

Java Persistence Advanced Concepts

Multitier Application Architecture

Java Persistence API Revisited

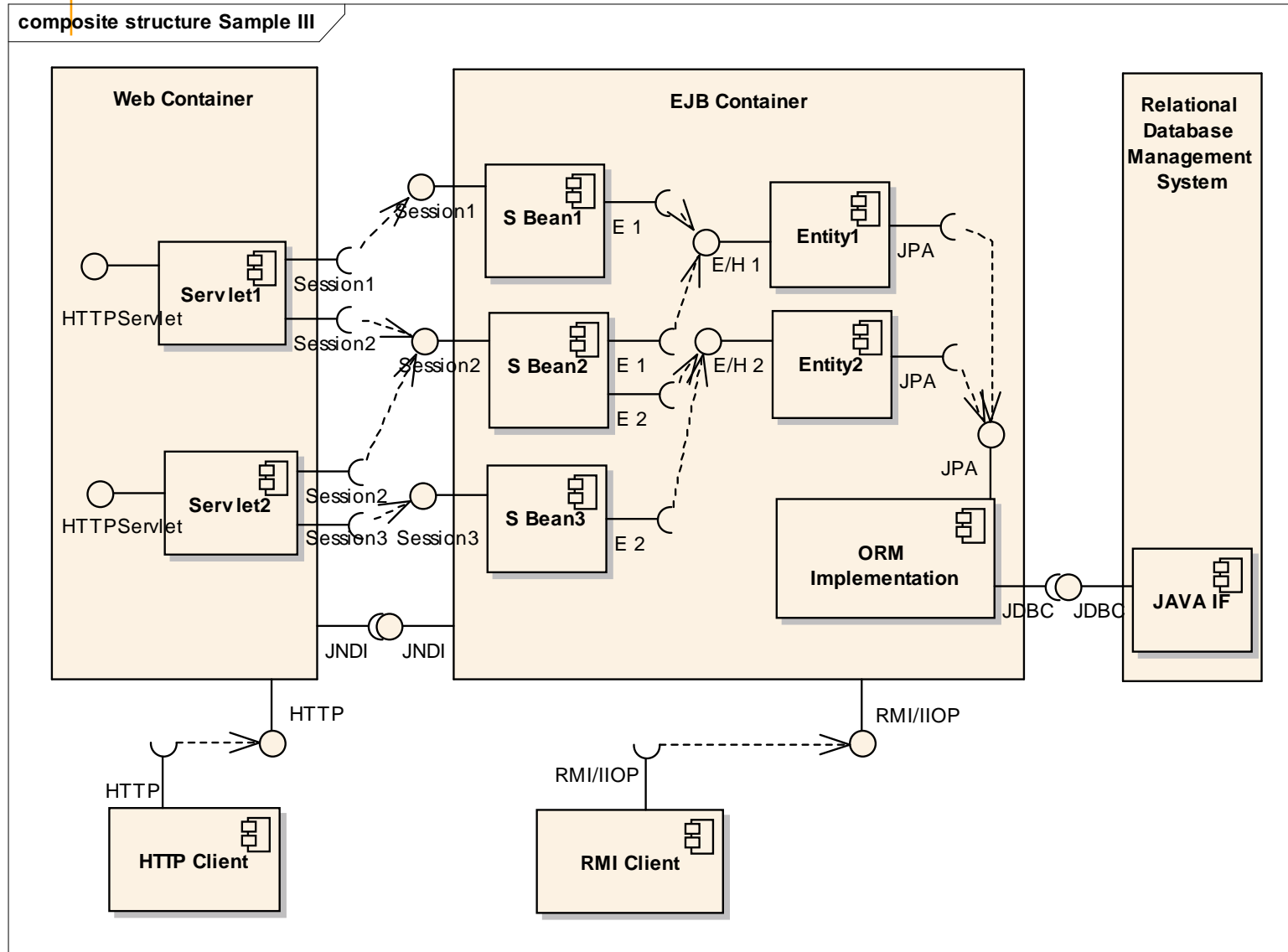
Transaction Management

EntityManager Detailed

JPA Queries Added to DAO

Cooperation with the Web Tier

Multitier Application Architecture



Java Persistence API

Description

- POJO (Plain Old Java Object) based standard
- Defines Object Relational Mapping for data persistence (entities)
- Use Java 5 annotations for persistence metadata

Features

- Java Persistence API is part of EJB 3.0 can also integrate with J2SE applications
- Can use different persistence providers (Hibernate, Toplink, iBatis)
- Declarative transaction management

Object Relational Mapping

- Transformation of the data between the class objects and (relational) databases
- Supported by automated tools (Object Model <-> DB model)
- Objects are transformed to records (entities), properties are mapped to fields, identity

JPQL for Defining Queries

- Queries are formulated in term of objects and properties (relations)
- Parameter passing by name and by position
- Named and dynamic queries
- Updates are handed on result set



Transaction Management

Transaction Management Features

- Provides a consistent programming model across different transaction APIs (JTA, JDBC, Hibernate, JPA)
- Supports declarative transaction management
- Provides a simpler API for programmatic transaction management than JTA
- Rollback on exception, rollback rules can be defined
- Atomicity, Consistency, Isolation, Durability

Isolation Level Definitions

READ_UNCOMMITTED - dirty reads, non-repeatable reads and phantom reads can occur

READ_COMMITTED - dirty reads are prevented

REPEATABLE_READ - dirty reads and non-repeatable reads are prevented

SERIALIZABLE - dirty reads, non-repeatable reads and phantom reads are prevented

Transaction Propagation

REQUIRED - support a current transaction, create a new one if none exists - DEFAULT

MANDATORY - support a current transaction, throw an exception if none exists

NESTED - subtransaction can be rolled back, behave like PROPAGATION_REQUIRED else

NEVER - execute non-transactionally, throw an exception if a transaction exists

NOT_SUPPORTED - execute non-transactionally, suspend the current transaction if one exists

REQUIRES_NEW - create a new transaction, suspend the current transaction if one exists

SUPPORTS - support a current transaction, execute non-transactionally if none exists



Declarative vs. Programmatic Transaction Management

Declarative

```
@Transactional(rollbackFor=ApplicationException.class)
public void processData {
    ...
    if( problem ) {
        throw new ApplicationException("It went wrong") ; // Rolled back
    }
}
```

Programmatic

```
public void processData {
    EntityTransaction userTransaction = entityManager.getTransaction();
    try{
        userTransaction.begin();
        // If everthing goes well, make a commit here.
        userTransaction.commit();
    } catch(Exception exception) {
        // Exception has occurred, roll-back the transaction.
        userTransaction.rollback();
    }
}
```



EntityManager find() vs. getReference()

find()

```
MobileEntity mobile = entityManager.find(MobileEntity.class, "ABC-123");  
If (mobile != null) { // mobile object may or may not be null  
    // Process the object  
} else {  
    // Do something when "ABC-123" not exist  
}
```

getReference()

```
try {  
    MobileEntity mobile = entityManager.getReference(MobileEntity.class, "ABC-123");  
    // mobile object may not contain the actual state values for model, manufacturer  
    // and imei number, the states may be loaded during the first access.  
    String model = mobile.getModel();  
    // The persistence engine may fetch the model value for the mobile here  
    // at this particular point of time ...  
} catch(EntityNotFoundException ex) {  
    // Do something when "ABC-123" not exist  
}
```



EntityManager merge(), flush() and refresh()

merge()

```
MobileEntity mobile = entityManager.find(MobileEntity.class, "ABC-123");  
// Transaction ends.  
mobile.set() // Updating the mobile object.  
  
.....  
// New transaction begins  
entityManager.merge(mobile);  
// The values in the object are merged into database state
```

flush()

```
MobileEntity mobile = ...  
mobile.set(); // Update the state values for the mobile object. ...  
entityManager.flush();  
// Calling this flush method will synchronize the database with the values of the entity object.
```

refresh()

```
MobileEntity mobile = ...  
mobile.set();  
// The state values for the mobile object is updated. ...  
entityManager.refresh();  
// The refresh() method will refresh the entity object with the values taken from the database.  
// All the updates that are done are lost.
```



Java Persistence Query Language (JPQL)

General Form

```
SELECT [<result>] [FROM <candidate-class(es)>] [WHERE <filter>]
      [GROUP BY <grouping>] [HAVING <having>] [ORDER BY <ordering>]
DELETE FROM [<candidate-class>] [WHERE <filter>]
UPDATE [<candidate-class>] SET item1=value1, item2=value2 [WHERE <filter>]
```

Named Parameters

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName =
                        :surname AND p.firstName = :forename");
q.setParameter("surname", theSurname);
q.setParameter("forename", theForename);
```

Numbered Parameters

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND
p.firstName = ?2"); q.setParameter(1, theSurname);
q.setParameter(2, theForename);
```

Range of Results

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.age>20");
q.setFirstResult(20);
q.setMaxResults(10);
```



Finding Entities in the Database

Named Query

```
@NamedQuery( name="findAllCustomersWithName", query="select c from Customer c
                                                    where c.name like :custName" )

Query query = em.createNamedQuery("findAllCustomersWithName");
List<Customer> customers = query.setParameter("custName", "Smith") .getResultList();
```

Dynamic Query

```
Query query = em.createNamedQuery("select c from Customer c where c.name like
:custName" );
List<Customer> customers = query.setParameter("custName", "Smith") .getResultList();
```

or

```
Query query = em.createNamedQuery("select c from Customer c where c.name like
:custName" );
Customer customer = query.setParameter("custName", "Smith") .getSingleResult();
```

Updating Queried Entities

```
query.setFlushMode(FlushModeType.COMMIT);
customer.setAddress("3. Dirty street");
```



Direct Integration with Struts

In struts-config.xml

```
<plug-in
  className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation" value="/WEB-INF/training-servlet.xml"/>
</plug-in>
```

Struts action class

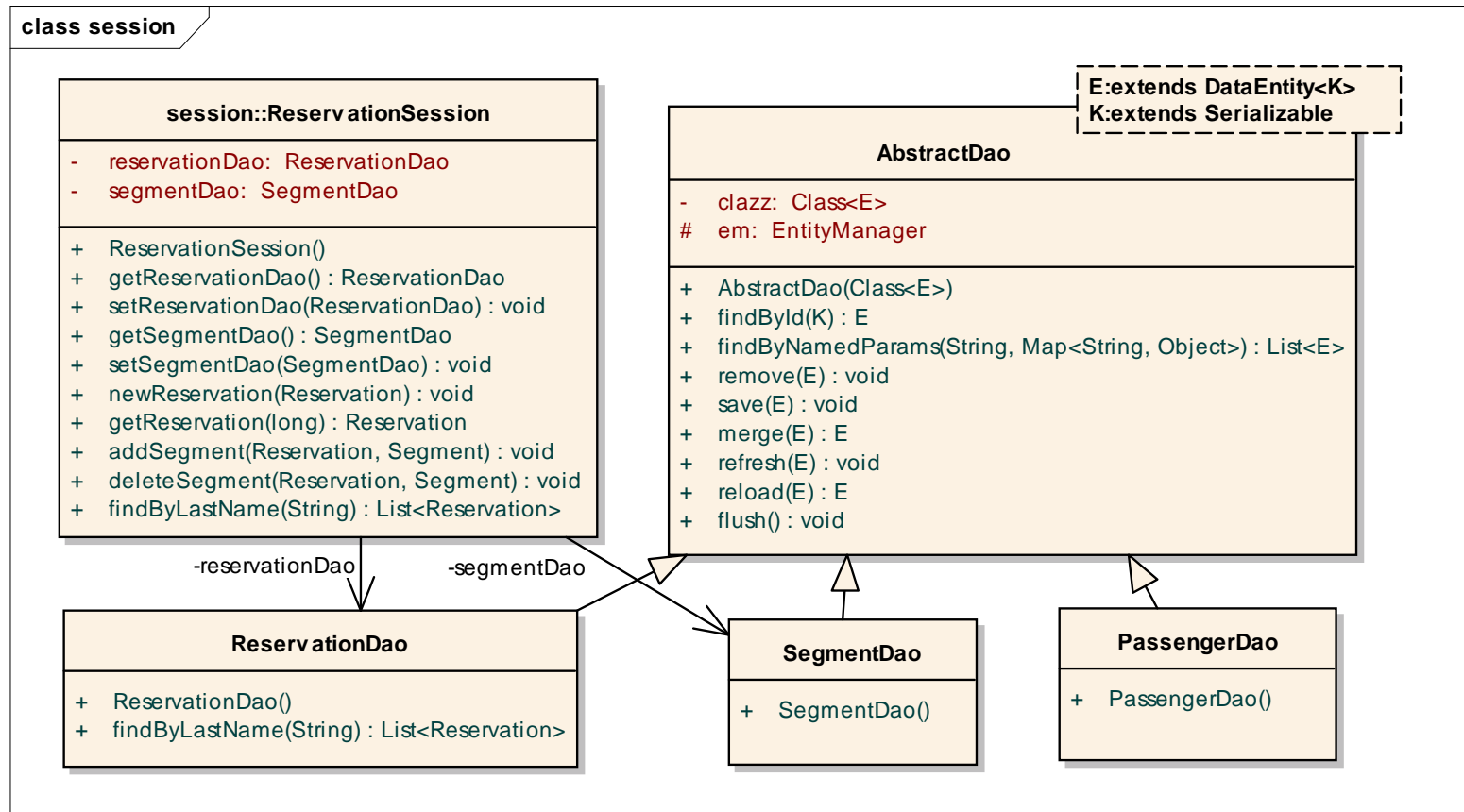
```
public class ListCourseAction extends ActionSupport {
  public ActionForward execute( ... ) {
    ApplicationContext context = getWebApplicationContext();
    CourseService courseService = (CourseService) context.getBean("courseService");
    Set allCourses = courseService.getAllCourses();
    ...
  }
}
```

In training-servlet.xml

```
<bean id="courseService" class="com.myapp.CourseService"/>
```



Sample Application Session and DAOs



Sample Application Modified Data Model

