Module 07: Hibernate Query Language

CS544: Enterprise Architecture

Retrieving Entities

- So far we've discussed how to:
  - Use load() or get() to find an entity by its ID
  - Access related entities from loaded entities

- But what if you don't know an entity's ID?
- Hibernate offers several ways to query the DB
  1. Using the Hibernate Query Language (HQL)
  2. Using the programmatic Criteria API
  3. Using Native SQL Queries in your DBs SQL dialect

Hibernate Query Language

- HQLs syntax is relatively similar to SQL, but is object oriented instead of relational
  - HQL understands objects and attributes
  - HQL understands associations between objects
  - HQL understands inheritance and polymorphism

```
List<Account> accounts = session.createQuery("from Account act "
    + "where act.class <> CheckingAccount "
    + "and act.owner.firstname = 'Frank'").list();
```

HQL query that retrieves a list of accounts that are not checking accounts and whose owners first name is Frank

Hibernate Query Language:

THE QUERY OBJECT

The HQL Query Object

- Basics of using HQL Queries:
  - Create a HQL Query: session.createQuery()

```
Query query = session.createQuery("from Person");
```

  - Retrieve a list of objects: query.list()

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Query query = session.createQuery("from Person");
List<Person> people = query.list();

tx.commit();
```

Queries should also be executed inside transactions

Create query

Execute query

Named Queries

- Named HQL queries can be stored inside mapping data
  - Can then be retrieved by name for execution
  - Separates HQL query strings from Java code
  - Can also be used to organize queries in one location

XML mapping:

```
<hibernate-mapping package="entities">
  <class name="Person">
    ...
  </class>
  <query name="Person.Everybody"><![CDATA[
     from Person
  ]]></query>
</hibernate-mapping>
```

Name needs to be Globally Unique

Use CDATA tags around the query to make sure < and > comparison operators do not conflict with XML

Annotation mapping:

```
@Entity
@NamedQueries({
  @NamedQuery(name="Person.Everybody", query="from Person")
})
public class Person {
  @Id
  @GeneratedValue
  private int id;
  ...
```

Optional @NamedQueries can be used to specify multiple @NamedQuery

Globally Unique name

Using Named Queries

- Retrieve a named query by its name
- Then use it like any other query

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Query query = session.getNamedQuery("Person.Everybody");
List<Person> people = query.list();
for (Person p : people) {
  System.out.println(p.getFirstname() + " " + p.getLastname());
}

tx.commit();
```

Query defined in metadata

Executed like a normal query

Hibernate Query Language:

THE FROM CLAUSE

From Clause

- The simplest HQL query only contains a ‘from’ clause – the select clause is optional*

```
List<Person> people = session.createQuery("from Person").list();
```

Returns all objects of the Person class in the database

- HQL keywords are not case sensitive
  - FROM, from, FrOm, and fRoM are the same keyword
- Class and property names are case sensitive
  - Person and person are two different classes!

Polymorphic Queries

- HQL has very solid polymorphism support
  - You can query for mapped inheritance hierarchies
    - Will return all accounts regardless of sub-type

```
List<Account> accounts = session.createQuery("from Account").list();
```

  - For classes that implement any interface
    - Will return all objects that implement the interface

```
Query q = session.createQuery("from java.io.Serializable");
List<Object> objects = q.list();
```

  - Or even for every any object what so ever
    - Be careful – Will return every object in the database

```
Query q = session.createQuery("from java.lang.Object");
List<Object> objects = q.list();
```

Aliases

- You can alias class names for ease of reference
  - Just like SQL allows you to alias tables names

```
List<Person> people = session.createQuery("from Person as p").list();
```

  - Just like SQL the 'as' keyword is optional

```
List<Person> people = session.createQuery("from Person p").list();
```

  - We will demonstrate this feature more in the queries later on in this module

Pagination

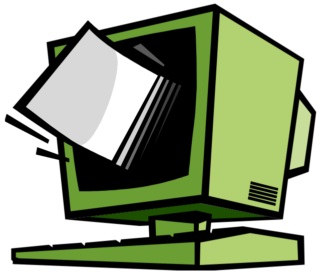
- HQL has build in pagination support to select part of a large result set
- Crucial feature when users need to work with large result sets (like a ‘from Person’ query)

```
Query query = session.createQuery("from Person");

query.setFirstResult(0);
query.setMaxResults(50);

List<Person> people = query.list();
for (Person p : people) {
  System.out.println(p.getFirstname() + " " + p.getLastname());
}
```

Zero based index – 0 is the beginning

Page size

Returns the first 50 people



Order By

- The 'order by' clause sorts the list in a particular order

Order people by lastname

```
Query query = session.createQuery("from Person p order by p.lastname");
List<Person> people = query.list();
for (Person p : people) {
  System.out.println(p.getFirstname() + " " + p.getLastname());
}
```

- The 'asc' and 'desc' keywords can be added to specify ascending or descending order

```
Query query = session.createQuery("from Person p order by p.lastname desc");
List<Person> people = query.list();
for (Person p : people) {
  System.out.println(p.getFirstname() + " " + p.getLastname());
}
```

By lastname descending

Hibernate Query Language:

THE WHERE CLAUSE

The Where Clause

- The where clause lets you add constraints to the result set, refining the list

Select all person objects whose first name is John

```
Query query = session.createQuery("from Person where firstname = 'John'");
List<Person> people = query.list();
for (Person p : people) {
  System.out.println(p.getFirstname() + " " + p.getLastname());
}
```

- HQL supports the same expressions as SQL in addition to several OO specific expressions

```
Query query = session.createQuery("from Person p where p.accounts[0].id = 3");
List<Person> people = query.list();
```

Select all people whose first account id is 3

HQL Expressions

Type	Operators
Literals	'string', 128, 4.5E+3, 'yyyy-mm-dd hh:mm:ss'
Arithmetic	+, -, *, /
Comparison	=, <>, >=, <=, !=, like
Logical	and, or, not
Grouping	(,)
Concatenation	||
Values	in, not in, between, is null, is not null, is empty, is not empty
Case	case … when … then … else … end, case when … then … else … end

Type	Functions
Temporal	current_date(), current_time(), current_timestamp(), second(…), minute(…), hour(…), day(…), month(…), year(…)
String	concat(… , …), substring(), trim(), lower(), upper(), length(), str()
Collection	Index(), size(), minindex(), maxindex()

Indexed Collection Expressions

- HQL provides the [] expression syntax for accessing elements of indexed collections

```
Query query = session.createQuery("from Person p where p.accounts[0].id = 3");
List<Person> people = query.list();
```

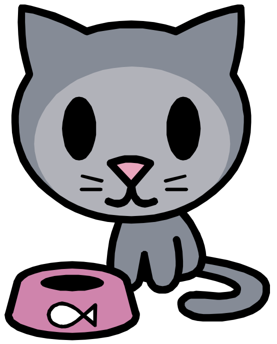
Accounts is mapped as List

- Both Lists and Maps can be accessed

```
Query query = session.createQuery("from Person p where p.pets['Mimi'].species = 'Cat'");
List<Person> people = query.list();
```

Retrieves all the people who have a cat named Mimi

Map of pets using the pet name as its key



Query Parameters

- It is considered very bad practice to use concatenation to insert values into Queries
  - Opens the door for HQL (SQL) injection
  - Makes query messy and less readable

```
String firstname = "John";
Query query = session.createQuery(
    "from Person where firstname = '"+firstname+"'");
```

Makes messy code, and breaks if there is a ‘ inside firstname

Don’t try this at home

- Named parameters should be used instead

```
String firstname = "John";
Query query = session.createQuery("from Person where firstname = :first");
query.setParameter("first", firstname);
```

Much cleaner, and can be used with named queries stored in XML / annotations

Setting Parameters

- Hibernate provides two ways for setting parameters on queries

1. The setParameter(name, param, [type]) method

```
String firstname = "John";
Query query = session.createQuery("from Person where firstname = :first");

query.setParameter("first", firstname, Hibernate.STRING);

List<Person> people = query.list();
```

Type argument is optional, except for temporal types

2. The setType(name, param) methods

```
String firstname = "John";
Query query = session.createQuery("from Person where firstname = :first");

query.setString("first", firstname);

List<Person> people = query.list();
```

Type is specified by the method name

Positional Parameters

- Hibernate also supports positional parameters
  - Similar to JDBC using ? to indicate a parameter

```
Query query = session.createQuery(
    "from CheckingAccount where balance < ? and overdraftLimit > ?");
query.setParameter(0, 1000.0);
query.setParameter(1, 100.0);
List<Account> accounts = query.list();
```

First parameter

Second parameter

Set parameters, 0 for the first, 1 for the second

- The use of positional parameters is not recommended
  1. Less self documenting than named parameters
  2. More brittle to change than named parameters

Unique Result

- Use query.uniqueResult() if you are certain that the result will only contain a single object
  - Throws NonUniqueResultException if the resultset contains more than one object
  - Method returns null if the result set is empty

```
Query query = session.createQuery("from Person");
query.setMaxResults(1);
Person p1 = (Person) query.uniqueResult();
```

MaxResults = 1, guarantees unique result

```
Person p = (Person)session.createQuery("from Person where id = 1").uniqueResult();
```

Specific Id guarantees unique result

Special Attribute: id

- Even if you did not call your identity column id, you can still refer to it in HQL as id*

```
Query query1 = session.createQuery("from Employee where employeeId = 1");
Employee e1 = (Employee)query1.uniqueResult();
System.out.println(e1.getFirstname() + " " + e1.getLastname());

Query query2 = session.createQuery("from Employee where id = 1");
Employee e2 = (Employee)query2.uniqueResult();
System.out.println(e2.getFirstname() + " " + e2.getLastname());
```

Both queries produce the same result

```
@Entity
public class Employee {
  @Id
  @GeneratedValue
  private int employeeId;
  private String firstname;
  private String lastname;
  @Temporal(TemporalType.DATE)
  private Date birthdate;

  ...
```

Id property "employeeId"

Hibernate Output:

```
Hibernate: /* from Employee where employeeId = 1 */ ...
Jim Grove
Hibernate: /* from Employee where id = 1 */ ...
Jim Grove
```

Special Attribute: class

- Hibernate also provides a special attribute that allows you to check the class of an object

```
List<Account> accounts = session.createQuery("from Account act "
    + "where act.class <> CheckingAccount "
    + "and act.owner.firstname = 'Frank'").list();
```

Hibernate Query Language:

JOINS

Joins

- Hibernate supports implicit and explicit joins
  - Explicit joins use the join keyword
  - Has to explicitly state which class to select

```
Query query = session.createQuery(
          "select p from Person p join p.address a where a.zip = :zip");
query.setParameter("zip", "90009");
List<Person> people = query.list();
```

Join keyword

Select p not a

  - An implicit join uses the dot notation to join an object of another class without using 'join'

```
Query query = session.createQuery("from Person p where p.address.zip = :zip");
query.setParameter("zip", "90009");
List<Person> people = query.list();
```

Joins Address

People often make mistakes with this on the exams

Inner Joins

- Inner joins require that both objects exist – unlike outer joins where one can be null
  - If one is null, the potential result is discarded
  - Implicit joins are always inner joins
  - Just using the ‘join’ keyword signifies an inner join
  - All previous join examples where inner joins

```
Query query = session.createQuery("from Person p where p.address.zip = :zip");
query.setParameter("zip", "90009");
List<Person> people = query.list();
```

Person has to have an address, a person without an address would not be included in the result set

Outer Joins

- Outer joins allow one side to be null
  - A left outer join allows the joined object to be null

```
Query query = session.createQuery("from Person p left outer join p.address a ");
List<Object[]> peopleAddr = query.list();
```

Outer keyword is optional

Resultset:

```
Frank Brown   Chicago, Illinois
John Willis
Marry Doe     Los Angeles, California
```

John Willis is included even though his address reference was null

  - A right outer join allows the initial side to be null

```
Query query = session.createQuery("from Person p right outer join p.address a ");
List<Object[]> peopleAddr = query.list();
```

Outer keyword is optional

Resultset:

```
Frank Brown   Chicago, Illinois
Marry Doe     Los Angeles, California
              New York, New York
```

New York is included even though we don't have a person there

Joining a Collection

- You have to use explicit joins for collections:

```
Query query = session.createQuery(
                 "from Person p join p.numbers n where n like :number");
query.setParameter("number", "641%");
List<Person> people = query.list();
```

- If you don’t include a select clause you get:

Person	Number
firstName: John, lastName: Brown	type: home, number: 641-472-1234
firstName: John, lastName: Brown	type: mobile, number: 641-919-1234
firstName: Alice, lastName: Long	type: mobile, number: 641-233-1234
firstName: Alice, lastName: Long	type: home, number: 641 469-1234

Only Persons

- You have to use explicit joins for collections:

```
Query query = session.createQuery(
                 "select p from Person p join p.numbers n where n like :number");
query.setParameter("number", "641%");
List<Person> people = query.list();
```

- Although we only have persons objects, we still have duplicates:

Person
firstName: John, lastName: Brown
firstName: John, lastName: Brown
firstName: Alice, lastName: Long
firstName: Alice, lastName: Long

Distinct Keyword

- The distinct keyword removes duplicates

```
Query query = session.createQuery(
       "select distinct p from Person p join p.numbers n where n like :number");
query.setParameter("number", "641%");
List<Person> people = query.list();
```

Person
firstName: John, lastName: Brown
firstName: Alice, lastName: Long

Fetch Joins

- Another feature of Hibernate joins is the ability to eagerly fetch collections
  - This is done by outer joining the collection in the sql statement

- For instance if we want to pre-load all the accounts associated with a person

```
Query query = session.createQuery("from Person p left join fetch p.accounts where p.id = 1");
Person p = (Person)query.uniqueResult();
```

If you do not outer join, people without an account will be omitted

Loads person and also pre-caches all associated accounts

Join Fetch Considerations

- You can not constrain a join fetch-ed collection
  - E.g. “from Person p join fetch accounts a where a = …” is an invalid query
- When eager fetching, it is possible that duplicate initial objects occur due to the outer join
  - Duplicates can be removed by using ‘select distinct’
- Do not fetch more than one collection in parallel, doing so will create a Cartesian product*
- You can not use pagination with eager fetching

Hibernate Query Language:

THE SELECT CLAUSE

Selecting

- The select clause specifies which entities and or properties the query should return

A result can contain:

- Entities
- Properties
- A mix of the two

What we've done so far

```
@Entity
public class Book {
  @Id
  @GeneratedValue
  private int id;
  private String title;
  private String author;
  private double price;
  @Temporal(TemporalType.DATE)
  private Date publish_date;

  ...
```

Selecting a property:

Only selects the book title

```
Query query = session.createQuery("select b.title from Book b");
List<String> titles = query.list();
for (String title: titles) {
  System.out.println(title);
}
```

Returns a list of Strings

Selecting Multiple Items

- Using select you can also specify more than one entity and / or property
  - By default these will be returned as an Object[]

```
Query query = session.createQuery(
  "select person, pet.species, adr.city "
  + "from Pet pet join pet.owner person "
  + "join person.address adr ");
List<Object[]> items = query.list();

Person p = null; String petType = null; String city = null;
for (Object[] item : items) {
  p = (Person) item[0]; petType = (String) item[1]; city = (String) item[2];

  System.out.println(p.getFirstname() + " " + p.getLastname()
        + " owns a " + petType + " in " + city);
}
```

Select an entity and two properties

Result set is a List<Object[]>

  - Alternately you can have the result formatted as java.util.List, java.util.Map or even a new Object

Select – List()

- You can format a mixed result as a java.util.List instead of an Object[] using “new list()”

```
Query query = session.createQuery(
  "select new list(person, pet.species, adr.city) "
  + "from Pet pet join pet.owner person "
  + "join person.address adr ");
List<List<Object>> items = query.list();

Person p = null; String petType = null; String city = null;
for (List<Object> item : items) {
  p = (Person) item.get(0);
  petType = (String) item.get(1);
  city = (String) item.get(2);

  System.out.println(p.getFirstname() + " " + p.getLastname()
    + " owns a " + petType + " in " + city);
}
```

Select new list(… , … , …)

Returns a List<List<Object>>

Aliases and Maps

- The select clause also allows you to specify aliases for selected values
- This is only really useful when using Maps

new map(.., ..., ...)

```
Query query = session.createQuery(
  "select new map(p as person, sum(a.balance) as liquid) "
  + "from Person p join p.accounts a group by p.id ");
List<Map<String,Object>> items = query.list();

Person p = null; Double liquid = null;
for (Map<String,Object> item : items) {
  p = (Person)item.get("person"); liquid = (Double)item.get("liquid");

  System.out.println(p.getFirstname() + " " + p.getLastname()
      + "'s liquid assets: " + liquid);
}
```

Returns List<Map<String, Object>>

Aliased values become map keys

As New Object

- Results can even be formatted as new objects, provided the required constructor exists
  - Mapped classes can use their simple class name
  - Unknown classes need Fully Qualified Domain Name

```
Query query = session.createQuery(
  "select new entities.Home(p, a) "
  + "from Person p "
  + "join p.address a ");
List<Home> homes = query.list();

Person p = null; Address a = null;
for (Home home : homes) {
  p = home.getPerson();
  a = home.getAddress();

  System.out.println(p.getFirstname()
      + " " + p.getLastname()
      + " has a home in "+ a.getCity());
}
```

Fully Qualified Domain Name used: new **entities.Home**(p, a)

Returns a List<Home>

```
package entities;

public class Home {
  private Person person;
  private Address address;

  public Home(Person p, Address a) {
    this.person = p;
    this.address = a;
  }

  ...
```

No annotations or XML configuration, class is unknown to Hibernate

Needs to have the constructor used in the query

Aggregates

- HQL also provides aggregate functions:
  - avg(…), sum(…), min(…), max(…)
  - count(*), count(…), count(distinct …), count(all …)

Sum of all account balances

```
Query query = session.createQuery(
  "select new map(p as person, sum(a.balance) as liquid) "
  + "from Person p join p.accounts a group by p.id ");
```

- Group by clause specifies groups to aggregate

```
Query query = session.createQuery(
  "select new map(p as person, sum(a.balance) as liquid) "
  + "from Person p join p.accounts a group by p.id ");
```

Group by person.id – in other words: a group is all the accounts for each person

- The having clause can filter groups

```
Query query = session.createQuery(
  "select new map(p as person, sum(a.balance) as liquid) "
  + "from Person p join p.accounts a group by p.id "
  + "having sum(a.balance) > 100 ");
```

Only show groups with a balance greater than 100

Hibernate Query Language:

BULK UPDATES / DELETES

Bulk Operations

- HQL also supports bulk updates and deletes
  - Similar to the DML features of SQL

Bulk Update:

```
Query query = session.createQuery("update Account set balance = balance - :fee");
query.setParameter("fee", 5.0);
int updated = query.executeUpdate();
```

Apply a fee to all accounts

Bulk Delete:

```
Query query = session.createQuery("delete Book where publish_date < :date");
query.setParameter("date", df.parse("01/01/2002"));
int deleted = query.executeUpdate();
```

Delete all old books

Active Learning

- If a person can have many cars; write an HQL query that selects everyone that owns a car that has color = silver.

- Assuming we have a ‘Person’ **class** with a ‘name’ **property** that is mapped to `Student` **table** with a `FullName` **column**. What would be the HQL query to select everyone whose name starts with an ‘M’?

Module Summary

- In this module we covered the different aspects of the Hibernate Query Language
  - Overall HQL is very similar to SQL
  - The minimal requirement for a query is a from clause
  - The where clause can be used to refine the result set
  - You can implicitly or explicitly join other tables
  - Using the select clause you can define what should be returned and in which format it should be returned
  - HQL can also be used for bulk updates and deletes