**Objectives**

Explain the need and benefit of ORM

ORM (Object-Relational Mapping), makes it easier to develop code that interacts with database, abstracts the database system, transactionality

* ORM Pros and Cons - https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a
* What is ORM? - <https://en.wikipedia.org/wiki/Object-relational_mapping>

Answer:

ORM (Object-Relational Mapping) is a programming methodology that simplifies database interaction for developers by utilizing objects in their code. Rather than crafting intricate SQL statements, ORM enables us to use Java classes (or objects from any programming language) that are automatically associated with database tables.

In software development, our focus is primarily on objects, while databases organize data in rows and columns. Without ORM, developers would need to:

* Compose elaborate SQL queries for each task.
* Manually transform database records into Java objects.

ORM addresses this challenge by automating interactions with the database and the mapping between database tables and program objects.

Advantages:

* Minimizes Coding Work: ORM takes care of most database tasks automatically.
* Streamlines CRUD Tasks: You can execute insert, update, delete, and read actions without needing to write SQL statements.
* Database Agnosticism: ORM facilitates easier transitions between various database systems.
* Accelerates Development: Developers can concentrate more on business functionalities rather than database management.
* Enhances Security: When used appropriately, ORM helps mitigate the risk of SQL injection.

Example:

Without ORM (SQL query):

SELECT \* FROM books WHERE id = 1;

With ORM (Java code):

Book book = bookRepository.findById(1L);

The SQL is automatically generated by the ORM for you.

Drawbacks of ORM:

* May struggle with very intricate queries.
* Improper configuration can sometimes hinder performance.

Conclusion:

ORM streamlines our interaction with databases by enabling us to utilize objects within code rather than relying on SQL queries. It enhances the speed of development, boosts the clarity of the code, and facilitates easier application maintenance.

Demonstrate the need and benefit of Spring Data JPA

Evolution of ORM solutions, Hibernate XML Configuration, Hibernate Annotation Configuration, Spring Data JPA, Hibernate benefits, open source, light weight, database independent query

* With H2 in memory database - https://www.mkyong.com/spring-boot/spring-boot-spring-data-jpa/
* With MySQL - https://www.mkyong.com/spring-boot/spring-boot-spring-data-jpa-mysql-example/
* XML Configuration Example -https://www.tutorialspoint.com/hibernate/hibernate\_examples.htm
* Hibernate Configuration Example -https://www.tutorialspoint.com/hibernate/hibernate\_annotations.htm

 Over the years, ORM solutions have progressed to streamline database programming and enhance efficiency. Initially, ORM frameworks such as Hibernate required extensive XML configuration files to link Java classes with database tables. Developers were required to manually create XML files to specify every table, column, and relationship, which was a tedious and error-prone task.

As ORM technology progressed, Hibernate introduced configuration using annotations. This innovation allowed developers to utilize straightforward Java annotations like @Entity, @Table, and @Id directly within their code, reducing the reliance on extensive external configurations and improving the readability and maintainability of the code. While annotation-based Hibernate enhanced the developer experience, there remained a necessity to write repository classes filled with repetitive CRUD methods.

To further simplify the ORM process, Spring Data JPA was developed. It builds on JPA and Hibernate but eliminates the need to write most repetitive data access code. Spring Data JPA provides a predefined repository structure that automatically manages basic CRUD operations. By merely extending the JpaRepository interface, developers gain access to methods such as save, findById, findAll, and deleteById without needing to write additional code. This functionality makes Spring Data JPA extremely valuable for rapidly developing modern applications.

Spring Data JPA is open-source, lightweight, and easy to integrate with Spring Boot. It supports both in-memory databases like H2 and external databases such as MySQL, enabling developers to switch between different database systems with minimal effort. Using the H2 in-memory database, developers can run applications and tests without requiring an actual database server, significantly accelerating development and testing. When working with databases like MySQL, Spring Data JPA automatically takes care of connection management and schema mapping.

A significant advantage of Spring Data JPA is its capability to generate database queries by simply adhering to naming conventions. For instance, if a method named findByName is created, Spring Data JPA will automatically create the SQL query in the background. This feature simplifies the process of writing complex queries and greatly cuts down development time. Spring Data JPA also accommodates JPQL (Java Persistence Query Language) and native SQL for developers seeking finer control over queries.

Another notable benefit of Spring Data JPA is its complete independence from any specific database. Applications can effortlessly transition from one database to another, such as from MySQL to PostgreSQL, with minimal modifications to the code. This characteristic enhances the application's flexibility and portability.

In comparison to the older XML-based Hibernate configurations, Spring Data JPA is significantly quicker to establish and requires fewer files. Even when contrasted with Hibernate utilizing annotations, Spring Data JPA provides a superior abstraction and decreases manual effort. It automates the oversight of entities, sessions, and transactions, enabling developers to concentrate more on business logic rather than data persistence code.

To sum up, Spring Data JPA streamlines the whole ORM process. It removes boilerplate code, accelerates development, accommodates both in-memory and external databases, and facilitates easy database transitions. Its integration with Spring Boot positions it as the ideal choice for contemporary, scalable applications needing rapid and efficient database access.

Explain about core objects of hibernate framework

Factory, Session, Transaction Factory, Transaction, Connection Provider

* Hibernate Architecture Reference - https://www.tutorialspoint.com/hibernate/hibernate\_architecture.htm

The Hibernate framework revolves around multiple fundamental objects that collaborate to effectively manage database functions. A key element is the SessionFactory. The SessionFactory is a substantial object that handles the creation and management of various sessions. It is generated one time at the beginning of the application and is utilized repeatedly during the application's lifespan. The SessionFactory interprets the Hibernate configuration files and establishes the necessary connection setup for database interaction. It is created to ensure thread safety and is usually implemented as a singleton in applications.   
  
The Session object is a simple, temporary entity that symbolizes one unit of work with the database. Every session serves to create, read, modify, or remove database entries. The Session object serves as the primary interface between the Java application and the database. It offers techniques for storing entities, retrieving data from the database, and handling transactions. In contrast to the SessionFactory, a Session is not safe for threads and must not be used concurrently by several threads.   
  
The TransactionFactory handles the creation of Transaction objects. It operates in the background of most applications and guarantees that transaction management is correctly configured according to the settings. Hibernate accommodates various transaction strategies, and the TransactionFactory provides an abstraction for the underlying transaction system in use.   
  
The Transaction object in Hibernate signifies a single task that needs to be finished completely or not at all. It guarantees that database functions adhere to the ACID (Atomicity, Consistency, Isolation, Durability) principles. The Transaction object offers methods to initiate, finalize, or revert a transaction. Transactions play a crucial role in preserving data integrity, particularly when executing several interconnected database operations that ought to either all succeed or all fail.

The Connection Provider is an essential element that provides database connections to Hibernate. It encapsulates the connection specifics and handles the initiation and termination of database connections. Hibernate generally employs a connection pool to enhance performance, and the Connection Provider manages the integration with these connection pools.

In conclusion, the primary components of Hibernate like SessionFactory, Session, TransactionFactory, Transaction, and Connection Provider collaborate to facilitate database access, oversee transactions, and guarantee effective connection management. These elements constitute the base of the Hibernate architecture and deliver a robust, dependable framework for ORM in Java applications

Explain ORM implementation with Hibernate XML Configuration and Annotation Configuration

XML Configuration - persistence class, mapping xml, configuration xml, loading hibernate configuration xml file; Annotation Configuration - persistence class, @Entity, @Table, @Id, @Column, hibernate configuration xml file Loading hibernate configuration and interacting with database get the session factory, open session, begin transaction, commit transaction, close session

* XML Configuration Example - https://www.tutorialspoint.com/hibernate/hibernate\_examples.htm
* Hibernate Configuration Example - <https://www.tutorialspoint.com/hibernate/hibernate_annotations.htm>

Hibernate offers two main methods for implementing ORM: XML configuration and configuration through annotations. Both methods enable Java classes to link with database tables, but they vary in the manner in which the mapping data is supplied.   
  
In Hibernate XML configuration, developers need to specify a persistence class, a mapping XML file, and a configuration XML file for Hibernate. The persistence class is a straightforward Java class (POJO) featuring private fields, public getter and setter methods, and a constructor with no parameters. The mapping XML file is where the developer manually associates each class with its corresponding database table and defines the mappings for each field and primary key. For instance, the mapping XML file specifies the association between each class field and the corresponding database column. The XML configuration file for Hibernate is utilized to supply the database connection information, including the database URL, username, password, and Hibernate dialect. It also enumerates all the XML mapping files that Hibernate is expected to utilize. With XML configuration, the application loads the Hibernate configuration file, creates the SessionFactory, opens a Session, starts a transaction, and after executing the necessary database operations, commits the transaction and closes the session.

In Hibernate annotation setup, the persistence class holds all the mapping details directly through Java annotations. Rather than creating individual mapping XML files, developers can utilize annotations such as @Entity to indicate the class as a database entity, @Table to specify the table name, @Id to identify the primary key field, and @Column to associate class fields with database columns. This method streamlines setup since all mapping information is contained within the Java class. Configuration based on annotations still necessitates a Hibernate configuration XML file to specify database connection settings and enumerate the classes that Hibernate should map. After the configuration is loaded, the steps to obtain the SessionFactory, open a Session, start a transaction, commit the transaction, and close the Session are identical to those in XML configuration.   
  
The key distinction between these two methods is that XML configuration maintains the mapping information in separate files external to the Java classes, whereas annotation configuration contains all the mapping details within the classes through the use of annotations. Both approaches are effective, but annotation-based configuration is more contemporary and commonly utilized due to its simplicity and ease of maintenance. No matter the approach taken, the general workflow with Hibernate stays consistent: load the configuration, create the SessionFactory, open a Session, initiate a transaction, execute database operations, commit the transaction, and ultimately close the Session.

Explain the difference between Java Persistence API, Hibernate and Spring Data JPA

JPA (Java Persistence API), JPA is a specification (JSR 338), JPA does not have implementation, Hibernate is one of the implementation for JPA, Hibernate is a ORM tool, Spring Data JPA is an abstraction above Hibernate to remove boiler plate code when persisting data using Hibernate.

* Difference between Spring Data JPA and Hibernate - https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1
* Intro to JPA - https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html

Java Persistence API, or JPA, is a specification established by Java for handling relational data within Java applications. JPA does not offer any functional code or tools. It is merely a collection of standards and regulations for managing object-relational mapping in Java. JPA specifies how Java entities can link to database tables, yet it lacks its own implementation. Developers must utilize a JPA provider to execute database operations.   
  
Hibernate is among the most commonly utilized implementations of the JPA specification. Hibernate is a comprehensive ORM (Object Relational Mapping) framework that supplies the necessary code to link Java objects to database tables. Hibernate can be utilized directly without JPA, or it may serve as a JPA provider while adhering to the JPA standard. Hibernate provides numerous extra capabilities in addition to the fundamental JPA specification, including caching, lazy loading, and sophisticated query options. While employing Hibernate, developers must still manually create repository classes and numerous data access methods.   
  
Spring Data JPA is an advanced framework built on JPA and Hibernate. It adds an extra layer that streamlines database access by eliminating repetitive code. Using Spring Data JPA, developers are not required to manually implement standard CRUD operations such as save, update, delete, or find by ID. These actions are offered automatically by merely extending predefined repository interfaces. Spring Data JPA enables developers to craft custom queries by adhering to naming conventions, and it automatically produces the necessary SQL. This renders Spring Data JPA very user-friendly for developers and greatly shortens development time.

In conclusion, JPA is a specification outlining the guidelines for ORM in Java. Hibernate is a well-known ORM tool that follows the JPA specification and delivers the functional framework. Spring Data JPA is a framework layered over JPA and Hibernate that streamlines database interactions and minimizes the code developers must produce. Collectively, they offer a robust and user-friendly framework for handling relational data in Java applications.

Demonstrate implementation of DML using Spring Data JPA on a single database table

Hibernate log configuration and ddl-auto configuration, JpaRepsitory.findById(), defining Query Methods, JpaRespository.save(), JpaRepository.deleteById()

* Spring Data JPA Ref Repository methods - https://docs.spring.io/spring-data/jpa/docs/2.2.0.RELEASE/reference/html/#repositories.core-concepts
* Query methods - https://docs.spring.io/spring-data/jpa/docs/2.2.0.RELEASE/reference/html/#repositories.query-methods

Implementing Data Manipulation Language (DML) operations with Spring Data JPA on one database table is straightforward and very efficient, as Spring Data JPA offers numerous built-in methods that automatically manage most database interactions. Spring Data JPA can execute actions such as insert, update, delete, and select with little coding through the JpaRepository interface.   
  
To begin the implementation, Hibernate logging and automatic schema creation need to be correctly set up in the application.properties file. The log configuration for Hibernate can be adjusted by setting spring.jpa.show-sql=true to show SQL queries in the console, along with spring.jpa.hibernate.ddl-auto=update to automatically generate or modify the database schema according to the entity classes. The ddl-auto feature includes options such as create, update, validate, and none for managing Hibernate's approach to schema generation.   
  
In Spring Data JPA, fundamental DML tasks such as reading, adding, modifying, and removing records can be executed through the methods offered by JpaRepository. The findById() function is utilized to fetch a record using its primary key. For instance, bookRepository.findById(1L) will provide the entry that has an ID of 1. If the entry is absent, the method yields an empty Optional.   
  
The save() function offered by JpaRepository is utilized to insert or modify a record. This process will insert a new record if the ID is absent or update an existing record if the ID is already there. For instance, invoking bookRepository.save(book) will automatically decide whether to insert or update depending on the ID of the book entity.   
  
Records can be easily removed using the deleteById() method. For instance, bookRepository.deleteById(1L) will eliminate the entry with ID 1 from the database. These repository methods do not need SQL queries and are automatically created by Spring Data JPA.

Besides fundamental CRUD operations, Spring Data JPA provides query methods that utilize naming conventions for methods. For instance, creating a method in the repository interface called findByTitle(String title) will automatically produce the appropriate SQL to look up books based on their title. The developer is not required to manually write the SQL. More intricate queries can be constructed utilizing keywords such as findBy, findAllBy, deleteBy, countBy, and others to automatically produce suitable queries.   
  
Spring Data JPA supports creating custom queries with the @Query annotation when there's a need for greater control. This enables developers to create JPQL or native SQL queries for intricate situations while continuing to utilize the repository interface.   
  
In general, Spring Data JPA offers a straightforward, robust, and effective approach to execute DML operations on database tables through built-in repository functions such as findById(), save(), and deleteById(). It minimizes the boilerplate code, accelerates development, and automatically manages the majority of the database interaction processes.

**Hands on 2**

**Hibernate XML Config implementation walk through**   
  
SME to provide explanation on the sample Hibernate implementation available in the link below:  
https://www.tutorialspoint.com/hibernate/hibernate\_examples.htm  
  
Explanation Topics

* Explain how object to relational database mapping done in hibernate xml configuration file
* Explain about following aspects of implementing the end to end operations in Hibernate:
  + SessionFactory
  + Session
  + Transaction
  + beginTransaction()
  + commit()
  + rollback()
  + session.save()
  + session.createQuery().list()
  + session.get()
  + session.delete()

1. SessionFactory

The SessionFactory is a heavyweight, thread-safe object that is created once per application.It is configured using the hibernate.cfg.xml file which contains database connection details and mapping file references.

Hibernate uses SessionFactory to create Session objects.

Example:

SessionFactory factory = new Configuration().configure().buildSessionFactory();

2. Session

The Session object is lightweight and is used to interact with the database.Each Session represents a single unit of work and is not thread-safe.Sessions are opened from the SessionFactory.

Example:

Session session = factory.openSession();

3. Transaction

A Transaction in Hibernate represents a single unit of work that should either completely succeed or completely fail.Hibernate transactions help maintain data consistency and rollback on failure.

4. beginTransaction()

The beginTransaction() method is used to start a new transaction.It ensures that all operations performed are treated as part of the same transaction block.

Example:

Transaction tx = session.beginTransaction();

5. commit()

The commit() method is used to save all changes permanently to the database.Once committed, the transaction is closed successfully.

Example: tx.commit();

6. rollback()

The rollback() method is used to undo all changes if any exception occurs or if the transaction fails.It is useful for handling errors safely.

Example:

tx.rollback();

7. session.save()

The save() method is used to insert a new record into the database.Hibernate converts the object into an SQL INSERT query.

Example:

session.save(employee);

8. session.createQuery().list()

The createQuery() method is used to write HQL (Hibernate Query Language) queries to fetch data.The list() method executes the query and returns the results as a list.

Example:

List employees = session.createQuery("FROM Employee").list();

9. session.get()

The get() method is used to fetch a single record based on its primary key.It returns null if the record is not found.

Example:

Employee emp = session.get(Employee.class, 1);

10. session.delete()

The delete() method is used to delete an object from the database.

Example:

session.delete(employee);

**Hands on 3**

**Hibernate Annotation Config implementation walk through**   
  
SME to provide explanation on the sample Hibernate implementation available in the link below:  
https://www.tutorialspoint.com/hibernate/hibernate\_annotations.htm  
  
Explanation Topics

* Explain how object to relational database mapping done in persistence class file Employee
* Explain about following aspects of implementing the end to end operations in Hibernate:
  + @Entity
  + @Table
  + @Id
  + @GeneratedValue
  + @Column
  + Hibernate Configuration (hibernate.cfg.xml)
    - Dialect
    - Driver
    - Connection URL
    - Username
    - Password

**1. @Entity**

* Marks the class as a Hibernate Entity.
* Tells Hibernate to map this Java class to a database table.
* Without @Entity, Hibernate will ignore the class.

Example:

@Entity

public class Employee { ... }

**2. @Table**

* Maps the class to a specific database table.
* If not provided, Hibernate assumes the table name is the same as the class name.

Example:

@Table(name = "EMPLOYEE")

**3. @Id**

* Marks the field as the Primary Key of the table.

Example:

@Id

private int id;

**4. @GeneratedValue**

* Automatically generates the primary key values.
* Hibernate will auto-increment the primary key when inserting new records.

Example:

@GeneratedValue

private int id;

**5. @Column**

* Maps a Java class field to a specific table column.
* The name attribute specifies the column name in the table.

Example:

@Column(name = "FIRST\_NAME")

private String firstName;

The hibernate.cfg.xml file is used to configure the Hibernate framework and to provide the database connection details. It also includes Hibernate-specific settings and mappings to the entity classes.

* **Dialect:**  
  Specifies which database is being used. Hibernate uses this to generate the correct SQL queries for that database.

Example:

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

* **Driver:**  
  Defines the JDBC driver that allows the Java application to connect to the database.

Example:

<propertyname="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

* **Connection URL:**  
  Provides the location and name of the database. It tells Hibernate where the database server is hosted.

Example:

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernate\_db</property>

* **Username and Password:**  
  These are the credentials used to authenticate and connect to the database.

Example:

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">2005</property>

**Hands on 4**

**Difference between JPA, Hibernate and Spring Data JPA**   
  
Java Persistence API (JPA)

* JSR 338 Specification for persisting, reading and managing data from Java objects
* Does not contain concrete implementation of the specification
* Hibernate is one of the implementation of JPA

Hibernate

* ORM Tool that implements JPA

Spring Data JPA

* Does not have JPA implementation, but reduces boiler plate code
* This is another level of abstraction over JPA implementation provider like Hibernate
* Manages transactions

**Refer code snippets below on how the code compares between Hibernate and Spring Data JPA  
Hibernate**

   /\* Method to CREATE an employee in the database \*/

   public Integer addEmployee(Employee employee){

      Session session = factory.openSession();

      Transaction tx = null;

      Integer employeeID = null;

      try {

         tx = session.beginTransaction();

         employeeID = (Integer) session.save(employee);

         tx.commit();

      } catch (HibernateException e) {

         if (tx != null) tx.rollback();

         e.printStackTrace();

      } finally {

         session.close();

      }

      return employeeID;

   }

**Spring Data JPA**  
EmployeeRespository.java

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

EmployeeService.java

@Autowire

  private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

  employeeRepository.save(employee);

  }

​​​​​​​   
  
**Reference Links:**   
<https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1>   
<https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>

**Java Persistence API (JPA)**

• JPA is a Java standard (JSR 338) utilized for storing, retrieving, and handling data between Java objects and relational databases.   
  
• JPA solely establishes the guidelines and techniques for data storage. It does not offer any functional code or application.   
  
• It serves as a common interface that can be adopted by various providers.   
  
• Hibernate ranks among the most commonly utilized implementations of JPA.

**Hibernate**

• Hibernate is a specific ORM (Object Relational Mapping) tool that adheres to the JPA specification.   
  
• It supplies the functional code to associate Java objects with database tables.   
  
• Hibernate can function as both a JPA implementation provider and a standalone ORM framework.   
  
• When utilizing Hibernate directly, developers must handle sessions, transactions, and exception management.

**Spring Data JPA**

• Spring Data JPA does not serve as a JPA implementation. It represents a more advanced abstraction built on JPA.   
  
• It aids in minimizing the repetitive code needed for executing database tasks.   
  
• Spring Data JPA automatically offers standard CRUD operations like save, delete, findById, and findAll without the need for explicit SQL or HQL queries.   
  
• It utilizes a JPA provider such as Hibernate for managing the actual data persistence.   
  
• Spring Data JPA automatically handles transactions through annotations such as @Transactional.   
  
• It enables developers to concentrate more on the business logic and less on the code for database access.

**Code Comparison: Hibernate vs. Spring Data JPA**

**🔹 Hibernate Example**

public Integer addEmployee(Employee employee) {

Session session = factory.openSession();

Transaction tx = null;

Integer employeeID = null;

try {

tx = session.beginTransaction();

employeeID = (Integer) session.save(employee);

tx.commit();

} catch (HibernateException e) {

if (tx != null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

}

return employeeID;

}

**Explanation:**

* Developer must open and close the session manually.
* Transaction handling, rollback, and exception management must be written explicitly.

**🔹 Spring Data JPA Example**

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

}

**Explanation:**

* The repository interface automatically provides CRUD operations without manual SQL.
* No session or transaction handling is required in the method body.
* Spring handles transactions using the @Transactional annotation.