

Hibernate

A decorative horizontal bar with a wavy, undulating shape, composed of various colored segments including black, blue, light blue, yellow, and teal, spanning the width of the slide.

Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.

www.fandsindia.com

fands@vsnl.com



www.fandsindia.com

Ground Rules

- **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**
- **If you have questions or issues, please let me know immediately.**
- **Let us be punctual.**

Basic Recap of Hibernate

- ORM
- Loosely coupled persistent layer
- Reduction in development time
- Flexibility to change table names or column names
- Relationship Management
- Inheritance Mapping

Code Vs. Implementation

- Working with JPA Annotations and Hibernate implementation
- JPA API – maintained by Sun Microsystems
- POJO classes are not tightly coupled with Hibernate
 - migration to toptlink / OpenJPA
- Code – Can be JPA or Hibernate

Types

■ Demo1

- Hibernate Code(SessionFactory, Session) +
Hibernate Implementation + XML Files
(hibernate.cfg.xml and __.hbm.xml)

■ Demo 2

- Hibernate Code(SessionFactory, Session) +
Hibernate Implementation + JPA annotations

■ Demo 3

- JPA Code (EntityManagerFactory, EntityManager)
+ Hibernate Implementation + persistence.xml
and JPA Annotations



A **F**ast **AND** **S**teady Approach

Annotations



Entity - @entity, Table, Id, Column

```
package demo;
@Entity
@Table(name="deptTable")
public class Dept {
    private int deptno;
    private String dname;
    private double salary;

    @Id
    @Column(name="deptno")
    public int getDeptno() {
        return deptno;
    }

    public void setDeptno(int deptno) {
        this.deptno = deptno;
    }

    @Column(name="dname")
    public String getDname() {
        return dname;
    }
}
```

```
public void setDname(String dname) {
    this.dname = dname;
}

@Column(name="salary")
public double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}

public Dept() {
    System.out.println("Dept Constructor");
}
}
```


Primary Key

- Entities must define an id field. Typically a generated id is fine for most cases, but sometimes we need to add multiple fields corresponding to the database primary key. The id can either be simple or composite value. JPA provides three Strategies with annotations,
 - @Id: single valued type – most commonly used
 - @GeneratedValue(strategy=GenerationType.IDENTITY)
 - @IdClass: map multiple fields to table PK
 - @EmbeddedId map PK class to table PK

@IdClass Vs @EmbeddedId

```
@Entity
@IdClass(PersonPK.class)
public class Person {
    @Id
    private Long name;
    @Id
    private Long dateOfBirth;
}

public class PersonPK implements
    Serializable {
    private Long name;
    private Long dateOfBirth;
    public boolean equals(Object obj);
    public int hashCode();
}
```

```
@Entity
public class Person {
    @EmbeddedId
    private PersonPK key;
}

@Embeddable
public class PersonPK implements
    Serializable {
    private Long name;
    private Long dateOfBirth;
    public boolean equals(Object obj);
    public int hashCode();
}
```

Relationship Mapping

- 1 .. 1
- 1 .. M
- M .. 1
- M .. M

OnetoOne

- Student – Department
- In student
 - @OneToOne (cascade=CascadeType.ALL)
 - @JoinColumn(name="DEPT_ID", unique= **true**, nullable=**true**, insertable=**true**, updatable=**true**)
 - private** Department department;
- In address class (only for bidirectional)
 - @OneToOne(mappedBy="department")
 - private** Student student;

OnetoMany

- Dept -> Emp
- In Dept class

```
@OneToMany
(mappedBy="dept",cascade={CascadeType.REM
OVE,CascadeType.PERSIST})
private Set<Emp> emps=new HashSet<Emp>();
```
- In Emp class

```
@ManyToOne
private Dept dept;
```

ManyToMany

- Student-Department
- In student class
 - @ManyToMany
 - private** Collection<Department> departments;
- In department class
 - @ManyToMany(mappedBy="departments")
 - private** Collection<Student> students;

Batch Processing

A decorative horizontal bar spans the width of the slide, positioned between the title and the list of topics. It is composed of a series of colored rectangular segments in shades of blue, teal, yellow, and black, arranged in a slightly wavy pattern.

Batch inserts
Batch updates
StatelessSession

Batch Processing

```
Session session = sessionFactory.openSession();  
    Transaction tx = session.beginTransaction();  
for ( int i=0; i<100000; i++ )  
{ Customer customer = new Customer(.....);  
    session.save(customer); }  
tx.commit();  
session.close();
```

- Why will this lead to OutOfMemoryException ?
 - Hibernate caches all the newly inserted Customer instances in the session-level cache

Batch Processing

- If you are undertaking batch processing you will need to enable the use of JDBC batching. This is absolutely essential if you want to achieve optimal performance. Set the JDBC batch size to a reasonable number
- `hibernate.jdbc.batch_size 20`

Batch Inserts

```

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ )
    { Customer customer = new Customer(.....);
      session.save(customer);
      if ( i % 20 == 0 )
      { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear(); }
    } tx.commit(); session.close();

```

Batch updates

List Vs. Scroll

```
ScrollableResults customers =
    session.getNamedQuery("GetCustomers")
        .setCacheMode(CacheMode.IGNORE)
        .scroll(ScrollMode.FORWARD_ONLY);
int count=0; while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush(); session.clear();
    } } tx.commit(); session.close();
```

StatelessSession

- Hibernate provides a command-oriented API that can be used for streaming data to and from the database in the form of detached objects.
- No persistence context, does not provide many of the higher-level life cycle semantics.

StatelessSession

- No implementation of a first-level cache nor interact with any second-level or query cache.
- No implementation of transactional write-behind or automatic dirty checking
- No implementation of Hibernate's event model and interceptors
- In short JDBC

Locking

A decorative horizontal bar spans the width of the slide, positioned between the 'Locking' title and the 'Optimistic'/'Pesimistic' text. It is composed of a series of vertical rectangular segments in various shades of blue, teal, yellow, and black, creating a mosaic-like effect.

Optimistic
Pesimistic

Session and transaction scopes

■ SessionFactory

- is an expensive-to-create, threadsafe object, intended to be shared by all application threads. It is created once, usually on application startup, from a Configuration instance.

■ Session

- is an inexpensive, non-threadsafe object that should be used once and then discarded for: a single request, a conversation or a single unit of work. A Session will not obtain a JDBC Connection, or a Datasource, unless it is needed. It will not consume any resources until used.

Transactions and Concurrency

- Hibernate directly uses JDBC connections and JTA resources without adding any additional locking behavior.
- Hibernate does not lock objects in memory.
- In addition to versioning for automatic optimistic concurrency control, Hibernate also offers, using the `SELECT FOR UPDATE` syntax, a (minor) API for pessimistic locking of rows.

Optimistic locking properties

- When using long transactions or conversations that span several database transactions, it is useful to store versioning data to ensure that if the same entity is updated by two conversations, the last to commit changes will be informed and not override the other conversation's work. It guarantees some isolation while still allowing for good scalability and works particularly well in read-often write-sometimes situations.

Optimistic locking properties

- When using long transactions or conversations that span several database transactions, it is useful to store versioning data to ensure that if the same entity is updated by two conversations, the last to commit changes will be informed and not override the other conversation's work. It guarantees some isolation while still allowing for good scalability and works particularly well in read-often write-sometimes situations.

Pessimistic locking

- It is not intended that users spend much time worrying about locking strategies. It is usually enough to specify an isolation level for the JDBC connections and then simply let the database do all the work. However, advanced users may wish to obtain exclusive pessimistic locks or re-obtain locks at the start of a new transaction

Pessimistic locking

- Hibernate will always use the locking mechanism of the database; it never lock objects in memory.
- The LockMode class defines the different lock levels that can be acquired by Hibernate
- LockMode.WRITE is acquired automatically when Hibernate updates or inserts a row.
- LockMode.UPGRADE can be acquired upon explicit user request using `SELECT ... FOR UPDATE` on databases which support that syntax.

Pessimistic locking

- LockMode.UPGRADE_NOWAIT can be acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAIT under Oracle.
- LockMode.READ is acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. It can be re-acquired by explicit user request.
- LockMode.NONE represents the absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start out in this lock mode.

Pessimistic locking

- The "explicit user request" is expressed in one of the following ways:
- A call to `Session.load()`, specifying a `LockMode`.
- A call to `Session.lock()`.
- A call to `Query.setLockMode()`

JPA Query language

A decorative horizontal bar with a wavy, undulating shape spans the width of the slide. It is composed of numerous vertical segments in various shades of blue, teal, yellow, and black.

Basic optimizations

Caching

The query cache

Query cache regions

Second-level cache providers

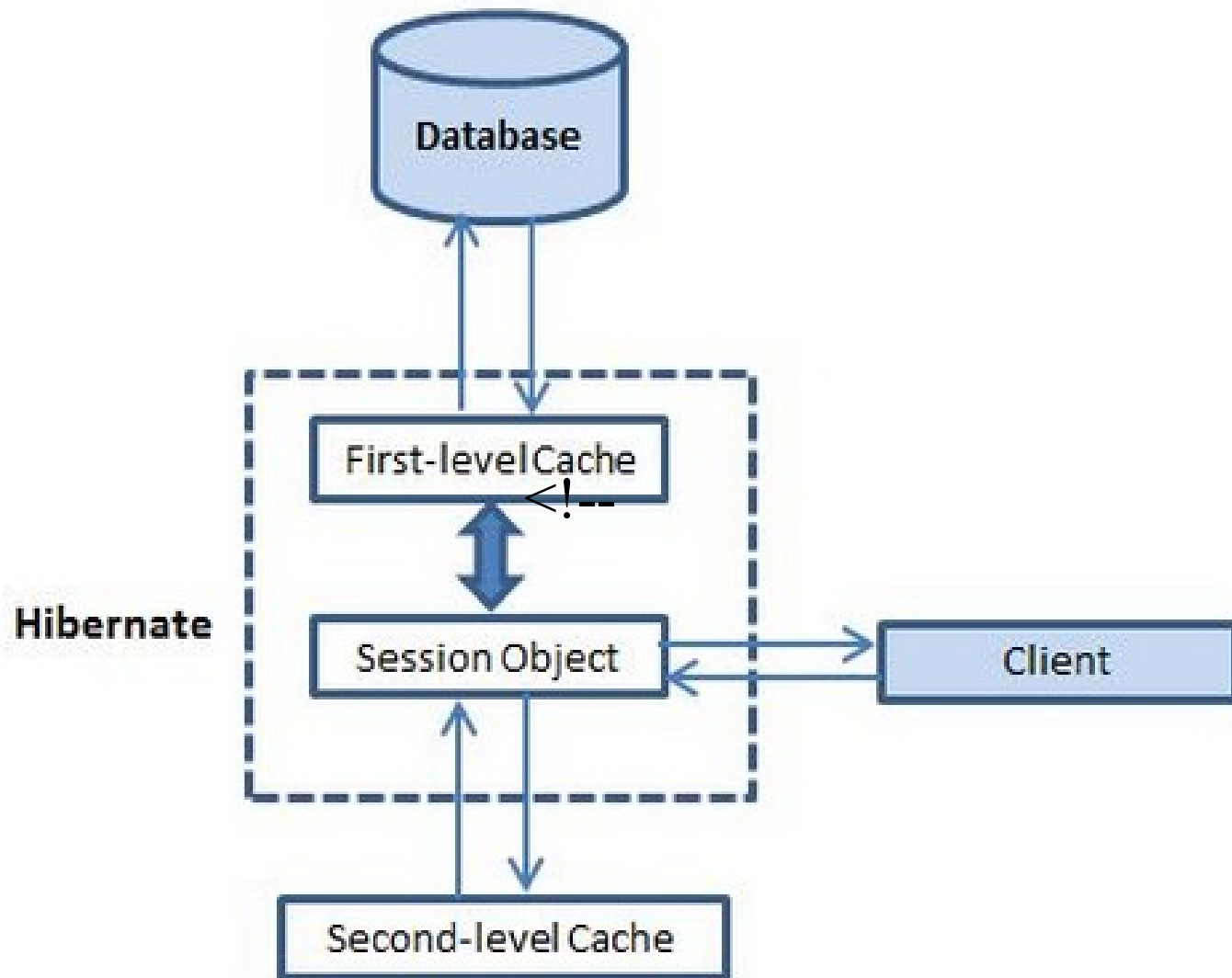
Configuring your cache providers

Caching strategies

Second-level cache providers for Hibernate

Managing the cache

Moving items into and out of the cache



Optional

The Second Level Cache

- A Hibernate Session is a transaction-level cache of persistent data.
- It is possible to configure a cluster or JVM-level.
- Define which caching implementation to use by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`

Cache Providers

- Hashtable (not intended for production use)
- EHCache
- OSCache
- SwarmCache
- JBoss Cache 1.x
- JBoss Cache 2
- Refer to Table 20.1 for details

Cache mappings

`<cache`

`usage="transactional | read-write | nonstrict-
read-write | read-only"`

`region="RegionName"`

`include="all | non-lazy" />`

Coding

- Cache Configurations
- Using Cache
- Hbm files
- Code

Cache Configuration - For ehcache

Optional

- Ehcache.properties file in src

```
<ehcache ...>
```

```
  <diskStore path="c:\\tmp" />
```

```
  <defaultCache
```

```
    maxElementsInMemory="100000"
```

```
    eternal="false"
```

```
    timeToIdleSeconds="1000"
```

```
    timeToLiveSeconds="1000"
```

```
    overflowToDisk="false" />
```

```
</ehcache>
```

Config file – enable caching

```
<property
  name="cache.provider_class">org.hibernate.cache.E
  hCacheProvider</property>

<property
  name="cache.region.factory_class">org.hibernate.ca
  che.ehcache.EhCacheRegionFactory</property>

<property
  name="cache.use_second_level_cache">true</prope
  rty>

<property
  name="cache.use_query_cache">true</property>
```

Declarative

- `<query name="testquery" cacheable="true" >`
 `select e from demo.Emp e`
 `</query>`
- `<class name="demo.Emp" table="etable">`
 `<cache usage="read-only"/>`
 `...`
 `</class`

Programmatic

- `Query qu = session.createQuery("select
e from Emp e ");`
- `qu.setCacheable(true);`
- `qu.setCacheMode(CacheMode.NORMAL);`

Caching Strategy

■ **read only**

- If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used. This is the simplest and optimal performing strategy. It is even safe for use in a cluster.

■ **read/write**

- If the application needs to update data, a read-write cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required


■ **nonstrict read/write**

- If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.

■ **transactional**

- The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache.

Validators

- 
- A decorative horizontal bar spans the width of the slide, positioned below the title. It is composed of a series of colored rectangular segments in shades of blue, teal, yellow, and black, arranged in a slightly wavy pattern.
- Defining constraints
 - Using the Validator framework

Defining constraints

- Constraints in Bean Validation are expressed via Java annotations.
- Three different type of constraint annotations
 - field
 - Property
 - class-level annotations.

Field-level constraints

- Constraints can be expressed by annotating a field of a class.

```
public class Car
{
    @NotNull
    private String manufacturer;
    @AssertTrue
    private boolean isRegistered;
}
```

Property-level constraints

- If your model class adheres to the JavaBeans standard, it is also possible to annotate the properties of a bean class instead of its fields.

@NotNull

```
public String getManufacturer() {  
    return manufacturer;  
}
```

Class-level constraints

- When a constraint annotation is placed on this level the class instance itself is passed to the `ConstraintValidator`.
- Are useful if it is necessary to inspect more than a single property of the class to validate it or if a correlation between different state variables has to be evaluated.

```
@PassengerCount  
public class Car {
```

Validating constraints

- The Validator interface is the main entry point to Bean Validation
- ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Car car = new Car(null);
Set<ConstraintViolation<Car>> constraintViolations = validator.validate(car);
assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());

QUESTION / ANSWERS



THANKING YOU !

