The Java™ Persistence API in the Web Tier

Linda DeMichiel, Sun Microsystems

Gavin King, JBoss

Craig McClanahan, Sun Microsystems

Session 1887

Goal of This Talk

Learn how to use the Java™ Persistence API in your web tier applications

Agenda

Java Persistence API—Key Concepts

JavaServer™ Faces—Key Concepts

Threading models and injection

Managing transactions

Extended persistence contexts

Summary

Java Persistence API

- Part of JSR-220 (Enterprise JavaBeans™ 3.0)
- Began as simplification of entity beans
- Evolved into POJO persistence technology
  - Rich modeling capabilities, inheritance, polymorphism
  - Standardized object/relational mapping
  - Powerful query capabilities
- Scope expanded at request of community
  - Into persistence technology for Java EE platform
  - To support use in Java SE environments
  - To support pluggable persistence providers

Java Persistence API: Key Concepts

- Entities
- Persistence Units
- Persistence Contexts

Entities

- Plain old Java objects
  - Created using **new**
  - No required interfaces
  - Support inheritance, polymorphism
  - Have persistent identity
  - May have both persistent and non-persistent state
- Usable outside the container
  - Serializable; usable as detached objects
- Queryable via Java Persistence query language
- Managed at runtime through EntityManager API

Entity Class

```
@Entity
public class Customer {

    @Id private Long id;

    private String name;

    @OneToMany Set<Order> orders = new HashSet();

    public Set<Order> getOrders() { return orders; }

    public void addOrder(Order order) {
        getOrders().add(order);
    }

}
```

Persistence Unit

- Unit of persistence packaging and deployment
- Set of managed classes (entities and related classes)
- Defines scope for
  - Queries
  - Entity relationships
- O/R mapping information
  - Java language annotations and/or XML files
  - Defines Java language view onto a relational database
- Configuration information for persistence provider
  - persistence.xml file

Persistence Context

- Runtime application context
- Set of managed entity instances, belonging to a single persistence unit
  - Entities that have been read from the database
  - Entities that will be written to the database
    - Including entities that are newly persistent
  - Persistent entity identity equivalent to Java identity
- Persistence context lifetime may be
  - Transaction-based—scoped to a single transaction
  - Extended—spanning multiple sequential transactions

Persistence Contexts

- May be managed by application or container
  - Container-managed persistence contexts
    - Provide ease-of-use in Java EE environments
    - Propagated across components with JTA transaction
    - Obtained by injection or lookup in JNDI
    - May be scoped to single transaction or extended
  - Application-manager persistence contexts
    - Provided for use in Java SE and Java EE environments
    - Obtained from EntityManagerFactory
    - Extended scope is managed by application
  - Web tier supports both

EntityManager API

- Entity lifecycle operations
  - persist; remove; refresh; merge
- Finder operations
  - find, getReference
- Factory for Query objects
  - createNamedQuery, createQuery, createNativeQuery
- Operations for managing persistence context
  - flush, clear, close, getTransaction, joinTransaction,…

Persisting an Entity

```
@PersistenceContext EntityManager em;

public Order addNewOrder(Customer customer, Product
product) {

    Order order = new Order(product);
    customer.addOrder(order);
    em.persist(order);
    return order;

}
```

Removing an Entity

```
@PersistenceContext EntityManager em;

public void dropCustomer(Long custId) {
    Customer customer = em.find(Customer.class, custId);
    em.remove(customer);
}
```

Agenda

Java Persistence API—Key Concepts

JavaServer™ Faces—Key Concepts

Threading models and injection

Managing transactions

Extended persistence contexts

Summary

JavaServer Faces Technology

- Comprehensive user interface component model
- Flexible rendering model
- JavaBeans™ style event and listener handling
- Per-component validation framework
- Basic page navigation support
- Extensible controller architecture

Standard Features

- Hierarchical tree of UI components
- Converters (bidirectional)
- Validators (input correctness checks)
- User Interface event handling:
  - Action events
  - Value change events
  - Custom component events
- Outcome based navigation:
  - Which command (on which view) was invoked?
  - What logical outcome was returned?

Unique Features

- Value binding expressions:
  - Bind UI components to model tier data
    - <h:outputText ... value="#{customer.address.city}"/>
  - On form submit, used to push data back to the model
- Method binding expressions:
  - Bind UI components to "action" methods
    - <h:commandButton ... action="#{mybean.save}"/>
  - On form submit, used to select method invoked based on which command component was activated

Unique Features

- Managed Beans:
  - Instantiate application beans on demand
  - Store instance in request/session/application scope
  - Optionally configure bean properties
    - Basic “setter injection” style dependency injection framework
- Extensibility Points:
  - View Handler—Use non-JSP™ view technology
  - Navigation Handler—Customize navigation
  - Variable Resolver/Property Resolver—Customize value binding and method binding evaluation

Agenda

Java Persistence API—Key Concepts

JavaServer™ Faces—Key Concepts

Threading models and injection

Managing transactions

Extended persistence contexts

Summary

Injection

- Resource injection:
  - for datasources, `UserTransaction`, JMS queues/topics, environment entries, etc.
  - `@Resource DataSource bookStoreDS;`
- Persistence units and contexts:
  - `@PersistenceContext EntityManager em;`
  - `@PersistenceUnit EntityManagerFactory emf;`
- EJB™ references:
  - `@EJB ShoppingCart shoppingCart;`

Threading Models and Injection

- Certain objects are not threadsafe
  - Especially stateful objects, like EntityManager
  - Some of these objects are injectable
- Some other objects are multithreaded
  - Especially stateless objects, like servlets
  - Some of these objects may be injected into
- It is dangerous to inject objects of the first kind into objects of the second kind!
  - In particular, don't inject EntityManager into a servlet

Bad

```
public class BookShoppingServlet extends HttpServlet {

    @PersistenceContext EntityManager em;

    protected void doPost(HttpServletRequest req,
     HttpServletResponse res) throws … {

     Order order = …;
     em.persist(order);

     }

}
```

Good

```
public class BookShoppingServlet extends HttpServlet {

    protected void doPost(HttpServletRequest req,
    HttpServletResponse res) throws … {

    Order order = …;
    EntityManager em = new InitialContext()
        .lookup("java:comp/env/persistence/bookStore");
    em.persist(order);

    }

}
```

Good

```
public class BookShoppingServlet extends HttpServlet {

    @PersistenceUnit EntityManagerFactory emf;

    protected void doPost(HttpServletRequest req,
     HttpServletResponse res) throws … {

     Order order = …;
     EntityManager em = emf.createEntityManager()
     em.persist(order);
     em.close();

     }

}
```

Agenda

Java Persistence API—Key Concepts

JavaServer™ Faces—Key Concepts

Threading models and injection

Managing transactions

Extended persistence contexts

Summary

Transaction Demarcation

- JTA transactions
  - UserTransaction API
- Resource-local transactions
  - EntityTransaction API
- Container-managed entity managers use JTA
- Application-managed entity managers are either JTA or resource-local
  - Determined by configuration of the persistence unit
  - Resource-local transactions needed in Java SE environments

Servlet Example

```
public class BookShoppingServlet extends HttpServlet {

    protected void doPost(HttpServletRequest req,
              HttpServletResponse res) throws … {

        String customer = req.getParameter("customer");
        int custId = Integer.parseInt(customer);
        String creditCard = req.getParameter("creditCard");
        String book = req.getParameter("book");

        buyBook(book, custId, creditCard);

    }

}
```

JTA Transaction

```
@Resource UserTransaction utx;
@PersistenceUnit EntityManagerFactory emf;

protected void buyBook(String book, int custId, String
creditCard) throws … {

    utx.begin();
    EntityManager em = emf.createEntityManager();

    Customer customer = em.find(Customer.class, custId);
    Order order = new Order(customer);
    order.setCreditCard(creditCard);
    order.setBook(book);

    em.persist(order);

    utx.commit();
    em.close();
}
```

Resource-local Transaction

```
@PersistenceUnit EntityManagerFactory emf;

protected void buyBook(String book, int custId, String
creditCard) {

    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    Customer customer = em.find(Customer.class, custId);
    Order order = new Order(customer);
    order.setCreditCard(creditCard);
    order.setBook(book);

    em.persist(order);

    em.getTransaction().commit();
    em.close();
}
```

Problems

- Propagation of EntityManager between components
- Messy exception handling
  - Does not belong in business logic

Container-managed EntityManager

```
protected void buyBook(String book, int custId, String
creditCard) throws … {

    utx.begin();

    Customer customer =
       new CustomerHelper().getCustomer(custId);
    Order order =
       new OrderHelper().create(book, creditCard, customer);

    utx.commit();

}
```

Container-managed EntityManager

```
public Customer getCustomer(int id) {
    EntityManager em = (EntityManager) new InitialContext()
       .lookup("java:comp/env/persistence/bookStore");
    return em.find(Customer.class, id);
}
```

```
public Order create(String book, String creditCard,
Customer customer) {
    EntityManager em = (EntityManager) new InitialContext()
       .lookup("java:comp/env/persistence/bookStore");
    Order order = new Order(customer);
    order.setCreditCard(creditCard);
    order.setBook(book);
    em.persist(order);
}
```

Servlet Filter for Tx Demarcation

```
@Resource UserTransaction utx;

public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain) throws … {

    utx.begin();
    try {
        chain.doFilter(request, response);
    }
    catch (Exception e) {
        utx.rollback();
        throw new ServletException(e);
    }
    utx.commit();

}
```

Servlet Filter for Tx Demarcation

```
protected void buyBook(String book, int custId, String
creditCard) throws … {

    Customer customer =
       new CustomerHelper().getCustomer(custId);
    Order order =
       new OrderHelper().create(book, creditCard, customer);

}
```

Caveat

- Transaction is not committed until response complete
  - In fact, SQL statements may not even have been executed
- Servlet container may flush reponse to browser at any time
- So we might display success message to user and then transaction subsequently fails
  - Solution: use a persistence context that spans two transactions, one for read/write, one for pure readonly

Using EJB Components

```
@EJB CustomerMgr customerMgr;
@EJB OrderMgr orderMgr;

protected void buyBook(String book, int custId, String
creditCard) throws … {

    Customer customer = customerMgr.getCustomer(custId);
    Order order = orderMgr.create(book, creditCard,
customer);

}
```

Injecting the EntityManager

```
@Stateless
public class CustomerMgrBean implements CustomerMgr {
    @PersistenceContext EntityManager em;

    public Customer getCustomer(int id) {
        return em.find(Customer.class, id);
    }
}

@Stateless
public class OrderMgrBean implements OrderMgr {
    @PersistenceContext EntityManager em;

    create(String book, String creditCard, Customer cust) {
        Order order = new Order(cust);
        order.setCreditCard(creditCard);
        order.setBook(book);
        em.persist(order);
    }
}
```

Non-transactional Reads

- What happens if there is no JTA transaction in progress when we execute a query?
- You are allowed to perform read-only operations upon an EntityManager when no JTA transaction is in progress
  - Resulting SQL queries run outside of well-defined transaction context
  - In practice this usually means that the SQL is executed against a connection with autocommit enabled
- Entity lifecycle depends on persistence context scope
  - Transaction scope (container managed EM)—entity instances returned by the query are immediately detached (a temporary persistence context is created and destroyed)
  - Extended scope—entity instances are managed

Agenda

Java Persistence API—Key Concepts

JavaServer™ Faces—Key Concepts

Threading models and injection

Managing transactions

Extended persistence contexts

Summary

Conversations

- A conversation takes place whenever a single user interaction spans more than one request
- May span multiple atomic database/JTA transactions
- Sometimes convenient to keep and reuse references to the entities that are the subject of the conversation
  - In the HTTPSession
  - In a stateful session bean that represents the conversation

Extended Persistence Context

- A natural cache of data that is relevant to a conversation
- Allows stateful components to maintain references to managed instances (instead of detached instances)
- Maintained across multiple sequential transactions
  - Until the EntityManager is closed
- Optimistic transaction semantics
  - With version checking

Extended Persistence Context Helper

```
public class ExtendedPersistenceContextServlet
    extends HttpServlet {

  @PersistenceUnit EntityManagerFactory emf;

  public EntityManager getEntityManager(HttpSession s) {
    EntityManager em = s.getAttribute("entityManager");
    if (em==null) {
        em = emf.createEntityManager();
        s.setAttribute("entityManager", em);
    }
    em.joinTransaction();
    return em;
  }

  public void endConversation(HttpSession s) {
    getEntityManager().close();
    s.removeAttribute("entityManager");
  }

}
```

Using the Helper

```
public class BuyBookServlet
      extends ExtendedPersistenceContextServlet {

    protected void doPost(HttpServletRequest req,
              HttpServletResponse res) throws … {

        String customer = req.getParameter("customer");
        int custId = Integer.parseInt(customer);
        String creditCard = req.getParameter("creditCard");
        String book = req.getParameter("book");

        Order order = buyBook(book, custId, creditCard);

        req.getSession().setAttribute("order", order);

    }

}
```

Starting the Conversation

```
protected Order buyBook(String book, int custId, String
creditCard) {

    EntityManager em = getEntityManager();

    Customer customer = em.find(Customer.class, custId);
    Order order = new Order(customer);
    order.setCreditCard(creditCard);
    order.setBook(book);

    em.persist(order);

    return order;
}
```

Ending the Conversation

```
public class ConfirmOrderServlet
      extends ExtendedPersistenceContextServlet {

    protected void doPost(HttpServletRequest req,
              HttpServletResponse res) throws … {

        Order order = (Order)
            req.getSession().getAttribute("order");
        confirmOrder(order);

        endConversation();
    }

}

public void confirmOrder() {
    order.confirm();
}
```

Using an EJB Component

```
public class BuyBookServlet {

    protected void doPost(HttpServletRequest req,
              HttpServletResponse res) throws … {

        String customer = req.getParameter("customer");
        int custId = Integer.parseInt(customer);
        String creditCard = req.getParameter("creditCard");
        String book = req.getParameter("book");

        OrderMgr orderMgr = (OrderMgr)
            new InitialContext().lookup("java:/comp/…");

        orderMgr.buyBook(book, custId, creditCard);

        req.getSession().setAttribute("orderMgr", orderMgr);

    }

}
```

Starting the Conversation

```
@Stateful public class OrderMgrBean implements OrderMgr {

    @PersistenceContext(type=EXTENDED)
    private EntityManager em;

    private Order order;

    protected void buyBook(String book, int custId, String
creditCard) {

        Customer customer =
            em.find(Customer.class, custId);
        order = new Order(customer);
        order.setCreditCard(creditCard);
        order.setBook(book);

        em.persist(order);
    }

}
```

Ending the Conversation

```
public class ConfirmOrderServlet {

    protected void doPost(HttpServletRequest req,
                          HttpServletResponse res) throws … {

        OrderMgr orderMgr = (OrderMgr)
            req.getSession().removeAttribute("orderMgr");
        orderMgr.confirmOrder(order);
        orderMgr.remove();
    }

}
```

Caveat

- What if we have multiple **concurrent** conversations?
  - In a multi-window application
  - The persistence context should not be shared between the different conversations
  - But we have shared the EntityManager across the whole session, by keeping it in the HttpSession!
  - Solution: associate the persistence context with a conversation id that is passed to the server in each request

Non-transactional Writes

- What happens when we update data outside of a transaction?
  - We are allowed to call persist(), merge(), remove() outside of a JTA transaction,
  - Or mutate entities associated with an extended persistence context outside of a JTA transaction
- The actual database updates will be made the first time a transaction commits
- This is very useful!
  - We can “queue” changes on the server, over multiple requests, and make them persistent at the end of the conversation

Using the Java Persistence API and JavaServer Faces Together

- Previous examples illustrated with servlets:
  - Technically feasible solution, however…
  - Most applications use an application framework
    - Already provides a controller servlet
    - Application logic encapsulated in actions or backing beans
- JavaServer Faces provides opportunities to leverage the Java Persistence API without writing servlets
  - Resource injection into backing beans
  - Expression-based access to JNDI API naming context
- Filter based designs (tx demarcation) are also compatible

Resource Injection

- Works with managed beans just like servlets:
  - Use request scope managed beans to avoid thread safety issues

```
public class MyBackingBean {
    @PersistenceContext EntityManager em;
    public String save() {
        Order order = ...; // Populate from input fields
        em.persist(order);
        return null;
    }
}
```

DEMO

Trying it all together

Agenda

Java Persistence API—Key Concepts

JavaServer™ Faces—Key Concepts

Threading models and injection

Managing transactions

Extended persistence contexts

Summary

Summary

- Java Persistence API provides range of mechanisms for use in web tier
  - Container-managed and application-managed EntityManagers
  - Integration with JTA and resource-local transactions
  - Support for use in servlets, filters, managed beans
  - Modeling of transaction-scoped interactions as well as conversations

For More Information

- http://jcp.org/en/jsr/detail?id=220
- http://jcp.org/en/jsr/detail?id=252

Q&A

The Java™ Persistence API in the Web Tier

Linda DeMichiel, Sun Microsystems

Gavin King, JBoss

Craig McClanahan, Sun Microsystems

Session 1887