Object Lifecycle, Persistence, and Session Management

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

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

For live Spring & Hibernate training, see courses at http://courses.coreservlets.com/.

Taught by the experts that brought you this tutorial. Available at public venues, or customized versions can be held on-site at <u>your</u> organization.

- Courses developed and taught by Marty Hall
  – Java 5, Java 6, intermediate/beginning servlets/JSP, advanced servlets/JSP, Struts, JSF, Ajax, GWT, custom mix of topics
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  – Spring, Hibernate/JPA, EJB3, Ruby/Rails

Contact hall@coreservlets.com for details

Topics in This Section

- **Learn the lifecycle of Hibernate entities, when they transition from state–to–state, and how this affects development**
- **Take a closer look at persistence, and discover how Hibernate manages dirty and related objects**
- **See how Hibernate uses a Session to keep track of, and optimize, user activities**

4

Hibernate Object Lifecycle

- **Hibernate considers objects it can manage to always be in one of four states**
  - Transient
  - Persistent
  - Removed
  - Detached
- **Objects transition from state to state through various method calls**

Transient State

- **All objects start off in the transient state**
  - Account account = new Account();
    - account is a transient object
- **Hibernate is not aware of the object instance**
- **Not related to database row**
  - No value for accountId
- **Garbage collected when no longer referenced by any other objects**

Persistent State

- **Hibernate is aware of, and managing, the object**
- **Has a database id**
  - Already existing object retrieved from the database
  - Formerly transient object about to be saved
- **This is the only state where objects are saved to the database**
  - Modifications made in other states are NOT saved to the database while the object remains in that state
  - Changes to objects in a persistent state are automatically saved to the database without invoking session persistence methods
- **Objects are made persistent through calls against the Hibernate session**
  - session.save(account);
  - session.update(account);
  - session.lock(account);
  - session.merge(account);

Persistent State

```
Session session =
  SessionFactory.getCurrentSession();

// 'transient' state – Hibernate is NOT aware that it exists
Account account = new Account();

// transition to the 'persistent' state. Hibernate is NOW
// aware of the object and will save it to the database
session.saveOrUpdate(account);

// modification of the object will automatically be
// saved because the object is in the 'persistent' state
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();
```

Removed State

- **A previously persistent object that is deleted from the database**
  – session.delete(account);
- **Java instance may still exist, but it is ignored by Hibernate**
  – Any changes made to the object are not saved to the database
  – Picked up for garbage collection once it falls out of scope
    - Hibernate does *not* null-out the in-memory object

Removed State

```
Session session = SessionFactory.getCurrentSession();

// retrieve account with id 1.  account is returned in a 'persistent' state
Account account  = session.get(Account.class, 1);

// transition to the 'removed' state. Hibernate deletes the
// database record, and no longer manages the object
session.delete(account);

// modification is ignored by Hibernate since it is in the 'removed' state
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();

// notice the Java object is still alive, though deleted from the database.
// stays alive until developer sets to null, or goes out of scope
account.setBalance(1000);
```

Detached State

- **A persistent object that is still referenced <u>after</u> closure of the active session**
  - session.close() changes object's state from persisted to detached
- **Still represents a valid row in the database**
- **No longer managed by Hibernate**
  - Changes made to detached objects are not saved to the database while object remains in the detached state
  - Can be reattached, returning it to the persistent state and causing it to save its state to the database
    - update();
    - merge();
    - lock(); // reattaches, but does not save state

Detached State

```
Session session1 = SessionFactory.getCurrentSession();

// retrieve account with id 1. account is returned in a 'persistent' state
Account account  = session1.get(Account.class, 1);

// transition to the 'detached' state. Hibernate no longer manages the object
session1.close();

// modification is ignored by Hibernate since it is in the 'detached'
// state, but the account still represents a row in the database
account.setBalance(500);

// re-attach the object to an open session, returning it to the 'persistent'
// state and allowing its changes to be saved to the database
Session session2 = SessionFactory.getCurrentSession();
session2.update(account);

// commit the transaction
session2.getTransaction().commit();
```

Hibernate Lifecycle
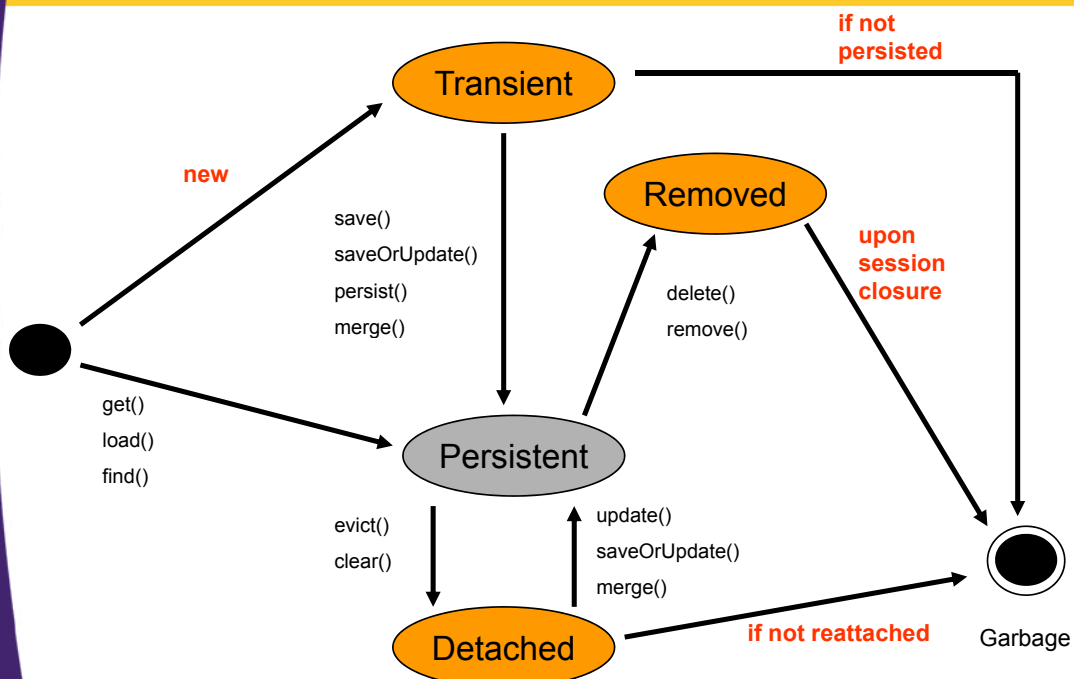
Saving Changes to the Database

- **Session methods do NOT save changes to the database**
  - save();
  - update();
  - delete();
- **These methods actually SCHEDULE changes to be made to the database**
- **Hibernate collects SQL statements to be issued**
- **Statements are later *flushed* to the database**
  - Once submitted, modifications to the database are not permanent until a commit is issued
    - session.getTransaction().commit();

The Persistence Context

- **Managed object environment**
  - No API or concrete object to reference, just conceptual
  - Thought of as containing:
    - Graph of managed persistent instances
    - List of SQL statements to send to the database
- **Each Hibernate session is said to have one 'persistence context'**
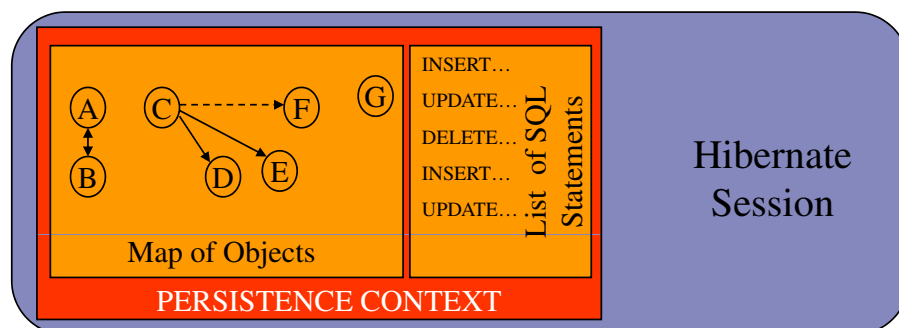
Flushing the Context

- **Submits the stored SQL statements to the database**
- **Occurs when:**
  - transaction.commit() is called
  - session.flush() is called explicitly
  - Before a query is executed
    - If stored statements would affect the results of the query
- **Flush modes**
  - session.setFlushMode()
    - FlushMode.AUTO: Default. Called as described above
    - FlushMode.COMMIT: Not flushed before queries
    - FlushMode.MANUAL: Only when explicit flush() is called

Hibernate Session

- **Single threaded non–shared object that represents a unit of work**
- **Used to retrieve objects from the database**
- **Contains the 'persistence context'**

- **Used to *schedule* changes to be made in the database**

Session API – Object Persistence

- ## session.save(Object o)
  - Schedules insert statements to create the new object in the database

- ## session.update(Object o)
  - Schedules update statements to modify the existing object in the database

Scheduled Changes

```
Session session = SessionFactory.getCurrentSession();

// 'transient' state – Hibernate is NOT aware that it exists
Account account = new Account();

// Transition to the 'persistent' state. Hibernate is NOW
// aware of the object and will save it to the database
// schedules the insert statements to create the object in the database
session.saveOrUpdate(account);

// modification of the object will automatically be
// saved scheduled because the object is in the 'persistent' state
// (actually alters the initial insert statement since it hasn't been sent yet)
account.setBalance(500);

// flushes changes to the database and commit the transaction
session.getTransaction().commit();
```

Session API –
Object Persistence

- **session.saveOrUpdate(Object o)**
  - Convenience method to determine if a 'save' or 'update' is required

- **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 object in the database.

Session API –
Object Retrieval

- **session.get(Object.class, Identifier)**
  - Retrieves an object instance, or null if not found

- **session.load(Object.class, Identifier)**
  - Retrieves an object instance but does NOT result in a database call
    - If managed instance not found, will return a *proxy*
      - Object fully initialized when non-id attribute is accessed
        » If 'detached', throws ObjectNotFoundException

Session API – Object Retrieval

- **session.lock(Object, LockMode)**
  - Reattaches a detached object to a session without scheduling an update
  - Also used to 'lock' records in the database

- **session.refresh(Object)**
  - Gets the latest version from the database

Session API – Other Methods

- **session.delete(Object)**
  - Schedule an object for removal from the database

- **session.evict(Object)**
  - Removes individual instances from persistence context, changing their state from persistent to detached

Session API – Other Methods

- **session.clear()**
  - Removes all objects from persistence context, changing all their states from persistent to detached

- **session.replicate(Object, ReplicationMode)**
  - Used for persisting records <u>across</u> databases
  - ReplicationModes
    - EXCEPTION: throw exception if row exists
    - IGNORE: don't create if already exists
    - LATEST_VERSION: choose the latest version to save
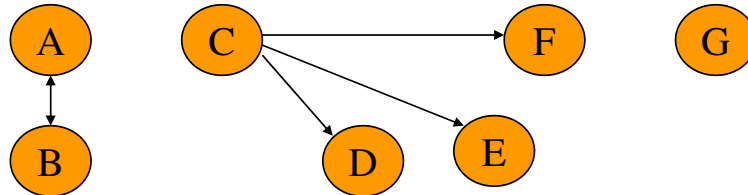   - OVERWRITE: overwrite existing row

Cascading

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 Attribute

- **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're removed from a collection
  - Enabling this tells Hibernate that the associated class is NOT SHARED, and can therefore be deleted when <u>removed from its associated collection</u>

Save-or-Update: Non-Cascade

- **Previously, to add relationships to new transient objects…**
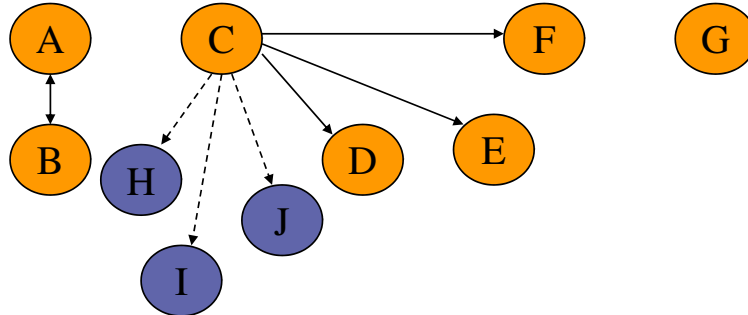


```
session.saveOrUpdate(H);      // make H persistent
session.saveOrUpdate(I);      // make I persistent
session.saveOrUpdate(J);      // make J persistent
C.add(H);                     // add relationship with H
C.add(I);                     // add relationship with I
C.add(J);                    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

Cascade="save-update"

```
<class name="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="balance" column="BALANCE"
            type="double" />

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

</class>
```
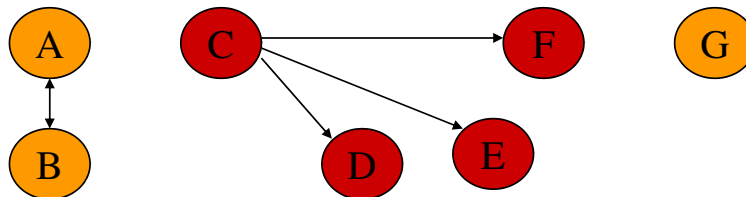
Save-or-Update: Cascade

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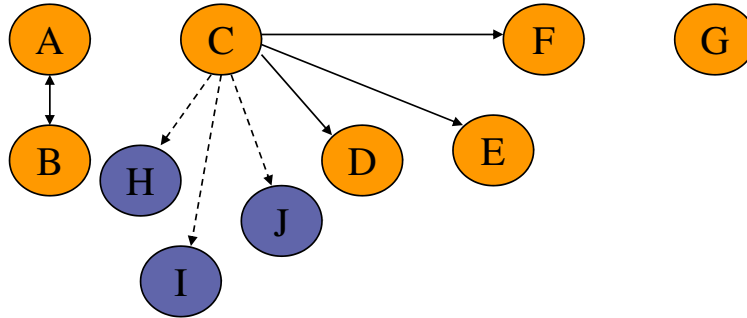

```
C.add(H);                    // add relationship with H
C.add(I);                    // add relationship with I
C.add(J);                    // add relationship with J
session.saveOrUpdate(C);     // save relationships through C
```
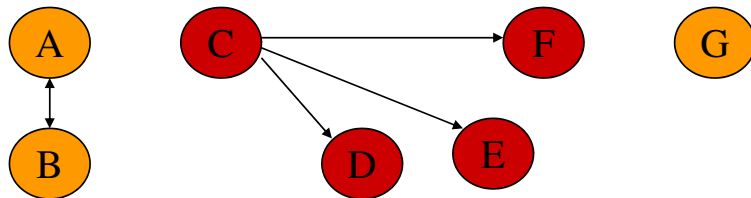
Save-or-Update: Non-Cascade

- **Previously, if several associated objects were modified…**


```
session.saveOrUpdate(C);
session.saveOrUpdate(D);
session.saveOrUpdate(E);
session.saveOrUpdate(F);
```
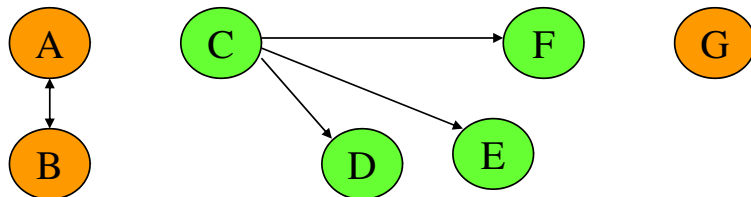
Save-or-Update: Cascade

- **Now, call one and let Hibernate cascade the action across associated objects…**


```
session.saveOrUpdate(C);
```

Delete: Non-Cascade

- **Before, deleting an object and its dependants required several calls…**


```
session.delete(C);
session.delete(D);
session.delete(E);
session.delete(F);
```

Cascade="delete"

```
<class name="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="balance" column="BALANCE"
            type="double" />

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

  <set name="ebillers" inverse="true" >
    <key column="EBILLER_ID" not-null="true" />
    <one-to-many class="EBiller" />
  </set>
</class>
```
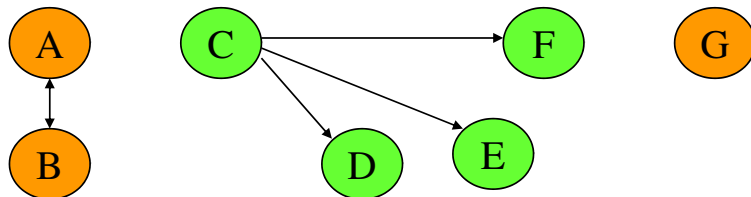
Delete: Cascade

- **Now, deleting an object and all of its dependants is much easier…**


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

Cascade="delete-orphan"

```xml
<class name="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="balance" column="BALANCE"
            type="double" />

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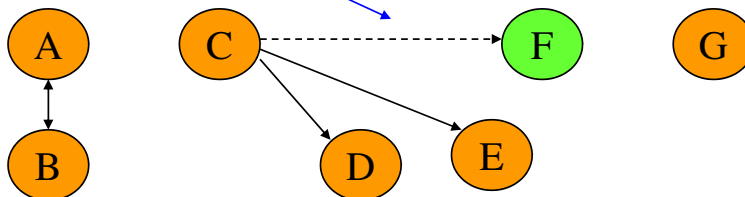
Cascade="delete-orphan"

```java
// remove EBill f from Account c
c.getEbills().remove(f);

// save current state of c (no longer referencing f).
// results in deleting f from database
// even without calling session.delete(f)
session.saveOrUpdate(c);
```

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 C F G

B D E

Other Cascade Settings

- **lock**
  - Reattaches detached associated objects
- **replicate**
  - Replicates associated objects
- **evict**
  - Evicts associated objects from the persistence context
- **all**
  - Cascades everything but delete-orphan
    - Can still include by setting cascade="all, delete-orphan"

Cascading Delete for M:M

- **Issue using M:M and setting cascade delete**
  - Delete Jack, tries to delete Account 1, which Jill still is a associated with – results in an error (if using database constraints)

OWNER_ID	NAME
1	JACK
2	JILL

OWNER_ID	ACCOUNT_ID
1	1
2	1

ACCOUNT_ID	BALANCE
1	100
2	500

- **Hibernate discourages it**
  - Recommends NOT using cascade on M:M

You should be using 1:M and M:1 to simulate M:M anyway!

Cascading Delete for M:M

- **Recommended way to handle this is to use two <one-to-many> relationships**
  - No <many-to-many> tag
  - Cascade on the <one-to-many>
- **When Jack gets deleted:**
  - Jack's record is deleted from the OWNER table
  - The relationship for his account is deleted from the join table
  - Use a database trigger on the join table to check for other existing owners of the same account, and if none, THEN delete the account from the account table

OWNER_ID	NAME
1	JACK
2	JILL

OWNER_ID	ACCOUNT_ID
1	1
2	1

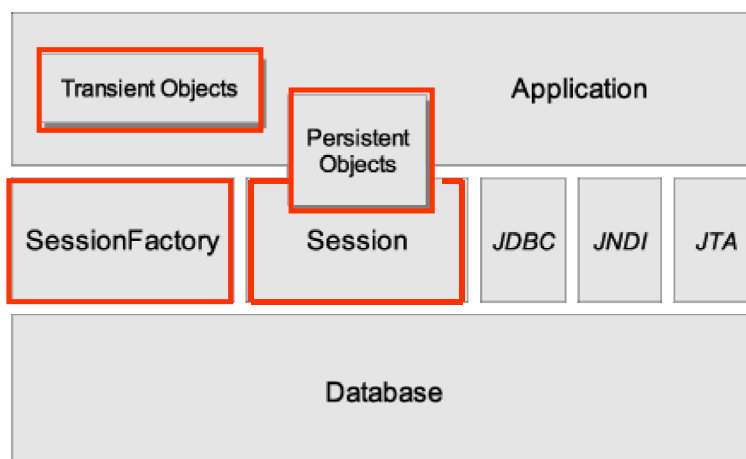| ACCOUNT_ID	BALANCE
1	100
2	500

Session Management

Hibernate Architecture – High Level

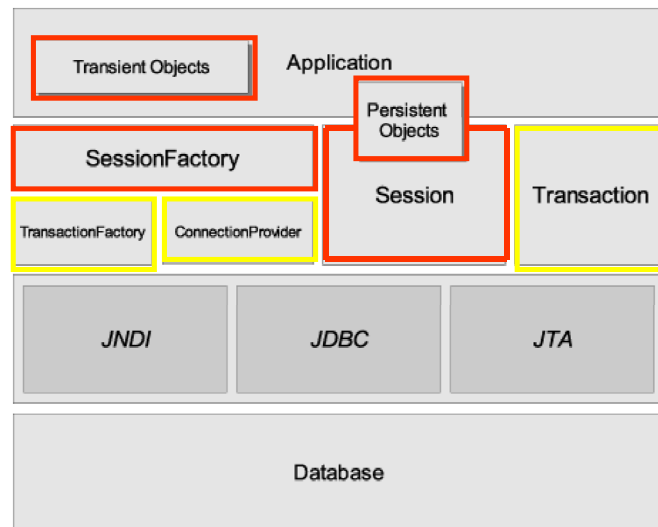*Source: Official Hibernate Documentation*

Hibernate Arch – "Lite"

- **The "Lite" architecture has the application provide its own JDBC connections and manage its own transactions**

Source: Official Hibernate Documentation

Hibernate Arch – "Full Cream"

- **The "full cream" architecture abstracts the application away from the underlying JDBC and Java Transaction API (JTA) and lets Hibernate take care of the details**

Source: Official Hibernate Documentation

Obtaining the Session

- **SessionFactory.getCurrentSession()**
  - Reuses an existing session
    - If none exists, creates one and stores it for future use
  - Automatically flushed and closed when transaction.commit() is executed
    - FlushMode.AUTO

- **SessionFactory.openSession()**
  - Always obtains a brand new session
  - Provides/requires more management
    - FlushMode.MANUAL
      - Developer must manually flush and close the session

- **http://www.hibernate.org/42.html**

Obtaining the Session

- **Within JavaSE environment, current sessions are saved on the thread**
  - hibernate.cfg.xml
    ```
    <property name="current_session_context_class">
      thread
    </property>
    ```
- **ThreadLocal Session Pattern**
  - Use ThreadLocal variable to maintain session data
- **ThreadLocal**
  - Specialized Java class which is associated to a users thread of execution and serves as a data container throughout the lifespan of the thread (or child threads)

Obtaining the Session

- **Within a container, managed as part of the JTA implementation**
  - hibernate.cfg.xml
    ```
    <property name="current_session_context_class">
      jta
    </property>
    ```
- **Session is scoped and bound to the JTA Transaction**
- **More about this when we talk about Transactions.**

Conversation

- **Unit of work that spans multiple requests**
  - Example:
    - User enters a form that spans several screens
    - New Bank Account Creation

- **Page 1 →Request 1**
  - Create Account Object
- **Page 2 → Request 2**
  - Collect user information, save account to database
- **Page 3 → Requests 3 – N**
  - Collect information for a single EBiller, build an EBiller object, and associate it to the account, save to database
- **Page 4 → Request N+1**
  - Display saved information summary

Conversations in Hibernate

- **Unit of work that spans multiple Hibernate Sessions (think multiple requests)**
- **Each request starts and ends its own Hibernate Session**
  - Possibly modifies persistent objects
    - Might want to persist to the database at a later point in time (i.e. subsequent request when in possession of more information)
- **Two ways of solving this**
  - Use detached objects
    - Session-per-request with detached objects
  - Use an extended Persistence Context
    - Session-per-conversation

Session-per-request with Detached Objects

- **Previously persisted objects that continue to represent rows of data in the database that are available to be modified <u>outside of the Hibernate Session</u>**

- **After changes are complete, these detached objects are 'reattached' via an update, merge, or lock**

Session-per-request with Detached Objects

- **Page 1 →Request 1**
  - Create Account Object
    - Account is 'transient'
- **Page 2 → Request 2**
  - Collect user information, save account to database
    - Account transitions to 'persistent', but we still want that account object available when we get to page 3, so we detach it from the Hibernate session
- **Page 3 → Request 3 – N**
  - Collect information for a single EBiller, build an EBiller object, and associate it to my account, save to database
    - EBiller starts off 'transient', transitions to 'persistent' for the save to the database, and then 'detached' to continue using for later pages
    - Also reattaching the Account object to save the relationship, and then detaching that again as well.
- **Page 4 → Request N+1**
  - Display saved information summary using 'detached' objects

Using Session-per-request with Detached Objects

1. **Create transient objects as needed**
2. **Get a handle to the current Session**
3. **Modify/obtain objects from the database through the session as needed**
   - These object will then be in the 'persistent' state and associated to the persistence context
4. **Close the session**
   - This will change the state of these objects to 'detached'
   - Be careful! Changes to objects in a 'persistent' state will be saved to the database when closing the session!
5. **Modify detached objects outside of the session as needed**
6. **Repeat steps 2 – 5 and create/modify/delete objects as needed across multiple requests**
7. **When the unit-of-work is complete, get a handle to the current session (if not already obtained)**
8. **Reattach the objects to be persisted**
   - Through an update(), merge() or lock() call

Reattach using update()

- **Need to reattach the object to the persistence context**
- **Using update() schedules an update SQL command, which might be an unnecessary database call**
  - Hibernate doesn't know what, if anything, changed while it was outside
  - Not harmful, but could be unnecessary

Detached Object - Update Example

```java
public void testUpdateAccountBalance() {
    // Get a handle to the current Session
    Session session =
        HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    // Modify objects
    Account account = new Account();
    session.saveOrUpdateAccount(account);

    // Close the Session without modifying the persistent object
    // Changes the state of the objects to detached
    session.getTransaction().commit();
    HibernateUtil.getSessionFactory().close();

    // Modify detached object outside of session
    account.setBalance(2000);
    ...
```

```java
    ...

    // Get handle to a subsequent Session
    Session session2 =
        HibernateUtil.getSessionFactory().getCurrentSession();
    session2.beginTransaction();

    // Reattach the object to be persisted.
    // An update statement will be scheduled regardless
    // of whether or not object was actually updated
    session2.update(account);

    // Commits/closes the session
    // Saves the changes made in the detached state
    session2.getTransaction().commit();
```

Reattach using lock()

- **Reattaches the object without forcing an update**
  - Used if you can be sure no changes took place
  - Changes made before locking are not persisted
- **Only guarantees that the object's state is changed from detached to persistent**

Reattach using merge()

- **Does NOT reattach the object passed in**
- **Checks to see if the object already exists in the system**
  - If so, gets the existing object
  - Creates a new instance otherwise
- **Copies the values of the submitted object into the persistent object from previous step**
- **Returns a reference to the final merged object instance back to the caller**
- **Note: SUBMITTED OBJECT STATE DOES NOT CHANGE**
  - Merging does not affect the submitted item reference. It is up to the developer to replace it if they so desire

Detached Object - Merge Example

```
...

// Get a handle a subsequent Session
Session session2 =
    HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();

// Reattach the object using merge.
// The data is persisted, but the passed in
// object is STILL in a detached state
session2.merge(account);

// Since this account object is NOT
// persistent, change is not saved
account.setBalance(100);

// Commits/closes session
// Saves the changes made in the detached state
session2.getTransaction().commit();
```

Detached Object - Merge Example

```
...

// Get a handle a subsequent Session
Session session2 =
    HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();

// Correct use of the merge operation.
// Now, my account reference is pointing
// to the updated object in memory
account = session2.merge(account);

// Change is NOW saved
account.setBalance(100);

// Commits/closes session
// Saves the changes made in the detached state
session2.getTransaction().commit();
```

Reattach using delete()

- **Used to remove detached objects directly from the database**

- **Though invisible to the user, delete() will first reattach the object via a proxy, and then scheduled it for deletion – just like a normal delete**

Session-per-Conversation

- **Extending the persistence context**
  - Keep the same persistence context <u>across server requests</u>
  - Reconnected upon subsequent requests
- **No more 'detached' objects**
  - All objects are either transient or persistent
  - No need for reattaching/merging
- **Synchronized with database at end of the conversation**

Session-per-Conversation

- **Session is disconnected from the underlying JDBC connection upon commit**
  - Persistence context is held on in some persistent state
    - HTTPSession
    - Stateful Session Bean
- **Persistence context is reconnected when the session begins the next transaction**
  - Objects loaded during the conversation are in the 'persistent' state; never detached
  - All changes are synchronized with the db when flush() is called
- **Need to disable automatic flushing of the session**
  - Set FlushMode.MANUAL when the conversation begins and the Session is opened
- **Manually trigger the synchronization with the database**
  - session.flush() at the end of the conversation

Session-per-Conversation

- **Still trying to gain traction**
- **Implemented by the JBoss Seam application framework**
- **Considered to be complicated by many developers**
  - Need to find a place to store the Session
  - Need to write an interceptor to bind and unbind the Session at the beginning and end of each request
  - Need to figure out if the conversation is complete by passing a token around
- **Refer to documentation for details**

Wrap-up

Summary

- **In this lecture, we:**
  - Walked through the Hibernate object lifecycle, and saw how an object's state affects its persistence behavior
    - Transient – not persisted
    - Persistent – changes managed by Hibernate
    - Detached – previously managed by Hibernate and requires re-attachment
  - Learned how to cascade object activities
    - cascade="delete " | cascade="delete-orphan"
  - Took a closer look at the Session object, explored the concept of the 'Persistence Context' and discovered many different persistent mechanisms
    - flush(), merge(), persist(), lock() etc...
  - Defined 'conversation' and the ways to implement them and carry our objects across multiple requests
    - Session-per-request with detached objects
    - Session-per-conversation

Preview of Next Sections

- **Transaction management**
- **Automatic versioning of objects**

Questions?