Java Persistence API

Good Morning Everybody...
Lets Start our journey to Java Persistence...

What is Data Persistence?

- Persistence is making some form of data last across multiple executions of an application.

- Actually it refers to the saving of entities within an application to some storage media.

What is an Entity ?

- In software development an entity contains data but usually no logic of significance.

- Entities are passed between units of a program and encapsulate properties that the program uses.

- Entities in JAVA are POJOs (Plain Old Java Objects).

- Entities can be anything – users, products, orders, and more.

Storage media

- Flat Files
- XML / JSON Files
- Database

Files are not the best option to store large and complex data in terms of efficiency and data association.

Databases

- Relational Database Systems
  - Oracle, Microsoft SQL Server

- Schema-less Databases / NoSQL
  - Key-Value stores : Apache Cassandra, Redis
  - Graph Databases: Neo4j

Databases

- Relational databases are the most common databases in use, currently backing nearly all major enterprise software.

- We will talk about bridging the gap between relational databases and program entities.

Object Relational Mapping

- Storing object-oriented entities in a relational database is often not a simple task and requires a great deal of repetitive code and conversion between data types.

- This is what ORM solutions try to solve.

- Before we move on, we should understand the problem of persisting entities.

Object Relational Mapping

```
public Product getProduct(long id) throws SQLException {
 try(Connection connection = this.getConnection();
 PreparedStatement s = connection.prepareStatement("SELECT * FROM dbo.Product WHERE productId = ?")) {
    s.setLong(1, id);
    try(ResultSet r = s.executeQuery())
    {
     if(!r.next()){return null;}
     Product product = new Product(id);
     product.setName(r.getNString("Name"));
     product.setDescription(r.getNString("Description"));
     product.setDatePosted(r.getObject("DatePosted", Instant.class));
     product.setPurchases(r.getLong("Purchases"));
     product.setPrice(r.getDouble("Price"));
     product.setBulkPrice(r.getDouble("BulkPrice"));
     product.setMinimumUnits(r.getInt("MinimumUnits"));
     product.setSku(r.getNString("Sku"));
     product.setEditorsReview(r.getNString("EditorsReview"));
     return product;
    }
 }
}
```

Object Relational Mapping

```
public void addProduct(Product product) throws SQLException {
        try (Connection connection = this.getConnection();
             PreparedStatement s = connection.prepareStatement(
                     "INSERT INTO dbo.PRODUCT (Name, Description, DatePosted," +
                             "Purchases, Price, BulkPrice, MinimumUnits, Sku," +
                             "EditorsReview) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)",
                     new String[]{"ProductId"}
             )) {
            s.setNString(1, product.getName());
            s.setNString(2, product.getDescription());
            s.setObject(3, product.getDatePosted(), JDBCType.TIMESTAMP);
            s.setLong(4, product.getPurchases());
            s.setDouble(5, product.getPrice());
            s.setDouble(6, product.getBulkPrice());
            s.setInt(7, product.getMinimumUnits());
            s.setNString(8, product.getSku());
            s.setNString(9, product.getEditorsReview());
            if (s.executeUpdate() != 1)
                throw new SaveException("Failed to insert record.");
            try (ResultSet r = s.getGeneratedKeys()) {
                if (!r.next())
                    throw new SaveException("Failed to retrieve product ID.");
                product.setProductId(r.getLong("ProductId"));
            }
        }
    }
```

Object Relational Mapping

- We observe lot of repetitive code for simple actions.

- Probably a developer would like to write a set of utilities to make things simpler.

- What if this was already written and tested extensively for you? Wouldn’t the above code be easier?

Object Relational Mapping

- The solution to this problem is ORMs.

- ORMs make developer life easier by handling all repetitive code itself and simplifying CRUD operations.

Object Relational Mapping

```
public Product getProduct(long id) {
        return this.getCurrentTransaction().get(Product.class, id);
}

public void addProduct(Product product) {
        this.getCurrentTransaction().persist(product);
}
```

Object Relational Mapping

- Of course, that’s not all there is to it. An ORM can’t just “know” what tables and columns to map an entity or entities to, nor can it always know how the data types should line up.

- When using an ORM, you must create formal mapping instructions telling the ORM how to map your entities.

- This can take many different forms depending on which ORM you use, which is kind of a problem.

- After you create dozens or hundreds of entity mappings using the proprietary format of a particular ORM, it becomes extremely difficult to switch to another ORM at any given point in the future.

- So there is a need to have a common ORM API.

JPA

- JPA provides a standard ORM API.

- Most popular providers:
  - EclipseLink (Reference implementation)
  - Hibernate

Mapping Entities To Tables Using JPA Annotations

```
@Entity
public class Author implements Serializable {
…..
}
@Entity(name=”AuthorEntity”)
public class Author implements Serializable {
...
}
```

By default the table name in database which an entity maps, is the entity name.

Mapping Entities To Tables Using JPA Annotations

```
@Entity
@Table(name = “Authors”)
public class Author implements Serializable {
...
}
@Entity(name = “AuthorsEntity”)
@Table(name = “Authors”)
public class Author implements Serializable {
...
}
```

By using the @Table annotation, you could specific table name for your entity.

Creating Simple IDs

- By using @Id Annotation either on field or field accessor method.

- As a best practice, entity IDs (primary keys) should be longs or Longs java data types.

- Typically you want your entity IDs to be automatically generated.

- By using @GeneratedValue annotation you may specify the ID generation strategy.

Creating Simple IDs

```
@Entity
...

public class Book  implements Serializable {
 ...

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public long getId() {
        return this.id;
    }
 ...
}
```

- This kind of generation strategy is compatible with AUTO_INCREMENT columns and indicates that the database column the ID is stored can generate its own value automatically.

Creating Composite IDs

- Composite IDs are those IDs that consist of multiple fields/columns-.

- There are 2 ways to define Composite IDS:
  - By using @Id & @IdClass annotations
  - By using @EmbeddedId & @Embeddable annotations.

Creating Composite IDs

```
public class JoinTableCompositeId implements Serializable
{
    private long fooParentTableSk;
    private long barParentTableSk;

    public long getFooParentTableSk() { ... }
    public long getBarParentTableSk() { ... }

}

@Entity
@Table(name = "SomeJoinTable")
@IdClass(JoinTableCompositeId.class)
public class JoinTableEntity implements Serializable
{
    private long fooParentTableSk;
    private long barParentTableSk;
 ...
    @Id
    public long getFooParentTableSk() { ... }

    @Id
    public long getBarParentTableSk() { ... }
 ...
}
```

Creating Composite IDs

```
@Embeddable
public class JoinTableCompositeId implements Serializable
{
    private long fooParentTableSk;
    private long barParentTableSk;

    public long getFooParentTableSk() { ... }
    public long getBarParentTableSk() { ... }

}

@Entity
@Table(name = "SomeJoinTable")
public class JoinTableEntity implements Serializable
{
    private JoinTableCompositeId id;

    @EmbeddedId
    public JoinTableCompositeId getId() { ... }

}
```

Creating Composite IDs

- But why we need extra class for Composite IDs?

JoinTableEntity entity = entityManager.find(JoinTableEntity.class, id);

- The find method doesn’t accept multiple ID arguments. It accepts only one ID argument, so you must create a JoinTableCompositeId instance with which to locate the entity:

JoinTableCompositeId compositeId = new JoinTableCompositeId();

compositeId.setFooParentTableSk(id1);

compositeId.setBarParentTableSk(id2);

JoinTableEntity entity = entityManager.find(JoinTableEntity.class, compositeId);

Mapping Basic Data types

- Properties of type short and Short are mapped to SMALLINT, INTEGER, BIGINT, or equivalent fields.
- int and Integer properties are mapped to INTEGER, BIGINT, or equivalent SQL data types.
- long, Long, and BigInteger properties are mapped to BIGINT or equivalent fields.
- Properties of type float, Float, double, Double, and BigDecimal are mapped to DECIMALor equivalent SQL data types.
- byte and Byte properties are mapped to BINARY, SMALLINT, INTEGER, BIGINT, or equivalent fields.
- Properties of type char and Char are mapped to CHAR, VARCHAR, BINARY, SMALLINT,INTEGER, BIGINT, or equivalent fields.
- Properties of type boolean and Boolean are mapped to BOOLEAN, BIT, SMALLINT, INTEGER,BIGINT, CHAR, VARCHAR, or equivalent fields.
- byte[] and Byte[] properties are mapped to BINARY, VARBINARY, or equivalent SQL data types.
- char[], Character[], and String properties are mapped to CHAR, VARCHAR, BINARY,VARBINARY, or equivalent SQL data types.
- Properties of type java.util.Date and Calendar are mapped to DATE, DATETIME, or TIME fields, but you must supply additional instructions using @Temporal.
- Properties of type java.sql.Timestamp are always mapped to DATETIME fields.
- Properties of type java.sql.Date are always mapped to DATE fields.
- Properties of type java.sql.Time are always mapped to TIME fields.
- Enum properties are mapped to SMALLINT, INTEGER, BIGINT, CHAR, VARCHAR, or equivalent fields. By default enums are stored in their ordinal form, but you can alter this behavior using @Enumerated.
- Any other properties implementing Serializable are mapped to VARBINARY or equivalent SQL data types and converted using standard Java serialization and deserialization.

Mapping Basic Data types

- Eager Fetching => @Basic(fetch = FetchType.EAGER)

Eager fetching means the field value is retrieved from the database at the same time the entity is retrieved from the database. This may or may not involve additional SQL statements, depending on whether the field value resides in a different table.

- Lazy Loading => @Basic(fetch = FetchType.LAZY)

Lazy loading causes the provider to retrieve the value only when the field is accessed — it is not retrieved when the entity is initially retrieved. Not specifying the fetch attribute (or explicitly specifying FetchType.EAGER) is a requirement that the provider fetch the value eagerly. However, specifying FetchType.LAZY is only a hint to fetch the value lazily. The provider may not support lazy access and may fetch the value eagerly anyway.

Specifying Column Names & Other Details

```
@Entity
public class Book implements Serializable {

    private long id;
    private String isbn;
    private String title;
    private String author;

@Id
@GeneratedValue( strategy = GenerationType.IDENTITY)
    public long getId() {
        return id;
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

}
```

Specifying Column Names & Other Details

- By default, JPA maps entity properties to columns with the same name.

- For example the id, isbn, title and author properties of the above entity are automatically mapped to database columns named Id, Isbn, Title and Author respectively.

- By using @Column annotation you may specify (and you should!!!!) several details regarding the column in database, through annotation attributes.

- The most basic configuration is the name. This allows you to completely decouple the entity property name from the database column name.Which is something that you want !!!

Specifying Column Names & Other Details

```
@Entity
public class Book implements Serializable {

    private long id;
    private String isbn;
    private String title;
    private String author;

@Id
@GeneratedValue( strategy = GenerationType.IDENTITY)
    public long getId() {
        return id;
    }
@Column (name = “BookIsbn”)
    public String getIsbn() {
        return isbn;
    }
@Column(name = “Booktitle”, length = 100)
    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

}
```

Persistence Unit

- A persistence unit is a configuration and a set of entity classes logically grouped together.

- This configuration controls the javax.persistence.EntityManager instances attached to the persistence unit, and an EntityManager in a particular persistence unit can manage only the entities defined in that persistence unit.

- For example, given two persistence units Foo and Bar, an EntityManager instantiated for persistence unit Foo can manage only the entities defined in persistence unit Foo. It cannot manage the entities defined only in Bar.

- However, it’s possible to define an entity in multiple persistence units. If an entity is defined in both Foo and Bar, an EntityManager instantiated for persistence unit Foo or Bar can access the entity.

Persistence Unit

- Persistence Unit is configured in /WEB-INF/classes/META-INF directory inside of a WAR file and therefore its scope is application wide.

- The Persistence Unit scope defines what code can access the persistence unit.

- persistence.xml example

```
 <?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
 version="2.1">
 <persistence-unit name="EntityMappings" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
 <non-jta-data-source> java:/comp/env/jdbc/EntityMappings </non-jta-data-source>
 <exclude-unlisted-classes>false</exclude-unlisted-classes>
 <validation-mode>NONE</validation-mode>
 <properties>
 <property name="javax.persistence.schema-generation.database.action" value="none" />
 </properties>
 </persistence-unit>
</persistence>
```

Using the Persistence API

```
public class BookRepo {

….

        @PersistenceContext("EntityMappings")

         EntityManagerFactory factory;

….

}
```

In a Container that supports DI , with this statement the Container creates and Injects an EntityManagerFactory instance based on the specified PersistenceUnit. From the factory instance , we will create EntityManager instances, that we will use in order to perform operations using our Entities.

Using the Persistence API

```
public class BookRepo {
…
 @PersistenceContext("EntityMappings")
  EntityManagerFactory factory;

 public void saveBook(Book book){
        EntityManager em = null;
        EntityTransaction tnx = null;
        try{
             em = this.factory.createEntityManager();
             tnx = em.getTransaction();
         tnx.begin();
         em.persist(book);
         tnx.commit();
        }
         catch(Exception e){
          if(transaction != null && transaction.isActive())
          transaction.rollback();
         }
          finally{
          if(manager != null && manager.isOpen())
          manager.close();
        }
    }
}
```

Defining Relationships Between Entities

- Relationship types:
  - 1-1
  - 1-N
  - M-N

1-1 Relationship

```
@Entity(name = "ORDER_INVOICE")
public class Invoice {

    @Id
    @Column(name = "INVOICE_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long invoiceId;

    @Column(name = "ORDER_ID")
    private long orderId;

    @Column(name = "AMOUNT_DUE", precision = 2)
    private double amountDue;

    @Column(name = "DATE_RAISED")
    private Date orderRaisedDt;

    @Column(name = "DATE_SETTLED")
    private Date orderSettledDt;

    @Column(name = "DATE_CANCELLED")
    private Date orderCancelledDt;

    @OneToOne(optional=false)
    @JoinColumn(name = "ORDER_ID")
    private Order order;
    ...
    //getters and setters goes here
}
```

```
@Entity(name = "ORDERS")
public class Order {

    @Id
    @Column(name = "ORDER_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long orderId;

    @Column(name = "CUST_ID")
    private long custId;

    @Column(name = "TOTAL_PRICE", precision = 2)
    private double totPrice;

    @Column(name = "OREDER_DESC")
    private String orderDesc;

    @Column(name = "ORDER_DATE")
    private Date orderDt;

    @OneToOne(optional=false,mappedBy="order",targetEntity=Invoice.class)
    private Invoice invoice;
    ....
    //setters and getters goes here
}
```

1-N Relationship

```
@Entity(name = "ORDERS")
public class Order {

    @Id //signifies the primary key
    @Column(name = "ORDER_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long orderId;

    @Column(name = "CUST_ID")
    private long custId;

    @ManyToOne(optional=false)
@JoinColumn(name="CUST_ID",referencedColumnName="CUST_ID")
    private Customer customer;
     .............
    The other attributes and getters and setters goes here
}
```

```
@Entity(name = "CUSTOMER")
public class Customer {

    @Id //signifies the primary key
    @Column(name = "CUST_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long custId;

    @Column(name = "FIRST_NAME", length = 50)
    private String firstName;

    @Column(name = "LAST_NAME", nullable = false,length = 50)
    private String lastName;

@OneToMany(mappedBy="customer",targetEntity=Order.class)
    private Collection orders;
    .........................
    // The other attributes and getters and setters goes here
}
```

M-N Relationship

```
@Entity(name = "ORDERS")
public class Order {
	@Id
	@Column(name = "ORDER_ID")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private long orderId;

	@Column(name = "CUST_ID")
	private long custId;

	@ManyToMany(fetch=FetchType.EAGER)
	@JoinTable(name="ORDER_DETAIL",
		joinColumns=
		@JoinColumn(name="ORDER_ID",
referencedColumnName="ORDER_ID"),
	 inverseJoinColumns=
		@JoinColumn(name="PROD_ID",
referencedColumnName="PROD_ID")
	)
	private List<Product> productList;
	.............
	The other attributes and getters and setters goes here

}
```

```
@Entity(name = "PRODUCT")
public class Product {
	@Id
	@Column(name = "PROD_ID")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private long prodId;

	@Column(name = "PROD_NAME", nullable = false,length = 50)
	private String prodName;

	@Column(name = "PROD_DESC", length = 200)
	private String prodDescription;

@ManyToMany(mappedBy="productList",fetch=FetchType.EAGER)
	private List<Order> orderList;
	.............
	The other attributes and getters and setters goes here
}
```

Fetching Data From 1-1

```
....

EntityManager em =
entityManagerFactory.createEntityManager();

Invoice invoice = em.find(Invoice.class, 1);

System.out.println("Order for invoice 1 : " + invoice.getOrder());

em.close();

entityManagerFactory.close();

....
```

```
....

EntityManager em =
entityManagerFactory.createEntityManager();

    Order order = em.find(Order.class, 111);

    System.out.println("Invoice details for order 111 : " +
    order.getInvoice());

    em.close();

    entityManagerFactory.close();

    ....
```

Fetching Data From 1-N

```
.....

EntityManager em =
entityManagerFactory.createEntityManager();

Order order = em.find(Order.class, 111);

System.out.println("Customer details for order 111 : " +
order.getCustomer());

em.close();

entityManagerFactory.close();

.....
```

```
.....

EntityManager em =
entityManagerFactory.createEntityManager();

Customer customer = em.find(Customer.class, 100);

System.out.println("Order details for customer 100 : " +
customer.getOrders());

em.close();

entityManagerFactory.close();

.....
```

Fetching Data From M-N

```
.....

EntityManager em
entityManagerFactory.createEntityManager();

Order order = em.find(Order.class, 111);

System.out.println("Product  : " +
order.getProductList());

em.close();

entityManagerFactory.close();

.....
```

```
.....

EntityManager em =
entityManagerFactory.createEntityManager();

Product product = em.find(Product.class, 2000);

System.out.println("Order details for product   : " +
product.getOrderList());

em.close();

entityManagerFactory.close();

.....
```

Using JPA in Spring Framework

- JdbcTemplate
- JpaTemplate
- EntityManager
- Spring Transactions

Understanding Transaction Scope

- What is a Transaction?

Transactions provide an "all-or-nothing" proposition, stating that each work-unit performed in a database must either complete in its entirety or have no effect whatsoever.

- Spring handles all the transaction management for you by using @Transactional annotation on Interfaces/Classes and their methods.

- Spring begins a transaction when it encounters an annotated method. The transaction’s scope covers the execution of that method, the execution of any methods that method invokes, and so on, until the method returns.

Understanding Transaction Scope

- The transaction terminates one of two ways:

  -Either the method completes execution directly and the transaction manager commits the transaction.

  - Or the method throws an exception and the transaction manager rolls the transaction back. By default, any java.lang.RuntimeException results in a rolled back transaction. Using the @Transactional annotation you can expand or restrict this filter to refine what triggers a transaction rollback.

- Spring also handles Exception Translation for you. It translates exceptions thrown from JDBC driver to Spring related Exceptions (more descriptive exceptions).

- As soon as you configure Exception Translation , you should mark your repository classes with @Repository annotation. This tells Spring that the annotated bean is eligible for exception translation using the configured PersistenceExceptionTranslators.

Implementing CRUD Operations

```
public interface AuthorRepository
{
 Author get(long id);
 void add(Author author);
 void update(Author author);
 void delete(Author author);
 void delete(long id);
}
```

Implementing CRUD Operations

```
@Repository
public class DefaultAuthorRepository implements AuthorRepository {
 @PersistenceContext
EntityManager entityManager;

 @Override
 public Author get(long id)
 {
  return this.entityManager.createQuery( "SELECT a FROM Author a WHERE a.id = :id", Author.class ).setParameter("id", id).getSingleResult();
 }
 @Override
 public void add(Author author)
 {
   this.entityManager.persist(author);
 }
 @Override
 public void update(Author author)
 {
   this.entityManager.merge(author);
 }
 @Override
 public void delete(Author author)
 {
   this.entityManager.remove(author);
 }
@Override
 public void delete(long id)
 {
 this.entityManager.createQuery( "DELETE FROM Author a WHERE a.id = :id" ).setParameter("id", id).executeUpdate();
 }
}
```

Use JPA Repositories Inside Service Classes

```
@Service
@Transactional
public class BookManager {
@Autowired
private AuthorRepository authorRepository;
…
    public void saveAuthor(Author author){
        if(author.getId() < 1)
          authorRepository.add(author);
        else
          authorRepository.update(author);
    }
...
}
```

Spring Data JPA Intro

- What is Spring Data JPA

Spring Data JPA is not another JPA provider.It is a framework that attempts to simplify even further common tasks regarding persistence operations.

- Why should I think to use this framework

Spring Data JPA eliminates boilerplate code regarding common operations.

Spring Data JPA Intro

```
public interface CustomerRepository extends CrudRepository<Customer, Long>
{


}
```

That’s all you need !!!!!! Awesome?

Check https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

Spring Data JPA Intro

```
public interface CustomerRepository extends
CrudRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

}
```

- User can add method definitions based on a DSL which Spring Data JPA translates internally into query to DB.

End Of Journey

I would like to thank you all for your time and your patience!!

Alexandros Tselonis
Software Engineer