

Exploring the Java Persistence API

By Bob McCune

Agenda

- ▶ JPA Overview
- ▶ Annotating Persistent Entities
- ▶ Understanding the EntityManager
- ▶ JPQL and the Query Interface
- ▶ Spring and JPA
- ▶ Demo



JPA Overview

What is the JPA?

- ▶ Developed under JSR-220
 - ▶ Initial goal was to simplify EJB CMP
- ▶ JSR-220 segmented into two specifications:
 - ▶ EJB 3.0
 - ▶ Java Persistence API
 - ▶ Complete, standalone ORM solution for both Java EE and Java SE environments
- ▶ Significant Community Involvement:
 - ▶ Leverages best ideas from Hibernate, Toplink, and JDO

Why should I care?

- ▶ Why not just use JDBC?
 - ▶ Low level API
 - ▶ Simple to use, but can be error prone
- ▶ Why not just use [INSERT ORM HERE]?
 - ▶ Standardized API leveraging best ideas from ORM community
 - ▶ Better for developers - one API to learn and use
 - ▶ Can choose between competing implementations
 - ▶ Vendor independence

Goals

- ▶ Provide complete ORM solution for Java SE and Java EE environments
- ▶ Easy to use
 - ▶ Standard POJOs - no framework interfaces or classes to implement or extend
 - ▶ Facilitate test-driven development
- ▶ Annotation driven, no XML mappings required.
- ▶ Configuration By Exception
 - ▶ Sensible defaults

JPA Features

- ▶ Simple POJO Persistence
 - ▶ No vendor-specific interfaces or classes
- ▶ Supports rich domain models
 - ▶ No more anemic domain models
 - ▶ Multiple inheritance strategies
 - ▶ Polymorphic Queries
 - ▶ Lazy loading of associations
- ▶ Rich Annotation Support
- ▶ Pluggable persistence providers



Persistent POJOs

POJO Requirements

- ▶ Annotated with `@Entity`
- ▶ Contains a persistent `@Id` field
- ▶ No argument constructor (public or protected)
- ▶ Not marked final
 - ▶ Class, method, or persistent field level
- ▶ Top level class
 - ▶ Can't be inner class, interface, or enum
- ▶ Must implement `Serializable` to be remotely passed by value as a detached instance

Persistent Entity Example

```
@Entity
public class AppUser {

    @Id
    private Long id;
    private String username;
    private String password;

}
```



| AppUser | | |
|---------|----------|----------|
| id | username | password |



Annotating Entities

JPA Annotations

- ▶ JPA annotations are defined in the `javax.persistence` package:
 - ▶ <http://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html>
- ▶ Annotations can be placed on fields or properties
 - ▶ Field level access is preferred to prevent executing logic
 - ▶ Property-level annotations are applied to "getter" method
- ▶ Can't mix style in inheritance hierarchy
 - ▶ Must decide on field OR property

Persistent Identifiers

- ▶ Entities must define an id field/fields corresponding the the database primary key
- ▶ The id can either be simple or composite value
- ▶ Strategies:
 - ▶ @Id: single valued type - most common
 - ▶ @IdClass: map multiple fields to table PK
 - ▶ @EmbeddedId map PK class to table PK
- ▶ Composite PK classes must:
 - ▶ implement Serializable
 - ▶ override equals() and hashCode()

@IdClass

- Maps multiple fields of persistent entity to PK class

```
@Entity
@IdClass(ArtistPK.class)
public class Artist {

    @Id
    private Long idOne;
    @Id
    private Long idTwo;
}
```

```
public class ArtistPK
    implements Serializable {

    private Long idOne;
    private Long idTwo;

    public boolean equals(Object obj);
    public int hashCode();
}
```

@EmbeddedId

- Primary key is formal member of persistent entity

```
@Entity
public class Artist {

    @EmbeddedId
    private ArtistPK key;
}
```

```
@Embedded
public class ArtistPK
    implements Serializable {

    private Long id1;
    private Long id2;

    public boolean equals(Object obj);
    public int hashCode();
}
```

@GeneratedValue

- ▶ Supports auto-generated primary key values
- ▶ Strategies defined by GenerationType enum:
 - ▶ GenerationType.AUTO (preferred)
 - ▶ GenerationType.IDENTITY
 - ▶ GenerationType.SEQUENCE
 - ▶ GenerationType.TABLE

@Id

```
@GeneratedValue(strategy = GenerationType.AUTO)  
private Long id;
```


@Table and @Column

- ▶ Used to define *name* mappings between Java object and database table/columns
- ▶ @Table applied at the persistent class level
- ▶ @Column applied at the persistent field/property level

```
@Entity
@Table(name = "TBL_ARTIST")
public class Artist {
```

```
    @Id
    @Column(name = "ARTIST_ID")
    private Long id;
```

```
    @Column(name = "ARTIST_NAME")
    private String name;
```

```
}
```

| TBL_ARTIST | |
|-------------|-------------|
| ARTIST_ID | NUMERIC |
| ARTIST_NAME | VARCHAR(50) |

@Temporal

- ▶ Used with `java.util.Date` or `java.util.Calendar` to determine how value is persisted
- ▶ Values defined by `TemporalType`:
 - ▶ `TemporalType.DATE` (`java.sql.Date`)
 - ▶ `TemporalType.TIME` (`java.sql.Time`)
 - ▶ `TemporalType.TIMESTAMP` (`java.sql.Timestamp`)

```
...  
@Temporal(value=TemporalType.DATE)  
@Column(name="BIO_DATE")  
private Date bioDate;  
...
```



| TBL_ARTIST | |
|------------|---------|
| ARTIST_ID | NUMERIC |
| BIO_DATE | DATE |

@Enumerated

- ▶ Used to determine strategy for persisting Java enum values to database
- ▶ Values defined by EnumType:
 - ▶ EnumType.ORDINAL (default)
 - ▶ EnumType.STRING

```
@Entity
public class Album {
    ...
    @Enumerated(EnumType.STRING)
    private Rating rating;
    ...
}
```



| ALBUM | |
|----------|-------------|
| ALBUM_ID | NUMERIC |
| RATING | VARCHAR(10) |

@Lob

- ▶ Used to persist values to BLOB/CLOB fields
- ▶ Often used with @Basic to lazy load value

```
@Entity
public class Album {
    ...
    @Lob
    @Basic (fetch = FetchType.LAZY)
    @Column(name = "ALBUM_ART")
    private byte[] artwork;
    ...
}
```



| ALBUM | |
|-----------|---------|
| ALBUM_ID | NUMERIC |
| ALBUM_ART | BLOB |

@Version

- ▶ JPA has automatic versioning support to assist optimistic locking
- ▶ Version field should not be modified by the application
- ▶ Value can be primitive or wrapper type of short, int, long or java.sql.Timestamp field

```
@Version  
private Integer version;
```

@Transient

- ▶ By default, JPA assumes all fields are persistent
- ▶ Non-persistent fields should be marked as transient or annotated with @Transient

@Entity

```
public class Genre {
```

@Id

```
private Long id; ← persistent
```

```
private transient String value1; ← not persistent
```

@Transient

```
private String value2; ← not persistent  
}
```

@Embedded & @Embeddable

```
@Entity
public class Artist {
    ...
    @Embedded
    private Bio bio;
}
```

| ARTIST | |
|----------|---------|
| ALBUM_ID | NUMERIC |
| BIO_DATE | DATE |
| BIO_TEXT | CLOB |



```
@Embeddable
public class Bio {

    @Temporal(value=TemporalType.DATE)
    @Column(name="BIO_DATE")
    private Date bioDate;

    @Lob
    @Column(name="BIO_TEXT")
    private String text;
}
```



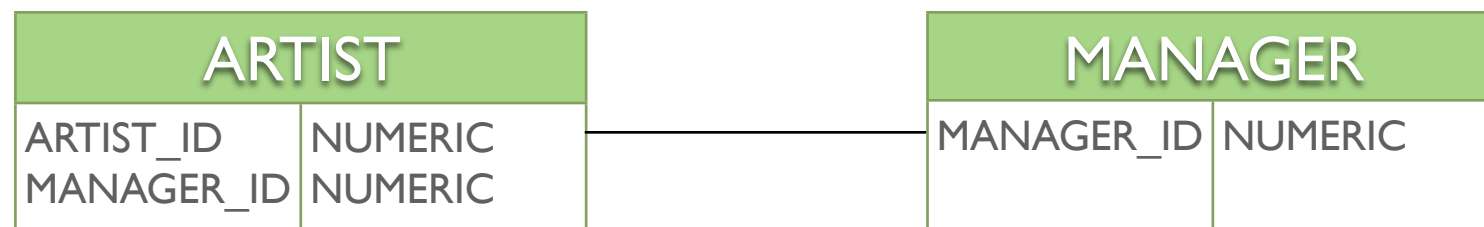
Annotating Relationships

Relationships

- ▶ JPA supports all standard relationships
 - ▶ One-To-One
 - ▶ One-To-Many
 - ▶ Many-To-One
 - ▶ Many-To-Many
- ▶ Supports unidirectional and bidirectional relationships
- ▶ Supports both composite and aggregate relationships

@OneToOne

- Can be based on shared primary key or foreign key relationship using either `@PrimaryKeyJoinColumn` or `@JoinColumn`



```
@Entity
public class Artist {
```

```
    @Id
    private Long id;
```

```
    @OneToOne
    @JoinColumn(name = "MANAGER_ID")
    private Manager manager;
```

```
}
```

```
@Entity
public class Manager {
```

```
    @Id
    private Long id;
```

```
    @OneToOne(mappedBy="manager")
    private Artist artist;
```

```
}
```

Specifies relationship based
on MANAGER_ID column

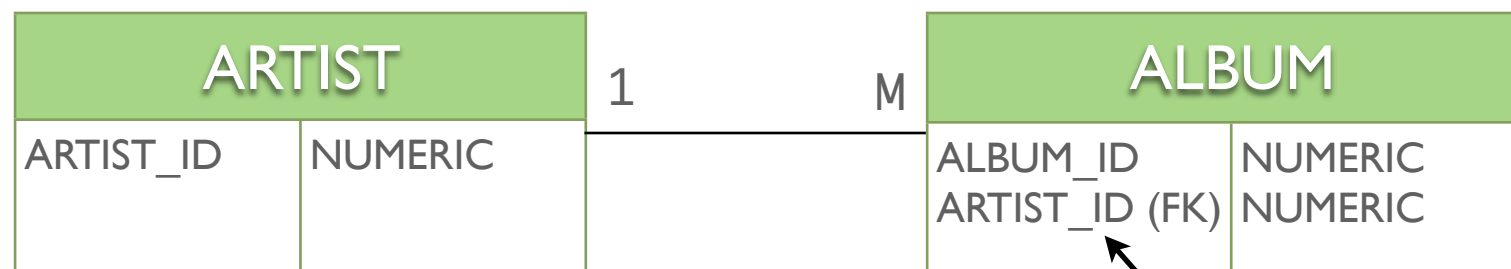
@OneToMany

- ▶ @OneToMany defines the *one* side of a one-to-many relationship
- ▶ The *mappedBy* element of the annotation defines the object reference used by the *child* entity
- ▶ @OrderBy defines an collection ordering required when relationship is retrieved
- ▶ The child (many) side will be represented using an implementation of the `java.util.Collection` interface

@ManyToOne

- ▶ @ManyToOne defines the *many* side of a one-to-many relationship
- ▶ @JoinColumn defines foreign key reference
- ▶ The *many* side is considered to be the owning side of the relationship

One-To-Many Example



```
@Entity
public class Artist {

    @Id
    @Column(name = "ARTIST_ID")
    private Long id;

    @OneToMany(mappedBy = "artist")
    private Set<Album> albums =
        new HashSet<Album>();
}
```

```
@Entity
public class Album {

    @Id
    @Column(name = "ALBUM_ID")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "ARTIST_ID")
    private Artist artist;
}
```

@ManyToMany

- ▶ @ManyToMany annotation is defined on both sides of the relationship
- ▶ Each entity contains a collection of the other
- ▶ @JoinTable is specified on the *owning* side of the relationship
 - ▶ The owning side in a many-to-many is arbitrary
- ▶ @JoinColumn is used to specify the *owning* and *inverse* columns of the join table

Many-To-Many Example

| TRACK | |
|----------|---------|
| TRACK_ID | NUMERIC |

| COMPOSER_TRACK | |
|----------------|---------|
| TRACK_ID | NUMERIC |
| COMPOSER_ID | NUMERIC |

| COMPOSER | |
|-------------|---------|
| COMPOSER_ID | NUMERIC |

```
@Entity
public class Track {

    @Id
    @Column(name = "TRACK_ID")
    private Long id;

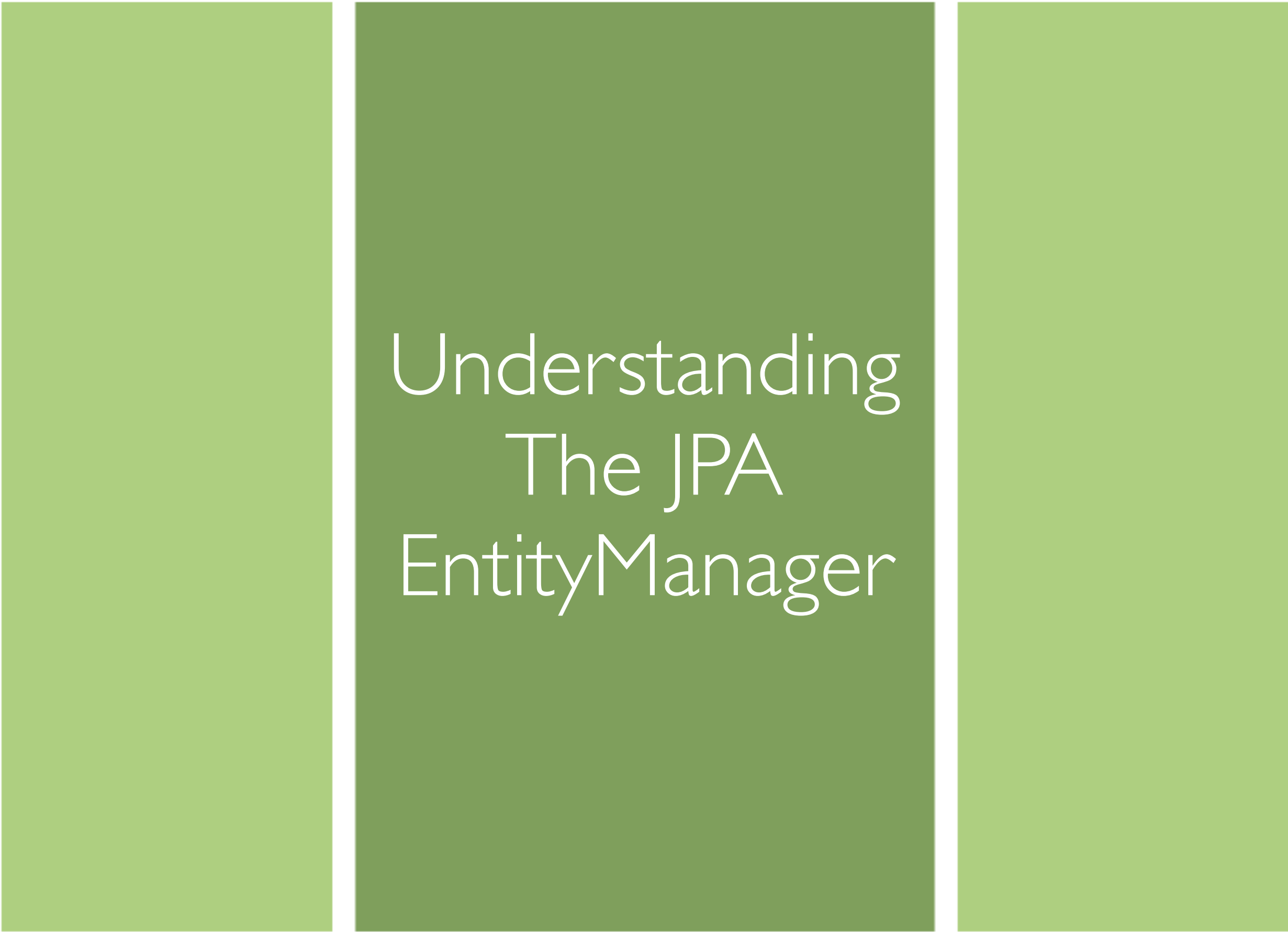
    @ManyToMany(mappedBy="compositions")
    private Set<Composer> composers
        = new HashSet<Composer>();
}
```

```
@Entity
public class Composer {
    @Id
    @Column(name = "COMPOSER_ID")
    private Long id;

    @ManyToMany
    @JoinTable(name="COMPOSER_TRACK",
        joinColumns = { @JoinColumn(name = "COMPOSER_ID") },
        inverseJoinColumns = { @JoinColumn(name = "TRACK_ID") }
    )
    private Set<Track> compositions;
}
```

Cascading Operations

- ▶ JPA supports multiple cascade styles
- ▶ Defined by the CascadeType enum:
 - ▶ CascadeType.PERSIST
 - ▶ CascadeType.MERGE
 - ▶ CascadeType.REMOVE
 - ▶ CascadeType.REFRESH
 - ▶ CascadeType.ALL



Understanding The JPA EntityManager

EntityManager

- ▶ Provides interface to *Persistence Context*
- ▶ Obtained from instance of EntityManagerFactory
 - ▶ Manually created in Java SE environment
 - ▶ Managed in Java EE or Spring and injects EntityManager instances where needed
- ▶ Provides core persistence operations
- ▶ Used to obtain Query interface instance
- ▶ Provides access to transaction manager for use in Java SE environments

Key EntityManager Methods

- ▶ `<T> T find(Class<T> entityClass, Object primaryKey)`
- ▶ `<T> T getReference(Class<T> entityClass, Object primaryKey)`
- ▶ `void persist(Object entity)`
- ▶ `<T> T merge(T entity)`
- ▶ `refresh(Object entity)`
- ▶ `remove(Object entity)`
- ▶ `void flush()`
- ▶ `void close();`

Persistence Context & Unit

- ▶ A Persistence Context is a collection of persistent entities managed by the Entity Manager
- ▶ Persistence Unit is defined in persistence.xml
 - ▶ The only XML required by JPA!
 - ▶ Must be defined loaded from META-INF directory
- ▶ A persistence-unit defines:
 - ▶ The persistence context name
 - ▶ Data source settings
 - ▶ Vendor specific properties and configurations

Example persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence> <!-- removed schema info to reduce clutter -->

    <!-- Demo Persistence Unit -->
    <persistence-unit name="jpademo" transaction-type="RESOURCE_LOCAL">

        <jta-data-source>java:/comp/env/jdbc/JpaDemo</jta-data-source>

        <properties>
            <!-- Only scan and detect annotated entities -->
            <property name="hibernate.archive.autodetection" value="class" />
            <!-- JDBC/Hibernate connection properties -->
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />

            <!-- Set hibernate console formatting options -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.use_sql_comments" value="false" />

        </properties>
    </persistence-unit>
</persistence>
```

Listeners & Callbacks

- ▶ JPA supports lifecycle callback and listener operations.
- ▶ Pre/Post operations supported:
 - ▶ @PrePersist/@PostPersist
 - ▶ @PreUpdate/@PostUpdate
 - ▶ @PreRemove/@PostRemove
 - ▶ @PostLoad

JPA Queries

Query Interface

- ▶ Obtained from the EntityManager using:
 - ▶ createQuery()
 - ▶ createNamedQuery()
 - ▶ createNativeQuery()
- ▶ Supports bind parameters, both named and ordinal
- ▶ Returns query result:
 - ▶ getSingleResult()
 - ▶ getResultList()
- ▶ Pagination Support:
 - ▶ setFirstResult()
 - ▶ setMaxResults()

JPA Queries

- ▶ Supports static & dynamic queries
- ▶ Queries can be written using JPQL or SQL
- ▶ Named and positional bind parameters
- ▶ Supports both static and dynamic queries
 - ▶ Static queries are written as annotations of the entity
- ▶ Supports eager fetching using the fetch keyword

JPQL Features

- ▶ Java Persistence Query Language (JPQL)
 - ▶ Extension of EJB QL language
- ▶ SQL like syntax
 - ▶ Reference objects/properties instead of tables/columns
- ▶ Supports common SQL features:
 - ▶ Projections
 - ▶ Inner & Outer Joins - Eager fetching supported
 - ▶ Subqueries
 - ▶ Bulk operations (update and delete)

JPQL Examples

Dynamic Query

```
public Album findById(Long id) {  
    String jpql = "select distinct a from Album a left join fetch a.artist art "  
    + "left join fetch art.genre left join fetch a.tracks where a.id = :id"  
    Query query = getEntityManager().createQuery(jpql);  
    query.setParameter("id", id);  
    return (Album) query.getSingleResult();  
}
```

Static Query

```
@NamedQuery(name="artist.all",  
            query="select distinct a from Artist a left join fetch a.albums")  
  
public List<Artist> findAll() {  
    Query query = getEntityManager().createNamedQuery("artist.all");  
    return query.getResultList();  
}
```



JPA Demo Spring & JPA

Spring 2.0's JPA Support

- ▶ Supports JPA in both managed and non-managed environments:
 - ▶ J2EE/Java EE environments
 - ▶ Servlet Containers
- ▶ No code or annotation Spring dependencies required
- ▶ EntityManagers can be injected using the JPA standard `@PersistenceContext` annotation
- ▶ Transparent transaction management & exception translation
- ▶ Additionally offers `JpaTemplate` & `JpaDaoSupport` classes
 - ▶ Simplified JPA usage, often single line of code

Spring JPA Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:/comp/env/jdbc/JpaDemo" />
  </bean>

  <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="MYSQL" />
        <property name="showSql" value="true" />
      </bean>
    </property>
  </bean>

  <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>

  <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
</beans>
```



Demo

Q & A

Resources

- ▶ Java Persistence with Hibernate
 - ▶ <http://www.manning.com/bauer2/>
- ▶ JPA 101 - Java Persistence Explained
 - ▶ <http://www.sourcebeat.com/books/jpa.html>
- ▶ JPA Annotation Reference
 - ▶ <http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html>
- ▶ JPA Sample Application
 - ▶ <http://www.bobmccune.com/>