Hibernate

Queries and Transactions

Agenda

1. Writing Native Queries
2. Writing HQL Queries
3. Writing Criteria Queries
4. Handling Transactions

Writing Native Queries

Writing Native Queries

- Native queries are just plain SQL queries written using the Hibernate API
- The results are return as `Object[]`

Writing Native Queries

- Example query:

```
private static List<Object[]>
getStudentsNames(Session hbSession) {
      SQLQuery studentsNamesQuery =
      hbSession.createSQLQuery(
      "select p.firstName, p.lastName " +
      "from students s inner join persons p
" + "on s.studentId = p.personId");
List<Object[]> studentsNames =
      studentsNamesQuery.list();
      return studentsNames;
}
```

Writing Native Queries

- SQL queries can return results transformed to an arbitrary JavaBean class:

```
SQLQuery studentsNamesQuery =
   session.createSQLQuery(
      "select p.firstName as \"firstName\", " +
      "p.lastName as \"lastName\" " +
      "from students s inner join persons p " +
      "on s.studentId = p.personId");
studentsNamesQuery.setResultTransformer(
   Transformers.aliasToBean(NameBean.class));
List<NameBean> names = studentsNamesQuery.list();
```

Writing HQL Queries

Writing HQL Queries

- HQL is very similar to SQL
- HQL queries work on entities and their properties and is object-oriented (meaning it can handle object-oriented features of the Hibernate entities)
- Entity names and properties are case sensitive in HQL - everything else is case insensitive as in SQL

Writing HQL Queries

- In HQL you can omit a `SELECT` clause:

```
from com.martin-toshev.hrm.entities.Employee
```

- You can omit the package of the entity:

```
from Employee
```

- You can use aliases:

```
from Employee emp
```

```
from Employee as emp
```

Writing HQL Queries

- Example query:

```
Query allEmployeesQuery =
session.createQuery("from Employee");
List<Employee> allEmployees =
      allStudentsQuery.list();
for (Employee emp : allEmployees) {
   System.out.printf(
   "Id=%d, Name=%s, Phone=%s\n",
   emp.getName(), emp.getPhone());
}
```

Writing HQL Queries

- In HQL there is also a `where` clause:

```
from Employee emp where emp.name like 'Ivan%'
```

- You can also use a `SELECT` clause:

```
select emp.name from Employee emp where
emp.name like 'Ivan%'
```

Writing HQL Queries

- You can use an `ORDER BY` clause:

```
from Employee emp order by emp.name desc
```

- You can have nested queries

```
from Employee e
where
e.salary = (select max(salary) from Employee)
```

Writing HQL Queries

- Navigating through many-to-one associations is performed directly

```
from Customer as cust
where cust.address.town = 'Sofia'
```

(Implemented with table joins in RDBMS)

This special field indicates the size of the collection

```
from Order o
where o.orderDate > '2006-1-1'
and (o.customer.lastName = 'Todorov'
or o.custmer.orders.size > 10)
```

Writing HQL Queries

- One-to-many and many-to-many navigations require joins

```
from Employee as emp
inner join emp.skills
```

- To avoid duplicate records use `disctinct`:

```
select distinct prof
from Professor prof inner join prof.courses c
inner join c.students stud
where stud.lastName = 'Nikolova'
```

Writing HQL Queries

- HQL queries can have named and unnamed parameters
- Example (unnamed parameters):

```
from Student stud
where stud.lastName = ?
```

- Example (named parameters):

```
from Student stud
where stud.firstName = :first_name
and stud.lastName = :last_name
and stud.courses.size = :count_of_courses
```

Writing HQL Queries

- Example query:

```
Query queryStudentByName = hbSession.createQuery(
      "from Student " +
      "where firstName = :fname " +
      "and lastName = :lname");
      queryStudentByName.setString("fname", "Ivan");
      queryStudentByName.setString("lname",
      "Ivanov");
List<Student> students = queryStudentByName.list();
if (students.size() == 0) {
      System.out.println("Student not found");
} else if (students.size() == 1) {
      Student stud = students.get(0);
      System.out.println("Id=" +
      stud.getPersonId());
} else {
      System.out.println("Too many students found");
}
```

Writing HQL Queries

- HQL queries are polymorphic (meaning that when a parent entity is retrieved - child entities are also retrieved)
- Polymorphic behavior can be controlled via Hibernate configuration
- When results are not entities then the returned values are stored as `Object[]`

Writing HQL Queries

- Example query:

```
Query deptsStudsQuery = session.createQuery(
      "select dept.deptId, dept.name, count(*) " +
      "from Department dept join dept.Courses c " +
      "group by dept.deptId, dept.name");
List<Object[]> deptsAndStuds =
deptsStudsQuery.list();
```

Writing HQL Queries

- Several ways to process Query results:
  - list() - returns the results as a java.util.List instance
  - Iterate() - returns the results as a java.util.Iterator instance
  - uniqueResult() - returns single object, null or NonUniqueResultException

Writing HQL Queries

- Example query (using iterators to process queries that retrieve a lot of records):

```
Query studQuery =
      session.createQuery("from Student");
      Iterator<Student> studIter = studQuery.iterate();
      while (studIter.hasNext()) {
         Student stud = studIter.next();
         System.out.printf("FirstName=%s, LastName=%s",
            stud.getFirstName(), stud.getLastName());
}
```

Writing HQL Queries

- You can specify additional join conditions using the `with` keyword:

```
from Employee as emp left join emp.skills as skill
with skill.name like '%Java%'
```

- You can also retrieve child records from joined entities using the `fetch` keyword:

```
from Cat as cat
inner join fetch cat.mate
left join fetch cat.kittens
```

Writing HQL Queries

- You can also create a new bean from an HQL query assuming it has the appropriate constructor:

```
select new EmployeeBean(emp.name, emp.phone)
from Employee emp
```

- HQL queries can even return the results of aggregate functions (such as **sum, max, count and avg)** on properties

Writing HQL Queries

- You can use indices to refer to particular elements:

```
from Employee emp where emp.skills[0].name like
'%Java%'
```

- Indices can only be used in a `WHERE` clause

Writing Criteria Queries

Writing Criteria Queries

- Criteria queries are a purely object-oriented mechanism for executing queries against an RDBMS from Hibernate
- Criteria queries are an alternative to HQL
- Hibernate provides an API for constructing Criteria queries

Writing Criteria Queries

- The Criteria API provides a mechanism for specifying `WHERE` conditions, joins, ordering, and basically most of the things supported in HQL (but not all of them)

- Criteria queries can also be generated at runtime

Writing Criteria Queries

- Example queries:

```
List<Course> coursesByCriteria =
      session.createCriteria(Course.class)
      .add(Restrictions.like("name", "%Java%"))
      .list();
```

```
List<Course> coursesByCriteria =
      session.createCriteria(Course.class)
      .add(Restrictions.in("name",
      new String[] { "Java", "C#", "C++" } ) )
      .createAlias( "professor", "prof" )
      .add(Restrictions.eq("prof.lastName","Sali" ) )
      .addOrder( Order.asc("name") )
      .list();
```

Handling Transactions

Handling Transactions

- Optimistic concurrency control means that conflicts are resolved at the end of a transaction (optimistically) - supported in Hibernate via automatic versioning

- Pessimistic concurrency control means that conflicts are resolved during transaction execution via database locks (pessimistically) - supported in Hibernate by means of the locking mechanisms in the underlying database

Handling Transactions

- Hibernate does not lock objects in memory - locking semantics are defined by the RDBMS by means of the supported transaction isolation levels in case of pessimistic locking

- In addition to versioning for automatic concurrency control, Hibernate provides an API for pessimistic locking of rows via `SELECT FOR UPDATE` syntax

Handling Transactions

- It is possible to implement transactions in the application by using explicit locks on the database objects being manipulated - however this is a bad practice since the application does not scale to many users

- When multiple users access the database concurrently it is recommended to uses Hibernate's built-in utilities for versioning in order to use optimistic locking for the entities

Handling Transactions

- Object identity != Entity identity
- In a single Hibernate session object identity is guaranteed to equal entity identity
- In concurrent sessions object identities are different but entity identities for the same database record are equivalent

Handling Transactions

- Transactions can be committed/roll-backed explicitly when using hibernate in a non-managed environment
- Transactions can be committed/roll-backed by the application container in a managed environment (such as a j2ee application server) when using Hibernate with JTA (either with bean-managed transaction or with container-managed transaction)

Handling Transactions

- Hibernate wraps any underlying JDBC exceptions (such as an SQLException) that can be thrown in a HibernateException instance

Handling Transactions

- In case an exception occurs then:
  - In a non-managed environment the application developer must ensure that the Hibernate session is closed
  - In a managed environment the application developer may not need to explicitly commit/rollback the transaction or close the session when an exception is thrown - this is typically done by the container

Questions?

Problems

1. What is the HQL language? Is it similar to SQL? What are the differences?
2. Write an HQL query to find all products that have price in the range [10…50].
3. Write an HQL query to find all categories that contain exactly 3 or 4 products.
4. Write an HQL query to find all orders that have total due (sum of prices * amount) of 100 or more.
5. Write an HQL query to find the max expensive order between given start and end date.

Problems

- 6. Write a Hibernate program that lists all orders from the database that contain a product with a quantity > 10.
- 7. Write a parameterized query to find all orders between given two dates that have total due above given amount. Use named parameters.
- 8. Write a query to print the sum of all orders by product category. Use the `GROUP BY` statement or a native SQL query.