Advanced Java

Agenda

- Code first vs Database first approach
- Transaction Management
- Hibernate
  - Configuration
  - Bootstrapping
  - Architecture

Code First vs Database First

- Two approaches for entity (POJO) creation and tables (RDBMS) creation.
  - Database first approach (Reverse engg)
  - Code first approach (Forward engg)

Database first approach

- step 1. Design database and create tables with appropriate relations.
- step 2. Implement entity classes with appropriate ORM annotations
- Entity classes can be implemented in one of the following ways.
  - Manual approach
  - Wizards/Tools (e.g. Hibernate Rev Engg Wizard or JPA Project)

Code first approach

- step 1. Implement entity classes with appropriate ORM annotations.
- step 2. In Hibernate/JPA configuration set appropriate setting (one of following). Database tables (with corresponding relations) will be automatically created when prorgram is executed.

```
# Hibernate project (in hibernate.cfg.xml or SessionFactory java config)
hibernate.hbm2ddl.auto=create
```

- This setting can take one of the following values.
  - create: RDBMS tables will be dropped (if exists) and then tables will be created.
  - create-drop: RDBMS tables will be dropped (if exists) and then tables will be created. All created tables will be dropped, while application is terminated.
  - update: Update the table schema (add columns if required).
  - validate: Validate the table structure. If not compatible, application is terminated by an exception.
  - none: No action will be performed. (default)
- In Spring Boot, SQL file can be execute after table creation to populate initial data. The default file name is import.sql.

```
spring.jpa.properties.hibernate.hbm2ddl.import_files=import.sql
```

Choosing right approach

- If the database already exists, or is meant to be shared with other applications, you have to create entities adhering to database. Be careful while doing changes in database. You need to update entities accordingly.
- If there are external constraints on the database structure like you must use stored procedures, or use certain DB naming conventions, or you use a tool for schema migration during updates (that imposes constraints). Here you need full control of the DB structure, so database first approach is recommended.
- If your application "owns" the database, then treat the database as an implementation detail, and let JPA create it. This is useful for testing and some relatively small applications (may be micro-services). Note that application entity model may change and will reflect changes in database.

Spring Transaction Management

- A @Transactional method calling another @Transactional method is like nested transaction.

```
@Repository
public class InventoryDao {
    @Transactional
    public void reserveInventory(int prodId, int quantity) { ... }
    // ...
}

@Repository
public class PaymentDao {
    @Transactional
    public void doPayment(String orderId, double amount) { ... }
    // ...
}

@Repository
public class OrderDao {
    @Transactional //(propogation=REQUIRED)
    public void newOrder(int custId, int prodId, int quantity) { ... }
    // ...
}
@Service
public class OrderService {
    @Autowired
    private OrderDao orderDao;
    @Autowired
    private PaymentDao paymentDao;
    @Autowired
    private InventoryDao inventoryDao;
    @Transactional
    public void placeOrder() {
        // ...
        orderDao.newOrder(...);
        inventoryDao.reserveInventory(...);
        paymentDao.doPayment(...);
    }
}
```

- The behaviour of "nested transaction" is defined by @Transactional(propogation=XXXX).
  - REQUIRED: Support a current transaction, create a new one if none exists.
  - SUPPORTS: Support a current transaction, execute non-transactionally if none exists.
  - REQUIRES_NEW: Create a new transaction, and suspend the current transaction if one exists.
  - MANDATORY: Support a current transaction, throw an exception if none exists.
  - NOT_SUPPORTED: Execute non-transactionally, suspend the current transaction if one exists.
  - NEVER: Execute non-transactionally, throw an exception if a transaction exists.
  - NESTED: Execute within a nested transaction if a current transaction exists (i.e. savepoint), behave like REQUIRED otherwise.

Hibernate

Hibernate Architecture

SessionFactory

- One SessionFactory per application (per db).
- Heavy-weight. Not recommended to create multiple instances in single application.
- Thread-safe. Can be accessed from multiple threads (synchronization is built-in).
- Typical practice is to create singleton utility class for that.

Session

- Created by SessionFactory & it encapsulate JDBC connection.
- All hibernate (CRUD) operations are done on hibernate session.
- Not thread-safe. Should not access same session from multiple threads.
- Light-weight. Can be created and destroyed as per need.

Transaction

- In hibernate, autocommit is false by default.
- DML operations should be performed using transaction.
- tx = session.beginTransaction(): to start new transaction.
- tx.commit(): to commit transaction.

- tx.rollback(): to rollback transaction.

Session Flush

- session.flush(): Forcibly synchronize in-memory state of hibernate session with database.
- Each tx.commit() automatically flush the state of session.
- Manually calling flush() will result in executing appropriate SQL queries into database.
- Note that flush() will not commit the data into the RDBMS tables.
- The flush mode can be set using session.setHibernateFlushMode(mode).
    - ALWAYS, AUTO, COMMIT, MANUAL

Dialect

- RDBMS have specific features like data types, stored procedures, primary key generation, etc.
- Hibernate support all RDBMS.
- Most of code base of Hibernate is common.
- Database level changes are to be handled specifically and appropriate queries should be generated. This is done by Dialect.
- Hibernate have dialects for all popular RDBMS (org.hibernate.dialect.*). Programmer should configure appropriate dialect to utilize full features of RDBMS.

Query and NativeQuery

- Query object represents HQL queries.
- NativeQuery object represents SQL queries.

Hibernate CRUD methods

- get()/find() -- select/retrieve entity from db for given id.
- persist() -- insert new entity in db.
- update() -- update existing entity in db.
- delete() -- delete entity from db

Hibernate CRUD and using hibernate.cfg.xml

- step 1: Create new "Maven Project" with hibernate-core and MySQL dependencies.

```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.14.Final</version>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.4.0</version>
</dependency>
```

- step 2: Create entity class with appropriate ORM annotations.

```java
@Entity
@Table(name="users")
public class User {
    @Id
    @Column
    private int id;
    // ...
}
```

- step 3: Create hibernate.cfg.xml under resources directory.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
```

```
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/dmcdb</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.show_sql">true</property>
        <mapping class="com.sunbeam.User"/>
    </session-factory>
</hibernate-configuration>
```

- step 4: Create HbUtil class to build SessionFactory.

```
public class HbUtil {
    private static SessionFactory sessionFactory = createSessionFactory();
    private static ServiceRegistry serviceRegistry;

    public static SessionFactory createSessionFactory() {
        serviceRegistry = new StandardServiceRegistryBuilder()
                .configure() // read hibernate.cfg.xml
                .build();

        Metadata metadata = new MetadataSources(serviceRegistry)
                .buildMetadata();

        return metadata.getSessionFactoryBuilder().build();
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        sessionFactory.close();
```

```
            serviceRegistry.close();
        }
    }
```

- step 5: Implement DAO class. Note use of transactions for DML operations.

```
public class UserDaoImpl implements AutoCloseable {
    private Session session;

    public UserDaoImpl() {
        SessionFactory factory = HbUtil.getSessionFactory();
        this.session = factory.openSession();
    }

    public void close() {
        if(session != null)
            session.close();
    }

    public User findById(int id) {
        User c = session.get(User.class, id);
        return c;
    }
    // ...
}
```

- step 6: In main class, test the methods.

```
try (UserDaoImpl dao = new UserDaoImpl()) {
    User c = dao.findById(1);
    // ...
} catch (Exception e) {
    e.printStackTrace();
}
```

Hibernate 3 Bootstrapping

- Bootstrapping refers to the process of building and initializing a SessionFactory.
- SessionFactory is typically initialized with following properties.
  - hibernate.connection.driver_class
  - hibernate.connection.url
  - hibernate.connection.username
  - hibernate.connection.password
  - hibernate.dialect
  - hibernate.show_sql
- SessionFactory can be configured using hibernate.cfg.xml or Java code to initialize Configuration object.
- Hibernate3 Bootstrapping

```
// step 1: Create Configuration object.
Configuration cfg = new Configuration();
// step 2: Read hibernate.cfg.xml file using its configure() method.
cfg.configure();
// step 3: Create SessionFactory using its buildSessionFactory() method.
SessionFactory factory = cfg.buildSessionFactory();
```