# Advanced Hibernate Features

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

## Topics in This Section

- Batch Processing
- Data Filtering
- Interceptors and Events
- Calling Triggers and Stored Procedures
- 2nd Level Cache
- Statistics
- DDL Generation
- Integration with Spring

## Batch Processing

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

## Batch Processing

- **When executing operations across large data sets, it is more optimal to run directly in the database (not in memory)**
  - Avoids loading potentially thousands of records into memory to perform the exact same action
- **In SQL, can be performed with 'Update' and 'Delete' commands**
  - `UPDATE ACCOUNT SET BALANCE=BALANCE*1.01;`
  - `DELETE FROM ACCOUNT;`

## Hibernate Batch Update and Delete

- **Data is modified directly in the database**
  - Changes made to database records are NOT reflected in any in-memory objects
  - Best to start with a clean persistence context, execute batch update, and THEN load any necessary objects
- **Can only be against a single object type**
- **Understands inheritance**
  - Batch made against a superclass/interface are executed against all subclasses
- **By default, does *not* affect any versioning columns (update only)**
  - Can execute in a fashion to update version numbers
    - `'versioned' keyword`

## Hibernate Batch Update

```
// Provide the monthly interest
// to savings accounts
Query q =
  session.createQuery(
    "update [versioned] Account set balance=
    (balance + (balance*interestRate))
     where accountType='SAVINGS' ");

// return number of objects updated
int updatedItems = q.executeUpdate();
```

## Hibernate Batch Delete

```
// Provide the monthly interest
// to savings accounts
Query q =
  session.createQuery(
    "delete from Account");

// return number of objects deleted
// across all subclasses
int deletedItems = q.executeUpdate();
```

## Hibernate Batch Inserts

- **Copy objects from one table to another**
  - Still modified directly in the database
- **Transfer object needs to be a concrete classes**
  - No superclasses or interfaces
- **Create a new object and mapping file to transfer the records**
  - Example
    - ArchivedAccount
      - Has its own mapping file, and its own table
      - Contains all the identical attributes of the Account class
      - Can obtain the ID from the copied object
      - If using versioning, copied record will start at zero if version column not included in select statement

## Hibernate Batch Insert

```
// Archive all existing accounts
Query q =
  session.createQuery(
    "insert into ArchivedAccount(
     accountId, creationDate, balance)
     select
     a.accountId, a.creationDate, a.balance
     from Account a");

int createdObjects = q.executeUpdate();
```

# Data Filtering

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

## Data Filtering

- **Limit the amount of data visible without modifying query parameters**
- **Often used for security purposes**
  - Users often only have access to certain levels of information
- **Similar to label security in the database**

## Setting up Data Filters

1. **Define the filter within the mapping file of the targeted entity**
   - Identify the attributes to filter on, and their types
2. **Apply the filter on the desired class or collection by indicating it within the <class> or <*collection-type*> tags**
3. **After obtaining a session with which to perform your actions, enable the appropriate filter, setting any applicable parameters**

## Account Class Filter

```
<class name="courses.hibernate.vo.Account"
  table="ACCOUNT">
  <id name="accountId" column="ACCOUNT_ID">
    <generator class="native" />
  </id>
  ...
  <filter name="creationDateFilter"
          condition="CREATION_DATE > :asOfDate"/>
</class>

<filter-def name="creationDateFilter">
  <filter-param name="asOfDate" type="date" />
</filter-def>
```

## Account Class Filter Test

```
Session session = HibernateUtil
 .getSessionFactory().getCurrentSession();

session.beginTransaction();

session.enableFilter("creationDateFilter")
  .setParameter("asOfDate",
    new Date(2008,12,08));

List accounts = accountService.getAccounts();
Assert.assertEquals(2, accounts.size());

session.disableFilter("creationDateFilter");

accounts = accountService.getAccounts();

Assert.assertEquals(5, accounts.size());
```

## Account Collection Filter

```
<class name="courses.hibernate.vo.Account"
  table="ACCOUNT">
  <id name="accountId" column="ACCOUNT_ID">
    <generator class="native" />
  </id>
  ...
  <set name="accountTransactions" inverse="true">
    <key column="ACCOUNT_ID" not-null="true"/>
    <one-to-many
      class="courses.hibernate.vo.AccountTransaction"/>
    <filter name="transactionDateFilter"
            condition="TXDATE > :asOfDate" />
  </set>
</class>

<filter-def name="transactionDateFilter">
  <filter-param name="asOfDate" type="date" />
</filter-def>
```

## Account Collection Filter Test

```
session.enableFilter("transactionDateFilter")
  .setParameter("asOfDaate", new Date(2008,12,08));

SortedSet accountTransactions =
  account.getAccountTransactions();
Assert.assertEquals(2,
  accountTransactions.size());

// Need to evict object from cache!
session.evict(account);

session.disableFilter("transactionDateFilter");

accountTransactions =
  account.getAccountTransactions();
Assert.assertEquals(3, accountTransactions2.size());
```

## Interceptors and Events

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

## Interceptors and Events

- **Callbacks that fire based on actions of a processing request**
- **Assists with separation of concerns**
  - "Non-business" processing
    - Auditing/logging

## Interceptors

- **Callbacks from the session allowing the application to inspect and/or manipulate properties of a persistent object**
  - Before it is saved, updated, deleted or loaded
- **Implemented one of two ways**
  - Implement Interceptor directly
  - Extend EmptyInterceptor (preferred)
- **Comes in two flavors**
  - Session-scoped
    - Specified when a session is opened
    - SessionFactory.openSession(Interceptor)
  - SessionFactory-scoped
    - Registered on the configuration during factory creation
    - Applies to all sessions

## Creating Interceptors

1. **Extend the EmptyInterceptor class**
2. **Implement the desired callback methods**
   1. afterTransactionBegin(…)
   2. afterTransactionCompletion (…)
   3. onSave (…)
   4. preFlush(…)
   5. postFlush(…)
   6. etc...
3. **Configure the interceptor use**
   - Either during factory creation
   - After obtaining a session

## Account Date Interceptor

```
public class AccountDateInterceptor
    extends EmptyInterceptor {

  public boolean onSave(Object entity,
      Serializable id, Object[] state,
      String[] propertyNames, Type[] types) {

    if (entity instanceof Account) {
      for (int i = 0; i < propertyNames.length; i++) {
        if (propertyNames[i]
              .equalsIgnoreCase("creationDate")) {
          state[i] = new Date();
          return true;
        }
      }
    }
    return false;
  }
}
```

## Setting an Interceptor

```
// when creating the SessionFactory.
// causes interception on ALL sessions
SessionFactory sessionFactory =
  Configuration().setInterceptor(
  new AccountDateInterceptor())
    .configure().buildSessionFactory();
```

---

```
// set when opening an individual session
Session session =
  HibernateUtil.getSessionFactory()
  .openSession(new AccountDateInterceptor());
```

## Events

- Can be used in addition to/or replacement of interceptors
- Triggered by extending default Hibernate implementations or implementing interfaces

| DEFAULT LISTENERS | INTERFACES |
|---|---|
| DefaultDeleteEventListener | DeleteEventListener |
| DefaultEvictEventListener | EvictEventListener |
| DefaultLoadEventListener | LoadEventListener |
| DefaultLockEventListener | LockEventListener |
| DefaultMergeEventListener | MergeEventListener |
| DefaultPersistEventListener | PersistEventListener |
| DefaultSaveOrUpdateEventListener | SaveOrUpdateEventListener |
| etc... | etc… |

## Creating Events

1. **Create a listener class in one of two ways**
   - Implementing the desired Hibernate listener interface
   - Extending an already existing Hibernate default listener
2. **List the listener class in the hibernate.cfg.xml**
   - If not extending, also list the default Hibernate event listener
     - Hibernate uses these too!

## Implementing a Listener

```
public class AccountTransactionDateEventListener
    implements SaveOrUpdateEventListener {

  // this method gets fired on a save or update
  public void onSaveOrUpdate(SaveOrUpdateEvent
      saveOrUpdateEvent) throws HibernateException {

    // check the object type passed in on the event
    if (saveOrUpdateEvent.getObject()
        instanceof AccountTransaction) {
      // if it's an accountTransaction, set the date
      AccountTransaction at = (AccountTransaction)
           saveOrUpdateEvent.getObject();
      at.setTransactionDate(new Date());
    }
  }
}
```

## Configure the SessionFactory

```
<hibernate-configuration>
  <session-factory>
  ...
    <event type="save-update">
      <listener class="courses.hibernate.util
       .AccountTransactionDateEventListener"/>

      <listener class="org.hibernate.event.def
       .DefaultSaveOrUpdateEventListener"/>
    </event>
  </session-factory>
</hibernate-configuration>
```

## Extending an Existing Listener

```
public class AccountTransactionDateEventListener
    extends DefaultSaveOrUpdateEventListener {

  // this method gets fired on a save or update
  public void onSaveOrUpdate(SaveOrUpdateEvent
      saveOrUpdateEvent) throws HibernateException {

    // check the object type passed in on the event
    if (saveOrUpdateEvent.getObject()
        instanceof AccountTransaction) {
      // if it’s an accountTransaction, set the date
      AccountTransaction at = (AccountTransaction)
           saveOrUpdateEvent.getObject();
      at.setTransactionDate(new Date());
    }
    super.onSaveOrUpdate(saveOrUpdateEvent);
  }
}
```

## Configure the SessionFactory

```
<hibernate-configuration>
  <session-factory>
  ...
    <event type="save-update">
      <listener class="courses.hibernate.util
       .AccountTransactionDateEventListener"/>
    </event>
  </session-factory>
</hibernate-configuration>
```

## Calling Triggers and Stored Procedures

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

## Triggers and Stored Procedures

- **Leveraging triggers in your database happens outside of Hibernate's knowledge**
  – Sets data on rows after Hibernate actions
  – Need to be able to obtain those set values
- **Call database stored procedures through Hibernate code**
- **Setup stored procedures as Hibernate's way of executing normal processes**

## Triggers

- **Identify columns that are modified automatically by the database in the object mapping file**
  – generated="insert | always“
  – Also need to tell Hibernate NOT to insert or update these columns, as appropriate
- **If an entity possesses columns identified to be populated by the database, Hibernate will re-read the object as appropriate**
  – For insert, after the insert statement is executed
  – For always, after insert or update statements

## Setting up Triggers

```
<class name="courses.hibernate.vo.EBill" table="EBILL">
 ...
  <!-- Causes a re-fetch upon insertion -->
  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"
            insert="false"
            update="false"
            generated="insert" />

  <!-- Causes a re-fetch upon insertion and update -->
  <property name="updateDate" column="UPDATE_DATE"
            type="timestamp"
            insert="false"
            update="false"
            generated="always" />
  ...
</class>
```

## Calling Stored Procedures

- **For querying, similiar syntax and process as named sql-query**
  - Defined inside or outside the class tags in the mapping file
  - If returning a value, can set an alias and return type
- **For insert, update, or delete, must be defined inside the class tag**
  - Overrides the default implementation for those events
  - Column order is random and unintuitive
    - Documentation says to look at the SQL log output to see what order Hibernate lists the columns
- **Must set the 'callable' attribute**

## Stored Procedure Setup

```
<class name="courses.hibernate.vo.EBill" table="EBILL">
  ...
  <!-- Calling procedure to execute the insert -->
  <sql-insert callable="true" check="param">
    {call create_ebill(?, ?, ?, ?, ?, ?,?, ?, ?, ?, ?)}
  </sql-insert>
</class>

<!-- named SQL query, but with callable
     and return value set -->
<sql-query name="getEbills" callable="true">
  <return alias="ebill"
          class="courses.hibernate.vo.EBill"/>
  { ? = call get_ebills() }
</sql-query>
```

## 2nd Level Cache

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

## 2nd Level Cache

- **Performance increase for objects with a much greater READ to WRITE ratio**
- **Great for reference or immutable objects**
- **Not advised for**
  - Frequently changing data
  - Tables accessed from other applications
- **Requires a cache strategy and pluggable cache provider**

## Setting up Caching

- **Four caching strategies; each level increases performance and risk of stale data**
  - Transactional
    - Slowest of caching, but most risk free
  - Read-write
  - Nonstrict-read-write
  - Read-only
    - Fastest performance, but most risky.
    - Use if the data never changes
- **Four cache providers are built into Hibernate**
  - EHCache: Simple process scope cache in a single JVM
  - OSCache: Richer set of expiration policies and support
  - SwarmCache: Cluster cache, but doesn't suppor 'Query Cache'
  - JBoss Cache: Fully transactional, replicated clustering

## ehcache.xml

```
<ehcache>
  <diskStore path="java.io.tmpdir" />
  <defaultCache maxElementsInMemory="10000"
                eternal="false"
                timeToIdleSeconds="120"
                timeToLiveSeconds="120"
                overflowToDisk="true" />

  <!-- setup special rules for Account objects -->
  <cache name="courses.hibernate.vo.Account"
         maxElementsInMemory="1000"
         eternal="false"
         timeToIdleSeconds="300"
         timeToLiveSeconds="600"
         overflowToDisk="false" />
</encache>
```

## Configuring Hibernate Cache

```
<session-factory>
  ...
  <property name="cache.provider_class">
    org.hibernate.cache.EhCacheProvider
  </property>

  <property name="cache.use_second_level_cache">
    true
  </property>
  ...
</session-factory>
```

## Account Mapping File

```
<class name="courses.hibernate.vo.Account"
       table="ACCOUNT">
  <cache usage="read-write" />
  <id name="accountId" column="ACCOUNT_ID">
    <generator class="native" />
  </id>
  <discriminator column="ACCOUNT_TYPE" type="string" />
  <version name="version" column="VERSION"
           type="long"    access="field" />
  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"    update="false" />
  ...

</class>
```

## Statistics

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

## Hibernate Statistics

- **Hibernate maintains statistics on which objects were queried, and how often**
  - Enable statistics in the configuration file
    - hibernate.generate_statistics=true
- **Can be leveraged to determine usage patterns and better optimize performance**
- **Hibernate Interfaces**
  - *Statistics* for global information
  - *EntityStatistics* for info on Object Type
  - *QueryStatistics* for SQL and HQL queries

## Hibernate Statistics - Example

```
public static void main(String args[]) {
  Statistics stats =
    HibernateUtil.getSessionFactory().getStatistics();

  stats.setStatisticsEnabled(true);

  AccountServiceTest testCase = new
  AccountServiceTest();
  testCase.testCreateAccount();
  testCase.testDeleteAccount();
  testCase.testGetAccount();
  testCase.testUpdateAccountBalance();

  stats.logSummary();

  EntityStatistics accountStats =
    stats.getEntityStatistics(
      "courses.hibernate.vo.Account");
}
```

## Hibernate Statistics Output

```
2562 [main] INFO org.hibernate.stat.StatisticsImpl - Logging statistics....
2562 [main] INFO org.hibernate.stat.StatisticsImpl - start time: 1228714626244
2562 [main] INFO org.hibernate.stat.StatisticsImpl - sessions opened: 20
2562 [main] INFO org.hibernate.stat.StatisticsImpl - sessions closed: 20
2562 [main] INFO org.hibernate.stat.StatisticsImpl - transactions: 10
2562 [main] INFO org.hibernate.stat.StatisticsImpl - successful transactions: 10
2562 [main] INFO org.hibernate.stat.StatisticsImpl - optimistic lock failures: 0
2562 [main] INFO org.hibernate.stat.StatisticsImpl - flushes: 9
2578 [main] INFO org.hibernate.stat.StatisticsImpl - connections obtained: 10
2578 [main] INFO org.hibernate.stat.StatisticsImpl - statements prepared: 22
2578 [main] INFO org.hibernate.stat.StatisticsImpl - statements closed: 22
2578 [main] INFO org.hibernate.stat.StatisticsImpl - second level cache puts: 5
2578 [main] INFO org.hibernate.stat.StatisticsImpl - second level cache hits: 2
2578 [main] INFO org.hibernate.stat.StatisticsImpl - second level cache misses: 1
2578 [main] INFO org.hibernate.stat.StatisticsImpl - entities loaded: 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - entities updated: 1
2594 [main] INFO org.hibernate.stat.StatisticsImpl - entities inserted: 4
2594 [main] INFO org.hibernate.stat.StatisticsImpl - entities deleted: 4
2594 [main] INFO org.hibernate.stat.StatisticsImpl - entities fetched (minimize this): 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - collections loaded: 8
2594 [main] INFO org.hibernate.stat.StatisticsImpl - collections updated: 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - collections removed: 12
2594 [main] INFO org.hibernate.stat.StatisticsImpl - collections recreated: 12
2594 [main] INFO org.hibernate.stat.StatisticsImpl - collections fetched (minimize this): 8
2594 [main] INFO org.hibernate.stat.StatisticsImpl - queries executed to database: 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - query cache puts: 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - query cache hits: 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - query cache misses: 0
2594 [main] INFO org.hibernate.stat.StatisticsImpl - max query time: 0ms
```

## DDL Generation

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

## DDL Generation

- Hibernate provides a tool to automatically generate database objects based on a domain model, or a domain model based on an already existing database.
- hbm2ddl
- Used through ANT tasks or with Hibernate configuration

## Hibernate Configuration

```
<!-- in the Hiberante.cfg.xml file -->
<session-factory>
  <property name="hibernate.hbm2ddl.auto">
    create|create-drop
  </property>
  ...
</sessionFactory>
```

---

```
// programmatically
Configuration cfg =
  new Configuration().configure();
SchemaUpdate schemaUpdate =
  new SchemaUpdate(cfg);
schemaUpdate.execute();
```

# Hibernate and The Spring Framework

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

## Spring Integration

- **First rate support for Hibernate**
  – Many IOC convenience features
- **Basic setup**
  – Configure a Spring data source (as normal)
  – Configure a PropertiesFactoryBean to setup the Hibernate properties
  – Configure a Spring LocalSessionFactoryBean to wrap the Hibernate SessionFactory
  – Setup Transaction Management

## Spring Data Source

```
<bean id="dataSource" class="org.springframework
    .jdbc.datasource.DriverManagerDataSource">

  <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
  </property>
  <property name="url">
    <value>
      jdbc:oracle:thin:@localhost:1521:XE
    </value>
  </property>
  <property name="username">
    <value>lecture9</value>
  </property>
  <property name="password">
    <value>lecture9</value>
  </property>
</bean>
```

## Spring PropertiesFactoryBean

```
<bean id="hibernateProperties" class="org.springframework
      .beans.factory.config.PropertiesFactoryBean">
    <property name="properties">
    <props>
      <prop key="dialect">
        org.hibernate.dialect.Oracle10gDialect
      </prop>
      <prop key="connection.pool_size">1</prop>
      <prop key="show_sql">true</prop>
      <prop key="format_sql">false</prop>
      <prop key="current_session_context_class">
        thread
      </prop>
      <prop key="hibernate.transaction.factory_class">
        org.hibernate.transaction.JDBCTransactionFactory
      </prop>
    </props>
  </property>
</bean>
```

## Hibernate Session Factory

```
<bean id="sessionFactory"
  class="org.springframework
    .orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref local="dataSource" />
  </property>
  <property name="hibernateProperties">
    <ref bean="hibernateProperties" />
  </property>
  <property name="mappingResources">
    <list>
      <value>Account.hbm.xml</value>
      ......
    </list>
  </property>
</bean>
```

References the previously defined data source

References the previously defined PropertiesFactoryBean with the defined hibernate properties

List the mapping files

## Add Transaction Management

- **Allow Hibernate to delegate transaction management to the Spring container**
  - Declarative transactions
- **To setup:**
  - Configure a HibernateTransactionManager in Spring
  - Create DAO and service implementations that perform the business functionality
  - Setup a TransactionProxyFactoryBean to wrap the service target implementation

## Hibernate Transaction Manager

```
<bean id="txManager"
   class="org.springframework.orm.
   hibernate.HibernateTransactionManager">

<property name="sessionFactory"
          ref="sessionFactory" />
</bean>
```

Previously defined SessionFactory is injected into the Spring HibernateTransactionManager

## AccountService Target Class

- Define the DAO and service implementations, and inject the DAO into the service

```
<bean id="accountDAO"
  class="courses.hibernate.dao.AccountDAO">
  <constructor-arg ref="sessionFactory" />
</bean>

<bean id="accountServiceTarget"
  class="courses.hibernate
    .service.AccountService">

  <property name="accountDAO">
    <ref bean="accountDAO" />
  </property>
</bean>
```

## AccountService Transaction Proxy

```
<bean id="accountService" class="org.springframework
 .transaction.interceptor.TransactionProxyFactoryBean">

  <property name="transactionManager">
    <ref local="txManager" />
  </property>
  <property name="target">
    <ref local="accountServiceTarget" />
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Provide the Service Proxy with the previously defined 'transactionManager' and 'accountServiceTarget' implementations

'Declare' the methods as requiring transaction management

## AccountDAO

```
public class AccountDAO {
  private SessionFactory sessionFactory;
  public AccountDAO(SessionFactory factory) {
    sessionFactory = factory;
  }

  // excluding try/catch for space purposes
  public void saveAccount(Account account) {
      Session session =
        sessionFactory.getCurrentSession();
  }
  ...
}
```

SessionFactory is automatically injected into the AccountDAO during construction

## Spring Test Case

```
ClassPathResource resource = new
  ClassPathResource("applicationContext.xml");

beanFactory = new XmlBeanFactory(resource);

AccountService accountService =
  (AccountOwnerService)
  beanFactory.getBean("accountService");

...
// create a new account
...

// wrapped in a declared transaction
accountService.saveOrUpdateAccount(account);
```

## Wrap-up

## Summary

- **In this lecture, we covered a TON of advanced features**
  - Batch Processing
    - Great for changing many records in the same fashion
  - Data Filtering
    - Restrict data, possibly for security reasons
  - Interceptors and Events
    - Separation of concerns
  - Calling Triggers and Stored Procedures
    - When the database possesses its own functionality
  - 2nd Level Cache
    - Also for optimization
  - Statistics
    - See how we're doing – is this really buying us anything?
  - DDL Generation
    - Let Hibernate create and drop our tables
- **Integration with Spring**
  - Setting up configuration and injecting SessionFactory
  - Declarative Transactions

## Preview of Next Sections

- **Java Persistence API**

## Questions?