Java Persistence API (JPA)

jim stafford

Fall 2022 v2022-07-24: Built: 2022-12-07 06:13 EST

Table of Contents

Chapter 1. Introduction

This lecture covers implementing object/relational mapping (ORM) to an RDBMS using the Java Persistence API (JPA). This lecture will directly build on the previous concepts covered in the RDBMS and show the productivity power gained by using an ORM to map Java classes to the database.

1.1. Goals

The student will learn:

- to identify the underlying JPA constructs that are the basis of Spring Data JPA Repositories
- to implement a JPA application with basic CRUD capabilities
- to understand the significance of transactions when interacting with JPA

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare project dependencies required for using JPA
2. define a DataSource to interface with the RDBMS
3. define a PersistenceContext containing an `@Entity` class
4. inject an EntityManager to perform actions on a PeristenceUnit and database
5. map a simple `@Entity` class to the database using JPA mapping annotations
6. perform basic database CRUD operations on an `@Entity`
7. define transaction scopes

Chapter 2. Java Persistence API

The Java Persistence API (JPA) is an object/relational mapping (ORM) layer that sits between the application code and JDBC and is the basis for Spring Data JPA Repositories. JPA permits the application to primarily interact with plain old Java (POJO) business objects and a few standard persistence interfaces from JPA to fully manage our objects in the database. JPA works off convention and customized by annotations primarily on the POJO, called an Entity. JPA offers a rich set of capability that would take us many chapters and weeks to cover. I will just cover the very basic setup and `@Entity` mapping at this point.

2.1. JPA Standard and Providers

The JPA standard was originally part of Java EE, which is now managed by the Eclipse Foundation within Jakarta. It was released just after Java 5, which was the first version of Java to support annotations. It replaced the older, heavyweight Entity Bean Standard — that was ill-suited for the job of realistic O/R mapping — and progressed on a path that was in line with Hibernate. There are several persistence providers of the API

- EclipseLink is now the reference implementation
- Hibernate was one of the original implementations and the default implementation within Spring Boot
- OpenJPA from the Apache Software Foundation
- DataNucleus

2.2. JPA Dependencies

Access to JPA requires declaring a dependency on the JPA interface (`jakarta.persistence-api`) and a provider implementation (e.g., `hibernate-core`). This is automatically added to the project by declaring a dependency on the `spring-boot-starter-data-jpa` module.

Spring Data JPA Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

The following shows a subset of the dependencies brought into the application by declaring a dependency on the JPA starter.

Spring Boot JPA Starter Dependencies

```
+- org.springframework.boot:spring-boot-starter-data-jpa:jar:2.7.0:compile
|  +- org.springframework.boot:spring-boot-starter-aop:jar:2.7.0:compile
|  +- org.springframework.boot:spring-boot-starter-jdbc:jar:2.7.0:compile
|  |  \- org.springframework:spring-jdbc:jar:5.3.20:compile
|  +- jakarta.transaction:jakarta.transaction-api:jar:1.3.3:compile
|  +- jakarta.persistence:jakarta.persistence-api:jar:2.2.3:compile ①
|  +- org.hibernate:hibernate-core:jar:5.6.9.Final:compile ②
```

① the JPA API module is required to compile standard JPA constructs

② a JPA provider module is required to access extensions and for runtime implementation of the standard JPA constructs

From these dependencies we have the ability to define and inject various JPA beans.

2.3. Enabling JPA AutoConfiguration

JPA has its own defined bootstrapping constructs that involve settings in `persistence.xml` and entity mappings in `orm.xml` configuration files. These files define the overall persistence unit and include information to connect to the database and any custom entity mapping overrides.

Spring Boot JPA automatically configures a default persistence unit and other related beans when the `@EnableJpaRepositories` annotation is provided. `@EntityScan` is used to identify packages for `@Entities` to include in the persistence unit.

Spring Boot Data Bootstrapping

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication
@EnableJpaRepositories ①
// Class<?>[] basePackageClasses() default {};
// String repositoryImplementationPostfix() default "Impl";
// ...(many more configurations)
@EntityScan ②
// Class<?>[] basePackageClasses() default {};
public class JPASongsApp {
```

① triggers and configures scanning for JPA Repositories

② triggers and configures scanning for JPA Entities

By default, this configuration will scan packages below the class annotated with the `@EntityScan` annotation. We can override that default using the attributes of the `@EntityScan` annotation.

2.4. Configuring JPA DataSource

Spring Boot provides convenient ways to provide property-based configurations through its standard property handing, making the connection areas of `persistence.xml` unnecessary (but still usable). The following examples show how our definition of the `DataSource` for the JDBC/SQL example can be used for JPA as well.

Table 1. Spring Data JPA Database Connection Properties

H2 In-Memory Example Properties

```
spring.datasource.url=jdbc:h2:mem:songs
```

Postgres Client Example Properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
```

2.5. Automatic Schema Generation

JPA provides the capability to automatically generate schema from the Persistence Unit definitions. This can be configured to write to a file to be used to kickstart schema authoring. However, the most convenient use for schema generation is at runtime during development.

Spring Boot will automatically enable runtime schema generation for in-memory database URLs. We can also explicitly enable runtime schema generation using the following hibernate property.

Example Explicit Enable Runtime Schema Generation

```
spring.jpa.hibernate.ddl-auto=create
```

2.6. Schema Generation to File

The JPA provider can be configured to generate schema to a file. This can be used directly by tools like Flyway or simply to kickstart manual schema authoring.

The following configuration snippet instructs the JPA provider to generate a create and drop commands into the same `drop_create.sql` file based on the metadata discovered within the PersistenceContext. Hibernate has the additional features to allow for formatting and line termination specification.

Schema Generation to File Example

```
spring.jpa.properties.javax.persistence.schema-generation.scripts.action=drop-and-
create
spring.jpa.properties.javax.persistence.schema-generation.create-source=metadata

spring.jpa.properties.javax.persistence.schema-generation.scripts.create-
target=target/generated-sources/ddl/drop_create.sql
spring.jpa.properties.javax.persistence.schema-generation.scripts.drop-
target=target/generated-sources/ddl/drop_create.sql

spring.jpa.properties.hibernate.hbm2ddl.delimiter=; ①
spring.jpa.properties.hibernate.format_sql=true ②
```

① adds ";" character to terminate every command — making it SQL script-ready

② adds new lines to make more human-readable

`action` can have values of `none`, `create`, `drop-and-create`, and `drop` [1]

`create/drop-source` can have values of `metadata`, `script`, `metadata-then-script`, or `script-then-metadata`. `metadata` will come from the class defaults and annotations. `script` will come from a location referenced by `create/drop-script-source`

Generate Schema to Debug Complex Mappings

Generating schema from `@Entity` class metadata is a good way to debug odd persistence behavior. Even if normally ignored, the generated schema can identify incorrect and accidental definitions that may cause unwanted behavior.

2.7. Other Useful Properties

It is useful to see database SQL commands coming from the JPA/Hibernate layer during early stages of development or learning. The following properties will print the JPA SQL commands and values that were mapped to the SQL substitution variables.

JPA/Hibernate SQL/JDBC Debug Properties

```
spring.jpa.show-sql=true ①
logging.level.org.hibernate.type=trace ②
```

① prints JPA SQL commands

② prints SQL parameter values

The following cleaned up output shows the result of the activated debug. We can see the individual SQL commands issued to the database as well as the parameter values used in the call and extracted from the response.

JPA/Hibernate SQL/JDBC Debug Example Output

```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?,
?)

binding parameter [1] as [VARCHAR] - [Rage Against The Machine]
binding parameter [2] as [DATE] - [2020-05-12]
binding parameter [3] as [VARCHAR] - [Recalled to Life]
binding parameter [4] as [INTEGER] - [1]
```

2.8. Configuring JPA Entity Scan

Spring Boot JPA will automatically scan for `@Entity` classes. We can provide a specification to external packages to scan using the `@EntityScan` annotation.

The following shows an example of using a String package specification to a root package to scan for `@Entity` classes.

@EntityScan example

```
import org.springframework.boot.autoconfigure.domain.EntityScan;
...
@EntityScan(value={"info.ejava.examples.db.repo.jpa.songs.bo"})
```

The following example, instead uses a Java class to express a package to scan. We are using a specific `@Entity` class in this case, but some may define an interface simply to help mark the package and use that instead. The advantage of using a Java class/interface is that it will work better when refactoring.

@EntityScan .class Example

```
import info.ejava.examples.db.repo.jpa.songs.bo.Song;
...
@EntityScan(basePackageClasses = {Song.class})
```

2.9. JPA Persistence Unit

The JPA Persistence Unit represents the overall definition of a group of Entities and how we interact with the database. A defined Persistence Unit can be injected into the application using an `EntityManagerFactory`. From this injected class, clients can gain access to metadata and initiate a Persistence Context.

Persistance Unit/EntityManagerFactory Injection Example

```
import javax.persistence.EntityManagerFactory;
...
    @Autowired
    private EntityManagerFactory emf;
```

2.10. JPA Persistence Context

A Persistence Context is a usage instance of a Persistence Unit and is represented by an `EntityManager`. An `@Entity` with the same identity is represented by a single instance within a Persistence Context.

Persistance Context/EntityManager Injection Example

```
import javax.persistence.EntityManager;
...
    @Autowired
    private EntityManager em;
```

Injected `EntityManagers` reference the same Persistence Context when called within the same thread. That means that a `Song` loaded by one client with ID=1 will be available to sibling code when using ID=1.

Use/Inject EntityManagers

Normal application code that creates, gets, updates, and deletes `@Entity` data should use an injected `EntityManager` and allow the transaction management to occur at a higher level.

[1] *"JavaEE: The JavaEE Tutorial, Database Schema Creation"*, Oracle, JavaEE 7

Chapter 3. JPA Entity

A JPA `@Entity` is a class that is mapped to the database that primarily represents a row in a table. The following snippet is the example Song class we have already manually mapped to the `REPOSONGS_SONG` database table using manually written schema and JDBC/SQL commands in a previous lecture. To make the class an `@Entity`, we must:

- annotate the class with `@Entity`
- provide a no-argument constructor
- identify one or more colums to represent the primary key using the `@Id` annotation
- override any convention defaults with further annotations

JPA Example Entity

```
@javax.persistence.Entity ①
@Getter
@AllArgsConstructor
@NoArgsConstructor ②
public class Song {
    @javax.persistence.Id ③ ④
    private int id;
    @Setter
    private String title;
    @Setter
    private String artist;
    @Setter
    private java.time.LocalDate released;
}
```

① class must be annotated with @Entity

② class must have a no-argument constructor

③ class must have one or more fields designated as the primary key

④ annotations can be on the field or property and the choice for `@Id` determines the default

Primary Key property is not modifiable

This Java class is not providing a setter for the field mapped to the primary key in the database. The primary key will be generated by the persistence provider at runtime and assigned to the field. The field cannot be modified while the instance is managed by the provider. The all-args constructor can be used to instantiate a new object with a specific primary key.

3.1. JPA @Entity Defaults

By convention and supplied annotations, the class as shown above would:

- have the entity name "Song" (important when expressing queries; ex. `select s from Song s`)

- be mapped to the `SONG` table to match the entity name
- have columns `id integer`, `title varchar`, `artist varchar`, and `released (date)`
- use `id` as its primary key and manage that using a provider-default mechanism

3.2. JPA Overrides

Many/all of the convention defaults can be customized by further annotations. We commonly need to:

- supply a table name that matches our intended schema (i.e., `select * from REPOSONGS_SONG` vs `select * from SONG`)
- select which primary key mechanism is appropriate for our use
- supply column names that match our intended schema
- identify which properties are optional, part of the initial `INSERT`, and `UPDATE` -able
- supply other parameters useful for schema generation (e.g., String length)

Common JPA Annotation Overrides

```
@Entity
@Table(name="REPOSONGS_SONG") ①
@NoArgsConstructor
...
public class Song {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE) ②
    @Column(name = "ID") ③
    private int id;
    @Column(name="TITLE", length=255, nullable=true, insertable=true, updatable=true
)④
    private String title;
    private String artist;
    private LocalDate released;
}
```

① overriding the default table name `SONG` with `REPOSONGS_SONG`

② overriding the default primary key mechanism with `SEQUENCE`. The default sequence name is `hibernate-sequence` for the Hibernate JPA provider.

③ re-asserting the default convention column name `ID` for the `id` field

④ re-asserting many of the default convention column mappings

Schema generation properties not used at runtime

Properties like `length` and `nullable` are only used during optional JPA schema generation and are not used at runtime.

Chapter 4. Basic JPA CRUD Commands

JPA provides an API for implementing persistence to the database through manipulation of `@Entity` instances and calls to the EntityManager.

4.1. EntityManager persist()

We create a new object in the database by calling `persist()` on the EntityManager and passing in an `@Entity` instance that represents something new. This will:

- assign a primary key if configured to do so
- add the instance to the Persistence Context
- make the `@Entity` instance managed from that point forward

The following snippet shows a partial DAO implementation using JPA.

Example EntityManager persist() Call

```
@Component
@RequiredArgsConstructor
public class JpaSongDAO {
    private final EntityManager em;

    public void create(Song song) {
        em.persist(song);
    }
...
```

A database `INSERT` SQL command will be queued to the database as a result of a successful call and the `@Entity` instance will be in a managed state.

Resulting SQL from persist Call()

```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?,
?)
```

In the managed state, any changes to the `@Entity` will result in a future `UPDATE` SQL command. Updates are issued during the next JPA session "flush". JPA session flushes can be triggered manually or automatically prior to or no later than the next commit.

4.2. EntityManager find() By Identity

JPA supplies a means to get the full `@Entity` using its primary key.

Example EntityManager find() Call

```
public Song findById(int id) {
    return em.find(Song.class, id);
}
```

If the instance is not yet loaded into the Persistence Context, `SELECT` SQL command(s) will be issued to the database to obtain the persisted state. The following snippet shows the SQL generated by Hibernate to fetch the state from the database to realize the `@Entity` instance within the JVM.

Resulting SQL from find() Call

```
Hibernate: select
    song0_.id as id1_0_0_,
    song0_.artist as artist2_0_0_,
    song0_.released as released3_0_0_,
    song0_.title as title4_0_0_
from reposongs_song song0_
where song0_.id=?
```

From that point forward, the state will be returned from the Persistence Context without the need to get the state from the database.

4.3. EntityManager query

JPA provides many types of queries

- JPA Query Language (JPAQL) - a very SQL-like String syntax expressed in terms of `@Entity` classes and relationship constructs
- Criteria Language - a type-safe, Java-centric syntax that avoids String parsing and makes dynamic query building more efficient than query string concatenation and parsing
- Native SQL - the same SQL we would have provided to JDBC

The following snippet shows an example of executing a JPAQL Query.

Example EntityManager Query

```
public boolean existsById(int id) {
    return em.createQuery("select count(s) from Song s where s.id=:id",①
            Number.class) ②
            .setParameter("id", id) ③
            .getSingleResult() ④
            .longValue()==1L; ⑤
}
```

① JPAQL String based on `@Entity` constructs

② query call syntax allows us to define the expected return type

③ query variables can be set by name or position

④ one (mandatory) or many results can be returned from query

⑤ entity exists if row count of rows matching PK is 1. Otherwise should be 0

The following shows how our JPAQL snippet mapped to the raw SQL issued to the database. Notice that our `Song` `@Entity` reference was mapped to the `REPOSONGS_SONG` database table.

Resulting SQL from Query Call

```
Hibernate: select
    count(song0_.id) as col_0_0_
from reposongs_song song0_
where song0_.id=?
```

4.4. EntityManager flush()

Not every change to an `@Entity` and call to an `EntityManager` results in an immediate 1:1 call to the database. Some of these calls manipulate an in-memory cache in the JVM and may get issued in a group of other commands at some point in the future. We normally want to allow the `EntityManager` to cache these calls as much as possible. However, there are times (e.g., prior to making a raw SQL query) where we want to make sure the database has the current state of the cache.

The following snippet shows an example of flushing the contents of the cache after changing the state of a managed `@Entity` instance.

Example EntityManager flush() Call

```
Song s = ... //obtain a reference to a managed instance
s.setTitle("...");
em.flush(); //optional!!! will eventually happen at some point
```

Whether is was explicitly issued or triggered internally by the JPA provider, the following snippet shows the resulting `UPDATE` SQL call to change the state of the database to match the Persistence Context.

Resulting SQL from flush() Call

```
Hibernate: update reposongs_song
    set artist=?, released=?, title=? ①
where id=?
```

① all fields designated as `updatable=true` are included in the `UPDATE`

4.5. EntityManager remove()

JPA provides a means to delete an `@Entity` from the database. However, we must have the managed `@Entity` instance loaded in the Persistence Context first to use this capability. The reason for this is that a JPA delete can optionally involve cascading actions to remove other related entities as well.

The following snippet shows how a managed `@Entity` instance can be used to initiate the removal from the database.

Example EntityManager remove() Call

```
public void delete(Song song) {
    em.remove(song);
}
```

The following snippet shows how the remove command was mapped to a SQL `DELETE` command.

Resulting SQL from remove() Call

```
Hibernate: delete from reposongs_song
where id=?
```

4.6. EntityManager clear() and detach()

There are two commands that will remove entities from the Persistence Context. They have their purpose, but know that they are rarely used and can be dangerous to call.

- clear() - will remove all entities
- detach() - will remove a specific `@Entity`

I only bring these up because you may come across class examples where I am calling `flush()` and `clear()` in the middle of a demonstration. This is purposely mimicking a fresh Persistence Context within scope of a single transaction.

clear() and detach() Commands

```
em.clear();
em.detach(song);
```

Calling `clear()` or `detach()` will evict all managed entities or targeted managed `@Entity` from the Persistence Context — loosing any in-progress and future modifications. In the case of returning redacted @Entities — this may be exactly what you want (you don't want the redactions to remove data from the database).

Use clear() and detach() with Caution

Calling `clear()` or `detach()` will evict all managed entities or targeted managed `@Entity` from the Persistence Context — loosing any in-progress and future modifications.

Chapter 5. Transactions

All commands require some type of transaction when interacting with the database. The transaction can be activated and terminated at varying levels of scope integrating one or more commands into a single transaction.

5.1. Transactions Required for Explicit Changes/Actions

The injected `EntityManager` is the target of our application calls and the transaction gets associated with that object. The following snippet shows the provider throwing a `TransactionRequiredException` when the calling `persist()` on the injected `EntityManager` when no transaction has been activated.

Example Persist Failure without Transaction

```
@Autowired
private EntityManager em;
...
@Test
void transaction_missing() {
    //given - an instance
    Song song = mapper.map(dtoFactory.make());

    //when - persist is called without a tx, an exception is thrown
    em.persist(song); ①
}
```

① `TransactionRequiredException` exception thrown

Exception Thrown when Required Transaction Missing

```
javax.persistence.TransactionRequiredException: No EntityManager with actual
transaction available for current thread - cannot reliably process 'persist' call
```

5.2. Activating Transactions

Although you will find transaction methods on the `EntityManager`, these are only meant for individually managed instances created directly from the `EntityManagerFactory`. Transactions for injected an `EntityManager` are managed by the container and triggered by the presence of a `@Transactional` annotation on a called bean method within the call stack.

This next example annotates the calling `@Test` method with the `@Transactional` annotation to cause a transaction to be active for the three (3) contained `EntityManager` calls.

Example @Transactional Activation

```
import org.springframework.transaction.annotation.Transactional;
```

```
...
@Test
@Transactional ①
void transaction_present_in_caller() {
    //given - an instance
    Song song = mapper.map(dtoFactory.make());

    //when  - persist called within caller transaction, no exception thrown
    em.persist(song); ②
    em.flush(); //force DB interaction ②

    //then
    then(em.find(Song.class, song.getId())).isNotNull(); ②
} ③
```

① `@Transactional` triggers an Aspect to activate a transaction for the Persistence Context operating within the current thread

② the same transaction is used on all three (3) `EntityManager` calls

③ the end of the method will trigger the transaction-initiating Aspect to commit (or rollback) the transaction it activated

5.3. Conceptual Transaction Handling

Logically speaking, the transaction handling done on behalf of `@Transactional` is similar to the snippet shown below. However, as complicated as that is — it does not begin to address nested calls. Also note that a thrown `RuntimeException` triggers a rollback and anything else triggers a commit.

Conceptual View of Transaction Handling

```
tx = em.getTransaction();
try {
  tx.begin();
  //call code ②
} catch (RuntimeException ex) {
  tx.setRollbackOnly(); ①
} catch (Exception ex) { ②
} finally {
    if (tx.getRollbackOnly()) {
        tx.rollback();
    } else {
        tx.commit();
    }
}
```

① `RuntimeException`, by default, triggers a rollback

② Normal returns and checked exceptions, by default, trigger a commit

5.4. Activating Transactions in @Components

We can alternatively push the demarcation of the transaction boundary down to the `@Component` methods.

The snippet below shows a DAO `@Component` that designates each of its methods being `@Transactional`. This has the benefit of knowing that each of the calls to `EntityManager` methods will have the required transaction in place — whether it is the right one is a later topic.

@Transactional Component

```
@Component
@RequiredArgsConstructor
@Transactional ①
public class JpaSongDAO {
    private final EntityManager em;

    public void create(Song song) {
        em.persist(song);
    }
    public Song findById(int id) {
        return em.find(Song.class, id);
    }
    public void delete(Song song) {
        em.remove(song);
    }
```

① each method will be assigned a transaction

5.5. Calling @Transactional @Component Methods

The following example shows the calling code invoking methods of the DAO `@Component` in independent transactions. The code works because there really is no dependency between the `INSERT` and `SELECT` to be part of the same transaction, as long as the `INSERT` commits before the `SELECT` transaction starts.

Calling @Component @Transactional Methods

```
@Test
void transaction_present_in_component() {
    //given - an instance
    Song song = mapper.map(dtoFactory.make());

    //when  - persist called within component transaction, no exception thrown
    jpaDao.create(song); ①

    //then
    then(jpaDao.findById(song.getId())).isNotNull(); ②
}
```

① INSERT is completed in separate transaction

② SELECT completes in follow-on transaction

5.6. @Transactional @Component Methods SQL

The following shows the SQL triggered by the snippet above with the different transactions annotated.

@Transactional Methods Resulting SQL

```
①
Hibernate: insert into reposongs_song (artist, released, title, id) values (?, ?, ?,
?)
②
Hibernate: select
    song0_.id as id1_0_0_,
    song0_.artist as artist2_0_0_,
    song0_.released as released3_0_0_,
    song0_.title as title4_0_0_
from reposongs_song song0_
where song0_.id=?
```

① transaction 1

② transaction 2

5.7. Unmanaged @Entity

However, we do not always get that lucky — for individual, sequential transactions to play well together. JPA entities follow the notation of managed and unmanaged/detached state.

- Managed entities are actively being tracked by a Persistence Context
- Unmanaged/Detached entities have either never been or no longer associated with a Persistence Context

The following snippet shows an example of where a follow-on method fails because the EntityManager requires that @Entity be currently managed. However, the end of the create() transaction made it detached.

Unmanaged @Entity

```
@Test
void transaction_common_needed() {
    //given a persisted instance
    Song song = mapper.map(dtoFactory.make());
    jpaDao.create(song); //song is detached at this point ①

    //when - removing detached entity we get an exception
    jpaDao.delete(song); ②
```

① the first transaction starts and ends at this call

② the `EntityManager.remove` operates in a separate transaction with a detached `@Entity` from the previous transaction

The following text shows the error message thrown by the `EntityManager.remove` call when a detached entity is passed in to be deleted.

```
java.lang.IllegalArgumentException: Removing a detached instance
info.ejava.examples.db.repo.jpa.songs.bo.Song#1
```

5.8. Shared Transaction

We can get things to work better if we encapsulate methods behind a `@Service` method defining good transaction boundaries. Lacking a more robust application, the snippet below adds the `@Transactional` to the `@Test` method to have it shared by the three (3) DAO `@Component` calls — making the `@Transactional` annotations on the DAO meaningless.

Shared Transaction

```
@Test
@Transactional ①
void transaction_common_present() {
    //given a persisted instance
    Song song = mapper.map(dtoFactory.make());
    jpaDao.create(song); //song is detached at this point ②

    //when - removing managed entity, it works
    jpaDao.delete(song); ②

    //then
    then(jpaDao.findById(song.getId())).isNull(); ②
}
```

① `@Transactional` at the calling method level is shared across all lower-level calls

② Each DAO call is executed in the same transaction and the `@Entity` can still be managed across all calls

5.9. @Transactional Attributes

There are several attributes that can be set on the `@Transactional` annotation. A few of the more common properties to set include

- propagation - defaults to REQUIRED, proactively activating a transaction if not already present
  - SUPPORTS - lazily initiates a transaction, but fully supported if already active

    - MANDATORY - error if called without an active transaction
    - REQUIRES_NEW - proactively creates a new transaction separate from the caller’s transaction
    - NOT_SUPPORTED - nothing within the called method will honor transaction semantics
    - NEVER - do not call with an active transaction
    - NESTED - may not be supported, but permits nested transactions to complete before returning to calling transaction
- isolation - location to assign JDBC Connection isolation
- readOnly - defaults to false, hints to JPA provider that entities can be immediately detached
- rollback definitions - when to implement non-standard rollback rules

Chapter 6. Summary

In this module we learned:

- to configure a JPA project in include project dependencies and required application properties
- to define a PersistenceContext and where to scan for `@Entity` classes
- requirements for an `@Entity` class
- default mapping conventions for `@Entity` mappings
- optional mapping annotations for `@Entity` mappings
- to perform basic CRUD operations with the database