Table of Content

Exercise 1
Entity definition

Duration

0,5 hour

Goal

Learn how to describe an entity in terms of Java Persistence API

Subject

You need to describe the Company entity that is mapped onto two database tables: Company and CompanyDetail.

Here are the tables DDLs:

```
CREATE TABLE Company (
	Company_id		INT PRIMARY KEY,
	Company_name	VARCHAR(50)
);
CREATE TABLE CompanyDetail (
	Company_id			INT PRIMARY KEY,
	CompanyDetail_address	VARCHAR(100)
);
```

Attributes of Company entity are mapped to tables' fields as follows:

- Company.id → Company.Company_id
- Company.name → Company.Company_name
- Company.address → CompanyDetail.CompanyDetail_address

Description

1. Open module jpa-lab-01
2. Open class edu.jpa.entity.Company
3. Specify class-level annotation @Entity. This lets JPA runtime to know that this particular class should be treated as an entity.
4. Specify class-level annotation @Table with the name of primary table "Country".
5. Specify class-level annotation @SecondaryTable with the name of secondary table "CompanyDetail". The parameter pkJoinColumns of @SecondaryTable annotation specifies how to join primary and secondary table when building an entity: it should be @PrimaryKeyJoinColumn(name = "Company_id", referencedColumnName = "Company_id").
6. Define Country entity fields: id (of the type Integer), name (of the type String) and address (of the type String).
7. Specify field-level annotation @Column for each field with information which field is to be mapped on this particular field. Example: @Column(name="Company_id", table="Company")
8. Specify field-level annotation @Id for the id field.

9. Open and run the class edu.jpa.Launcher. There should be no errors if entity is defined correctly.
10. Open database DB_LAB_01 using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Exercise 2
Identity of entity definition

Duration

0,5 hour

Goal

Learn how to define simple and composite identity of entity in terms of Java Persistence API.

Subject

There is a Department entity (with fields: companyName, name and description) and 3 cases to defined identity for it:

1. Single identity
2. Composite with identity as separate class (using @EmbeddedId)
3. Composite with identity fields inside the entity class (using @IdClass)

You need to implement all three cases.

Description

Single identity

1. Open module jpa-lab-02
2. Open class edu.jpa.entity.Department_1
3. Specify class-level annotation @Entity. This lets JPA runtime to know that this particular class should be treated as an entity.
4. Specify field-level annotation @Id for id field. This lets JPA runtime to know that this fields is used as key
5. Specify field-level annotation @GeneratedValue for id field. This applies particular identity generation strategy for this field (here the default one will be applied).

Composite with identity as separate class (using @EmbeddedId)

6. Open class edu.jpa.entity.DepartmentKey. This class represents the entity identity (company name + department name).
7. Define this class as implementing java.io.Serializable (composite-id class must implement Serializable)
8. Open class edu.jpa.entity.Department_2
9. Specify class-level annotation @Entity. This lets JPA runtime to know that this particular class should be treated as an entity.
10. Specify field-level annotation @EmbeddedId for field id (of type DepartmentKey).

Composite with identity fields inside the entity class (using @IdClass)

11. Open class edu.jpa.entity.DepartmentKey

12. Specify class-level annotation @Embeddable. This lets JPA runtime to know that this class is embeddable one and can be a part of an entity.
13. Open class edu.jpa.entity.Department_3
14. Specify class-level annotation @Entity. This lets JPA runtime to know that this particular class should be treated as an entity.
15. Specify class-level annotation @IdClass with identity class DepartmentKey. This lets JPA runtime to know that DepartmentKey should be used as identity foe this entity (and entity class should contain fields with the same names as in DepartmentKey class).
16. Specify field-level annotation @Id for any of key fields (companyName or departmentName).
17. Open the class edu.jpa.Launcher and analyze its content with help of trainer (since EntityManager is not known for you yet).
18. Run the class edu.jpa.Launcher. There should be no errors if entity is defined correctly.
19. Open database DB_LAB_02 using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Exercise 3
Entities relations definition

Duration

0,5 hour

Goal

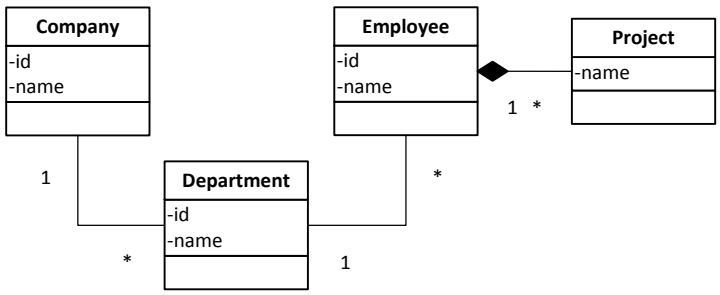
Learn how to describe entities relations in terms of Java Persistence API.

Subject

There are the following entities: Company, Department and Employee. The entities relates to each other as follows:

- Company relates to Department as one-to-many
- Department relates to Company as many-to-one
- Department relates to Employee as one-to-many
- Employee relates to Department as many-to-one
- Relations Company-Department and Department-Employee are bidirectional.
- Employee contains collection of Project objects.

Project is not an entity. It means that it does not have its own identifier and cannot be persisted separately (without owning entity).



You need to describe relations between the entities and objects using annotations @OneToMany, @ManyToOne and @ElementCollection.

Description

1. Open module jpa-lab-03
2. Open class edu.jpa.entity.Company:
3. Look on the field departments of type List<Department>. This field contains all departments that belong to company. Specify field-level annotation @OneToMany for the field departments. Annotation argument “mappedBy” should be specified to let JPA runtime know that this is bidirectional relation, and relation is defined by metadata specified for field defined by “mappedBy”: @OneToMany(mappedBy = "company")
4. Open class edu.jpa.entity.Department:
5. Look on the field company of type Company. This field contains the reference to Company entity this department belongs to. Specify field-level annotation @ManyToOne for the field company.

6. Look on the field employees of type List<Employee>. This field contains all employees that belong to the department. Specify field-level annotation @OneToMany for the field employees. Annotation argument “mappedBy” should be specified to let JPA runtime know that this is bidirectional relation, and relation is defined by metadata specified for field defined by “mappedBy”: @OneToMany(mappedBy = "department")
7. Open class edu.jpa.entity.Employee:
8. Look on the field department of type Department. This field contains the reference to Department entity this employee belongs to. Specify field-level annotation @ManyToOne for the field department.
9. Look on the field projects of type List<Project>. This field contains all the projects the employee is involved in. Specify field-level annotation @ElementCollection for the field projects.
10. Open the class edu.jpa.entity.embeddables.Project and marl this as embeddable one. Specify the class-level annotation @Embeddable and make this class implementing java.io.Serializable.
11. Open the class edu.jpa.Launcher and analyze its content with help of trainer (since EntityManager is not known for you yet).
12. Run the class edu.jpa.Launcher. There should be no errors if entity is defined correctly.
13. Analyze queries JPA runtime sends to database for data extraction (see STDOUT).
14. Open database DB_LAB_03 using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Exercise 4
Entities class hierarchy

Duration

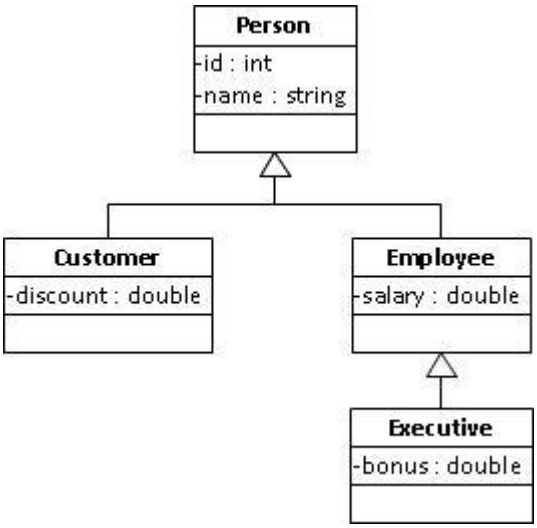
1 hour

Goal

Learn how to map entity-classes hierarchy (OOP paradigm) into relational model in terms of Java Persistence API.

Subject

There are 3 domain objects: Customer, Employee and Executive. All of them have common parent class Person that is abstract and contains shared fields (id and name). The following class diagram represents the hierarchy of classes (with its attributes):



You need to map this OO model into relational model using facilities provided by JPA: Single table per class hierarchy (*InheritanceType.SINGLE_TABLE*); Table per concrete class (*InheritanceType.TABLE_PER_CLASS*); and Table per class (*InheritanceType.JOINED*).

Description

Inheritance type SINGLE_TABLE

1. Open module jpa-lab-04
2. Look on the package edu.jpa.TABLE_PER_HIERARCHY.entity. This package contains entity-classes for the domain.
3. Open class edu.jpa.TABLE_PER_HIERARCHY.entity.Person and add class-level annotations:

   @Entity – marks class as entity-class

   @Inheritance(strategy=InheritanceType.SINGLE_TABLE) – defines the hierarchy mapping strategy to use.

@DiscriminatorColumn(name="TYPE", discriminatorType=DiscriminatorType.STRING) – defines the database table field that will be used to keep discriminator value

4. Open class edu.jpa.TABLE_PER_HIERARCHY.entity.Customer and add class-level annotations:

   @Entity – marks class as entity-class

   @DiscriminatorValue("Customer") – defines value “Customer” as discriminator value for this type

5. Open class edu.jpa.TABLE_PER_HIERARCHY.entity.Employee and add class-level annotations:

   @Entity – marks class as entity-class

   @DiscriminatorValue("Employee") – defines value “Employee” as discriminator value for this type

6. Open class edu.jpa.TABLE_PER_HIERARCHY.entity.Executive and add class-level annotations:

   @Entity – marks class as entity-class

   @DiscriminatorValue("Executive") – defines value “Executive” as discriminator value for this type

7. Open class edu.jpa.TABLE_PER_HIERARCHY.Launcher and analyze its content:

   method init() create objects and saves them into database

   method sample() finds the entity by identified and prints the “name” attribute to console

8. Run class edu.jpa.TABLE_PER_HIERARCHY.Launcher. There should be no errors if entity is defined correctly.
9. Analyze queries JPA runtime sends to database for data extraction.
10. Open database DB_LAB_04_TABLE_PER_HIERARCHY using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Inheritance type TABLE_PER_CLASS

11. Look on the package edu.jpa.TABLE_PER_CLASS.entity. This package contains entity-classes for the domain.
12. Open class edu.jpa.TABLE_PER_CLASS.entity.Person and add class-level annotations:

    @MappedSuperclass – marks this class as template for child entity clasess.

    @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) – defines the hierarchy mapping strategy to use.

13. Open class edu.jpa.TABLE_PER_CLASS.entity.Customer and add class-level annotations:

    @Entity – marks class as entity-class

14. Open class edu.jpa.TABLE_PER_CLASS.entity.Employee and add class-level annotations:

    @Entity – marks class as entity-class

15. Open class edu.jpa.TABLE_PER_CLASS.entity.Executive and add class-level annotations:

    @Entity – marks class as entity-class

16. Open class edu.jpa.TABLE_PER_CLASS.Launcher and analyze its content:

    method init() create objects and saves them into database

    method sample() finds the entity by identified and prints the “name” attribute to console

17. Run class edu.jpa.TABLE_PER_CLASS.Launcher. There should be no errors if entity is defined correctly.
18. Analyze queries JPA runtime sends to database for data extraction.
19. Open database DB_LAB_04_ TABLE_PER_CLASS using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Inheritance type JOINED

20. Look on the package edu.jpa.TABLE_PER_SUBCLASS.entity. This package contains entity-classes for the domain.
21. Open class edu.jpa.TABLE_PER_SUBCLASS.entity.Person and add class-level annotations:

    @Entity – marks class as entity-class

    @Inheritance(strategy = InheritanceType.JOINED) – defines the hierarchy mapping strategy to use.

22. Open class edu.jpa.TABLE_PER_SUBCLASS.entity.Customer and add class-level annotations:

    @Entity – marks class as entity-class

23. Open class edu.jpa.TABLE_PER_SUBCLASS.entity.Employee and add class-level annotations:

    @Entity – marks class as entity-class

24. Open class edu.jpa.TABLE_PER_SUBCLASS.entity.Executive and add class-level annotations:

    @Entity – marks class as entity-class

25. Open class edu.jpa.TABLE_PER_SUBCLASS.Launcher and analyze its content:

    method init() create objects and saves them into database

    method sample() finds the entity by identified and prints the “name” attribute to console

26. Run class edu.jpa.TABLE_PER_SUBCLASS.Launcher. There should be no errors if entity is defined correctly.
27. Analyze queries JPA runtime sends to database for data extraction.

28. Open database DB_LAB_04_TABLE_PER_SUBCLASS using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Exercise 5
Working with EntityManager

Duration

0,5 hour

Goal

Learn how to create and use EntityManager abstraction for managing entity persistent state.

Subject

You have the following domain objects (with all persistence attributes defined): Company, Department and Employee. Also you have CompanyService object that is purposed for managing the Company entity. It has the following methods:

- getCompany(int)::Company – finds and returns a company entity by the given identifier
- saveCompany(Company)::void – saves the state of given company entity to database
- init()::void – (initializes database) creates a single company entity with name “Microsoft”

Description

1. Open module jpa-lab-05.
2. Look on the classes in “edu.jpa.entity” package to be aware about domain objects and its structure.
3. Open file META-INF/persistence.xml and configure Persistence Unit (PU):
    - Specify name of PU for persistence-unit (ex: persistenceUnits.lab05)
    - Specify entity classes (class element inside the persistence-unit): Company, Department, Employee
    - Take a look on “properties” element and learn the properties
4. Open class CompanyService and create EntityManagerFactory instance: emf = Persistence.createEntityManagerFactory("persistenceUnits.lab05") in static block. EntityManagerFactory is needed to create EntityManager instance.
5. Open class CompanyService and implement methods: getCompany(int), saveCompany(Company) and init(). See the “Subject” section for the purpose of the methods.

Implementing methods *getCompany(int)*

6. Create EntityManager instance:

```
EntityManager em = emf.createEntityManager()
```

7. Begin transaction:

```
em.getTransaction().begin()
```

8. Perform the entity search:

```
Company res = em.find(Company.class, id)
```

9. Finish the transaction:

```
em.getTransaction().rollback()
```

10. Return the found entity

Implementing methods *saveCompany(Company)*

11. Create EntityManager instance:

```
EntityManager em = emf.createEntityManager()
```

12. Begin transaction:

```
em.getTransaction().begin()
```

13. Merge given entity into persistence context:

```
em.merge(company)
```

14. Finish the transaction:

```
em.getTransaction().commit()
```

Question: Why the *merge()* method is used (not *persist()*)?

Implementing methods *init()*

15. Create EntityManager instance:

```
EntityManager em = emf.createEntityManager()
```

16. Begin transaction:

```
em.getTransaction().begin()
```

17. Create the company object and persist it:

```
final Company company = new Company();
company.setName("Microsoft");
em.persist(company);
```

18. Finish the transaction:

```
em.getTransaction().commit()
```

Question: Why the *persist()* method is used (not *merge()*)?

19. Open class Launcher and analyze it.
20. Run the class Launcher.
21. Open database DB_LAB_05 using dbVisualizer application, and look on the created database objects (tables, constraints, etc.) and data. Analyze it.

Exercise 6
Working with JPQL and Criteria API

Duration

0,5 hour

Goal

Learn the facilities JPA provides for searching entities: JPQL and Criteria API.

Subject

You have domain objects represented by classes: Company, Department and Employee. Also you have service layer objects represented by classes:

- EntityService – base class for all service classes. Contains logic for entities initialization (see the class constructor) and defines the methods:
  - getEntityManagerFactory() returns EntityManagerFactory instance that is created and initialized.
  - getEmployeesByDepartmentName(String)::List<Employee> - method for searching employees by name of department they belong to. This method is abstract and should be implemented in EntityService1 and EntityService2.
  - getDepartmentsInfo()::List<DepartmentInfo> - method for gathering information about all existing departments in form of DepartmentInfo (auxiliary JavaBean class). This method is abstract and should be implemented in EntityService1 and EntityService2.
- EntityService1 – **implementation** of search logic for EntityService **that should use JPQL** for searching.
- EntityService2 – **implementation** of search logic for EntityService that **should use JPA Criteria API** for searching.

You need to implement EntityService1 and EntityService2 classes.

Description

1. Open module jpa-lab-06.
2. Look on the classes in “edu.jpa.entity” package to be aware about domain objects and its structure.
3. Open class EntityService and analyze its structure:
   - Class constructor with logic for creating EntityManagerFactory instance and initializing the entities
   - getEntityManagerFactory()::EntityManagerFactory method that should be used in child classes for getting EntityManagerFactory instance
   - abstract methods getEmployeesByDepartmentName(String) and getDepartmentsInfo() that you need to implement in child classes EntityService1 and EntityService2

Using JPQL

4. Open class EntityService1

5. Implement method getEmployeesByDepartmentName(String):
6. Create the EntityManager:

```
EntityManager em = getEntityManagerFactory().createEntityManager()
```

7. Start the transaction:

```
em.getTransaction().begin()
```

8. Define the JPQL query:

```
String queryText = "select e from Employee e where e.department.name =
:name"
```

9. Create TypedQuery object:

```
TypedQuery<Employee> query = em.createQuery(queryText, Employee.class)
```

10. Specify the value for parameter “name”:

query.setParameter("name", name)

11. Execute the query:

```
List<Employee> result = query.getResultList()
```

12. Finalize the transaction:

```
em.getTransaction().rollback()
```

13. Implement method getDepartmentsInfo():
14. Create the EntityManager:

```
EntityManager em = getEntityManagerFactory().createEntityManager()
```

15. Start the transaction:

```
em.getTransaction().begin()
```

16. Define the JPQL query:

```
String queryText = "select new
edu.jpa.service.DepartmentInfo(e.department.name,count(e.department)) from
Employee e group by e.department.name"
```

17. Create TypedQuery object:

```
TypedQuery<DepartmentInfo> query = em.createQuery(queryText,
DepartmentInfo.class)
```

18. Execute the query:

```
List<DepartmentInfo> result = query.getResultList()
```

19. Finalize the transaction:

```
em.getTransaction().rollback()
```

Using Criteria API

20. Open class EntityService2
21. Implement method getEmployeesByDepartmentName(String):

```
Create the EntityManager: EntityManager em =
getEntityManagerFactory().createEntityManager()
```

22. Create CriteriaBuilder object:

```
CriteriaBuilder cb = em.getCriteriaBuilder()
```

23. Create CriteriaQuery object:

```
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class)
```

24. Specify the FROM statement (entity you are looking for):

```
Root e = cq.from(Employee.class)
```

25. Specify WHERE condition for the query:

```
cq.where(cb.equal(e.get("department").get("name"),
cb.parameter(String.class, "name")))
```

26. Start the transaction:

```
em.getTransaction().begin()
```

27. Create TypedQuery object:

```
TypedQuery<Employee> query = em.createQuery(cq);
```

28. Execute the query:

```
List<Employee> result = query.getResultList()
```

29. Finalize the transaction:

```
em.getTransaction().rollback()
```

30. Implement method getDepartmentsInfo():
31. Create the EntityManager:

```
EntityManager em = getEntityManagerFactory().createEntityManager()
```

32. Create CriteriaBuilder object:

```
CriteriaBuilder cb = em.getCriteriaBuilder()
```

33. Build the query:

    Create CriteriaQuery object:

    ```
    CriteriaQuery<DepartmentInfo> cq = cb.createQuery(DepartmentInfo.class)
    ```

    Specify the FROM statement:

    ```
    Root e = cq.from(Employee.class)
    ```

    Since the result we need differs from the entity we use in FROM statement the SELECT statement should be defined (with creation of DepartmentInfo basing on data selected from target entity):

    ```
    cq.select(cb.construct(DepartmentInfo.class,e.get("department").get("nam
    e"), cb.count(e.get("department"))))
    ```

    Specify the GROUP BY statement:

    ```
    cq.groupBy(e.get("department"))
    ```

34. Create TypedQuery object:

```
TypedQuery<DepartmentInfo> query = em.createQuery(cq)
```

35. Execute the query:

```
List<DepartmentInfo> result = query.getResultList()
```

36. Finalize the transaction:

```
em.getTransaction().rollback()
```

Running

37. Open class Launcher and analyze it.

38. Run the class Launcher.
39. Analyze the log written to STDOUT

Exercise 7
Integration with Spring Framework

Duration

0,5 hour

Goal

Learn facilities the Spring Framework provides for supporting JPA infrastructure.

Subject

You have an entity Company and service class CompanyService for managing the Company entity.
You need to do the following:

- Implement logic of CompanyService class using Spring Framework facilities to inject reference to EntityManager
- Provide transaction configuration for methods of CompanyService (declarative transaction management)
- Configure Spring Framework context with all needed objects (JDBC data source, JPA EntityManagerFactory, Transaction manager, CompanyService, etc.)

Description

1. Open module jpa-lab-07

Implementing CompanyService

2. Open class edu.jpa.service.CompanyService
3. Specify object stereotype @Repository for CompanyService class (optional, needed if component-scan is used)

```
@Repository
public class CompanyService {
        . . .
}
```

4. Define the class fields of type EntityManager to inject the reference to entoty manager:

```
@PersistenceContext
private EntityManager em;
```

5. Implement init() method. See the code sample below:

```
public void init() {
    String[] companies = {"Microsoft", "IBM"};
    for (String name : companies) {
        Company company = new Company();
        company.setName(name);
        em.persist(company);
    }
}
```

6. Specify transaction attributes via using @Transactional annotation:

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
```

7. Implement getCompany(int)::Company method. See the code sample below:

```
    public Company getCompany(int id) {
        return em.find(Company.class, id);
    }
```

8. Specify transaction attributes via using @Transactional annotation:

```
@Transactional(propagation = Propagation.REQUIRES_NEW, readOnly = true)
```

Spring Framework configuration

9. Open application-context.xml file
10. Enable loading configuration parameters from application-context.properties file: *<context:property-placeholder location="application-context.properties"/>*
11. Enable annotations support for dependencies injection and configuration: *<context:annotation-config/>*
12. Define JDBC DataSource that will be used to get JDBC connection to database:

```
  <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
```

13. Define EntityManagerFactory factory bean with all JPA configuration included (persistence unit name, dialect, reference to data source, properties, etc.):

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="persistenceUnits.lab07" />
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaProperties">
        <props>
            <prop
key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
        </props>
    </property>
    <property name="jpaDialect">
        <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
    </property>
</bean>
```

14. Define transaction management strategy bean:

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
    <property name="dataSource" ref="dataSource" />
</bean>
```

15. Define CompanyService as sprint bean to let Spring Framework inject the needed dependencies:

```
<bean class="edu.jpa.service.CompanyService"/>
```

Running

16. Open file META-INF/persistence.xml and look how simple it becomes (no properties, no connection info).
17. Open class edu.jpa.Launcher and analyze it.
18. Run class edu.jpa.Launcher and analyze the output.

Exercise 8
Blog Application development

Duration

4 hours

Goal

Summarize the stuff learned during the training and apply the knowledge on the practice.

Subject

The task is to develop the web-application implementing blog facilities. Application operates with entities Blog and BlogPost. The entities are defined in edu.jpa.blog.domain package and have the following attributes:

Blog:
- id – unique identifier of the entity
- version – version of the entity
- name – name of the blog
- author – author of the blog

BlogPost:
- id – unique identifier of the entity
- version – version of the entity
- date – date when post is created
- title – headline of the post
- text – text of the post

Since Blog and Blog post have common attributes (id and version) it makes sense to define base class for the entities and put all the common attributes here. The only thing that you need to do in this case is to override the attributes mapping to database (see @AttributeOverrides and @AttributeOverride annotations).

Application functionality:

Here is a list of functions application should provide:

1. Manage list of Blog entities:
   a. Show the list of blogs
   b. Add new blog entity
   c. Edit the particular blog entity
   d. Remove the blog entity
2. Manage list of BlogPost entities:
   a. Show the list of post for particular blog
   b. Add new post to particular blog
   c. Edit the particular post entity
   d. Remove the particular post
   e. Search the post inside the particular blog by text and date

Application Structure

Application has three layers: DAO, service (see package edu.jpa.blog.service) and presentation (see package edu.jpa.blog.web):

- DAO layer is represented by the JPA implementation (Hibernate Entity Manager).
- Service layer is represented by the BlogService and BlogPostService interfaces, and their implementation classes BlogServiceImpl and

BlogPostServiceImpl. These classes accumulate an entities management logic. To pass the data between service and presentation layers the DTO classes are used (see package edu.jpa.blog.service.dto).

- Presentation layer is implemented using Spring MVC and represented by:
  - Controller classes BlogController and BlogPostController
  - Resources (css, jsp, deployment descriptors) which are located in src/main/webapp directory.

Data storage is represented by HSQL database. See file src\main\resources\db\schema.sql for database structure.

Description

1. Open module jpa-lab-08

Entities description: class DomainObject

2. Open abstract class edu.jpa.blog.domain.DomainObject
3. Specify @MappedSuperclass annotation for this class
4. Define the “id” attribute:

```
@Id
@GeneratedValue
private int id;
```

5. Define the “version” attribute:

```
@Version
@private int version;
```

6. Implement the equals() and hashCode() methods (use IDE code generation facilities)
7. Write the setters and getters.

Entities description: class Blog

8. Open class edu.jpa.blog.domain.Blog
9. Make it extending edu.jpa.blog.domain.DomainObject
10. Specify @Table(name = "BLOG") annotation for class to map to DB table
11. Override “id” and “version” attributes mappings:

```
@AttributeOverrides({
    @AttributeOverride(name="id", column=@Column(name="BLOG_ID")),
    @AttributeOverride(name="version", column=@Column(name="BLOG_VERSION"))
})
```

12. Define the JPQL named query that will be used later in service layer classes:

```
@NamedQuery(
    name ="findAll",
    query = "select B from Blog B order by B.name")
```

13. Define the “name” attribute:

```
@Column(name = "BLOG_NAME")
private String name;
```

14. Define the “author” attribute:

```
@Column(name = "BLOG_AUTHOR")
```

```
private String author;
```

15. Define the blog-to-posts relation:

```
@OneToMany(mappedBy = "blog", fetch = FetchType.EAGER, cascade =
CascadeType.ALL)
private List<BlogPost> posts;
```

16. Write the setters and getters.

Entities description: class BlogPost

17. Open class `edu.jpa.blog.domain.BlogPost`
18. Make it extending `edu.jpa.blog.domain.DomainObject`
19. Specify `@Table(name = "BLOG_POST")` annotation for class to map to DB table
20. Override "id" and "version" attributes mappings:

```
@AttributeOverrides({
    @AttributeOverride(name="id", column=@Column(name="POST_ID")),
    @AttributeOverride(name="version", column=@Column(name="POST_VERSION"))
})
```

21. Define the JPQL named query that will be used later in service layer classes:

```
@NamedQuery(
    name = "findByBlog",
    query = "select P from BlogPost P where P.blog.id = :id"
)
```

22. Define the "date" attribute:

```
@Column(name = "POST_DATE")
private Date date;
```

23. Define the "title" attribute:

```
@Column(name = "POST_TITLE")
private String title;
```

24. Define the "text" attribute:

```
@Lob
@Basic(fetch = FetchType.EAGER)
@Column(name = "POST_TEXT")
private String text;
```

25. Define the post-to-blog relation:

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "BLOG_ID")
private Blog blog;
```

26. Write the setters and getters.

Learning DTO objects

27. Open class edu.jpa.blog.service.dto.BlogDTO. This class is used to pass Blog entity information from presentation layer to service layer and vice versa.
28. Open class edu.jpa.blog.service.dto.BlogPostDTO. This class is used to pass BlogPost entity information from presentation layer to service layer and vice versa.
29. Open class edu.jpa.blog.service.dto. SearchCriteriaDTO. This class is used to pass search criteria (to find BlogPost entities) from presentation layer to service layer.

That means that presentation layer does not operate over the entities. Service layer acts as transaction entry point and provides entities' management service for presentation layer communicating with it through DTO objects.

Implementing service objects: class BlogServiceImpl

30. Open class `edu.jpa.blog.service.BlogService`. This interface defines the following methods:

    `getBlogs()::List<BlogDTO>` – retrieves list of existing blogs in form of BlogDTO objects

    `getBlog(int)::BlogDTO` – retrieves the particular blog in form of BlogDTO object by its identifier

    `removeBlog(int)::void` – removes the blog by its identifier

    `modifyBlog(BlogDTO)::void` – saves the changes made to blog that is represented by BlogDTO

31. Open class `edu.jpa.blog.service.BlogServiceImpl`
32. Specify the @Repository annotations for the class (defines the class stereotype in terms of Spring Container)
33. Specify transaction configuration for class:

```
@Transactional(Transactional.TxType.REQUIRED)
```

34. Inject the reference to JPA EntityManager (Spring Framework is responsible for dependencies injection to its beans)

```
@PersistenceContext
private EntityManager em;
```

35. Implement method getBlogs()::List<BlogDTO>

```
public List<BlogDTO> getBlogs() {
    final TypedQuery<Blog> query = em.createNamedQuery("findAll",
    Blog.class);
    final List<Blog> blogs = query.getResultList();
    final List<BlogDTO> result = new ArrayList<BlogDTO>(blogs.size());

    for (final Blog blog : blogs) {
           result.add(new BlogDTO(blog));
    }
    return result;
}
```

This method uses named query "findAll" that is be defined for Blog class.

36. Implement method getBlog(int)::BlogDTO

```
public BlogDTO getBlog(final int id) {
    final Blog blog = em.find(Blog.class, id);
    return new BlogDTO(blog);
}
```

37. Implement method removeBlog (int)::void

```
public void removeBlog(int id) {
    final Blog blog = em.find(Blog.class, id);
    em.remove(blog);

}
```

38. Implement method modifyBlog(BlogDTO)::void

```
public void modifyBlog(BlogDTO blog) {
    final Blog persistedBlog;
```

```
    if (blog.getId() > 0) {
            persistedBlog = em.find(Blog.class, blog.getId());
    } else {
            persistedBlog = new Blog();
    }

    persistedBlog.setAuthor(blog.getAuthor());
    persistedBlog.setName(blog.getName());
    em.persist(persistedBlog);
}
```

Implementing service objects: class BlogPostServiceImpl

39. Open class `edu.jpa.blog.service.BlogPostService`. This interface defines the following methods:

    `getBlogPosts(int)::List<BlogPostDTO>` – retrieves existing blogs posts (by blod identifier) in form of BlogPostDTO objects

    `getBlogPost(int)::BlogPostDTO` – retrieves the particular post in form of BlogPostDTO object by its identifier

    `removePost(int)::void` – removes the post by its identifier

    `modifyBlogPost(BlogPostDTO)::void` – saves the changes made to post that is represented by BlogPostDTO

    `findBlogPosts(SearchCriteriaDTO)::List<BlogPostDTO>` – finds posts that match the given search criteria

40. Open class `edu.jpa.blog.service.BlogPostServiceImpl`
41. Specify the @Repository annotations for the class (defines the class stereotype in terms of Spring Container)
42. Specify transaction configuration for class:

    ```
    @Transactional(Transactional.TxType.REQUIRED)
    ```

43. Inject the reference to JPA EntityManager (Spring Framework is responsible for dependencies injection to its beans)

    ```
    @PersistenceContext
    private EntityManager em;
    ```

44. Implement method getBlogPosts(int)::List<BlogPostDTO>

    ```
    public List<BlogPostDTO> getBlogPosts(int id) {
        final TypedQuery<BlogPost> query = em.createNamedQuery("findByBlog",
        BlogPost.class);
        query.setParameter("id", id);
        final List<BlogPost> posts = query.getResultList();

        final List<BlogPostDTO> result = new
        ArrayList<BlogPostDTO>(posts.size());
        for (final BlogPost post : posts) {
                result.add(new BlogPostDTO(post));
        }
        return result;
    }
    ```

    This method uses named query "findByBlog" that is be defined for BlogPost class.

45. Implement method getBlogPost(int)::BlogPostDTO

    ```
    public BlogPostDTO getBlogPost(int id) {
        final BlogPost post = em.find(BlogPost.class, id);
    ```

```
    return new BlogPostDTO(post);
}
```

46. Implement method removePost (int)::void

```
public void removePost(int id) {
    final BlogPost post = em.find(BlogPost.class, id);
    post.setBlog(null);
    em.remove(post);
}
```

"post.setBlog(null)" is required to avoid canceling of removal due to having CascadeType.PERSIST on blog-to-post relation defined for Blog entity.

47. Implement method modifyBlogPost(BlogPostDTO)::void

```
public void modifyBlogPost(BlogPostDTO post) {
    final BlogPost persistedBlogPost;
    if (post.getId() > 0) {
        persistedBlogPost = em.find(BlogPost.class, post.getId());
    } else {
        final Blog persistedBlog = em.find(Blog.class, post.getBlogId());
        persistedBlogPost = new BlogPost();
        persistedBlogPost.setBlog(persistedBlog);
        persistedBlogPost.setDate(new Date());
    }

    persistedBlogPost.setTitle(post.getTitle());
    persistedBlogPost.setText(post.getText());

    em.persist(persistedBlogPost);
}
```

48. Implement method findBlogPosts(SearchCriteriaDTO)::List<BlogPostDTO>

```
public List<BlogPostDTO> findBlogPosts(final SearchCriteriaDTO criteria) {
    // build query
    CriteriaBuilder cb = em.getCriteriaBuilder();

    CriteriaQuery<BlogPost> cq = cb.createQuery(BlogPost.class);
    Root<BlogPost> bp = cq.from(BlogPost.class);

    Expression<Boolean> where = cb.equal(bp.<Integer>get("blog").get("id"),
    cb.parameter(Integer.class, "blogid"));
    if (StringUtils.hasText(criteria.getText())) {
        Expression<Boolean> expression = cb.like(bp.<String>get("text"),
        cb.parameter(String.class, "text"));
        where = cb.and(where, expression);
    }
    if (StringUtils.hasText(criteria.getFromDate())) {
        Expression<Boolean> expression =
        cb.greaterThanOrEqualTo(bp.<Date>get("date"),
        cb.parameter(Date.class, "from"));
        where = cb.and(where, expression);
    }
    if (StringUtils.hasText(criteria.getTillDate())) {
        Expression<Boolean> expression =
        cb.lessThanOrEqualTo(bp.<Date>get("date"), cb.parameter(Date.class,
        "till"));
        where = cb.and(where, expression);
    }
    cq.where(where);

    // execute query
    final TypedQuery<BlogPost> query = em.createQuery(cq);
    query.setParameter("blogid", criteria.getBlogId());
    if (StringUtils.hasText(criteria.getText())) {
        query.setParameter("text", criteria.getText());
    }
    if (StringUtils.hasText(criteria.getFromDate())) {
```

```
            query.setParameter("from",
            BlogUtils.convertStringToDate(criteria.getFromDate()));
        }
        if (StringUtils.hasText(criteria.getTillDate())) {
            query.setParameter("till",
            BlogUtils.convertStringToDate(criteria.getTillDate()));
        }

        // prepare result
        final List<BlogPost> posts = query.getResultList();

        final List<BlogPostDTO> result=new ArrayList<BlogPostDTO>(posts.size());
        for (final BlogPost post : posts) {
            result.add(new BlogPostDTO(post));
        }
        return result;
    }
```

Here we use Criteria API for dynamic building of JPQL query basing on search criteria.

Learning Spring MVC Controllers

49. Open class edu.jpa.blog.web.controller.BlogController. This class calls the appropriate methods of BlogService object in response to user's actions, and forms the response to user.
50. Open class edu.jpa.blog.web.controller.BlogPostController. This class calls the appropriate methods of BlogPostService object in response to user's actions, and forms the response to user.

Spring Container Configuration

51. Open the file application-context.xml
52. Define SQL DataSource to the database we use for data storing (here we use HSQL):

```
<jdbc:embedded-database id="dataSource" type="HSQL"/>

<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:db/schema.sql"/>
</jdbc:initialize-database>
```

53. Define LocalContainerEntityManagerFactoryBean (Hibernate JPA EntitManager entry point):

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="persistenceUnits.Blog" />
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaProperties">
        <props>
            <prop
        key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
            <prop key="hibernate.show_sql">false</prop>
            <prop key="hibernate.format_sql">true</prop>
        </props>
    </property>
    <property name="jpaDialect">
        <bean
        class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
    </property>
</bean>
```

54. Define transaction manager to use (neede for declarative transaction management):

```
<bean id="txManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="dataSource" ref="dataSource"/>
</bean>

<tx:annotation-driven transaction-manager="txManager"/>
```

55. Initialize service objects:

```
<context:component-scan base-package="edu.jpa.blog.service"/>
```

Persistence Unit configuration

56. Open file META-INF/persistence.xml
57. Create element <persistence-unit/> with name “persistenceUnits.Blog”
58. Define JPA org.hibernate.jpa.HibernatePersistenceProvider using <provider> element
59. Specify entity classes using <class> element
60. Here is a what you should finally have:

```
<persistence-unit name="persistenceUnits.Blog">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>edu.jpa.blog.domain.Blog</class>
    <class>edu.jpa.blog.domain.BlogPost</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
</persistence-unit>
```

Running

61. Run the application and make sure that it works correctly.
62. Analyze the application source code. Discuss with trainer the questions you have.