Hibernate

Manipulating Persistent Objects

Agenda

1. Basic Hibernate Configuration

2. Creating entity classes

Agenda

3.Hibernate Mapping with XML Files

4.Hibernate Mapping with Annotations

5.Working with Persistent Objects

Basic Hibernate Configuration

Basic Hibernate Configuration

- Hibernate configuration can be provided with a XML file (e.g. hibernate.cfg.xml) or a properties file (hibernate.properties)

- It specifies the different parameters required for connecting to a target database (Hibernate dialect, JDBC driver, DB user and password and others)

Basic Hibernate Configuration

- Sample `hibernate.cfg.xml`:

```
<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
   org.hibernate.dialect.OracleDialect
   </property>
   <property name="hibernate.connection.driver_class">
      oracle.jdbc.OracleDriver
   </property>
   <property name="hibernate.connection.url">
      jdbc:oracle:thin:@localhost:1521/academy
   </property>
   <property name="hibernate.connection.username">
      hrm
   </property>
   <property name="hibernate.connection.password">
      A1d2m3i4n5
   </property>
<property name="hibernate.jdbc.use_get_generated_keys">true</property>
   <!-- List of XML mapping files -->
   <mapping resource="Employee.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Basic Hibernate Configuration

- Business model objects (like Student and Course) can be in one of these three Hibernate states:
  - Persistent – associated with an active Hibernate session
  - Detached – associated to a Hibernate session, but the session is closed
  - Transient – object was never associated with a session

Basic Hibernate Configuration

- Applications start by creating a Hibernate `Configuration` and a `SessionFactory`
- Configuration reads the Hibernate settings from the `hibernate.cfg.xml` file
- The `SessionFactory` is used to create `Session` instances
- A `Session` instance represents a dialog with the database through Hibernate
- Hibernate sessions usually work with an associated `Transaction`

Basic Hibernate Configuration

- Example:

```
// Load the settings from hibernate.cfg.xml
Configuration cfg = new Configuration();
cfg.configure();
// Create the Hibernate session factory
SessionFactory sessionFactory =
cfg.buildSessionFactory();
// Start new Hibernate session
Session session =
sessionFactory.openSession();
// Start a new transaction (this is recommended)
session.beginTransaction();
```

Basic Hibernate Configuration

- A typical transaction flow:

```
Session session = factory.openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   ...
   tx.commit();
}
catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
}finally {
   session.close();
}
```

Creating Entity Classes

Creating Entity Classes

- After hibernate configuration is created typically the next step is to create the entity classes.
- If the relational database schema is present - entity classes can be generated out of it
- If the relational database schema is not present - it can be generated after creating the entity classes

Creating Entity Classes

- An entity is a simple Java class (POJO - Plain Old Java Class):

```
public class Employee {

private Integer id;
private String name;
…
public Integer getId() {
        return id;
}

public void setId(Integer id) {
        this.id = id;
}

public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}
…
```

Creating Entity Classes

- Each entity class corresponds to a relational table
- References to other tables are represented with Java collections (typically lists, sets or maps)
- Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity

Creating Entity Classes

- `Many-to-one/one-to-one` relationships are specified by declaring a property of the type of the parent/referenced entity

- For example:

```
public class Employee {

private Department deparment;
…
public Department getDepartment() {
        return department;
}

public void setDepartment(Department department) {
        this.department = department;
}
…
```

Creating Entity Classes

- `One-to-many` relationships are specified by declaring a collection field that keeps instances of the type of the child entity

- For example:

```
public class Employee {

private List<Vacation> vacations;
…
public List<Vacation> getVacations() {
        return vacations;
}

public void setVacations(List<Vacation> vacations) {
        this.vacations = vacations;
}
…
```

Creating Entity Classes

- `Many-to-many` relationships can be implemented in Hibernate as:

- Two many-to-one relationships (if an entity for the many-to-many table is created)

- One many-to-many relationship (if an entity for the many-to-many table is not created) - child collections for referenced entities may be present in both entities

Hibernate Mapping with XML Files

Hibernate Mapping with XML files

- After Hibernate entities are created the next step typically is to map them - this can be done using XML files
- XML files provide the required information to Hibernate on how to map Java classes to Hibernate entities
- Mapping types are used by Hibernate to map between Java data types and SQL data types

Hibernate Mapping with XML files

Mapping type	Java type	ANSI SQL Type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes/no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

Hibernate Mapping with XML files

Mapping type	Java type	ANSI SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Hibernate Mapping with XML files

- Sample `Employee.xbm.xml`:

```
<hibernate-mapping>
<class name="com.example.hibernate.entities.Employee"
table="EMPLOYEES">
<meta attribute="class-description">
Represents an HRM employee
</meta>
<id name="id" type="int" column="id">
<generator class="sequence">
<param name="sequence">employees_id_seq</param>
</generator> </id>
<property name="name" column="name" type="string" />
<property name="titleId" column="titleid" type="int" />
<property name="departmentId" column="departmentid" type="int" />
<property name="email" column="email" type="string" />
<property name="phone" column="phone" type="string" />
<property name="salary" column="salary" type="double" />
<property name="cv" column="cv" type="blob" />
<property name="hireDate" column="hiredate" type="date" />
<property name="endDate" column="enddate" type="date" />
<property name="referrerId" column="referrerid" type="int" />
<property name="managerId" column="managerid" type="int" />
</class>
</hibernate-mapping>
```

Hibernate Mapping with XML files

- Hibernate provides mechanisms to designate any of the following types of relationships:
  - one-to-one
  - many-to-one
  - one-to-many
  - many-to-many
- Different collection types can be used for the purpose - collection, list, map, set

Hibernate Mapping with XML files

- Unidirectional `Many-to-one` example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <many-to-one name="department"
                        column="deparmentId"
                        not-null="true"/>
</class>
<class name="Departments">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
</class>
```

Hibernate Mapping with XML files

- Unidirectional `One-to-one` example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <one-to-one name="department"
                class="Department"
                column="departmentId"
                not-null="true"/>
</class>
<class name="Departments">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
```

Hibernate Mapping with XML files

- Unidirectional `One-to-one` (using many-to-one and a unique constraint) example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <many-to-one name="department"
                        column="deparmentId"
                         unique="true",
                        not-null="true"/>
</class>
<class name="Departments">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
</class>
```

Hibernate Mapping with XML files

- Unidirectional `One-to-many` example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
<class name="Departments">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
        <set name="employees">
                <key column="departmentId" not-
null="true"/>
                <one-to-many class="Employee"/>
        </set>
</class>
```

Hibernate Mapping with XML files

- Unidirectional `One-to-many` (using a join table and a `many-to-many` with unique constraint) example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <set name="skills"
        table="EmployeeSkills">
                <key column="employeeId"
                        not-null="true"/>
                <many-to-many class="Skills"
                        column="skillId"
                        unique="true"/>
        </set>
</class>
<class name="Skills">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
```

Hibernate Mapping with XML files

- Unidirectional `Many-to-many` **example:**

```
<class name="Employee">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
<class name="Department">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
        <set name="employees">
                <key column="departmentId" not-
null="true"/>
                <one-to-many class="Employee"/>
        </set>
</class>
```

Hibernate Mapping with XML files

- When the mapping is bidirectional (meaning that both the parent and the child entity define the mapping) you always have to mark one of the mappings with `inverse="true"` so that Hibernate can determine which relationship endpoint to use for persistence

Hibernate Mapping with Annotations

Hibernate Mapping with Annotations

- Annotations:
  - Less verbose than XML
  - The preferred way in mapping entities if applicable

- A number of predefined annotations are provided by `javax.persistence` JPA and Hibernate

Hibernate Mapping with Annotations

- Entity annotations are two types:
  - Physical - describe the association between entities
  - Logical - describe the database schema (tables, columns, etc.)
- Hibernate-specific annotations are in the `org.hibernate.annotations` package
- JPA-specific annotations are in the `javax.persistence` package

Hibernate Mapping with Annotations

- The `@Entity` annotation marks a class as an entity

```
@Entity
public class Employee {
     …
}
…
```

- The `@Table` annotation specifies the referenced database table

```
@Entity
@Table(name="EMPLOYEES")
public class Employee {
     …
}
…
```

Hibernate Mapping with Annotations

- The `@Column` annotation maps a class attribute to a column in the corresponding table

```
@Entity
@Table(name = "EMPLOYEES")
public abstract class Employee {
    @Column(name = "Name", nullable = false, length = 50)
    private String name;
    …
}
```

- The `@Column` annotation provides:
  - name - table column name
  - unique - can set a unique constraint on the column
  - nullable - can set a null constraint on the column
  - length, precision, scale (if applicable)

Hibernate Mapping with Annotations

- The `@Id` annotation maps a field to a primary key field (ID field)

```
@Entity
@Table(name = "EMPLOYEES")
public abstract class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    …
}
```


```
@Entity
@Table(name = "EMPLOYEES")
public abstract class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    …
}
```

Hibernate Mapping with Annotations

- Hibernate provides different strategies for setting the `@Id` field:
  - IDENTITY - set by the RDBMS by means of an identity column (is supported by the RDBMS)
  - TABLE - uses a hi/lo algorithm to generate identifiers given a table and column as a source of hi values
  - SEQUENCE - uses a hi/lo algorithm to generate identifiers using a named database sequence
  - AUTO (default one) - selects any of the previous based on the capabilities of the underlying database
  - Custom - custom ID generator

Hibernate Mapping with Annotations

- Every non-static non-transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as `@Transient`

Hibernate Mapping with Annotations

- By default the access type of a class hierarchy is defined by the position of the `@Id` or `@EmbeddedId` annotations
- If Hibernate annotations are on a field, then only fields are considered for persistence and the state is accessed via the field

Hibernate Mapping with Annotations

- If Hibernate annotations are on a getter, then only the getters are considered for persistence and the state is accessed via the getter/setter

- Access type can be enforced by means of the `@Access` annotation specified on the entity class (not recommended - should be used only if necessary)

Hibernate Mapping with Annotations

- Fields can be derived by an entity by referenced classes (also called embedded objects or components)
- Component classes must be marked with an `@Embeddable` annotation
- Using components you can improve reusability of source code by extracting common entity fields to components (if applicable)

Hibernate Mapping with Annotations

- Example (using components):

```
@Entity
@Table(name = "EMPLOYEES")
public class Employee {

      private User id;
      …
}

@Embeddable
Public class User {
       private String name;
       private String phone;
       …
}
```

- Column mappings from a components can be overridden by using the `@Embedded` and `@AttributeOverrides` annotations

Working with Persistent Objects

Working with Persistent Objects

- The Session class represents a connection to the database through Hibernate - used for issuing commands

  get(Class, <primary key>)

  - loads entity from database by primary key

  save(Object)

  - Persists transient object to the database
  - Issues INSERT / UPDATE SQL commands
  - Populates the primary key column value
  - Modifies related tables when necessary

Working with Persistent Objects

update(Object)

- Reattaches detached object to the session - issues SELECT SQL command

- For persistent objects updates the database (issues INSERT / UPDATE / DELETE)

saveOrUpdate(Object)

- Combines save() and update()

delete(Object)

- Removes given persistent entity object from the database

- Cascade delete ca be set in the mappings

Working with Persistent Objects

refresh(Object)

- Refreshes given persistent entity object from the database
- Loads its most recent state along with the state of its related entities

lock(Object, LockMode.NONE)

- Reattaches detached object to the session
- Use this method instead of update() for reattaching detached objects

Working with Persistent Objects

flush(Object)

- Executes all needed SQL statements in the following order:
    - insertions
    - updates
    - collection deletions
    - collection element deletions, updates and insertions
    - collection insertions
    - entities deletion

Working with Persistent Objects

createQuery(<HQL query>) -> Query

- Creates an HQL query

createSQLQuery(<SQL query>) -> SQLQuery

- Creates an SQL query

createCriteria(<Class>) -> Criteria

- Creates a criteria query

Working with Persistent Objects

- Sample HQL query

```
Query allStudentsQuery =
      session.createQuery("from Student");
List<Student> allStudents =
      allStudentsQuery.list();
for (Student stud : allStudents) {
      System.out.printf(
      "Id=%d, FirstName=%s, LastName=%s\n",
      stud.getPersonId(), stud.getFirstName(),
      stud.getLastName());
}
```

Working with Persistent Objects

- Finding an entity by primary key:

```
session.get(<entity class>, <primary key>);
```

- Example:

```
private static Course getCourseByPrimaryKey(
      Session session, long primaryKey) {
   Course course = (Course)
      session.get(Course.class, primaryKey);
   return course;
}
```

Working with Persistent Objects

- To create a new entity - create a new instance of the entity class, assign properties of the instance and save it

- Example:

```
private static Professor addNewProfessor(
      Session session, String firstName,
      String lastName, String title) {
   Professor newProf = new Professor();
   newProf.setFirstName(firstName);
   newProf.setLastName(lastName);
   newProf.setTitle(title);
   session.save(newProf);
   return newProf;
}
```

Working with Persistent Objects

- Only parent entities are saved by default with `save()` - to save also modifications to child entities specify a `save-update` cascade option:

- Example:

```
<class name="model.Professor" table="PROFESSORS">
...
<set name="courses" table="COURSES"
      cascade="save-update">
      <key column="PROFESSORID"/>
      <one-to-many class="model.Course"/>
</set>
</class>
```

Working with Persistent Objects

- Changes to the entities (create, modify, delete, ...) are not executed immediately

- Hibernate holds them in a list and executes them at once in a sequence

```
session.flush();
```

- To force posting pending changes use:

- This executes the necessary SQL commands

- Does not commit the active transaction

Working with Persistent Objects

- Sometimes entities or their associated entities hold old data and need to be refreshed (reloaded from the database)

```
session.refresh(object);
```

- To refresh a persistent entity object use:

- Use case:
  - We load an entity
  - Meanwhile another transaction updates it
  - We need to refresh it

Working with Persistent Objects

- To modify an entity:
  1. Obtain persistent entity object by `Session.get(…)` or `Query.list()`
  2. Modify the persistent object
  3. Invoke `Session.update(Object)`

- Example:

```
Course course = (Course)
session.get(Course.class, 5L);
course.setName("New name");
session.update(course);
```

Working with Persistent Objects

- Suppose we get some object from the database and close the Hibernate session:
  1. This object becomes detached
  2. If we try to get its child entities an exception will be thrown
- Example:

```
Course course = (Course)
session.get(Course.class, 5L);
session.close();
List<Student> students = course.getStudents();
// org.hibernate.LazyInitializationException:
// no session or session was closed
```

Working with Persistent Objects

- Detached objects can be attached again with `session.buildLockRequest(LockMode.NONE).lock(Object)` method
- Example:

```
Course course = (Course)
session.get(Course.class, 5);
session.close();
// Now the course object is "detached"
session = sessionFactory.getCurrentSession();
session.beginTransaction();
session.buildLockRequest(LockMode.NONE).lock(course);
// Attach "course" object to the new session
List<Student> students = course.getStudents();
```

Questions?

Problems

1. How do we map Java POJO classes (entities) to database tables ? What about fields to columns and Java types to SQL types ?

2. How can we attach/detach objects in a Hibernate session ?

3. How do we create HQL/SQL/Criteria queries in Hibernate ?