Hibernate API

Session 2

Version 1.2

September 2018

Agenda

- **Entity States**
- Interfaces in the Hibernate API
- Locking

Entity States

- Transient – Object constructed with the new operator that is not associated with any database row
- Persistent – An entity instance associated with a database row and contained within a persistence context
- Removed – An object scheduled for deletion
- Detached – Object reference to an entity once associated with a closed persistence context

Persistence Context

- Caches of all persistent entity instances
- Corresponds with a session

```
SessionFactory  factory = HibernateUtil.getSessionFactory();

Session session = factory.openSession();
   // perform operation on entities
session.close();
```

Persistence Context

CitiusTech

Agenda

- Entity States
- **Interfaces in the Hibernate API**
- Locking

Session Interface

- Session interface provides methods for lifecycle operations
- Following table describes the actions and their corresponding end state:

Initial State	Action	End State
None	▪ get() ▪ load()	▪ Persistent
Transient	▪ save() ▪ saveorUpdate()	▪ Persistent
Persistent	▪ delete()	▪ Removed
Detached	▪ update() ▪ saveorUpdate()	▪ Persistent
Persistent	▪ Evict()	▪ Detached

Saving Entities (1/2)

- Creating an instance of a class you map with Hibernate mapping does not automatically persist the object to the database until you save the object with a valid Hibernate session
- An object remains to be in “transient” state until it is saved and moved into “persistent” state
- The class of the object that is being saved must have a mapping file (myclass.hbm.xml)
- The Hibernate API methods for saving objects are

```
•   public Serializable save(Object object)
•   public void save(Object object, Serializable id)
•   public Serializable save(String entityName, Object object)
```

- Session Interface schedules insert statements to create the new object in the database

Saving Entities (2/2)

- Following code snippet shows the use of save() method to persist bank object in the database:

```
Bank bank = new Bank(); // transient
//bank.setProperties();
SessionFactory  factory = HibernateUtil.getSessionFactory();
Session session = factory.openSession(); // Persistence Context 1
Transaction transaction = session.beginTransaction();
session.save(bank);
transaction.commit();
session.close(); // close Persistence Context 1
```

- save() result in an SQL INSERT

Loading Entities

- Used for loading objects from the database
- Each *load(..)* method requires object's primary key as an identifier
  - The identifier must be *Serializable* – any primitive identifier must be converted to object
- Requires which domain class or entity name to use to find the object with the id
- load() method will throw an exception if unique id is not found in the database
- If instance is not found in cache but it is there in the database, then it returns proxy containing only id value
  - Remaining proxy instance is fully initialized only when non-id attributes are accessed in persistent state
  - However, if remaining non-id attributes are accessed in 'detached' state it throws `ObjectNotFoundException`
- Java methods for loading objects:

```
•   public Object load(Class theClass, Serializable id)
•   public Object load(String entityName, Serializable id)
•   public void load(Object object, Serializable id)
```

Getting Entities

- Works like load() method
- Retrieves an object instance, or null if not found
- Retrieves fully initialized object instance and not the proxy
- Java methods for getting objects:
  - public Object get(Class theClass, Serializable id)
  - public Object get(String entityName, Serializable id)

- Following code snippet shows the use of load() methods in the application:

```
try {
 org.hibernate.Transaction transaction =
  session.beginTransaction();
  Bank bank = (Bank) session.load(Bank.class, 123L);
  System.out.println("Method Executed");
  System.out.println(bank.getName());
  transaction.commit();
} catch (Exception e) { e.printStackTrace(); }
. . .
```

Refreshing Objects

- Refreshes persistent object if it is not in sync with the database representation
- Gets the latest version from the database
- Scenarios you might want to use it:
  - If the data is getting modified by applications other than hibernate
  - SQL queries are directly applied on the database
  - Triggers are used to populate data in the database
- Java method for refreshing objects (from session interface)

```
public void refresh(Object object)
```

CitiusTech

Updating Entities

- Hibernate automatically manages any changes made to the persistent objects
  - The objects should be in “persistent” state and not in the transient state
- If a property changes on a persistent object, Hibernate session will perform the change in the database (when transaction commits possibly by queuing them first)
- From developer perspective, you do not have to any work to store these changes to the database
- You can force Hibernate to commit all of this changes using flush() method
- You can also determine if the session is dirty through isDirty() method
- Java method for updating objects (from session interface)

```
public void update(Object object)
```

- update() result in an **SQL UPDATE**
- Changes to persistent instances are detected at flush time and also result in an **SQL UPDATE**

Delete Entities

- Removes an object (means its associated row and persistent identity) from the database
- When delete method is called on POJO in session, it goes from persistent state to transient state
- The Java object is still alive, though deleted from the database and stays alive until developer sets to null, or goes out of scope
- Java method for deleting objects (from session interface)

```
public void update(Object object)
```

- delete() results in an SQL DELETE

Session Methods

- Similar to save() method, there are:
  - saveOrUpdate() and replicate() methods that result in either an **INSERT** or an **UPDATE**
- session.merge(Object o)
  - Retrieves a fresh version of the object from the database and based on that, as well as modifications made to the object being passed in, schedules update statements to modify the existing objects in the database
- session.evict(Object o)
  - Removes individual instances from persistence context, changing their state from persistent to detached

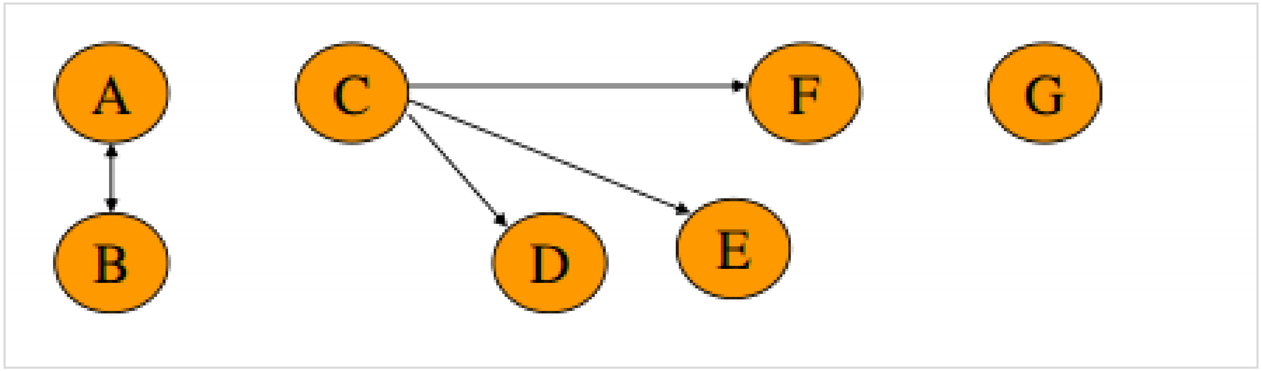
Difference Between Merge and Update

Both update() and merge() methods in hibernate are used to attach the object which is in detached state into persistence state

- Update():- if you are sure that the session does not contains an already persistent instance with the same identifier then use update to save the data in hibernate
- Merge():-if you want to save your modifications at any time without knowing about the state of an session then use merge() in hibernate

Cascading Object Persistence

- Hibernate represents domain object relationships through a graph model



- Propagate the persistence action not only to the object submitted, but also to any objects associated with that object

Cascade Attributes (1/2)

- Lifecycle operations can be cascaded
- Cascading configuration flags (specified in the mapping file)
  - **none**
    - Default behavior
  - **save-update**
    - Saves or updates associated objects
    - Associated objects can be transient or detached
  - **delete**
    - Deletes associated persistent instances
  - **delete-orphan**
    - Enables deletion of associated objects when they are removed from a collection
    - Enabling this tells hibernate that the associated class is not SHARED and can therefore be deleted when removed from its associated collection

Cascade Attributes (2/2)

- **lock**
  - Reattaches detached associated objects
- **replicate**
  - Replicates associated objects
- **evict**
  - Evicts associated objects from the persistence context
- **all**
  - Cascades everything, but delete-orphan

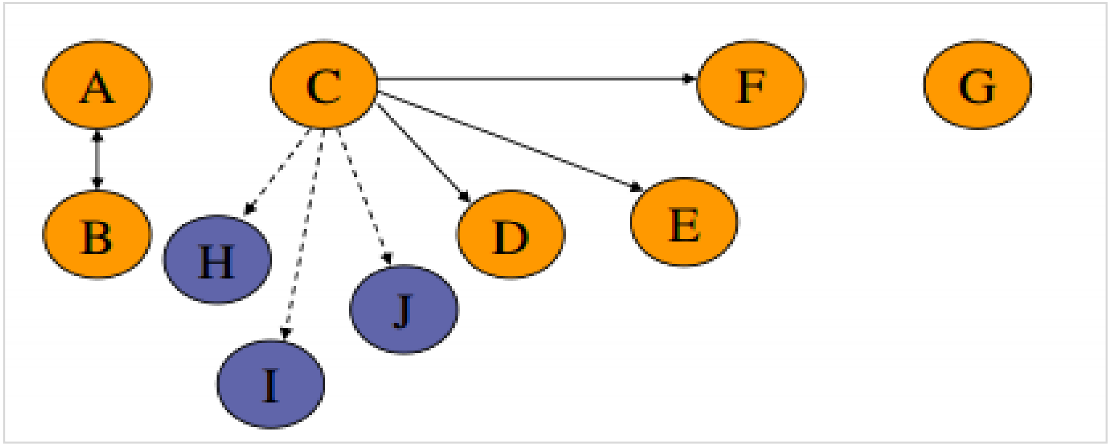
Cascading Operations Example

```
 <hibernate-mapping >
<class name="Event" table="events">
   <id name="id" column="uid" type="long">
       <generator class="increment"/>
   </id>
    <property name="name" type="string"/>
    <property name="startDate" column="start_date“
                                type="date"/>
     <property name="duration" type="integer"/>


    <many-to-one name="location" column="location_id"

    class="Location" cascade="save-update" />

</class>
</hibernate-mapping>
```

CitiusTech

Save-or-Update: Cascade

- With cascade, no need to persist the transient objects manually
- Hibernate will persist them automatically

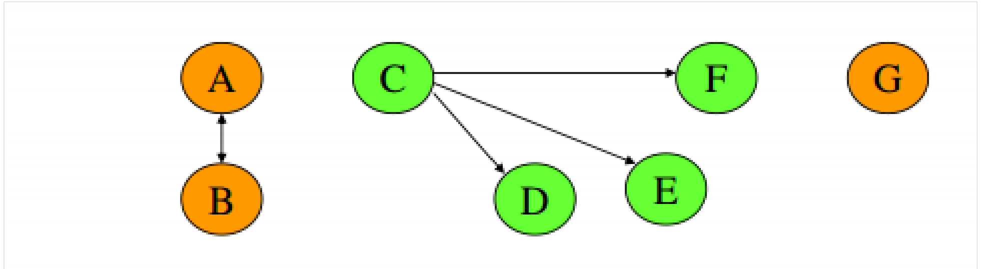


```
C.add(H);                    // add relationship with H
C.add(I);                    // add relationship with I
C.add(J);                    // add relationship with J
session.saveOrUpdate(C);     // save relationships through C
```

```
<set name="ebills" cascade="save-update"
    inverse="true">
  <key column="ACCOUNT_ID" not-null="true" />
  <one-to-many class="EBill" />
</set>
```

Delete: Cascade

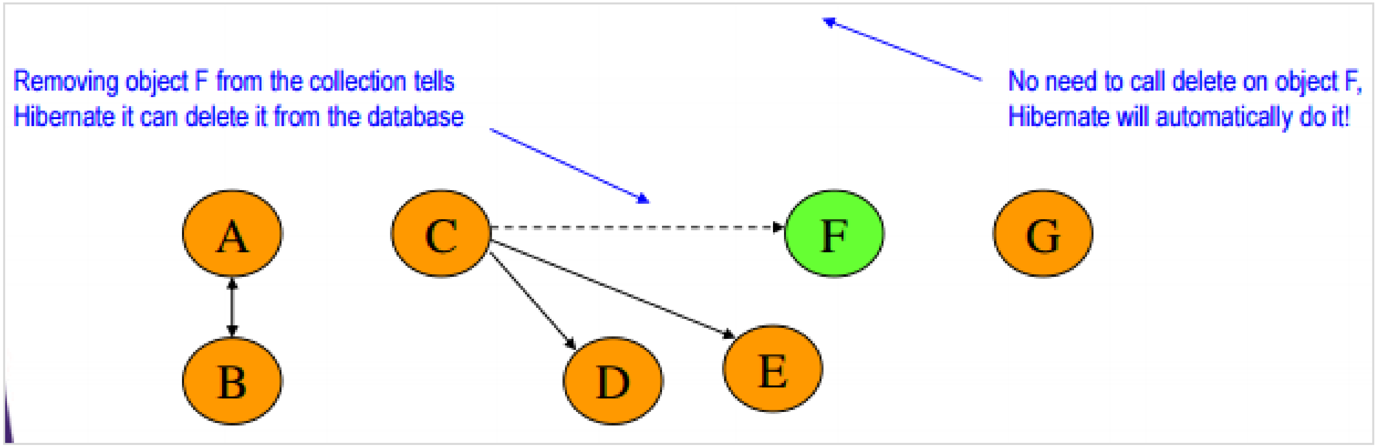
- Deleting an object and all its dependent's



```
session.delete(C);
```

Delete-orphan: Cascade

- Removing object from the collection tells Hibernate to delete from the database
- Example:
  - c.getBills().remove(f); // remove ebills from Account c
  - Session.saveorUpdate(c); // save current state results in deleting object f


Removing object F from the collection tells Hibernate it can delete it from the database
No need to call delete on object F, Hibernate will automatically do it!
A
B
C
D
E
F
G

Agenda

- Entity States
- Interfaces in the Hibernate API
- **Locking**

Locking

- Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used
- Hibernate does not lock objects in memory

Strategies

- Locking strategy can be either optimistic or pessimistic
- Optimistic:
  - Optimistic locking assumes that multiple transactions can complete without affecting each other
  - Transactions proceed without locking the data resources that they affect
  - Before committing, each transaction verifies that no other transaction has modified its data
  - If the check reveals conflicting modifications, the committing transaction rolls back
- Pessimistic:
  - Pessimistic locking assumes that concurrent transactions will conflict with each other
  - Resources are locked after they are read and only unlocked after the application has finished using the data

Optimistic Locking

- Used when the application uses long transactions or conversations that span several database transactions
- This approach guarantees some isolation, but scales well and works particularly well in Read-Often Write-Sometimes situations
- Hibernate provides two different mechanisms for this
  - The version number mechanism for optimistic locking is provided through a @Version annotation

    ```
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
    ```

  - Timestamps are a less reliable way of optimistic locking than version numbers, but can be used by applications for other purposes as well
  - Timestamping is automatically used if you use the @Version annotation on a Date or Calendar

Pessimistic Locking (1/2)

- Hibernate will always use the locking mechanism of the database; it never lock objects in memory
- The LockMode class defines the different lock levels that can be acquired by Hibernate. A lock is obtained by the following mechanisms:
  - LockMode.WRITE is acquired automatically when Hibernate updates or inserts a row
  - LockMode.UPGRADE can be acquired upon explicit user request using SELECT ... FOR UPDATE on databases which support that syntax
  - LockMode.UPGRADE_NOWAIT can be acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAIT under Oracle
  - LockMode.READ is acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. It can be re-acquired by explicit user request
  - LockMode.NONE represents the absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start in this lock mode

Pessimistic Locking (2/2)

- The "explicit user request" is expressed in one of the following ways:
  - A call to Session.load(), specifying a LockMode
  - A call to Session.lock()
  - A call to Query.setLockMode()
  - If you call Session.load() with option UPGRADE or UPGRADE_NOWAIT, and the requested object is not already loaded by the session, the object is loaded using SELECT ... FOR UPDATE
  - If you call load() for an object that is already loaded with a less restrictive lock than the one you request, Hibernate calls lock() for that object
  - Session.lock() performs a version number check if the specified lock mode is READ, UPGRADE, or UPGRADE_NOWAIT
  - In the case of UPGRADE or UPGRADE_NOWAIT, SELECT ... FOR UPDATE syntax is used
  - If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email univerct@citiustech.com

www.citiustech.com

CitiusTech