Lecture 02&03: Hibernate Introduction

What is Hibernate?

- **Definition**: Hibernate is an Object-Relational Mapping (ORM) framework for Java
- **Purpose**: To increase developer productivity by simplifying database operations
- **Core Concept**: Java is object-oriented programming, so we should work with objects rather than SQL statements directly

Problem Without Hibernate

When working without Hibernate, developers must:

- Write complex SQL queries manually
- Handle database connections explicitly
- Manage data type conversions between Java objects and database tables
- Write boilerplate code for CRUD operations

Example of a manual approach:

```
String sql = "INSERT INTO student (rollNo, name, age) VALUES (?, ?, ?)";
PreparedStatement pstmt = connection.prepareStatement(sql);
pstmt.setInt(1, student.getRollNo());
pstmt.setString(2, student.getName());
pstmt.setInt(3, student.getAge());

pstmt.executeUpdate();
```

ORM Mapping Concept

- **Object**: Java classes and their instances
- **Relational**: Database tables and rows
- **Mapping**: Automatic conversion between objects and database records

The ORM mapping shows how a Student class maps to a student table:

- Class fields → Table columns
- Object instances → Table rows
- Java data types → SQL data types

Lecture 04: Project Setup

Creating a New Hibernate Project

1. **IDE Setup**: Use IntelliJ IDEA or similar IDE
2. **Project Type**: Maven project for dependency management
3. **Java Version**: Use Java 8 or higher
4. **Build System**: Maven for managing dependencies

Adding Dependencies

Add the following dependencies to your `pom.xml`:

```
<dependencies>
    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.3.Final</version>
    </dependency>

    <!-- PostgreSQL Driver -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.3</version>
    </dependency>
</dependencies>
```

Key Dependencies Explained

- **hibernate-core**: Main Hibernate functionality
- **postgresql**: Database driver for PostgreSQL connectivity

Lecture 05: Failed Attempt to Save Data

Creating a Student POJO Class

```
public class Student {
    private int rollNo;
    private String name;
    private int age;

    // Constructors
    public Student() {}

    // Getters and Setters
    public int getRollNo() { return rollNo; }
    public void setRollNo(int rollNo) { this.rollNo = rollNo; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

Initial Save Attempt (Failed)

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Navin");
        s1.setRollNo(101);
        s1.setAge(30);

        Configuration cfg = new Configuration();
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        session.save(s1);  // This will fail

        System.out.println(s1);
    }
}
```

Why This Failed

- No Hibernate configuration file
- Entity class not properly annotated

- No mapping between Java class and database table
- Missing transaction management

Lecture 06: Successful Attempt to Save Data

Important Version Note

Save() Method Deprecation:

- `save()` method was **removed in Hibernate 7.1.0**
- `save()` method was **deprecated in version 6.6.3**
- Use `persist()` method instead for newer versions

Creating Hibernate Configuration File

Create `hibernate.cfg.xml` in the `src/main/resources` directory:

```
<!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>
        <!-- Database connection properties -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/telusko</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">0000</property>

        <!-- Hibernate properties -->
        <property name="hibernate.hbm2ddl.auto">update</property>

    </session-factory>
</hibernate-configuration>
```

Updated Student Entity with Annotations

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Student {
    @Id
    private int rollNo;
    private String name;
    private int age;

    // Constructors, getters, and setters remain the same
```

```
}
```

Successful Save Implementation

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Navin");
        s1.setRollNo(101);
        s1.setAge(30);

        Configuration cfg = new Configuration();
        cfg.addAnnotatedClass(com.telusko.Student.class);
        cfg.configure();

        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        Transaction transaction = session.beginTransaction();

        session.save(s1);  // or use session.persist(s1) for newer versions

        transaction.commit();

        System.out.println(s1);
    }
}
```

Key Success Factors

1. **Configuration file**: Properly configured database connection
2. **Entity annotations**: `@Entity` and `@Id` annotations
3. **Class registration**: Adding annotated class to configuration
4. **Transaction management**: Beginning and committing transactions

Lecture 07: Show SQL Configuration

Adding SQL Visibility Properties

Update `hibernate.cfg.xml` to show generated SQL:

```
<property name="hibernate.hbm2ddl.auto">update</property>
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
```

Configuration Properties Explained

- **hibernate.hbm2ddl.auto**:
  - `update`: Creates table if it does not exists, updates if schema changes
  - `create`: Drops and creates table every time
  - `validate`: Only validates the schema
  - `none`: No automatic schema management
- **hibernate.show_sql**: Displays generated SQL in console
- **hibernate.format_sql**: Formats SQL for better readability

Additional Useful Properties

```
<property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
```

Lecture 08: Refactoring – Using Persist Method

Updated Code with Persist Method

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Gaurav");
        s1.setRollNo(105);
        s1.setAge(22);

        Configuration cfg = new Configuration();
        cfg.addAnnotatedClass(com.telusko.Student.class);
        cfg.configure();

        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        Transaction transaction = session.beginTransaction();

        session.persist(s1);  // Using persist instead of save

        transaction.commit();

        session.close();
        sf.close();

        System.out.println(s1);
    }
}
```

Persist vs Save Method

- **persist()**: JPA standard method, preferred for new code
- **save()**: Hibernate-specific, deprecated in newer versions
- **Functionality**: Both methods save transient objects to the database.
- **Return value**: save() returns generated ID; persist() doesn't

Best Practices

1. Always use transactions for data modifications
2. Close session and session factory properly
3. Use persist() for better portability
4. Handle exceptions appropriately

Lecture 09: Fetching Data

Reading Data from Database

```
public static void main(String[] args) {
    Student s1 = new Student();
    s1.setName("Arya");
    s1.setRollNo(106);
    s1.setAge(21);

    Student s2 = null;

    SessionFactory sf = new Configuration()
            .addAnnotatedClass(com.telusko.Student.class)
            .configure()
            .buildSessionFactory();

    Session session = sf.openSession();

    s2 = session.get(Student.class, 102);  // Fetch student with rollNo 102

    session.close();
    sf.close();

    System.out.println(s2);
}
```

Get vs Load Methods

- **get()**: Returns null if object not found, eager loading
- **load()**: Throws exception if not found, lazy loading
- **Usage**: get() is more commonly used and safer

Lecture 10: Update and Delete Operations

Update Operation

```
public static void main(String[] args) {
    Student s1 = new Student();
    s1.setName("Harsh");
    s1.setRollNo(103);
    s1.setAge(25);

    SessionFactory sf = new Configuration()
            .addAnnotatedClass(com.telusko.Student.class)
            .configure()
            .buildSessionFactory();

    Session session = sf.openSession();

    session.merge(s1);  // Updates existing record or inserts new one

    session.close();
    sf.close();

    System.out.println(s1);
}
```

Delete Operation

```
public static void main(String[] args) {
    SessionFactory sf = new Configuration()
            .addAnnotatedClass(com.telusko.Student.class)
            .configure()
            .buildSessionFactory();

    Session session = sf.openSession();

    Student s1 = session.get(Student.class, 109);  // First fetch the object

    Transaction transaction = session.beginTransaction();

    session.remove(s1);  // Delete the object

    transaction.commit();

    session.close();
    sf.close();
}
```

Important Version Notes

- **get() method**: Deprecated in Hibernate 7.1.0 but **not removed**
- **remove()**: Use instead of delete() for JPA compliance
- **merge()**: Use instead of update() or saveOrUpdate()

Lecture 11: Changing Table and Column Names

Custom Table Name

```
@Entity
@Table(name="alien_table")
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;

    // getters and setters
}
```

Custom Column Names

```
@Entity
public class Alien {
    @Id
    private int aid;

    @Column(name="alien_name")
    private String aname;

    @Transient
    private String tech;

    // getters and setters
}
```

Annotation Explanations

- **@Entity**: Marks class as JPA entity
- **@Table(name="custom_name")**: Specifies custom table name
- **@Id**: Marks primary key field
- **@Transient**: Ignore the field in data base
- **@Column(name="custom_name")**: Specifies custom column name

Entity vs Table Differences

- **Entity**: Java class representation
- **Table**: Database table representation
- **Mapping**: Annotations bridge the gap between object and relational models
- **Flexibility**: Can have different names for class/field vs table/column

Lecture 12: Embeddables - Complex Data Types

Creating an Embeddable Class

```java
@Embeddable
public class Laptop {
    private String brand;
    private String model;
    private int ram;

    // Constructors
    public Laptop() {}

    // Getters and Setters
    public String getBrand() { return brand; }
    public void setBrand(String brand) { this.brand = brand; }

    public String getModel() { return model; }
    public void setModel(String model) { this.model = model; }

    public int getRam() { return ram; }
    public void setRam(int ram) { this.ram = ram; }
}
```

Using Embeddable in Entity

```java
@Entity
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;
    private Laptop laptop;  // Embedded object

    // Constructors, getters, and setters
}
```

Key Concepts

- **@Embeddable**: Marks a class as embeddable (value type)
- **Composition**: Alien "has-a" Laptop relationship
- **Table Structure**: Laptop fields become columns in Alien table
- **No Separate Table**: Embeddable objects don't get their own table

Benefits of Embeddables

1. **Code Organization**: Group related fields together
2. **Reusability**: Same embeddable can be used in multiple entities
3. **Type Safety**: Better than using primitive types for complex data
4. **Maintenance**: Changes to embeddable affect all using entities

Summary of Key Hibernate Concepts

Core Annotations

- `@Entity`: Marks a class as a JPA entity
- `@Id`: Designates the primary key
- `@Table`: Specifies table name and properties
- `@Column`: Specifies column properties
- `@Embeddable`: Marks a class as embeddable

Session Methods (Modern Approach)

- `persist()`: Save new entity
- `merge()`: Update existing or save new
- `get()`: Retrieve by primary key
- `remove()`: Delete entity

Configuration Essentials

1. Database connection properties
2. Hibernate-specific properties
3. Entity class registration
4. Transaction management

Best Practices

1. Always use transactions for modifications
2. Close resources properly
3. Use JPA standard methods when possible
4. Handle exceptions appropriately
5. Configure logging for development

Lecture 13: Mapping Relationship Tables

Theory

Database relationships define how entities are connected to each other. In Hibernate, these relationships are mapped using annotations to maintain referential integrity and enable efficient data retrieval.

Types of Relationships

1. One-to-One Relationship

Definition: Each record in one table corresponds to exactly one record in another table.

Example: Alien ↔ Laptop relationship

- One alien owns exactly one laptop
- One laptop belongs to exactly one alien

Table Structure:

alien table:
- aid (Primary Key): 101
- aname: Navin
- tech: Java
- laptop_lid (Foreign Key): 1

laptop table:
- lid (Primary Key): 1
- brand: Asus
- model: rog
- ram: 16

Key Points:

- Uses Foreign Key to establish relationship
- Foreign key can be placed in either table
- Maintains referential integrity

Lecture 14: One-to-One Mapping Implementation

Theory

OneToOne mapping in Hibernate uses JPA annotations to establish bidirectional relationships between entities. This eliminates the need for manual foreign key management.

Code Implementation

Entity Classes

Laptop Entity:

```
@Entity
public class Laptop {
    @Id
    private int lid;
    private String brand;
    private String model;
    private int ram;

    // Getters and setters
}
```

Alien Entity with OneToOne:

```
@Entity
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;

    @OneToOne
    private Laptop laptop;

    // Getters and setters
}
```

Session Management Example:

```
public class Main {
    public static void main(String[] args) {
        // Create entities
        Laptop l1 = new Laptop();
        l1.setLid(1);
```

```
        l1.setBrand("Asus");
        l1.setModel("Rog");
        l1.setRam(16);

        Alien a1 = new Alien();
        a1.setAid(101);
        a1.setAname("Navin");
        a1.setTech("Java");
        a1.setLaptop(l1);

        // Session operations
        Session session = sf.openSession();
        Transaction transaction = session.beginTransaction();

        session.persist(a1);
        session.persist(l1);

        transaction.commit();
        session.close();
    }
}
```

Configuration Setup:

```
SessionFactory sf = new Configuration()
    .addAnnotatedClass(com.telusko.Alien.class)
    .addAnnotatedClass(com.telusko.Laptop.class)
    .configure()
    .buildSessionFactory();
```

Key Features

- **@OneToOne**: Establishes one-to-one relationship
- **Automatic FK Management**: Hibernate manages foreign keys
- **Bidirectional Support**: Can navigate from both entities

Lecture 15: One-to-Many and Many-to-One Mapping

Theory

When you want to avoid creating junction tables for certain relationships, you can use OneToMany and ManyToOne annotations directly.

Problem Statement

If we don't want the third table (alien_laptop), we can establish direct relationships.

Implementation

Alien Entity (One Side):

```
@Entity
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;

    @OneToMany(mappedBy = "alien")
    private List<Laptop> laptops;
}
```

Laptop Entity (Many Side):

```
@Entity
public class Laptop {
    @Id
    private int lid;
    private String brand;
    private String model;
    private int ram;

    @ManyToOne
    private Alien alien;
}
```

Key Concepts

- **@OneToMany**: One alien can have multiple laptops
- **@ManyToOne**: Many laptops can belong to one alien
- **mappedBy**: Specifies the owning side of the relationship
- **No Junction Table**: Direct foreign key relationship

Important Note

The `get()` method has been **deprecated in Hibernate 7.1.0** but not removed.

Lecture 16: Many-to-Many Mapping

Theory

Many-to-many relationships occur when multiple records in one table can relate to multiple records in another table. This requires a junction table.

Problem & Solution

- **Problem**: Gets four tables by default
- **Solution**: Use proper annotations to control table creation

Implementation

Alien Entity:

```
@Entity
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;

    @ManyToMany
    private List<Laptop> laptops;
}
```

Laptop Entity:

```
@Entity
public class Laptop {
    @Id
    private int lid;
    private String brand;
    private String model;
    private int ram;

    @ManyToMany(mappedBy = "laptops")
    private List<Alien> aliens;
}
```

Table Structure Result

- **alien** table
- **laptop** table
- **alien_laptop** (junction table)
- **laptop_alien** (avoided with proper mapping)

Junction Table Example:

```
alien_laptop:
aliens_aid | laptops_lid
-----------|------------
101        | 1
101        | 2
102        | 1
103        | 2
```

Key Benefits

- **Bidirectional Navigation**: Can access from both sides
- **Automatic Junction Table**: Hibernate creates and manages
- **Flexible Relationships**: Supports complex many-to-many scenarios

Lecture 17: Hibernate Eager and Lazy Fetch

Theory

Hibernate provides different fetching strategies to optimise database queries and improve performance.

Caching System

Hibernate provides two types of cache:

1. **L1 Cache**: Session-level cache (default)
2. **L2 Cache**: SessionFactory-level cache

Fetch Types

Default Behavior

- **By default**: Lazy fetch is used
- **Lazy Loading**: Related entities are loaded only when accessed
- **Benefits**: Improved initial query performance

Eager Fetching Implementation:

```
@Entity
public class Alien {
    @Id
    private int aid;
    private String aname;
    private String tech;

    @OneToMany(fetch = FetchType.EAGER)
    private List<Laptop> laptops;
}
```

Key Differences

Aspect	Lazy Loading	Eager Loading
When Loaded	On-demand	Immediately
Performance	Faster initial load	Slower initial load
Memory Usage	Lower	Higher

Use Cases

- **Lazy**: Use when related data is not always needed
- **Eager**: Use when related data is frequently accessed

Lecture 18: Hibernate Caching

Theory

Caching is a performance optimisation technique that stores frequently accessed data in memory to reduce database hits.

L1 Cache (First-Level Cache)

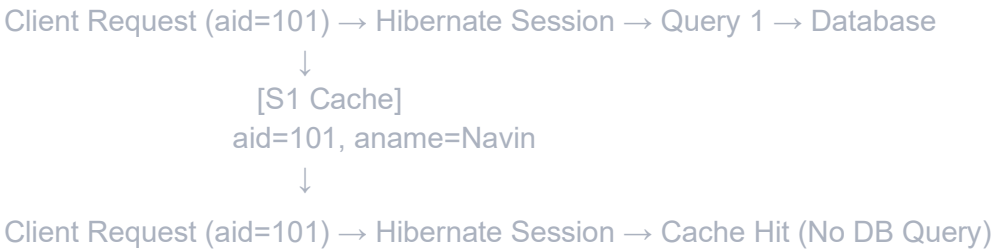
Characteristics:

- **Session-scoped:**Available within a single session
- **Automatic**: Enabled by default
- **Mandatory**: Cannot be disabled

How It Works:

1. **First Query**: Data fetched from database and cached
2. **Subsequent Queries:**Data served from cache
3. **Session Ends**: Cache is cleared

Visual Representation:



Benefits of L1 Cache:

- **Reduced Database Hits**: Same entity retrieved from cache
- **Improved Performance**: Faster data access
- **Automatic Management**: No manual configuration needed

Cache Behaviour Example:

```
// First call – hits database
Alien a1 = session.get(Alien.class, 101);

// Second call - served from cache
Alien a2 = session.get(Alien.class, 101);


// a1 == a2 (same object reference)
```

Lecture 19: HQL Introduction

Theory

HQL (Hibernate Query Language) is an object-orientated query language similar to SQL but operates on Hibernate entity objects rather than database tables.

L2 Cache

- **Implementation**: JCache
- **Scope**: SessionFactory level
- **Configuration**: Requires explicit setup

HQL Fundamentals

Key Concepts:

- **Entity Names**: Use entity class names instead of table names
- **Property Names**: Use entity property names instead of column names
- **Object-Oriented**: Operates on objects, not tables

Basic Syntax:

```
// HQL Query
String hql = "from Student";

// Equivalent SQL
String sql = "select * from student";
```

Example Usage:

```
// Create query
Query query = session.createQuery("from Student");

// Execute query
List<Student> students = query.getResultList();
```

HQL vs SQL Comparison:

Aspect	HQL	SQL
Focus	Entity objects	Database tables
Names	Entity/property names	Table/column names

Portability	Database independent	Database-specific
Type Safety	Compile-time checking	Runtime checking

Advanced HQL Features:

- **Where Clauses**: `from Student where name = 'John'`
- **Joins**: `from Student s join s.courses`
- **Aggregations**: `select count(*) from Student`
- **Parameters**: `from Student where id = :studentId`

Lecture 20: Fetching Data Using HQL

Theory

HQL (Hibernate Query Language) provides a powerful way to retrieve data from databases using object-orientated syntax. Unlike SQL, HQL operates on entity objects rather than database tables.

Basic HQL Implementation

Configuration Setup:

```
SessionFactory sf = new Configuration()
    .addAnnotatedClass(com.telusko.Laptop.class)
    .configure()
    .buildSessionFactory();
```

Simple Entity Retrieval:

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

// Direct entity retrieval using get() method
Laptop l1 = session.get(Laptop.class, 3);
System.out.println(l1);

session.close();
sf.close();
```

HQL Query Execution:

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

// HQL: select * from laptop where ram=32 -> SQL equivalent
// HQL: from Laptop where ram=32
Query query = session.createQuery("from Laptop");
List<Laptop> laptops = query.getResultList();

System.out.println(laptops);


session.close();
```

Key Concepts

- **Entity-Based Queries**: Use entity names instead of table names
- **Object-Oriented**: Returns objects, not primitive data
- **Type Safety**: Compile-time checking for entity properties

Important Deprecation Notice

The `createQuery()` method has been **deprecated in Hibernate 6.6.3** but not removed. Consider using newer alternatives for future compatibility.

Lecture 21: Fetching with Filters and Specific Properties

Theory

HQL supports advanced filtering capabilities and selective property retrieval, allowing for optimised database queries and reduced memory usage.

Filtered Queries with Parameters

Basic Filtering Implementation:

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

// Parameterized query for security and reusability
String brand = "Asus";
Query query = session.createQuery("from Laptop where brand like ?1");
query.setParameter(1, brand);
List<Laptop> laptops = query.getResultList();

System.out.println(laptops);

session.close();
```

Selective Property Retrieval

Fetching Specific Columns:

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

String brand = "Asus";
Query query = session.createQuery("select model from Laptop where brand like ?1");
query.setParameter(1, brand);
List<String> laptops = query.getResultList();

System.out.println(laptops);

session.close();
```

Multiple Properties Selection:

```
String brand = "Asus";
Query query = session.createQuery("select brand, model from Laptop where brand like ?1");
query.setParameter(1, brand);
List<Object[]> laptops = query.getResultList();

// Processing multiple properties
for (Object[] data : laptops) {
    System.out.println((String)data[0] + " " + (String)data[1]);

}
```

Key Features

Feature	Description	Example
Parameter Binding	Secure parameter substitution	`?1`, `?2` for positional
Selective Fetching	Retrieve only needed columns	`select model from Laptop`
Multiple Properties	Fetch multiple specific fields	`select brand, model from Laptop`
Type Safety	Return appropriate data types	`List<String>` vs `List<Object[]>`

Benefits

- **Performance**: Fetch only required data
- **Security**: Prevents SQL injection attacks
- **Memory Efficiency**: Reduced object overhead
- **Network Optimization**: Less data transfer

Lecture 22: Get vs Load Methods

Theory

Hibernate provides different methods for entity retrieval, each with specific use cases and performance characteristics.

Method Comparison

Using the get() Method:

```
// Traditional approach (deprecated in 7.1.0)
Laptop laptop = session.get(Laptop.class, 2);

System.out.println(laptop);
```

Using the load() Method:

```
// Load method (also deprecated)
Session session = sf.openSession();
Laptop laptop = session.load(Laptop.class, 2);
System.out.println(laptop);

session.close();
```

Modern Alternatives

Recommended Approaches:

```
// Modern alternative to get()
session.find(Laptop.class, 2);

// Modern alternative to load()

session.getReference(Laptop.class, 2);
```

Key Differences

Aspect	get() Method	load() Method
Execution	Immediate database hit	Lazy loading (proxy)
Null Handling	Returns null if not found	Throws exception if not found
Performance	Direct retrieval	Proxy-based retrieval
Status	Deprecated in 7.1.0	Deprecated in 7.1.0

Best Practices

- **Use session.find()** instead of get() for immediate loading
- **Use session.getReference()** instead of load() for lazy loading
- **Consider performance implications** of each approach
- **Plan for future Hibernate versions** by avoiding deprecated methods

Deprecation Timeline

Both the get() and load() methods are deprecated in Hibernate 7.1.0 but remain functional for backward compatibility.

Lecture 23: Level 2 Cache Using EHCache

Theory

Level 2 (L2) cache operates at the SessionFactory level, providing application-wide caching capabilities that persist across different sessions.

Cache Hierarchy

- **L1 Cache**: Session-level (automatic)
- **L2 Cache**: SessionFactory-level (requires configuration)

Dependencies Setup

Required Dependencies in pom.xml:

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jcache</artifactId>
</dependency>
```

Configuration Implementation

hibernate.cfg.xml Configuration:

xml

```
<hibernate-configuration xmlns="http://www.hibernate.org/xsd/orm/cfg">
    <session-factory>
        <!-- Database connection properties -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/telusko</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">5000</property>

        <!-- Hibernate properties -->
        <property name="hibernate.hbm2ddl.auto">update</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <!-- L2 Cache Configuration -->
```

```
        <property name="hibernate.cache.use_second_level_cache">true</property>
        <property
name="hibernate.cache.region.factory_class">org.hibernate.cache.jcache.JCacheRegionFa
ctory</property>
        <property
name="hibernate.jakarta.cache.provider">org.ehcache.jsr107.EhcacheCachingProvider</pr
operty>
    </session-factory>
</hibernate-configuration>
```

Code Implementation

Modern Session Usage:

```
// Using modern find() method instead of deprecated get()
Laptop l1 = session1.find(Laptop.class, 2);
System.out.println(l1);
```

L2 Cache Benefits

Performance Advantages:

- **Cross-Session Sharing**: Data cached across multiple sessions
- **Reduced Database Hits**: Frequently accessed data served from cache
- **Application-Level Optimization**: Improves overall application performance

Cache Levels Comparison:

Feature	L1 Cache	L2 Cache
Scope	Single Session	SessionFactory-wide
Lifetime	Session duration	Application lifetime
Configuration	Automatic	Manual setup required
Sharing	No sharing	Shared across sessions
Provider	Built-in Hibernate	EHCache, Hazelcast, etc.

Implementation Steps:

1. **Add Dependencies**: Include ehcache and hibernate-jcache
2. **Configure Properties**: Enable L2 cache in hibernate.cfg.xml
3. **Entity Annotation**: Add @Cache annotation to entities
4. **Provider Setup**: Configure cache provider (EHCache)

Cache Strategy Options:

- **READ_ONLY**: For immutable data
- **READ_WRITE**: For mutable data with read/write operations
- **NONSTRICT_READ_WRITE**: For rarely updated data
- **TRANSACTIONAL**: For transactional cache operations

Summary

This extended lecture series (L13-L23) covers comprehensive Hibernate concepts from basic mapping to advanced caching:

Core Mapping Concepts (L13-L19):

1. **L13**: Database relationship mapping fundamentals
2. **L14**: One-to-one mapping implementation
3. **L15**: One-to-many/many-to-one relationships
4. **L16**: Many-to-many mapping with junction tables
5. **L17**: Fetch strategies and caching introduction
6. **L18**: L1 cache mechanics and benefits
7. **L19**: HQL introduction and basic concepts

Advanced HQL and Caching (L20-L23):

8. **L20**: HQL data fetching with practical implementation
9. **L21**: Advanced filtering and selective property retrieval
10. **L22**: Method comparison (get vs load) and modern alternatives
11. **L23**: L2 cache implementation using EHCache

Best Practices Summary:

- **Use Modern Methods:**Replace deprecated get()/load() with find()/getReference()
- **Optimize Queries**: Use HQL parameter binding and selective fetching
- **Leverage Caching**: Implement L2 cache for frequently accessed data
- **Choose Appropriate Strategies:**Select proper fetch types and cache strategies
- **Plan for Future**: Avoid deprecated methods and use current Hibernate features
- **Performance Focus**: Balance between immediate loading and lazy loading based on use case