The Data Tier of the Java EE Architecture: Advanced Java Persistence

Author: Simon Pickin

Address: Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
Spain

Version: 1.1

Communication Software 2010-2011

Contents

Communication Software 2010-2011

© The Authors

The Persistence Layer

- JDBC not enough for independence of DBMS
  - supposes RDBMS & SQL
  - even between different RDBMS, SQL used may vary
  - java developer may not know (or want to know) SQL
  - too low-level
- Use of a persistence layer
  - object-oriented abstraction of database
  - part of business tier
  - between business logic and data tier
  - what form for the persistence layer?

The Data Access Object (DAO) Pattern

- *Data Access Object* (DAO) pattern
  - object that encapsulates database access
    - separates client interface from access mechanisms
    - provides generic API
  - mentioned in Sun's “Core J2EE Patterns”
  - similar to Fowler’s *Table Data Gateway* pattern
  - often used together with DTO pattern
- *Data Transfer Object* (DTO) pattern
  - object that encapsulates database data
  - similar to Sun's (but not Fowler’s) *Value Object* pattern

DAO Layer Considerations

- DAO layer of an application
  - usually comprises many DAOs
  - some of the DAOs may encapsulate
    - SQL joins
    - updates on multiple tables
- DAO layer solution
  - will be too heavy without some DAO code generation
    - unless application very simple
  - metadata used for code-generation could be obtained from
    - developer-defined descriptor file
    - database schema
    - both

DAO Implementation Strategies (1/3)

- Code each DAO class explicitly
  - simplest but least flexible
  - change of DBMS often implies change of DAO implementation
- Use a DAO factory
  - *Factory Method* pattern (Gamma et al.)
  - uni-class factories
    - create instances of a single DAO class
  - multi-class factories
    - create instances of multiple DAO classes

DAO Implementation Strategies (2/3)

- Use an abstract DAO factory
  - *Abstract Factory* pattern (Gamma et al.)
  - encapsulates set of DAO factories
    - one for each DBMS or DBMS type
  - returns pointer to a DAO factory
    - each factory implements same abstract interface
  - client creates DAOs via same abstract interface
    - whatever the DBMS or DBMS type

DAO Implementation Strategies (3/3)

- Use Generic DAO classes
  - handle different DBMS of same type
  - data source dependent details loaded from config. file
    - e.g. RDBMS: all DBMS-dependent SQL in configuration file
- Use abstract DAO factory and generic DAO classes
  - returns pointer to a DAO factory
  - client uses abstract interface...
    - covers different DBMS types
  - ... to create generic DAOs
    - covers different DBMSs of a given type

The Active Record Pattern

- Active Record pattern
  - can be viewed as pattern combining DAO and DTO
  - data and behaviour
  - "An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data." Fowler

Communication Software 2010-2011

Active Record of Ruby on Rails

- Model part of Ruby-on-Rails MVC framework
- Fowler's Active Record pattern
  - + inheritance + object associations.
    - former via use of *Single Table Inheritance* pattern
    - latter via a set of macros
- CoC and DRY principles
- Imposes
  - some aspects of database schema
  - some naming conventions
- Following RoR restrictions/conventions
  - very rapid development
- Not following RoR restrictions/conventions
  - "derails" the rapid development

Communication Software 2010-2011

From Active Record to ORM

- ORM can be viewed as active record, or similar, with maximum code generation and support added for many of following:
  - transactions
  - caching
  - relational integrity
  - some implicit persistence
  - mapping of relationships
    - object associations (1:1, 1:N, N:1, M:N) ↔ foreign key constraints
  - flexible mapping between objects and table rows:
    - single object represents single row
    - multiple objects represent single row
    - single object represents multiple rows (SQL join)
  - inheritance relations between persistent objects
  - lazy loading

OR Impedence Mismatch Overview (1/2)

- Current situation
  - programming languages, design languages:
    - usually OO
  - database management systems (DBMS)
    - usually relational

  => many, many developments/applications use both
- Conceptual basis
  - object model:
    - identity, encapsulation of state + behaviour,
    - inheritance, polymorphism
  - relational model
    - relation, attribute, tuple,
    - relation value, relation variable

  => non-trivial mapping: object model ↔ relational model

OR Impedence Mismatch Overview (2/2)

- Possible solutions
  1. Avoid the problem
     - use XML databases
     - use OO languages + OODBMS
     - use non-OO languages + RDBMS
     - use OO languages incorporating relational concepts + RDBMS
  2. Manual mapping
     - hand-code SQL using relational-oriented tools
       e.g. JDBC, ADO.NET
  3. Use DAO layer
     - accept limitations while working to reduce them
     - know when to stop trying to reduce them!
  4. Use ORM framework
     - accept limitations while working to reduce them
     - know when to stop trying to reduce them!
  5. Mix and match
     - e.g. DAO layer, part of which uses ORM

OR Impedence Mismatch Details (1/9)

- Management issues
  - to what extent should type of application or chosen application technology constrain the DB schema?
    - relational model may be used by multiple applications
      - care also needed with database caching
  - need to manage two separate but tightly-coupled models
    - different teams may be responsible for each model
    - evolution/refactoring difficulties
    - which model is to be considered primary?

OR Impedence Mismatch Details (2/9)

- Associations (also called relationships)
  - associations can have following multiplicities
    - object level: 1-1,1-N, N-1 and M-N; unidirectional & birectional
    - relational level: 1-1 and unidirectional N-1 associations only (foreign keys cannot point to collections)
  - relational level cannot model:
    - unidirectional 1-N associations
    - bidirectional N-1 associations
  - relational static model incomplete
    - associations at object level in fact map to keys+joins
    - to generate domain model from relational model
      - information must be added
  - M-N associations map to:
    - one relation for each class (R1, R2) + join relation Rx
    - one N-1 association from Rx to R1
    - one M-1 association from Rx to R2
  - navigating M-N associations between persistent objects
    - performance implications

OR Impedence Mismatch Details (3/9)

- Inheritance
  1. table-per-class (containing only fields specific to that subclass)
     - Fowler's *Class Table Inheritance* pattern
     - efficiency problems: e.g. querying value of class attribute involves join on all relations that are mapped to a derived classes
  2. table-per-concrete-class (usually same as table-per-leaf-class)
     - Fowler's *Concrete Table Inheritance* pattern
     - integrity problems, e.g, uniqueness of id defined in abstract class not easily enforced across the set of relations that are mapped to derived classes
     - not normalized
  3. table-per-inheritance-hierarchy
     - Fowler's *Single Table Inheritance* pattern
     - unlikely to be normalized
     - often needs discriminator field to identify mapped class
     - sparse table: many fields must by nullable
     - efficiency problems reduced if most fields inherited from base class
  4. between options 1 and 3: table-per-class-family

OR Impedence Mismatch Details (4/9)

- Polymorphism
  - inclusion polymorphism in OO languages:
    - mechanism allowing different types to expose different behaviour through the same interface
    - uses inheritance
    - most common use: override superclass method in a subclass (late binding)
  - ORM context
    - can sometimes mimic method overriding
      - depending on chosen inheritance mapping

OR Impedence Mismatch Details (5/9)

- Identity
  - relational model
    - two entries are identical if their content is identical
  - OO model
    - two entries are identical if their object reference is identical
      - c.f. difference between "==" and ".equals" in Java
  - most RDMBS permit duplicate tuples
    - often considered dubious practice from theoretical point of view
    - tuples are indistinguishable in any case
  - same entity queried twice, loaded into different object instances
    - could spell trouble
  - solution?: import OO world into relational world
    - add database-generated key field and create new value for each object instance
    - may be impractical
  - interaction with concurrent access, caching, clustering,...

OR Impedence Mismatch Details (6/9)

- Composition
  - usual understanding:
    - composite deleted → all components deleted
  - OO model: no problem (garbage collection)
  - relational model: must be explicit
    - composite relation is 1-N: problematic
  - possibilities in relational model
    - use database triggers
    - maintain coherence at application level

OR Impedence Mismatch Details (7/9)

- Containment
  - containers
    - OO model: collections, lists, sets
    - relational model: only relations
  - difficulty in modelling OO containers in relational model
  - example: lists with duplicates, solutions:
    - many DBMS allow duplicates, even if theoretically dubious, though not normalized
    - split over two or more tables to normalize

OR Impedence Mismatch Details (8/9)

- Encapsulation
  - OO encapsulation
    - access modifiers: private, protected, public
    - interfaces
  - only relational equivalent: views
  - some problems:
    - use of access modifiers:
      - may make it difficult to access values in order to persist them
      - consequence: access considerations may influence domain model
    - use of OO encapsulation: to control access to data
      - but databases often shared between applications

OR Impedence Mismatch Details (9/9)

- Data retrieval mechanism
  - query-by-example, query-by-API
  - query-by-language
    - use of object query languages
- Object Navigation and Loading
  - OO world
    - cost of navigation is very low
  - relational world
    - cost of navigation (joins, multiple queries) is high
    - particularly across a network
  - Stategies to reduce costs:
    - lazy/on-demand loading of objects linked to retrieved object
    - aggressive loading of objects linked to retrieved object
    - lazy/on-demand loading of attribute values
      - allows query optimization at database communication level

Approaches to ORM (1/3)

- Top-Down
  - relational schema optimized for the domain model
  - may involve generation of a database schema by ORM framework
  - schema oriented towards a single application
    - not easily used by other applications
    - integrity often enforced at application level
  - often uses database-generated primary keys for all relations
    - though most likely breaks normalization
  - common for prototypes / proof of concept

Approaches to ORM (2/3)

- Bottom-Up
  - domain model optimized for the relational schema
  - may involve generation of application code from database schema
  - often uses a 1-1 mapping from tables to classes
    - what about inheritance and encapsulation?
  - objects may lose their associations
    - navigation is via the relational model (inefficient)
  - domain model produced not very understandable to non-developers
  - relational schema not easily evolved
    - code generation cannot generate complete application
  - OR impedance mismatch simply moved up towards UI?

Approaches to ORM (3/3)

- Meet-in-the-Middle
  - domain model optimized for communication with non-developers
  - relational schema optimized for reliability and efficiency
  - most common approach
    - partly due to long life of database schemas

Bibliography for Section 1

- *Persistence in the Enterprise: A Guide to Persistence Technologies*. Roland Barcia, Geoffrey Hambrick, Kyle Brown, Robert Peterson and Kulvir Singh Bhogal. IBM Press, 2008
  `http://my.safaribooksonline.com/9780768680591`
- *Patterns of Enterprise Application Architectures*. Martin Fowler. Addison-Wesley 2002. Summary available at:
  `http://martinfowler.com/eaaCatalog/`
- *J2EE Patterns*
  `http://java.sun.com/blueprints/patterns/`
- *The Object-Relational Impedance Mismatch*. Scott Ambler.
  `http://www.agiledata.org/essays/mappingObjects.html`
- *The Vietnam of Computer Science*. Ted Neward
  `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`
- *Object-Relational Mapping as a Persistence Strategy*. Douglas Minnaar.
  `http://www.eggheadcafe.com/tutorials/aspnet/e957c6de-8400-4748-b42d-027f7a228063/objectrelational-mapping.aspx`
- *Object Relational Mapping Strategies*. ObjectMatter.
  `http://www.objectmatter.com/vbsf/docs/maptool/ormapping.html`

Contents

1. Object Relational Mapping
   - the persistence layer and persistence patterns
   - the object-relational impedance mismatch
   - approaches to ORM
2. Java Persistence Technologies
3. The Java Persistence API
   - entities, the entity manager and persistence units
   - the `Query` interface and JPQL
   - relationships and inheritance
   - database synchronisation & locking and local transactions
   - the entity life-cycle and life-cycle call-back methods
   - dependency injection

JDBC / Top Link

- JDBC: 1997 (v1), 1998 (v2), 2002 (v3), 2006 (v4)
  - based on Microsoft's ODBC
    - based, in turn, on X/Open's SQL CLI
  - foundation: objects for
    - connection
    - statement
    - result set
- Toplink: 1995 (Smalltalk), 1997 (Java)
  - developed by Carleton University spin-off company
  - contained most features expected of modern ORM
  - TopLink Essentials: JPA 1.0 implementation
    - basis for Glassfish JPA v1.0 reference implementation
  - (note that the JDK1 released in 1996)

Enterprise Java Beans v1 and v2 (1/2)

- EJB: 1998 (v1), 2001 (v2), 2003 (v2.1)
  - specification for enterprise components
  - includes entity beans for persistence
    - bean-managed persistence (BMP)
    - container-managed persistence (CMP)
- Differences EJB1 & EJB2 entity beans
  - better support for relationships (1:1, 1:N, M:N)
  - advances in CMP
  - introduction of local interfaces
    - for entity beans: facilitates *Session Facade* pattern

Enterprise Java Beans v1 and v2 (2/2)

- EJB entity beans CMP
  - object-relational mapping (ORM)
  - not use Plain Old Java Objects (POJOs); EJBs must
    - implement special interfaces
    - extend special classes
  - heavyweight and verbose
  - classes automatically persistent
  - persistent objects cannot execute outside EJB container
    - unit testing difficult
  - object-relational mapping not fully specified
    - details left to container implementer
  - defines an object query language (EJBQL)

Hibernate

- Hibernate: 2002 (v1.0)
  - open source product
  - object relational mapping
    - sometimes used together with DAO layer or generic DAOs
  - lightweight
  - uses POJOs
  - persistent objects can execute outside an EJB container
  - object-relational mapping fully specified
  - to make classes persistent
    - explicitly invoke methods on a persistence manager
  - defines an object query language (HQL)
  - uses JDBC
  - Hibernate Annotations + Hibernate EntityManager
    - provides JPA implementation on top of Hibernate Core

iBatis

- iBatis: 2003 (v1.0)
  - open source product (now Apache)
  - lightweight
  - basically, a DAO layer
    - with support for transactions and inheritance
  - all SQL externalised to XML files (called SQLMaps)
  - objects persisted and queried are Java Beans
  - POJOs can be mapped to input parameters of an SQL statement or to results of executing an SQL query
  - uses JDBC

Java Data Objects (JDO)

- Java Data Objects: 2002 (v1.0)
  - specification for persistable objects
  - inspiration from OODBMS, in particular, ODMG specs.
  - general object-data source mapping, not just ORM
  - support for transactions, caching, associations, inheritance
  - uses POJOs
  - to make classes persistent
    - explicitly invoke method on a persistence manager
  - defines an object query language (JDOQL)
  - usually implemented via JDBC

The Java Persistence API (JPA)

- JPA (2006)
  - part of EJB3 spec. but can be used independently
  - inspiration in Hibernate, TopLink, JDO,…
  - specification for flexible object-relational mapping
  - lightweight
  - uses POJOs
  - persistent objects can execute outside an EJB container
  - object-relational mapping fully specified
  - to make classes persistent
    - explicitly invoke an Entity Manager
  - defines an object query language (JPQL)
  - alternative to deployment descriptors: Java annotations

Bibliography for Section 2

- *Persistence in the Enterprise: A Guide to Persistence Technologies*. Roland Barcia, Geoffrey Hambrick, Kyle Brown, Robert Peterson and Kulvir Singh Bhogal. IBM Press, 2008
  `http://my.safaribooksonline.com/9780768680591`

Contents

1. Object Relational Mapping
   - the persistence layer and persistence patterns
   - the object-relational impedance mismatch
   - approaches to ORM
2. Java Persistence Technologies
3. The Java Persistence API
   - entities, the entity manager and persistence units
   - the **`Query`** interface and JPQL
   - relationships and inheritance
   - database synchronisation & locking and local transactions
   - the entity life-cycle and life-cycle call-back methods
   - dependency injection

Entities

- Entity
  - grouping of state treated as single unit
  - can be made persistent
  - normally only created, deleted & updated within a transaction
- JPA entity
  - fine-grained POJO with following restrictions
    - public or protected, no-argument constructor
    - persistent fields are private, protected, or package-private
      - clients access state via accessor or business methods
    - neither class nor its methods nor its fields are `final`
  - normally stored in a single place
  - detachable from / attachable to persistence layer
    - attached: has persistent identity, unique in persistence context
    - detached: can be passed by value to other JVM (if serializable)
    - so also serve as data transfer objects

Entity Metadata

- JPA entities have associated metadata
  - enabling persistence layer to recognize & manage entity
  - specified either using Java annotations or as XML
    - annotations easier
  - "annotation parameters" give more info about annotations
    - annotation parameters called *elements*
    - expressions `element=value` called *element specifications*
- Default metadata
  - JPA defines default values for many annotations
    - particularly element specifications
    - "configuration by exception"
  - ease-of-use (especially first use)
    - c.f. CoC principle of Ruby on Rails
- Java EE container
  - web container + EJB container + persistence provider

Managing Entities

- Entities are managed by an *entity manager*
  - each entity manager is associated with a *persistence context*
  - persistence context:
    - set of managed entity instances that exist in a particular data store
- The **`EntityManager`** API
  - creates and removes persistent entity instances
  - finds entities by the entity's primary key
  - allows queries to be run on entities
    - uses the Java Persistence Query Language (JPQL)
  - detaches entities from / re-attaches entities to persistent storage
- Entity managers may be
  - container-managed
  - application-managed

  differ in transactional behaviour & life-cycle management

Persistence Units

- A persistence unit
  - represents the data contained in a single data store
  - has associated **`EntityManagerFactory`**
    - create entity manager instances with given configuration
- A persistence unit comprises
  - a collection of entities that are managed together
    - i.e. a set of entity classes
  - the corresponding configuration information
    - e.g. persistence provider, data source
- File **`persistence.xml`**:
  - contains the definition of one or more persistence units
  - JPA inside a Java EE container
    - many more things to define in **`persistence.xml`** file

Defining a JPA Entity

- An entity is a class that:
  - satisfies restrictions mentioned above
  - is annotated with **`@Entity`**
  - uses one of the following to denote the primary key:
    - an attribute annotated with **`@Id`**
    - a mutator method (**`getXxx`**) annotated with **`@Id`**
  - if primary key auto-generated
    - annotate also with **`@GeneratedValue`** + element spec
  - if primary key composite
    - annotate with **`@EmbeddedId`** or **`@IdClass`** instead of **`@Id`**
- Defaults element values defined for the **`@Entity`** annotation:
  - entity mapped to relation of same name
  - all attributes of entity persistent
  - each entity attribute mapped to relation attribute of same name

  can be overridden (by using element specifications)

Communication Software 2010-2011

ORM: Entities and Simple Attributes

- Entity class ↔ table:
  - default: table name = unqualified name of entity class
  - if other mapping required: use **`@Table(name="...")`**
  - mapping a class to multiple tables:
    - use annotations **`@SecondaryTables`** & **`@SecondaryTable`**
- Entity attribute whose value is a simple type ↔ column:
  - notes regarding simple types:
    - primitive types, wrapper classes of primitive types, Strings, temporal types, enumerated types, serializable objects,…
    - can be denoted explicitly with **`@Basic`** but not necessary unless adding elements, e.g. **`@Basic(fetch=FetchType.LAZY)`**
    - set of allowable types for primary key (recall denoted with **`@Id`** or **`@EmbeddedId`** / **`@IdClass`**) is subset of simple types
  - default: attribute is persistent and column name = attribute name
    - not persistent: use **`transient`** modifier or **`@Transient`** annotation
  - if other mapping required: use **`@Column(name="...")`**

Communication Software 2010-2011

Constraints on Schema Generation

- To specify unique constraints
  - use **`unique`** element of **`@Column`** & **`@JoinColumn`**
  - use **`uniqueConstraints`** element of **`@Table`** & **`@SecondaryTable`**
- To specify not-null constraint
  - use **`nullable`** element of **`@Column`** & **`@JoinColumn`**
- To specify string-length constraints
  - use **`length`** element of **`@Column`** (no such element in **`@JoinColumn`**)
  - default is 255
- To specify floating-point constraints
  - use **`precision`** and **`scale`** elements of **`@Column`** (not **`@JoinColumn`**)
- To specify a particular SQL DDL string for a column
  - Use **`columnDefinition`** element of **`@Column`**, **`@JoinColumn`**, **`@PrimaryKeyJoinColumn`**, **`@DiscriminatorColumn`**

Communication Software 2010-2011

The `Query` Interface

- Method **`find()`** used to find single entity by primary key
  - more complicated search needs **`Query`** inteface
  - same interface used for bulk updates and deletes (new to EJB3)
- To obtain an object implementing the **`Query`** interface:
  - two entity manager methods for dynamic queries

```
public Query createQuery(String jpqlString)
public Query createNativeQuery(String sqlString)
```

  - one entity manager method (with 3 variants) for static queries

```
public Query createNamedQuery(String sqlOrJpqlString)
```

- Main methods on the **`Query`** interface

```
public List getResultList()
public Object getSingleResult()
public int executeUpdate() // bulk update or delete
public Query setMaxResults(int maxResult)
public Query setFirstResult(int startPosition)
public Query setParameter(String name Object value)
Public Query setFlushMode(FlushModetype flushMode)
```

Communication Software 2010-2011

The Java Persistence Query Language

- Similar to SQL:
  - **SELECT FROM WHERE**, **GROUP BY**, **HAVING**, **ORDER BY**
  - **UPDATE SET WHERE**
  - **DELETE FROM WHERE**

  and even more similar to HQL (Hibernate Query Language)
- Subqueries allowed in **WHERE** and **HAVING** clauses
- Set functions for use in grouping / aggregate queries
  - **AVG**, **COUNT**, **MAX**, **MIN**, **SUM**
- Navigation through entity graphs
  - using path expressions (dot notation)
    - allowed in **SELECT**, **SET**, **FROM**, **WHERE** clauses
  - though illegal for path expressions to navigate beyond
    - collection-valued relationship attribute
    - persistent attribute

Example 1: Simple CRUD (1/3)

```
// source: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
@Entity
public class Employee {
   @Id private int id;
   private String name;
   private long salary;

   public Employee(){}
   public Employee(int id) {this.id = id;}

   public int getId() {return id;}
   public void setId(int id) {this.id = id;}
   public String getName() {return name;}
   public void setName(String name) {this.name = name;}
   public long getSalary() {return salary;}
   public void setSalary(long salary) {this.salary = salary;}
}
```

Example 1: Simple CRUD (2/3)

```
import javax.persistence.*;
import java.util.Collection;

public class EmployeeService {
   protected EntityManager em;

   public EmployeeService(EntityManager em) {
       this.em = em;
   }

   public Employee createEmployee(int id, String name, long salary) {
       Employee emp = new Employee(id);
       emp.setName(name);
       emp.setSalary(salary);
       em.persist(emp);
       return emp;
   }

   public Employee findEmployee(int id) {
       return em.find(Employee.class, id);
   }
```

Example 1: Simple CRUD (3/3)

```
   public RemoveEmployee(int id) {
       Employee.emp = findEmployee(id);
       if (emp != null){
           em.remove(emp);
       };
   }

   // entity is managed so change made to persistent representation
   public raiseEmployeeSalary (int id, long raise) {
       Employee.emp = findEmployee(id);
       if (emp != null){
           emp.setSalary(emp.getSalary()+ raise);
       };
       return emp;
   }

   public Collection<Employee> findAllEmployees() {
       Query query = em.createQuery("SELECT e FROM Employee e");
       return (Collection<Employee>) query.getResultList();
   }

}
```

Example 2: Single-Table Mapping

```
// source: JBoss EJB3 tutorial, B. Burke
@Entity
@Table(name="AUCTION_ITEM")
public class Item {
  private long id;
  private String description;
  private String productName;
  private Set<Bid> bids = new HashSet();
  private User seller;

  @Id
  @GeneratedValue(stategy=AUTO)
  @Column(name="ITEM_ID")
  public long getId() {
      return id;
  }
  public void setId(long id) {
      this.id = id;
  }
  ….
}
```

```
create table AUCTION_ITEM
(
ITEM_ID Number,
DESC varchar(255),
ProductName varchar(255),
USER_ID Number
);
```

Example 2: Multi-Table Mapping (1/2)

```
// source: JBoss EJB3 tutorial, B. Burke
@Entity
@Table(name="OWNER")
@SecondaryTable(name="ADDRESS",
  pkJoinColumns=
   @PrimaryKeyJoinColumn(name="ADDR_ID",
   referencedColumnName="ADDR_ID"))
public class Owner {
    private long id;
    private String name;
    private String street;
    private String city;
    private String state;

    @Id
    @GeneratedValue(stategy=AUTO)
    @Column(name="OWNER_ID")
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
```

```
create table OWNER
(
OWNER_ID Number,
NAME varchar(255),
ADDR_ID Number,
);

create table ADDRESS
(
ADDR_ID Number,
STREET varchar(255),
CITY varchar(255),
STATE varchar(255)
);
```

Example 2: Multi-Table Mapping (2/2)

```
…

@Column(name="STREET",
     secondaryTable="ADDRESS")
public String getStreet() {
    return street;
}
public void setStreet(String street) {
    this.street = street;
}


@Column(name="CITY",
     secondaryTable="ADDRESS")
public String getCity() {
    return city;
}
protected void setCity(String city) {
    this.city  = city;
}

…
```

```
create table OWNER
(
OWNER_ID Number,
NAME varchar(255),
);

create table ADDRESS
(
ADDR_ID Number,
STREET varchar(255),
CITY varchar(255),
STATE varchar(255)
);
```

ORM: Relationships (1/2)

- Entity attribute whose value is another entity
  - can be annotated with one of the following multiplicities:
    - **`@OneToOne`**
    - **`@OneToMany`**
    - **`@ManyToOne`**
    - **`@ManyToMany`**
  - foreign key denoted with **`@JoinColumn(name="...")`**
- Every association/relationship is said to have an "owning side"
  - a bidirectional association is also said to have an "inverse side"
  - **`OneToMany`** & **`ManyToOne`**: owner must be many side
  - **`OneToOne`**: owner must be side that contains foreign key
  - inverse side uses **`mappedBy`** element of multiplicities annotation to identify corresponding attribute on owning side
- Composition relations denoted using the element spec.:
  - **`cascade=REMOVE`** on the inverse side

ORM: Relationships (2/2)

- Join table required for
  - many-to-many relations
  - unidirectional one-to-many relations
  - Example (bidirectional many to many):

```
@Entity
Public class Employee
@Id private ind id;
Private String name;
@ManyToMany // inverse side has @ManyToMany(mappedBy="projects")
@JoinTable(name="EMP_PROJ", // annotation on owning side only
     joinColumns=@JoinColumn(name="EMP_ID"),
     inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
Private Collection<Project> projects;
```

- Join column
  - to owning side described in **`joinColumns`** element
  - to inverse side described in **`inverseJoinColumns`** element
  - default names used if not specified (also for join table)

ORM: Inheritance

- Base class of inheritance hierarchy can be annotated via:
  - **`@Inheritance(InheritanceType=X)`**

  where **X** is one of the following (default is 1)
  1. **`SINGLE_TABLE`**: *Single Table Inheritance* pattern
  2. **`JOINED`**: *Class Table Inheritance* pattern
  3. **`TABLE_PER_CLASS`**: *Concrete Table Inheritance* pattern
- 2 & 3: hierarchy may contain classes that are not persistable
  - "transient classes" (non-entities) or "mapped superclasses".
- **`SINGLE_TABLE`** and sometimes **`JOINED`** inheritance
  - base clase has **`@DiscriminatorColumn`** annotation
    - elements specify properties of the discriminator column, e.g. type
  - each concrete entity uses **`@DiscriminatorValue`** annotation
- Queries on class hierarchy are polymorphic

Database Synchronisation

- Database writes queued until **`EntityManager`** synchronises
  - calls to **`EntityManager`** methods **`persist()`**, **`merge()`**, **`remove()`**
  - not coordinated with updates / deletes sent via **`Query`** interface
- Writes flushed to database according to one of policies:
  - when a transaction commits
    `javax.persistence.FlushModeType = COMMIT`
  - before a query concerning this data is executed
    `javax.persistence.FlushModeType = AUTO`
- Explicit management of flushing:
  - to find out / change flushing policy
    `EntityManager.getFlushMode()`
    `EntityManager.setFlushMode(FlushModeType flushMode)`
    `Query.setFlushMode(FlushModeType flushMode)`
  - to flush changes to database explicitly
    `EntityManager.flush()` (c.f. also **`EntityManager.refresh()`**)

Database Locking

- Default locking policy: optimistic concurrency control
  - at transaction commit
    - if data changed with respect to in-memory version: rollback
    - if not, go ahead with the commit
  - based on use of a version field
    - entity field annotated with **`@Version`**
    - version number checked to detect changes
- Other possibilities
  - set isolation level globally (see transactions section)
  - method **`EntityManager.lock()`**, pass entity as parameter
    - read lock: parameter **`LockModeType.READ`**
      - a transaction update may fail if other transaction has a read lock
    - write lock: parameter **`LockModeType.WRITE`**
      - forces increments of version field whether entity updated or not

Resource-Local Transactions in JPA

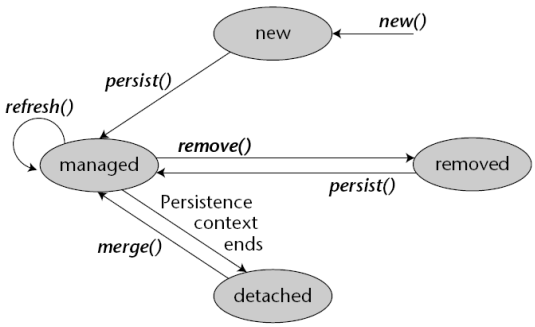
- Entities can be used inside resource-local transactions
  - transactions completely under control of the application
  - via **`EntityTransaction`** interface
- The **`EntityTransaction`** interface
  - similar to JTA **`UserTransaction`** interface (see later)
  - operations: **`begin`**, **`commit`**, **`rollback`**, **`isActive`**, **`setRollbackOnly`**, **`getRollbackOnly`**,
  - operations implemented via JDBC **`Connection`** transaction methods
  - obtained by **`EntityManager.getTransaction()`**
- Not generally used when in Java EE environment
  - JTA transactions recommended instead (see later)

Life-Cycle Call-Back Methods

- Methods called by persistence provider
  - entity optionally registers for call-back on life-cycle events
    - by annotating methods of bean class
  - restrictions on method that can be life-cycle call-back:
    - has no arguments, returns **`void`**, doesn't throw application exceptions
  - may be placed in a separate listener class
    - declared via annotation **`@EntityListener`**
- Possible entity life-cycle call-back methods
  - method annotated **`@PrePersist`** / **`@PostPersist`**
    - called before/after executing **`EntityManager.persist()`**
  - method annotated **`@PreRemove`** / **`@PostRemove`**
    - called before/after executing **`EntityManager.remove()`**
  - method annotated **`@PreUpdate`** / **`@PostUpdate`**
    - called before/after database synchronization
  - method annotated **`@PostLoad`**
    - called after instance loaded via EM **`find()`** or **`getReference()`**

Entity Life-Cycle

Source: Pro EJB3. Java Persistence API. Mike Keith et al



- Transactional persistence context
  - persistence context ends when transaction ends
- Extended persistence context
  - persistence context ends when caller is destroyed

Inversion of Control (IoC)

- Inversion of control (IoC) refers to situation where
  - an external system calls your code
  - rather than your code calling the external system
- Sometimes referred to as the 'Hollywood principle':
  - "don't call us, we'll call you".
- Example: EJB callback methods

Dependency Injection (DI) (1/2)

- To obtain a given service, components often need to know:
  - which other components they need to communicate with
  - where to locate them
  - how to communicate with them
- Naive approach
  - embed service location / instantiation logic in code of its clients
  - problems with naive approach:
    - changes in service access imply changes in code of many clients
    - difficult to unit test client components without using real service
- Dependency injection (DI) approach
  - clients declare their dependency on the services
  - "external code" *injects* the service
    - i.e. assumes responsibility for locating and instantiating the services and then supplies the service reference
  - "external code": often DI container / DI framework

Dependency Injection (DI) (2/2)

- Dependency injection
  - is a particular form of IoC
  - ensures that configuration of services is separated from their use
  - dependencies configured externally in one place
    - for multiple clients
- Externalising service-access dependency makes code
  - more reusable
  - more testable: easy to inject dependency on mock service
  - more readable
- Ways of carrying out dependency injection
  - constructor injection
  - setter injection
  - interface injection

Example: Dependency Injection (1/3)

- Obtaining application-managed entity manager, Java SE
  - using **`Persistence`** class, no dependency injection

```
// source: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
public class EmployeeClient {
   public static void main(String[] args) {
      EntityManagerFactory emf
      = Persistence.createEntityManagerFactory("EmployeeService");
      EntityManager em = emf.CreateEntityManager();

      Collection emps
      = em.CreateQuery("SELECT e FROM Employee e").getResultList();
      for(Iterator I = emps.iterator(); i.hasNext();){
         Employee e = (Employee) i.next();
         System.out.println(e.getId() + ", " + e.getName());
      }
   em.close();
   emf.close();
}
```

Example: Dependency Injection (2/3)

- Obtaining application-managed entity manager, Java EE
  - injection of Persistence Unit to obtain **`EntityManagerFactory`**
  - Java EE container peforms the injection

```
// source: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
public class LoginServlet extends HttpServlet {
   @PersistenceUnit(unitName="EmployeeService")
   EntityManagerFactory emf;
   protected void doPost(HttpServletRequest request,
                         HttpServletResponse response){
      String userId = request.getParameter("user")
      // check valid user
      EntityManager em = emf.createEntityManager();
      try{
         User user = user.find(User.class, userId);
         if (user == null) {
            // return error page...
         }
      } finally {em.close();}
      // ...
   }
}
```

Example: Dependency Injection (3/3)

- Obtaining container-managed entity manager, Java EE
  - injection of Persistence Context to obtain **`EntityManager`**
  - Java EE container peforms the injection
  - explanation of **`@stateless`** annotation: see later

```
// source: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
@Stateless
public class ProjectServiceBean implements ProjectService {
   @PersistenceContext(unitName="EmployeeService")
   EntityManager em;

   public void assignEmployeeToProject(int empId, int projectId) {
      Project project = em.find(Project.class, projectId);
      Employee employee = em.find(Employee.class, empId);
      project.getEmployees().add(employee);
      employee.getProjects().add(project);
   }
   // ...
}
```

Bibliography for Section 3

- *Pro EJB3. Java Persistence API*. Mike Keith and Merrick Schincariol. Apress, 2006
  `http://books.google.com/books?id=fVCuB_Xq3pAC`
- *The Java Persistence API*. Sun Microsystems
  `http://java.sun.com/javaee/technologies/persistence.jsp`
- *The Java EE 5 Tutorial. Part V: Persistence*. Sun Microsystems
  `http://java.sun.com/javaee/5/docs/tutorial/doc/bnbpy.html`
- *Inversion of Control Containers and the Dependency Injection pattern.* Martin Fowler, 2004
  `http://www.martinfowler.com/articles/injection.html`