Hibernate Query Language and Native SQL

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/hibernate.html

Topics in This Section

- Spend some time learning about the Hibernate Query Language, and how to leverage it to write database queries
- Prepare ourselves for cases where we need to write our own SQL by understanding how to accomplish its execution through Hibernate

The Hibernate Query Language

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/hibernate.html

Hibernate Query Language (HQL)

- **Similar to SQL**
  - Object based. Instead of tables and columns, syntax includes objects and attributes
- **Understands inheritance**
  - Can issue a query using a superclass or interface
- **Hibernate engine may turn one HQL statement into several SQL statements**
  - Does not allow for SQL database hints
- **Bypasses any object caches, such as the persistence context or 2nd Level Cache**

org.hibernate.Query

- **Main class used for building and executing HQL**
- **Similar to a JDBC prepared statement**
  - Bind parameter values
    - setLong(), setString(), setDate() etc…
    - setParameter();
      - Generic way of binding variables
  - Submit Requests
    - list() ;
      - Execute, and return a collection of result objects
    - uniqueResult();
      - Execute and return a single result object
- **Created using the Hibernate Session**

Basic Object Queries

```
// return all CheckingAccounts
Query getAllCheckingAccounts =
  session.createQuery("from CheckingAccount");

List checkingAccounts = getAllCheckingAccounts.list();


// return all Account types
Query getAllAccounts =
  session.createQuery("from Account");

List accounts = getAllAccounts.list();


// return ALL object types
Query getAllAccounts =
  session.createQuery("from java.lang.Object");

List objects = getAllObjects.list();
```

Does not require a select clause, just the object class name

Binding Query Parameters

- **Position-based**
  - Just like JDBC
  - Set parameters in an ordered fashion, starting with zero
- **Name-based**
  - Use names as placeholders
  - Set parameters by name
- **Pros/Cons**
  - Position-based faster on executing variable substitution
  - Name-based doesn't require code changes if a new parameter gets added in the middle of the statement

Position-Based Parameters

```
// return all Accounts based on
// balance and creation date
String query = "from Account a where"
             + " a.balance > ?"
             + " and a.creationDate > ?";

// deprecated. for demo only
Date date = new Date(2008, 12, 01);

Query getAccounts = session.createQuery(query)
  .setLong(0, 1000)
  .setDate(1, date);

List accounts = getAccounts.list();
```

Can alias objects, just like in SQL

Can set parameters in order, just like a JDBC PreparedStatement

Name-Based Parameters

```
// return all Accounts based on
// balance and creation date
String query = "from Account a where"
             + " a.balance > :someBalance"
             + " and a.creationDate > :someDate";

// deprecated. for demo only
Date date = new Date(2008, 12, 01);

// order doesn’t matter
Query getAccounts = session.createQuery(query)
  .setDate("someDate", date)
  .setLong("someBalance", 1000);

List accounts = getAccounts.list();
```

Setting Parameters Generically

```
// return all Accounts based on
// balance and creation date
String query = "from Account a where"
            + " a.balance > :someBalance"
            + " and a.creationDate > :someDate";

// deprecated. for demo only
Date date = new Date(2008, 12, 01);

// order doesn't matter.
// Temporal (time) values need to be specified
Query getAccounts = session.createQuery(query)
  .setParameter("someBalance", 1000)
  .setParameter("someDate", date, Hibernate.DATE);

List accounts = getAccounts.list();
```

Binding by Object

- **Name-based binding accepts an entire object for setting query parameters**
  - Placeholder names must match object attribute names
  - Hibernate uses reflection/java bean properties to map the attributes
- **Doesn't work with temporal data types**
  - Like Date

Binding by Object

```
// return all Accounts based on
// balance and creation date
String query = "from EBill e where"
            + " e.balance > :balance"
            + " and e.ebillerId > :ebillerId";

EBill queryParams = new EBill();
queryParams.setBalance(1000);
queryParams.setEbillerId(1);

// this will use java bean properties/reflection
// to bind the variables
Query getEBills = session.createQuery(query)
  .setProperties(queryParams);

List accounts = getEBills.list();
```

Assume an object with attribute names that matched the placeholder names...

...pass that object in to set the parameter values

Pagination

- **Break up large result sets into smaller groups (pages)**
  - setFirstResults(int startRow);
    - Set the starting record position
    - Zero-based indexing
  - setMaxResults(int numberToGet);
    - Set the number of records to retrieve
- **Keep track of current index in order to continue paging data through the data**

Pagination

```
// retrieve initial page, up to 50 records
Query getAccountsPage1 =
  session.createQuery("from Account")
    .setMaxResult(50);

...

// retrieve subsequent pages, passing
// in the first record to start with
Query getAccountsNextPage =
  session.createQuery("from Account")
    .setFirstResult(:startingIndex)
    .setMaxResult(50);
```

Setting Timeout

- **Set the time allowed for a specified query to execute**
  - setTimeout(int second);
  - Hibernate will throw an exception if limit is exceeded
- **Based on the JDBC timeout implementation**

Setting Timeout

```
try {

  // retrieve accounts, allow 30 seconds
  Query getAccounts =
    session.createQuery("from Account")
      .setTimeout(30);

  List accounts = getAccountsPage1.list();
}
catch (HibernateException) {
  ...
}
...
```

Setting Fetch Size

- **Optimization hint leveraged by the JDBC driver**
  – Not supported by all vendors, but if available, Hibernate will user this to optimize data retrieval
- **Used to indicate the number of records expected to be obtained in a read action**
  – If paging, should set to page size

Setting Fetch Size

```
// retrieve initial page, up to 50 records
Query getAccountsPage1 =
  session.createQuery("from Account")
    .setMaxResult(50)
    .setFetchSize(50);

...

// retrieve subsequent pages, passing
// in the first record to start with
Query getAccountsNextPage =
  session.createQuery("from Account")
    .setFirstResult(:startingIndex)
    .setMaxResult(50)
    .setFetchSize(50);
```

Adding Comments to Query

- **Developer provided comments included in the log along with the Hibernate SQL statement**
  - setComment(String comment);
  - Need to enable 'user_sql_comments' in the Hibernate configuration
- **Assists in distinguishing user-generated queries vs. Hibernate-generated**
  - Also be used to explain query intention

Adding Comments to Query

```
// retrieve initial page, up to 50 records
Query getAccountsPage1 =
  session.createQuery("from Account")
    .setMaxResult(50)
    .setComment("Retrieving first page of
                 Account objects");

...

// retrieve subsequent pages, passing
// in the first record to start with
Query getAccountsNextPage =
  session.createQuery("from Account")
    .setFirstResult(:startingIndex)
    .setMaxResult(50)
    .setComment("Retrieving page: " + pageNum);
```

Combining Settings

- Settings can be combined together on a single query
- Set on individual queries, not across all HQL queries

Combined Settings

```
Query getAccountPage1 =
  session.createQuery("from Account")
    .setMaxResult(50)
    .setFetchSize(50)
    .setTimeout(60)
    .setComment("Retrieving all account objects");

List accounts = getAccounts.list();

...

Query getAccountNextPage =
  session.createQuery("from Account")
    .setFirstResult(:startingIndex)
    .setMaxResult(25)
    .setFetchSize(25)
    .setTimeout(30)
    .setComment("Retrieving page " + pageNum);
```

Externalizing Queries

- **Define queries in object mapping files**
- **Can be 'global' or included inside class definition**
  - If inside class definition, need to prefix with fully qualified class name when calling
- **Isolates the SQL statements**
  - Useful if you want to modify all queries
    - Optimize queries
    - Switch vendors
    - May not require recompiling code

External: Global

```
<hibernate-mapping>
  <class name="courses.hibernate.vo.Account"
         table="ACCOUNT">
    <id name="accountId" column="ACCOUNT_ID">
      <generator class="native" />
    </id>
    <property name="creationDate" column="CREATION_DATE"
              type="timestamp"    update="false" />
    <property name="accountType" column="ACCOUNT_TYPE"
              type="string"      update="false" />
    <property name="balance" column="BALANCE"
              type="double" />
  </class>
  <query name="getAllAccounts" fetch-size="50"
        comment="My account query" timeout="30">
     <![CDATA[from Account]]>
   </query>
</hibernate-mapping>
```

External: Inside Class

```
<hibernate-mapping>
  <class name="courses.hibernate.vo.Account"
         table="ACCOUNT">
    <id name="accountId" column="ACCOUNT_ID">
      <generator class="native" />
    </id>
    <property name="creationDate" column="CREATION_DATE"
              type="timestamp"    update="false" />
    <property name="accountType" column="ACCOUNT_TYPE"
              type="string"      update="false" />
    <property name="balance" column="BALANCE"
              type="double" />
    <query name="getAccountByBalance"   fetch-size="50"
           comment="Get account by balance"
           timeout="30">
           <![CDATA[from Account where
                    balance=:balance]]>
    </query>
  </class>
</hibernate-mapping>
```

Calling Externalizing Queries

```
// globally named query
Query getAccounts =
  session.getNamedQuery("getAllAccounts")

List accounts = getAccounts.list();

...

// defined within class definition
Query getAccountByBalance =
  session.getNamedQuery(
    "courses.hibernate.vo.Account.getAccountByBalance")
  .setParameter("someBalance", 1000)

List accounts = getAccountByBalance.list();
```

Specifying Order

```
...

Query getAccounts =
   session.createQuery("from Account
  order by balance desc, creationDate
  asc")

List accounts = getAccounts.list();

...
```

Specifying Columns

- **Requires the use of the ‘select’ keyword**
- **Returns a list of object arrays**
  - Each index in the list contains an object array of the values for that row
  - Within each object array, columns are ordered as listed
    - Index 0 is the first identified column
    - Index 1 is the second identified column
    - Index n-1 is the nth identified column
- **Loop through the returned list of returned row column objects**

Specifying Columns

```
Query getAccountInfo = session.createQuery(
  "select accountId, balance from Account");

// get a list of results, where each result is
// an object array representing one row of data
List listOfRowValues = getAccountsInfo.list();

// for each object array...
for (Object[] singleRowValues : listOfRowValues) {
  // ...pull off the accountId and balance
  long accountId = (Long)singleRowValues[0];
  double balance = (Double)singleRowValues[1];
}
```

Using SQL/Database Functions

```
Query getAccountOwners =
  session.createQuery(
  "select upper(lastName),
          lower(firstName),
          sysdate
          from AccountOwner");
```

Performing Joins

- Implicit association join
- Ordinary join in the from clause
- Fetch join in the from clause
- Theta-style join in the where clause

Implicit Association Join

- Leverages the associations identified in the object's mapping file to figure out what SQL needs to be generated
- Uses dot notation to access the associated object in the query
- Only works for a single association reference
  - Does not work against collections of objects

Implicit Association Join

- Search for EBills by the name of the EBiller, through the EBill object

```
Query getVisaCardEbills =
  session.createQuery(
  "from EBill ebill where
   ebill.ebiller.name like '%VISA%' "

List ebills = getVisaCardEbills.list();
```

EBill issued from EBiller

```
<!-- EBill Mapping -->
<class name="courses.hibernate.vo.EBill" table="EBILL">
  ...
  <many-to-one name="ebiller" column="EBILLER_ID"
               class="courses.hibernate.vo.EBiller"/>
</class>
```

```
<!-- EBiller Mapping->
<class name="courses.hibernate.vo.EBiller"
  table="EBILLER">
  ...
  <property name="name" column="NAME" type="string" />
  ...
</class>
```

Ordinary Join

- **Join object types in the statement’s 'from' clause, bringing back all associated objects, or just specified ones**
- **Returns a list of a single object type, or an array of objects containing returned types**
  - For single object type, use the ‘select’ clause
  - For multiple types, returns a list of objects arrays
    - For repeated items, uses copies of object *references*, not *instances*
- **Works for collections of associated objects**

Ordinary Join

```
Query getVisaCardEbills =
  session.createQuery(
  "from EBill ebill
   join ebill.ebiller ebiller
   where ebiller.name like '%VISA%' "

// get a list of results, where each result is
// an object array representing one row of data
List listOfRowValues = getVisaCardEbills.list();

// returns BOTH object types
for (Object[] singleRowValues : listOfRowValues) {
  // ...pull off the EBill and EBiller
  EBill ebill = (EBill)singleRowValues[0];
  EBiller ebiller = (EBiller)singleRowValues[1];
  ...
}
```

EBill issued from EBiller

```
<!-- EBill Mapping -->
<class name="courses.hibernate.vo.EBill" table="EBILL">
  ...
  <many-to-one name="ebiller" column="EBILLER_ID"
               class="courses.hibernate.vo.EBiller"/>
</class>
```

```
<!-- EBiller Mapping->
<class name="courses.hibernate.vo.EBiller"
  table="EBILLER">
  ...
  <property name="name" column="NAME" type="string" />
  ...
</class>
```

Ordinary Join – Return One Type

```
Query getVisaCardEbills =
  session.createQuery(
    "select ebill from EBill ebill
    join ebill.ebiller ebiller
    where ebiller.name like '%VISA%' "

  List visaBills =
    getVisaCardEbills.list();
```

Ordinary Join – Collections

```
Query getVisaCardEbills =
  session.createQuery(
  "from EBiller ebiller
   join ebiller.ebills ebill
   where ebill.balance > 500"

// get a list of results, where each result is
// an object array representing one row of data
List listOfRowValues = getVisaCardEbills.list();

// go through the rows of object arrays
for (Object[] singleRowValues : listOfRowValues) {
  // ...pull off the EBiller and EBill
  EBiller ebiller = (EBiller)singleRowValues[0];
  EBill ebill = (EBill)singleRowValues[1];
  ...
}
```

Left Outer Joins

- Bring back all items of the 'left' side of a relationship, even if there is no matching 'right' side
  - If there IS a matching right side, bring that back too
  - Returns all objects in an object array per row
- Returns all objects in an object array per row
- Used for eager loading of objects

AccountTransactions may have EBills

```
<!-- EBill Mapping -->
<class name="courses.hibernate.vo.EBill" table="EBILL">
  ...
  <many-to-one name="accountTransaction"
    class="courses.hibernate.vo.AccountTransaction"
    column="ACCOUNT_TRANSACTION_ID"/>
</class>

<!-- AccountTransaction Mapping -->
<class name="courses.hibernate.vo.AccountTransaction"
      table="ACCOUNT_TRANSACTION">
  ...
  <one-to-one name="ebill"
    class="courses.hibernate.vo.EBill"
    property-ref="accountTransaction" />
</class>
```

Left Outer Join

```
Query getEBills =
  session.createQuery("from EBill ebill
   left join ebill.accountTransaction where
   ebill.balance > 500";

List listOfRowValues = getDebitTransactions.list();

for (Object[] singleRowValues : listOfRowValues) {
  // pull off the EBill
  EBill ebill = (EBill)singleRowValues[0];

  // we may or may not have an AccountTransaction.
  // if no related AccountTransaction, value is null
  AccountTransaction atx =
    (AccountTransaction)singleRowValues[1];

...
}
```

Fetch Join

- **Return a single object type with specified associations fully initialized**
- **Results in fewer, more optimized, SQL statements**
- **Used for eager loading or objects**
- **Never fetch more than one collection in parallel**
  - Will result in a Cartesian product
  - Can fetch many single-valued associations

Fetch Join

```
Query getEBills =
  session.createQuery("from EBill ebill
  join fetch ebill.accountTransaction where
  ebill.balance > 500";

List listOfRowValues = getDebitTransactions.list();

for (Object[] singleRowValues : listOfRowValues) {
  // pull off the EBill
  EBill ebill = (EBill)singleRowValues[0];

 // we may or may not have an AccountTransaction.
  // if no related AccountTransaction, value is null
  AccountTransaction atx =
    (AccountTransaction)singleRowValues[1];

  ...
}
```

Theta-Style Join

- **Join in a traditional SQL-like format**
- **Does not support outer joins**
- **Can join otherwise unrelated objects**
  - Objects not associated in mapping files

Theta-Style Join

```
Query getVisaCardEmployees =
  session.createQuery(
  "select owner
   from AccountOwner owner, EBiller ebiller
   where
     owner.cellPhone = ebiller.phone and
     ebiller.name like '%VISA%' "

  List visaEmployees =
    getVisaCardEmployees.list();
  ...
}
```

Aggregations

HQL Aggregation Functions

- Functions that operate against groups of resulting records
- Supported functions include:
  - count();
  - min();
  - max();
  - sum();
  - avg();

Count Function

```
Query countQuery =
  session.createQuery(
    "select count(ao) from
     AccountOwner ao "

long cnt =
  (Long)countQuery.uniqueResult();
```

Min, Max, and Avg Functions

```
Query accountStatsQuery =
  session.createQuery(
    "select min(a.balance), max(a.balance),
            avg(a.balance) from Account a");

List listOfRowValues = accountStatsQuery.list();

for (Object[] singleRowValues : listOfRowValues) {
  // pull off the values
  double min = (Double)singleRowValues[0];
  double max = (Double)singleRowValues[1];
  double avg = (Double)singleRowValues[2];
}
```

Group By and Having

- **Group subsets of returned results**
  - 'group by' clause, just like SQL
- **Restrict groups returned**
  - 'having' clause, also like SQL

Group By Aggregation

```
Query avgTxAmountPerAccountQuery =
  session.createQuery(
    "select atx.account.accountId,
            avg(atx.amount)
     from
            AccountTransaction atx
     group by
            atx.account.accountId");

List listOfRowValues =
  avgTxAmountPerAccountQuery.list();

for (Object[] singleRowValues : listOfRowValues) {
  // pull off the values
  long accountId = (Long)singleRowValues[0];
  double average = (Double)singleRowValues[1];
}
```

Having Aggregation Restriction

```
Query avgTxAmountPerAccountQuery =
  session.createQuery(
    "select atx.account.accountId,
            avg(atx.amount)
     from
            AccountTransaction atx
     group by
            atx.account.accountId
     having
            count(atx) > 20");

List listOfRowValues =
  avgTxAmountPerAccountQuery.list();

for (Object[] singleRowValues : listOfRowValues) {
  // pull off the values
  long accountId = (Long)singleRowValues[0];
  double average = (Double)singleRowValues[1];
}
```

Native SQL

Native SQL Queries

- **Write traditional SQL statements and execute them through the Hibernate engine**
  – Hibernate can handle the result set
- **Needed for very complicated queries or taking advantage of some database features, like hints**

Returning Scalar Values – All Columns

```
Query getEBills =
  session.createSQLQuery("SELECT * FROM EBILL");

List listOfRowValues =
    getEBills.list();

for (Object[] singleRowValues : listOfRowValues) {
  // returned in the order on the table
  long id = (long)singleRowValues[0];
  double balance = (balance)singleRowValues[1];
  ...
}
```

Return List of Objects

```
Query getEBills =
  session.createSQLQuery(
    "SELECT * FROM EBill")
  .addEntity(EBill.class);

List ebills =
    getEBills.list();
```

Returning Scalar Values – Projection

```
Query getScalarVariables =
  session.createSQLQuery(
    "SELECT E.EBILL_ID AS ID,
     EB.BALANCE AS BALANCE
     FROM EBILL EB")
  .addScalar("id", Hibernate.LONG)
  .addScalar("balance", Hibernate.DOUBLE);

List listOfRowValues =
  getScalarVariables.list();

for (Object[] singleRowValues : listOfRowValues) {
  long id = (Long)singleRowValues[0];
  double balance = (Double)singleRowValues[1];
}
```

Combining Scalars and Objects

```
Query getComboInfo =
  session.createSQLQuery(
    "SELECT
       E.EBILL_ID AS ID,
       EBLR.*
     FROM
       EBILL E, EBILLER EBLR")
  .addScalar("id", Hibernate.LONG)
  .addEntity("EBLR", EBiller.class);

List listOfRowValues = getComboInfo.list();

for (Object[] singleRowValues : listOfRowValues) {
  long id = (Long)singleRowValues[0];
  EBiller eblr = (EBiller)singleRowValues[1];
}
```

Wrap-up

Summary

- **Learned how to use HQL to execute queries by binding dynamic parameters and settings**
  - Named and position based binding
  - Paging, fetch-size, timeout, comments
- **Saw how to externalize our queries for maintenance purposes**
  - In mapping files globally, or within class definitions
- **Joins, Joins, Joins**
  - Implicitly; in from clause; with eager loading; traditional SQL-style
- **Aggregations:**
  - Grouping and Having
- **Native SQL**
  - Returning both scalar and object results

Preview of Next Sections

- Hibernate Advanced Features

Questions?