CSE 530A

More JPA

Washington University
Fall 2013
EntityManagerFactory

- Only need one EntityManagerFactory
  - Actually, one per DataSource
  - Expensive to create
  - Singleton pattern?
  - For web apps, can create factory in context listener and set as servlet context attribute
public class ContextListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
        sc.setAttribute("emf", emf);
        LOGGER.info("Created entity manager factory emf");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        EntityManagerFactory emf = (EntityManagerFactory) sc.getAttribute("emf");
        if (emf != null) {
            emf.close();
        }
    }
}
EntityManagerFactory

- Can then get the factory from the servlet context in the Servlet

```java
private void process(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    ... 

    EntityManagerFactory emf =
        (EntityManagerFactory) request.getServletContext().getAttribute("emf");

    ...

}```
private void process(HttpServletRequest request, HttpServletResponse response) 
    throws ServletException, IOException {

    EntityManagerFactory emf = 
        (EntityManagerFactory) request.getServletContext().getAttribute("emf");

    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = null;

    try {
        tx = em.getTransaction();
        tx.begin();

        User user = UserDao.retrieveUser(em, username);

        request.getRequestDispatcher("WEB-INF/jsp/Welcome.jsp").forward(request, response);
        tx.commit();
    } catch (Exception e) {
        LOGGER.log(Level.SEVERE, "error retrieving user", e);
        request.getRequestDispatcher("WEB-INF/jsp/Error.jsp").forward(request, response);
        tx.rollback();
    } finally {
        em.close();
    }
}
Accessing Entities

• There are several ways to access entities
  – find() by primary key
  – SQL-like queries
  – Criteria queries
Accessing Entities

• find() by primary key
  – Takes an entity class and a key value
  – Entity object is automatically created and populated

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
EntityManager em = emf.createEntityManager();

try {
    User user = em.find(User.class, id);

    String password = user.getPassword();

    ...
} finally {
    em.close();
}
```

...
Accessing Entities

• Select by query
  – Similar to SQL but uses entity class and field names instead of table and columns
  – Can use dot-notation to reference fields

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
EntityManager em = emf.createEntityManager();

try {
    Query query = em.createQuery("SELECT x FROM User AS x WHERE x.username = '" + name + "'");
    List<User> users = query.getResultList();
    if (users != null && users.size() > 0) {
        ...
    }

} finally {
    em.close();
}
```
Accessing Entities

• Concatenating strings is inelegant and dangerous
  – Beware SQL injection
• Can use named placeholders
  – Note the use of :name in the query
  – The placeholders in the query must be set using `setParameter` before the query is executed

```java
Query query = em.createQuery("SELECT x FROM User AS x WHERE x.username = :name");
query.setParameter("name", name);

List<User> users = query.getResultList();
```
Named Queries

• Queries can be attached to Entity classes
  – Note the use of the placeholders here, too

```java
@Entity
@Table(name = "users")
@SequenceGenerator(name = "USER_SEQ", sequenceName = "users_user_id_seq")
@NamedQueries({
    @NamedQuery(name = "User.findByName",
        query = "SELECT x FROM Users AS x WHERE x.username = :name")
})
public class User implements Serializable {
    private static final long serialVersionUID = 6803370824626024108L;

    ...
}
```
Named Queries

• Those queries can then be used by DAO methods
  – Again, the placeholders in the query must be set using `setParameter` before the query is executed

    ```java
    public static User retrieveUser(EntityManager em, String name) {
      Query query = em.createNamedQuery("User.findByName");
      query.setParameter("name", name);

      // If there could be multiple results then:
      // List<User> users = query.getResultList();

      return (User) query.getSingleResult();
    }
    ```
JPA Criteria

• A query can be built using criteria rather than a SQL statement
  – Doesn't work with all Java compilers and JPA implementations

// Get a criteria builder
CriteriaBuilder builder = em.getCriteriaBuilder();

// User is the expected return type
CriteriaQuery<Employee> criteria = builder.createQuery(User.class);

// User is also the FROM entity type
Root<User> root = criteria.from(User.class);

// The path expression User_.name doesn't seem to work with all versions of Java or JPA
Predicate condition = builder.equal(root.get(User_.name), username);

// Add the condition to the criteria query
criteria.where(condition);

// Finally get and execute the query
TypedQuery<User> query = em.createQuery(criteria);
User user = query.getSingleResult();
Hibernate Criteria

- I much prefer hibernate's way of doing it

```java
// Get the underlying hibernate Session from the EntityManager
Session hibernateSession = em.unwrap(Session.class);

// Create the criteria
Criteria criteria = hibernateSession.createCriteria(User.class);

// Add the restriction
criteria.add(Restrictions.eq("username", name));

// Execute the query
return (User) criteria.uniqueResult();
```
Hibernate

• Hibernate is library which implements the JPA API
  – Just one of many JPA implementations
• Hibernate pre-dates the existence of JPA
  – Personally prefer it over the JPA
Hibernate

• If we're going to use hibernate then we might as well use it everywhere
  – Create a SessionFactory instead of an EntityManagerFactory
  – Get Session objects instead of EntityManager objects
  – Use hibernate's query syntax
public class ContextListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        Configuration configuration = new Configuration();
        configuration.configure();
        ServiceRegistry serviceRegistry = new ServiceRegistryBuilder()
            .applySettings(configuration.getProperties()).buildServiceRegistry();
        SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        sc.setAttribute("hibernateSessionFactory", sessionFactory);
        LOGGER.info("Created hibernate session factory");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        SessionFactory sessionFactory = (SessionFactory) sc.getAttribute("hibernateSessionFactory");
        if (sessionFactory != null) {
            sessionFactory.close();
        }
    }
}
private void process(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    ... 
    SessionFactory sessionFactory =
            (SessionFactory) request.getServletContext().getAttribute("hibernateSessionFactory");
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        User user = UserDao.retrieveUser(session, username);
        ... 
        request.getRequestDispatcher("WEB-INF/jsp/Welcome.jsp").forward(request, response);
        tx.commit();
    } catch (Exception e) {
        LOGGER.log(Level.SEVERE, "error retrieving user", e);
        request.getRequestDispatcher("WEB-INF/jsp/Error.jsp").forward(request, response);
        tx.rollback();
    } finally {
        session.close();
    }
}
Accessing Entities

- Hibernate's pseudo-SQL syntax is a little different
  - Notice the similar use of placeholders, though
- The `Query` is now an `org.hibernate.Query` instead of a `javax.persistence.Query`

```java
Query query = session.createQuery("from User user where user.username = :name");
query.setString("name", name);

User user = (User) query.uniqueResult();
```
Accessing Entities

• Hibernate also has the equivalent of `find` to retrieve by ID

```java
public static User retrieveUser(Session session, Long id) {
    return (User) session.get(User.class, id);
}
```
Creating Entities

• Create the entity object but do not set the ID (assuming an auto-generated ID)
• Call `session.save`
  – The object will be assigned a new ID and saved to the database

```java
public static User createUser(Session session, String username, String password) {
    User user = new User();
    user.setUsername(username);
    user.setPassword(password);
    session.save(user);
    return user;
}
```
Updating Entities

• Update the fields on the entity object but do not change the
• Call `session.save`
  – The row in the database will be updated

```java
public static void updateUser(Session session, User user) {
    session.save(user);
}
```
Deleting Entities

• Call `session.delete`
JPA Configuration

• JPA configuration is done through the file /src/META-INF/persistence.xml
  – Eclipse will copy this to WEB-INF/classes during the build process
  – The example assumes the existence of a DataSource named jdbc/postgres configured in context.xml

```xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="..."
             version="2.0">
  <persistence-unit name="emf">
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
    </properties>
  </persistence-unit>
</persistence>
```
Hibernate Configuration

• Hibernate configuration is done through the file /src/hibernate.cfg.xml
  – Eclipse will copy this to WEB-INF/classes during the build process
  – The example assumes the existence of a DataSource named jdbc/postgres configured in context.xml

• Unlike with JPA, the Entity classes must be listed in the hibernate configuration
Hibernate Configuration

<!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD//EN"
   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.datasource">java:comp/env/jdbc/postgres</property>

    <!-- Disable second-level cache. -->
    <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
    <property name="cache.use_query_cache">false</property>
    <property name="cache.use_minimal_puts">false</property>
    <property name="max_fetch_depth">3</property>

    <!-- Print SQL to stdout. -->
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>

    <!-- Bind the getCurrentSession() method to the thread. -->
    <property name="current_session_context_class">thread</property>

    <mapping package="cse530a.model" />
    <mapping class="cse530a.model.User" />
    ...
  </session-factory>
</hibernate-configuration>
Show SQL

• Whether get, HQL, or Criteria are used to get Entities, it is translated to SQL under the hood
  – Setting `show_sql` to true in the configuration causes hibernate to print the actual SQL executed
Show SQL

Query query = session.createQuery("from User user where user.username = :name");
query.setString("name", username);

return (User) query.uniqueResult();

becomes

Hibernate:
select
    user0_.user_id as user1_0_,
    user0_.password as password0_,
    user0_.username as username0_
from
    users user0_
where
    user0_.username=?
Lazy vs Eager Loading

• When using a relationship between Entity objects hibernate will
  – Lazy load collections (@OneToMany and @ManyToMany)
  – Eager load single-valued references (@OneToOne and @ManyToOne)
Lazy vs Eager Loading

• Lazy loading means that the SQL to load the objects is not executed until they are actually accessed
  – Hibernate accomplishes this through the use of proxy objects
• An attempt to access a lazily-loaded field after transaction end will result in an exception
  – To be safe, all Entity objects should be considered invalid after transaction end
Lazy vs Eager Loading

• The fetch type can be changed via the annotations
  – Be careful! In this example, all of the Employee objects in the Department will be fetched from the database

```java
public class Department {
    …

    private List<Employee> employees;

    @OneToMany(mappedBy = "department", targetEntity = Employee.class,
                fetch = FetchType.EAGER)
    public List<Employee> getEmployees() {
        return employees;
    }

    …
}
```
N+1 Select Problem

• Assume we have a one-to-many relationship from Department to Employee and a one-to-one relationship from Employee to Address
• Assume we set the FetchType of employees in Department to EAGER
• What happens when get a Department?
N+1 Selects Problem

• What happens when get a Department?
  – One SELECT to get the department record
  – One SELECT to get all of the employee records for that department
  – One SELECT for each of the employee records to get its address record

• This is the N+1 selects problem
N+1 Selects Problem

• Changing the FetchType of employees in Department partially solves the problem
  – As long as we don't access employees we're OK, but once we do we get the extra SELECTs for the addresses

• We could change the FetchType of address in Employee to LAZY
  – When the employees are loaded the addresses are not
  – But what if we actually need the addresses?
    • Every address we access causes a SELECT
Join Fetch

- We can use join fetch in HQL to force the use of joins instead of multiple selects

"from Department dep left join fetch d.employees emp left join fetch emp.address where ..."