A Brief Introduction to Java Persistence API & Hibernate

Department of Computer Science & Engineering

Christopher M. Bourke
cbourke@cse.unl.edu

Abstract

Abstract

Complex applications make frequent use of an underlying data model. In development, much effort is put toward the mundane tasks of coding CRUD (Create-Retrieve-Update-Destroy). Recent developments of Data Access Object frameworks have freed developers from needing to worry about loading, persisting, and managing data, keeping them closer to the application layer. In this seminar, we will introduce the basic concepts and use of one such framework: Java Persistence API (JPA) using Hibernate.

Java Persistence API

Java Persistence API (JPA) is a framework for managing relational data

- Provides an abstract data layer between a database and Plain Old Java Objects
- API (`javax.persistence`) provides methods for querying and managing data
- JPQL (Java Persistence Query Language) – an SQL-like query language
- Built on top of JDBC
- JPA 1.0 (May 2006)
- JPA 2.0 (Dec 2009)
- Intended to replace heavy-weight EJB entity beans

A Simple Database

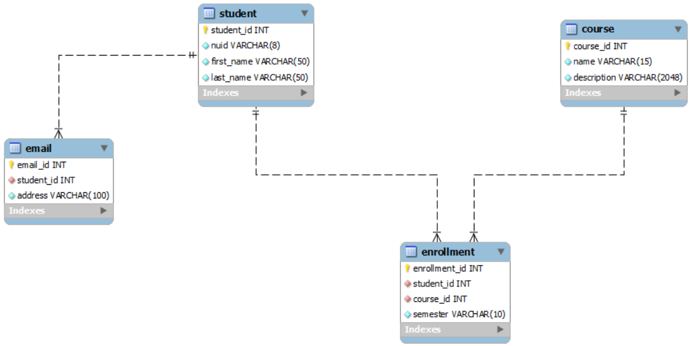

Figure: Database to support course enrollments

JDBC CRUD

Using JDBC:

- Need to manage our own connections
- Need to pull and handle each record column by column
- Need to join or make additional queries to pull related objects
- Lots of boiler-plate CRUD
- Let's take a look at an example...

JPA CRUD: Automated For Us!

Using JPA:

- We can annotate our java classes to map them to tables, columns
- (Alternatively: all relations can be enumerated in an XML configuration file)
- Basic CRUD is taken care of for us
- An `EntityManager` handles loading and persisting to the database
- JPQL is standardized and makes queries platform independent
- Now we can just use objects instead of worrying about loading and saving them!

JPA Advantages & Disadvantages

Additional advantages

- Closer to OOP
- Portable across application servers, persistence products
- Can be configured to provide connection pooling, caching, etc.

Disadvantages

- Slight performance hit (due to reflection, auto-generated queries)
- Resource hog
- Good design needs a clean object-database mapping

Persistence Until Configuration

- A *persistence unit* is a data source configuration
- Defined in `persistence.xml` configuration file
- Defines configuration such as database URL, login, JDBC driver, etc.
- Object Relational Mapping (ORM) can be configured here or via annotations
- Let's take a look at an example. . .

Class annotations

- JPA annotations defined in `javax.persistence`
- `@Entity` - makes a POJO into a JPA entity
- Note: JPA Entities are expected to be serializable
- `@Table` - maps the object to a schema/table in the database
- `@Id` - identifies which field is the primary key (optional)
- `@GeneratedValue(strategy=GenerationType.AUTO)`
- `@Column(name="...", nullable=false)`
- Some type coercion support is available
- Some libraries provide add-ons for reflective type coercion (Joda time)

Join Annotations

- `@OneToOne`
- `@ManyToOne`
- `@OneToMany`
- `@ManyToMany`
- JPA relations are uni-directional; need to explicitly define any bi-directional relationships
- "Many" is implemented via `java.util.Set` interface
- `@JoinColumn` - allows you to specify which column should be used to join
- Many-to-many requires a `@JoinTable` specification
- If not a *pure* join table, an intermediate Entity needs to be defined (example: `http://en.wikibooks.org/wiki/Java_Persistence/ManyToMany`)

Loading

- JPA supports two loading strategies: `LAZY` (hibernate default) and `EAGER`
- Eager fetching will completely load an entity and its related entities even if you don't need them
- Lazy fetching only loads an entity when its needed (when you get a field)
- Example: `@ManyToOne(fetch=FetchType.LAZY)`

Other Annotations & Issues

- `@Transient` – identifies a non-persistent field (JPA will ignore)
- JPA works through reflection
- Requires an available default (no-arg) constructor
- Hibernate is a particular implementation of JPA (with Hibernate-specific features added on)
- Hibernate only throws runtime exceptions, but still good practice to `try-catch-finally`

Example Entities

- `StudentEntity`
- `EmailEntity`
- `CourseEntity`

Using JPA in Java

Basic JPA usage pattern:

- Create an `EntityManagerFactory` (automatically loads a persistence unit from the `persistence.xml` config file)
- Create an `EntityManager` from the EMF
- Start a transaction
- Use the entity manager to load, update, persist JPA Entities
- Rollback or commit the transaction
- Close entity manager(s), factories

Using an Entity Manager

`EntityManager em = ...`

- `em.find(Class c, Object o)` – loads an entity from the database
- `em.persist(Object o)` – Persists the object to the database (update or insert)
- `em.detach()` – Removes the entity from the entity manager
- `em.merge()` – reloads the entity with values from the database (overwriting any in-memory changes)
- `em.flush()` – synchs up the database with all managed entities (saves all)
- `em.remove(Object o)` – deletes the entity from the database
- `em.refresh(Object o)` – syncs up the object with the database

Java Persistence Query Language

- JPA provides an SQL-like query language: JPQL
- Still provides a `NativeQuery` interface for regular SQL
- Allows you to refer to Objects rather than tables

```
String query = "FROM StudentEntity se WHERE se.nuid
      = :nuid";
...
StudentEntity se = (StudentEntity) em.createQuery(
     query)
.setParameter("nuid", myNUID)
.getSingleResult();
```

Handling Inheritance I

JPA supports several strategies for modeling inheritance

- Single Table Mapping (`InheritanceType.SINGLE_TABLE`) – One table corresponds to all subclasses. Every record has *every* possible state (even if it is not used in a subclass)
- Joined (`InheritanceType.JOINED`) – Additional state provided in subclasses is provided in a separate table; a subclass instance is generated by joining all the way up the inheritance hierarchy (multiple inserts/deletes/updates are also needed)
- Table Per Class (`InheritanceType.TABLE_PER_CLASS`) - One table is defined for each class in the hierarchy; common state is repeated in each table

Handling Inheritance II

Illustrative example for Single Table Mapping stategy: `unl.cse.employee`
To discriminate which class a record corresponds to, we need a `DiscriminatorColumn`

- Column defined in the superclass
- `DiscriminatorType` can be `STRING`, `CHAR`, or `INTEGER` (or you can allow the provider to choose)
- Subclasses can define `DiscriminatorValue`
- Subclasses can have additional annotations for subclass state

Handling Inheritance III

Good resources:

- http://openjpa.apache.org/builds/1.0.4/apache-openjpa-1.0.4/docs/manual/jpa_overview_mapping_inher.html
- http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/jpa_overview_mapping_discrim.html

References & Resources

- Hibernate Download: http://www.hibernate.org/downloads.html
- Sun JPA Tutorial: http://java.sun.com/javaee/5/docs/tutorial/doc/bnbpz.html
- Another JPA Tutorial: http://schuchert.wikispaces.com/JPA+Tutorial+1+-+Getting+Started
- Using JPA in Eclipse: http://wiki.eclipse.org/EclipseLink/Examples/JPA