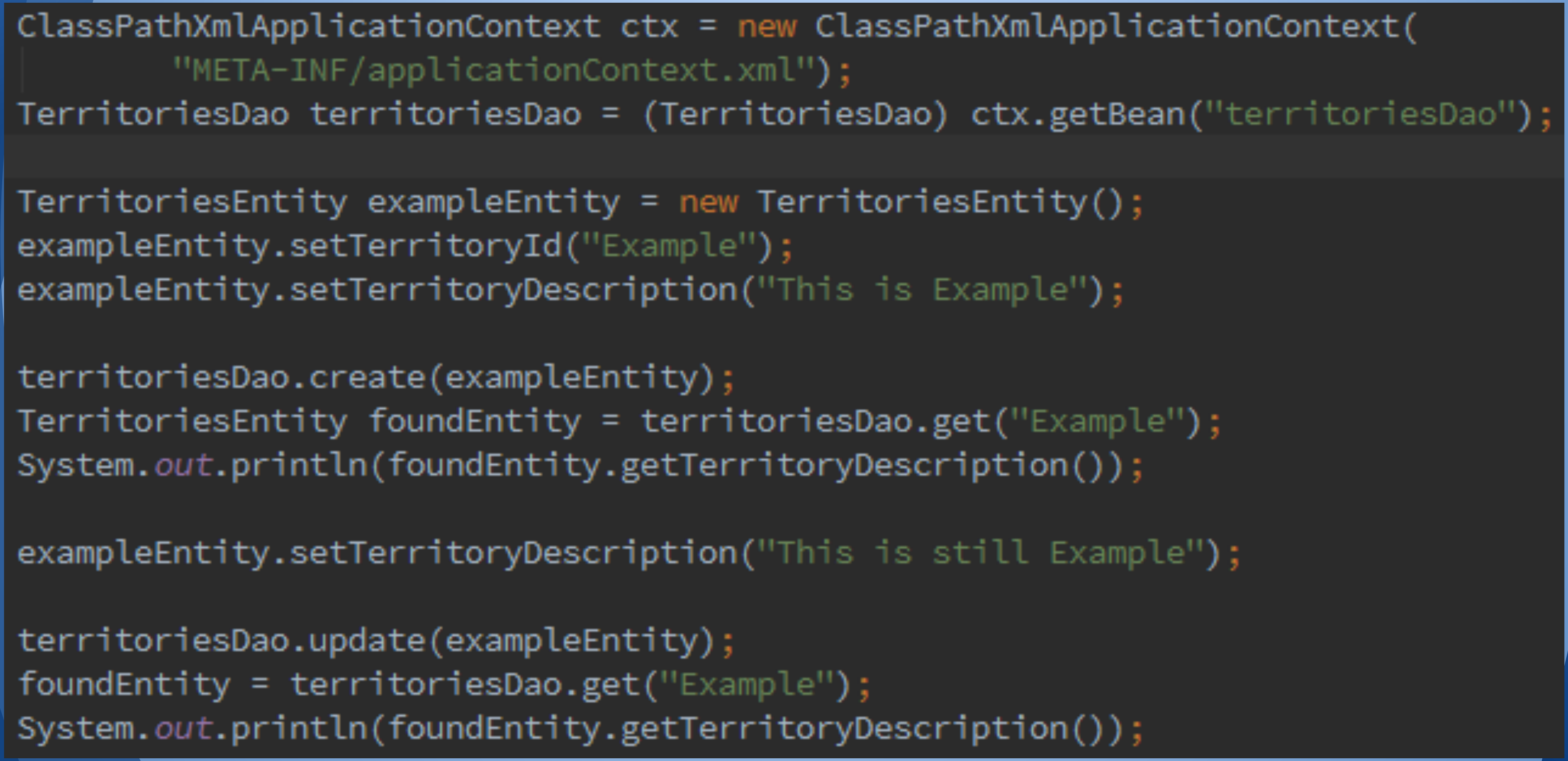


```java
@Repository
public class TerritoriesDao extends GenericDao<TerritoriesEntity, String> {

    @Autowired
    public TerritoriesDao(SessionFactory sessionFactory) {
        super(sessionFactory, TerritoriesEntity.class);
    }
}
```

- sometimes you may extend concrete dao
- using SpringIOC takse care of providing session factory (see @Autowired)
- registering this dao by using @Repository even more simplifies DAO usage

Accessing and modifying database - DAO usage

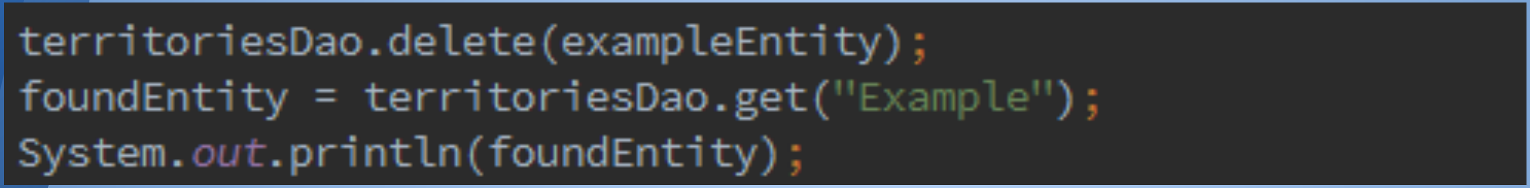
lets convert our runner which directly used sesion methods to employ our DAO



```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
        "META-INF/applicationContext.xml");
TerritoriesDao territoriesDao = (TerritoriesDao) ctx.getBean("territoriesDao");

TerritoriesEntity exampleEntity = new TerritoriesEntity();
exampleEntity.setTerritoryId("Example");
exampleEntity.setTerritoryDescription("This is Example");

territoriesDao.create(exampleEntity);
TerritoriesEntity foundEntity = territoriesDao.get("Example");
System.out.println(foundEntity.getTerritoryDescription());

exampleEntity.setTerritoryDescription("This is still Example");

territoriesDao.update(exampleEntity);
foundEntity = territoriesDao.get("Example");
System.out.println(foundEntity.getTerritoryDescription());
```

cdn...

Accessing and modifying database - DAO usage

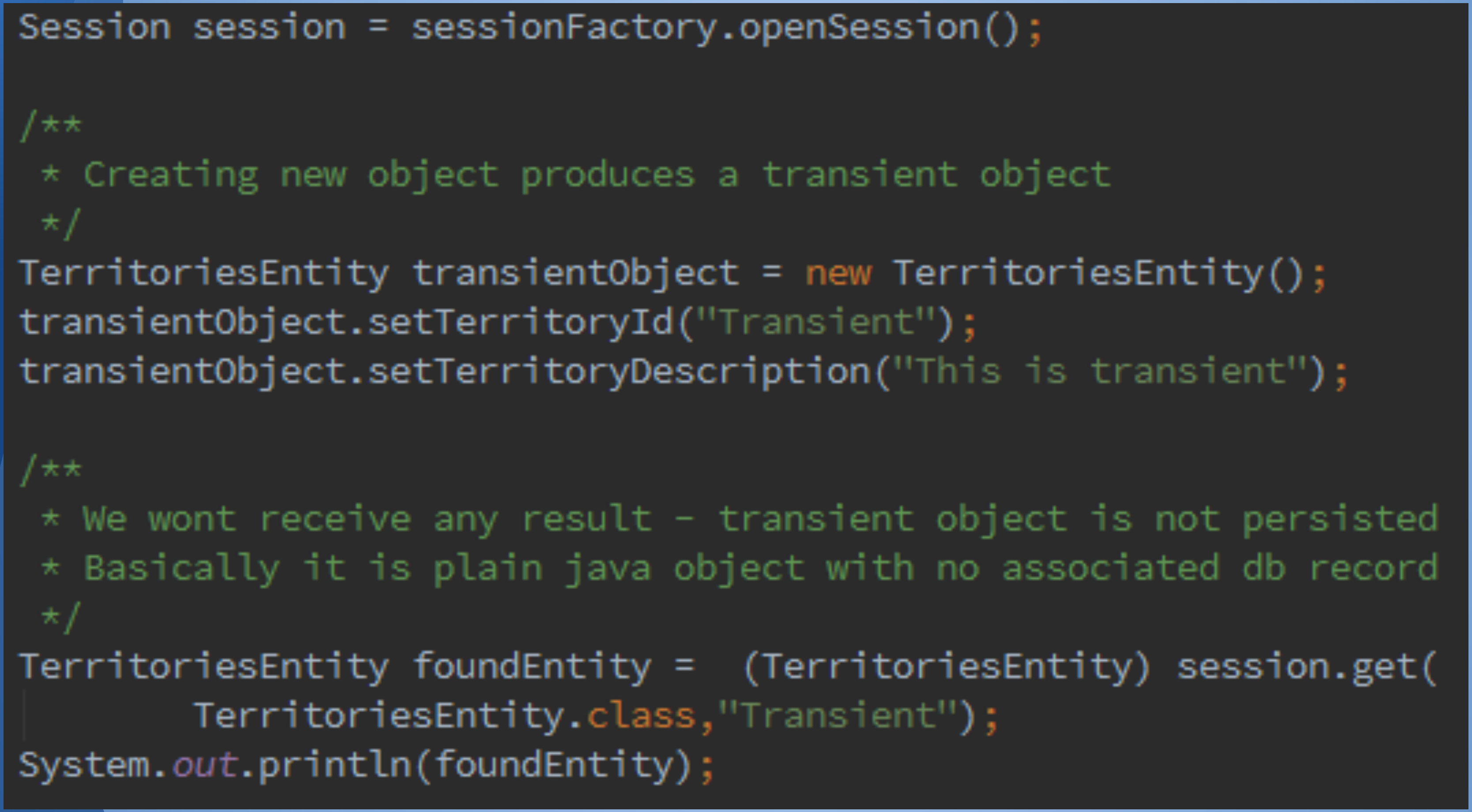


```
territoriesDao.delete(exampleEntity);
foundEntity = territoriesDao.get("Example");
System.out.println(foundEntity);
```

take a look at benefits from using dao

- session management is externalised
- DRY rule is satisfied - generic dao encapsulates often used procedures
- in concrete dao’s you may store part of your logic, for instance have methods ‘getForCurrentBillingInterval’
  - even though including business logic in repositories is highly controversial ;)

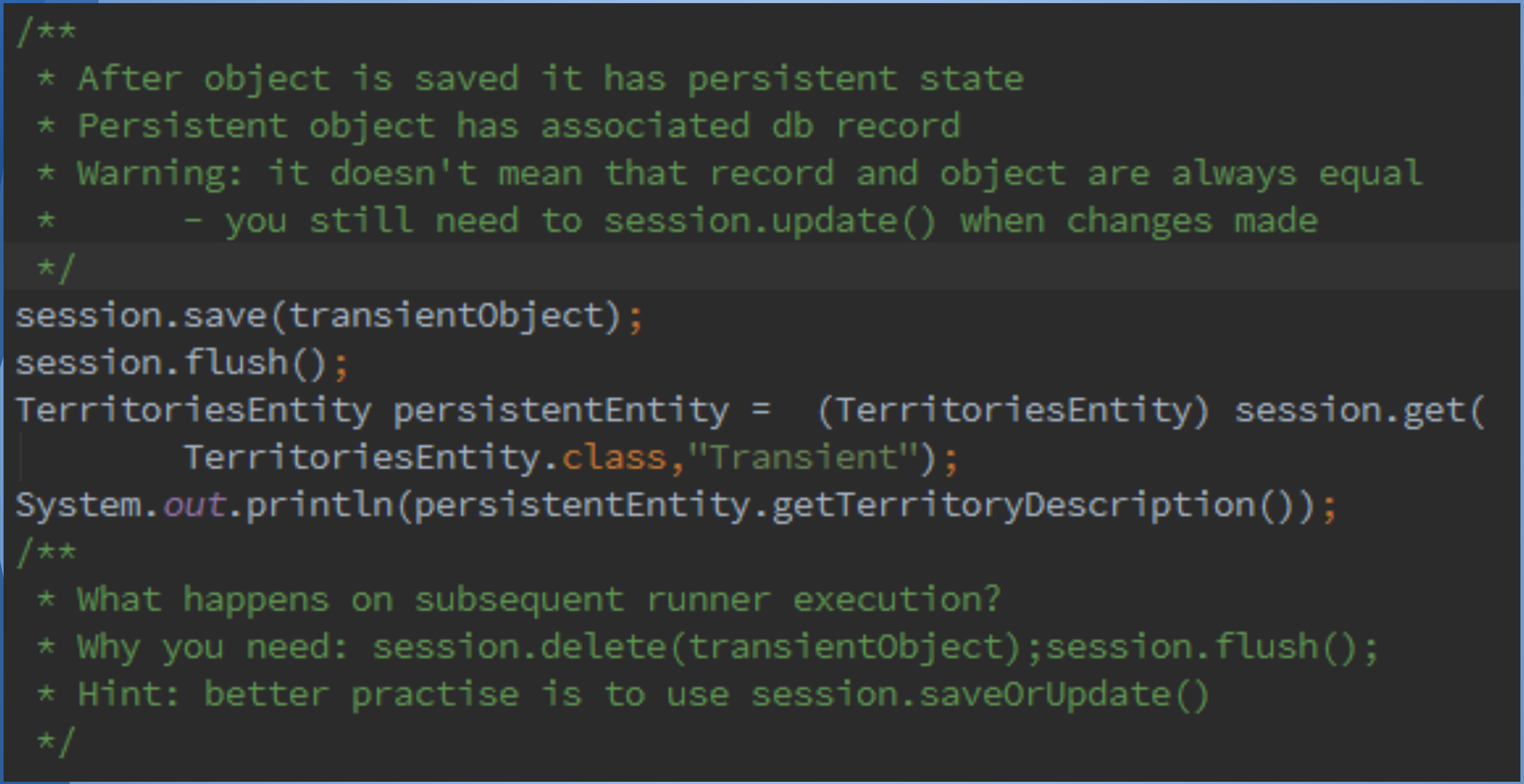
Object states - overview by example



```
Session session = sessionFactory.openSession();

/**
 * Creating new object produces a transient object
 */
TerritoriesEntity transientObject = new TerritoriesEntity();
transientObject.setTerritoryId("Transient");
transientObject.setTerritoryDescription("This is transient");

/**
 * We wont receive any result - transient object is not persisted
 * Basically it is plain java object with no associated db record
 */
TerritoriesEntity foundEntity =  (TerritoriesEntity) session.get(
        TerritoriesEntity.class,"Transient");
System.out.println(foundEntity);
```

cdn...

Object states - overview by example

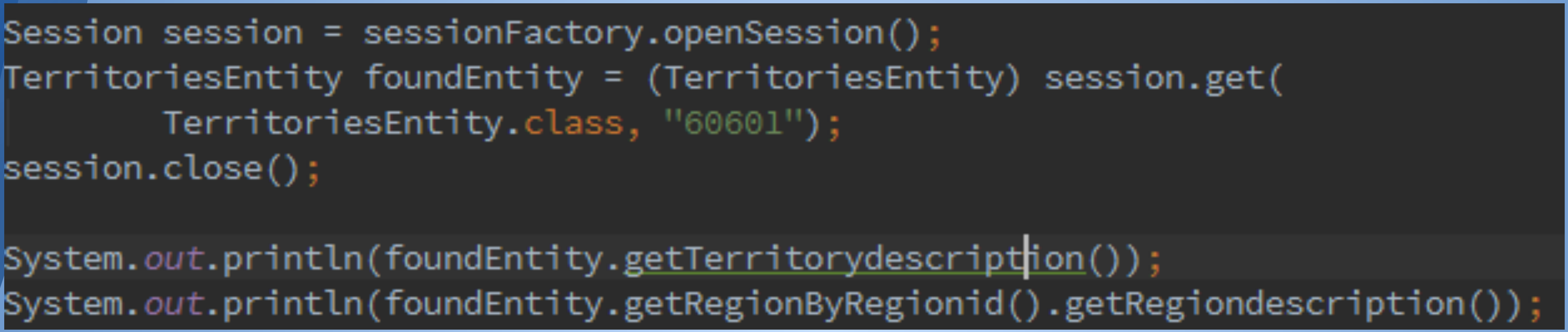


```
/**
 * After object is saved it has persistent state
 * Persistent object has associated db record
 * Warning: it doesn't mean that record and object are always equal
 *       - you still need to session.update() when changes made
 */
session.save(transientObject);
session.flush();
TerritoriesEntity persistentEntity =  (TerritoriesEntity) session.get(
        TerritoriesEntity.class,"Transient");
System.out.println(persistentEntity.getTerritoryDescription());
/**
 * What happens on subsequent runner execution?
 * Why you need: session.delete(transientObject);session.flush();
 * Hint: better practise is to use session.saveOrUpdate()
 */
```

cdn...

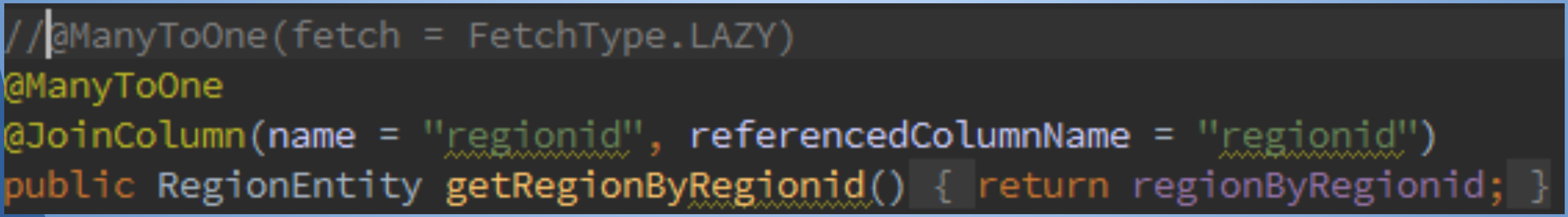
Fetching types - case study

try the following code:



```
Session session = sessionFactory.openSession();
TerritoriesEntity foundEntity = (TerritoriesEntity) session.get(
        TerritoriesEntity.class, "60601");
session.close();

System.out.println(foundEntity.getTerritorydescription());
System.out.println(foundEntity.getRegionByRegionid().getRegiondescription());
```

try changing (uncommenting) line in TerritoriesEntity:

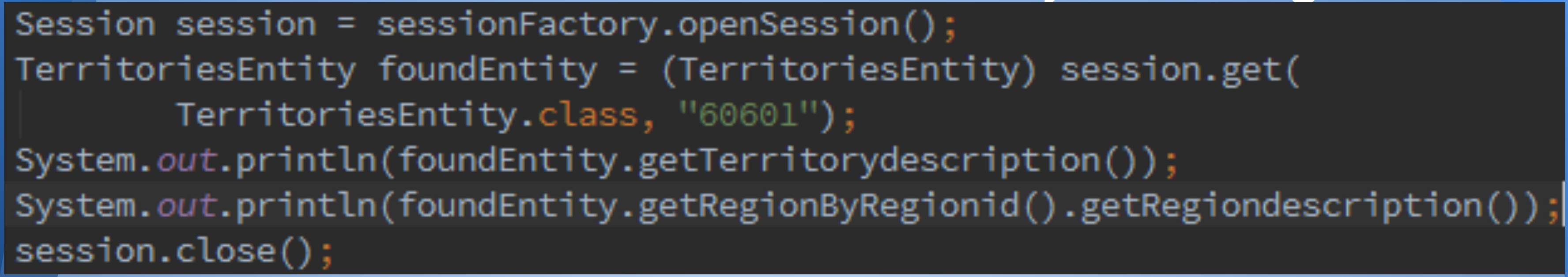


```
//@ManyToOne(fetch = FetchType.LAZY)
@ManyToOne
@JoinColumn(name = "regionid", referencedColumnName = "regionid")
public RegionEntity getRegionByRegionid() { return regionByRegionid; }
```

observe changes - what are your feelings about lazy fetching? do you see any advantages?

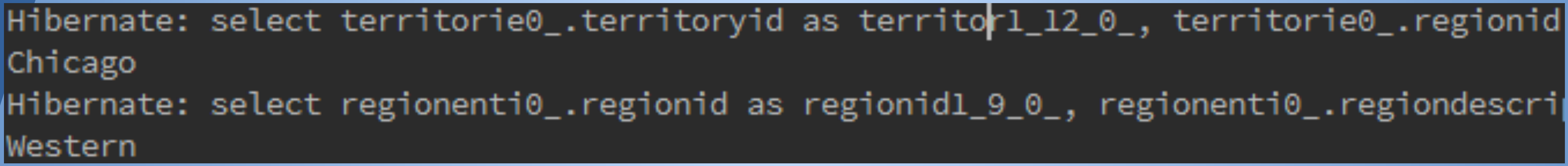
Fetching types - case study

- take a look at executed statements by following code:



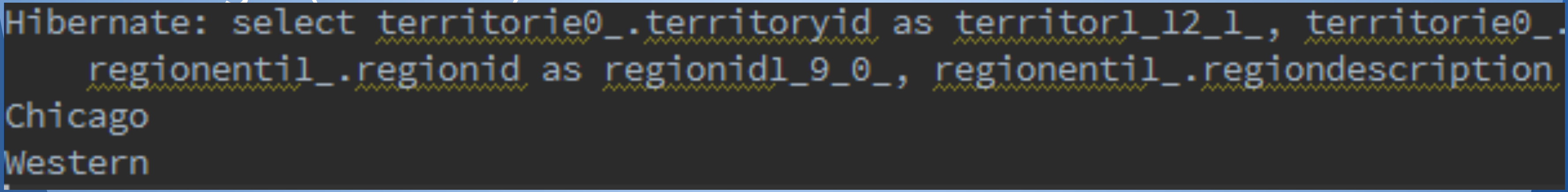
```
Session session = sessionFactory.openSession();
TerritoriesEntity foundEntity = (TerritoriesEntity) session.get(
        TerritoriesEntity.class, "60601");
System.out.println(foundEntity.getTerritorydescription());
System.out.println(foundEntity.getRegionByRegionid().getRegiondescription());
session.close();
```

when lazy fetching enabled:



```
Hibernate: select territorie0_.territoryid as territor1_12_0_, territorie0_.regionid
Chicago
Hibernate: select regionenti0_.regionid as regionid1_9_0_, regionenti0_.regiondescri
Western
```

when eager(default) enabled:



```
Hibernate: select territorie0_.territoryid as territor1_12_1_, territorie0_.
    regionenti1_.regionid as regionid1_9_0_, regionenti1_.regiondescription
Chicago
Western
```

- can you see what eager types cause?
- do you see link beetween moving session closing and susscessful execution of this/previous example with lazy fetching?

Fetching types - explanation

Fetching type is an manner in whitch nested data is aquired.

- Lazy means that additional select is executed on accessing data referenced by foreign key.
- When Eager (default) is set, select follows foreign keys and fetches all data at once.

Pros/Cons

- Eager ends witch fetching potentially huge amount of data, that could be unneccessary
- Lazy can lead to numerous, small select statements and high usage of db connection (and high amount of transactions, whitch will be discused another time)

Association Types

Relational databases are characterized by existance of … well, relations. In Hibernate, you define those relations using a certain set of annotations.

- @OneToOne
  - It defines that there exists one-to-one relation between two entities
- @OneToMany / @ManyToOne
  - They are both pretty much the same relation type. They only differ by the perspective of the owner
  - They are directional. The other (opposite to the owning) side of relation can reference the owner, by specifying ***mappedBy*** parameter

Association Types (cont.)

- @ManyToMany
  - It defines a many-to-many relation between entities
  - It requires a join table
  - It can be both directional and bi-directional, depending on the use of ***mappedBy*** parameter

There are several helper annotations that can be used to futher configure model to adhere to the schema (or influence the schema):

- @JoinColumn
  - specifies column names on both sides of relation
- @JoinTable
  - describes join column
  - especially helpful when you want to have full