WHAT IS: JAVA
By Matthew Tyson, Java Developer, JavaWorld
APR 2, 2019 10:53 AM PDT

About
Everything you need to know about Java programming tools and APIs, with code and examples.

THE JAVA PERSISTENCE SERIES

What is JPA? Introduction to the Java Persistence API

The Java ORM standard for storing, accessing, and managing Java objects in a relational database

As a specification, the Java Persistence API is concerned with *persistence*, which loosely means any mechanism by which Java objects outlive the application process that created them. Not all Java objects need to be persisted, but most applications persist key business objects. The JPA specification lets you define *which* objects should be persisted, and *how* those objects should be persisted in your Java applications.

By itself, JPA is not a tool or framework; rather, it defines a set of concepts that can be implemented by any tool or framework. While JPA's object-relational mapping (ORM) model was originally based on Hibernate, it has since evolved. Likewise, while JPA was originally intended for use with relational/SQL databases, some JPA implementations have been extended for use with NoSQL datastores. A popular framework that supports JPA with NoSQL is EclipseLink, the reference implementation for JPA 2.2.

JPA 2.2 in Jakarta EE

The Java Persistence API was first released as a subset of the EJB 3.0 specification (JSR 220) in Java EE 5. It has since evolved as its own spec, starting with the release of JPA 2.0 in Java EE 6 (JSR 317). As of this writing, JPA 2.2 has been adopted for continuation as part of Jakarta EE.

JPA and Hibernate

Because of their intertwined history, Hibernate and JPA are frequently conflated. However, like the Java Servlet specification, JPA has spawned many compatible tools and frameworks; Hibernate is just one of them.

Developed by Gavin King and released in early 2002, Hibernate is an ORM library for Java. King developed Hibernate as an alternative to entity beans for persistence. The framework was so popular, and so needed at the time, that many of its ideas were adopted and codified in the first JPA specification.

Today, Hibernate ORM is one of the most mature JPA implementations, and still a popular option for ORM in Java. Hibernate ORM 5.3.8 (the current version as of this writing) implements JPA 2.2. Additionally, Hibernate's family of tools has expanded to include popular tools like Hibernate Search, Hibernate Validator, and Hibernate OGM, which supports domain-model persistence for NoSQL.

JPA and EJB

As noted earlier, JPA was introduced as a subset of EJB 3.0, but has since evolved as its own specification. EJB is a specification with a different focus from JPA, and is implemented in an EJB container. Each EJB container includes a persistence layer, which is defined by the JPA specification.

What is Java ORM?

While they differ in execution, every JPA implementation provides some kind of ORM layer. In order to understand JPA and JPA-compatible tools, you need to have a good grasp on ORM.

Object-relational mapping is a *task*–one that developers have good reason to avoid doing manually. A framework like Hibernate ORM or EclipseLink codifies that task into a library or framework, an *ORM layer*. As part of the application architecture, the ORM layer is responsible for managing the conversion of software objects to interact with the tables and columns in a relational database. In Java, the ORM layer converts Java classes and objects so that they can be stored and managed in a relational database.

By default, the name of the object being persisted becomes the name of the table, and fields become columns. Once the table is set up, each table row corresponds to an object in the application. Object mapping is configurable, but defaults tend to work well.

JPA with NoSQL

Until fairly recently, non-relational databases were uncommon curiosities. The NoSQL movement changed all that, and now a variety of NoSQL databases are available to Java developers. Some JPA implementations have evolved to embrace NoSQL, including Hibernate OGM and EclipseLink.

Figure 1 illustrates the role of JPA and the ORM layer in application development.

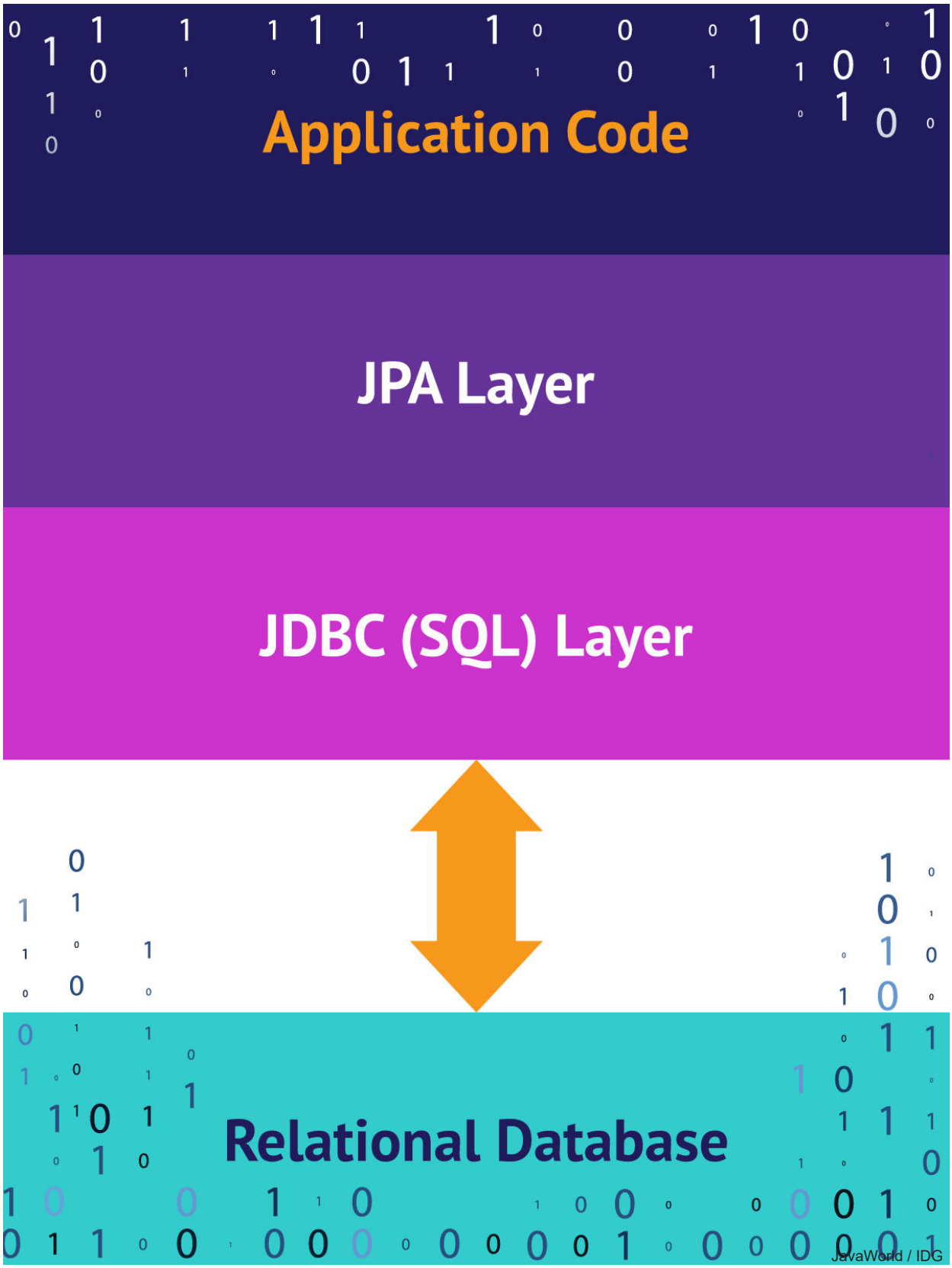

Figure 1. JPA and the ORM layer

Configuring the Java ORM layer

When you set up a new project to use JPA, you will need to configure the datastore and JPA provider. You'll configure a *datastore connector* to connect to your chosen database (SQL or NoSQL). You'll also include and configure the *JPA provider*, which is a framework such as Hibernate or EclipseLink. While you can configure JPA manually, many developers choose to use Spring's out-of-the-box support. See "**JPA installation and setup**" below for a demonstration of both manual and Spring-based JPA installation and setup.

Java Data Objects

Java Data Objects is a standardized persistence framework that differs from JPA primarily by supporting persistence logic in the object, and by its longstanding support for working with non-relational data stores. JPA and JDO are similar enough that JDO providers frequently also support JPA. See the Apache JDO Project to the learn more about JDO in relation to other persistence standards like JPA and JDBC.

Data persistence in Java

From a programming perspective, the ORM layer is an *adapter layer*: it adapts the language of object graphs to the language of SQL and relational tables. The ORM layer allows object-oriented developers to build software that persists data without ever leaving the object-oriented paradigm.

When you use JPA, you create a *map* from the datastore to your application's data model objects. Instead of defining how objects are saved and retrieved, you define the mapping between objects and your database, then invoke JPA to persist them. If you're using a relational database, much of the actual connection between your application code and the database will then be handled by JDBC, the Java Database Connectivity API.

As a spec, JPA provides *metadata annotations*, which you use to define the mapping between objects and the database. Each JPA implementation provides its own engine for JPA annotations. The JPA spec also provides the `PersistanceManager` or `EntityManager`, which are the key points of contact with the JPA system (wherein your business logic code tells the system what to do with the mapped objects).

To make all of this more concrete, consider Listing 1, which is a simple data class for modeling a musician.

Listing 1. A simple data class in Java

```
public class Musician {
  private Long id;
  private String name;
  private Instrument mainInstrument;
  private ArrayList performances = new ArrayList<Performance>();

  public Musician( Long id, String name){ /* constructor setters... */ }

  public void setName(String name){
    this.name = name;
  }
  public String getName(){
    return this.name;
  }
  public void setMainInstrument(Instrument instr){
    this.instrument = instr;
  }
  public Instrument getMainInstrument(){
    return this.instrument;
  }
  // ...Other getters and setters...
}
```

The `Musician` class in Listing 1 is used to hold data. It can contain primitive data such as the *name* field. It can also hold relations to other classes such as `mainInstrument` and `performances`.

`Musician`'s *reason for being* is to contain data. This type of class is sometimes known as a DTO, or *data transfer object*. DTOs are a common feature of software development. While they hold many kinds of data, they do not contain any business logic. Persisting data objects is a ubiquitous challenge in software development.

Data persistence with JDBC

One way to save an instance of the `Musician` class to a relational database would be to use the JDBC library. JDBC is a layer of abstraction that lets an application issue SQL commands without thinking about the underlying database implementation.

Listing 2 shows how you could persist the `Musician` class using JDBC.

Listing 2. JDBC inserting a record

```
Musician georgeHarrison = new Musician(0, "George Harrison");

    String myDriver = "org.gjt.mm.mysql.Driver";
      String myUrl = "jdbc:mysql://localhost/test";
      Class.forName(myDriver);
      Connection conn = DriverManager.getConnection(myUrl, "root", "");

      String query = " insert into users (id, name) values (?, ?)";
      PreparedStatement preparedStmt = conn.prepareStatement(query);
      preparedStmt.setInt    (1, 0);
      preparedStmt.setString (2, "George Harrison");
      preparedStmt.setString (2, "Rubble");

      preparedStmt.execute();
      conn.close();

// Error handling removed for brevity
```

The code in Listing 2 is fairly self-documenting. The `georgeHarrison` object could come from anywhere (front-end submit, external service, etc.), and has its ID and name fields set. The fields on the object are then used to supply the values of an SQL `insert` statement. (The `PreparedStatement` class is part of JDBC, offering a way to safely apply values to an SQL query.)

While JDBC allows the control that comes with manual configuration, it is cumbersome compared to JPA. In order to modify the database, you first need to create an SQL query that maps from your Java object to the tables in a relational database. You then have to modify the SQL whenever an object signature change. With JDBC, maintaining the SQL becomes a task in itself.

Data persistence with JPA

Now consider Listing 3, where we persist the `Musician` class using JPA.

Listing 3. Persisting George Harrison with JPA

```
Musician georgeHarrison = new Musician(0, "George Harrison");
musicianManager.save(georgeHarrison);
```

Listing 3 replaces the manual SQL from Listing 2 with a single line, `session.save()`, which instructs JPA to persist the object. From then on, the SQL conversion is handled by the framework, so you never have to leave the object-oriented paradigm.

Metadata annotations in JPA

The magic in Listing 3 is the result of a *configuration*, which is created using JPA's annotations. Developers use annotations to inform JPA which objects should be persisted, and how they should be persisted.

Listing 4 shows the `Musician` class with a single JPA annotation.

Listing 4. JPA's @Entity annotation

```
@Entity
public class Musician {
  // ..class body
}
```

Persistent objects are sometimes called *entities*. Attaching `@Entity` to a class like `Musician` informs JPA that this class and its objects should be persisted.

XML vs. annotation-based configuration
JPA also supports using external XML files, instead of annotations, to define class metadata. But why would you do that to yourself?

Configuring JPA

Like most modern frameworks, JPA embraces *coding by convention* (also known as convention over configuration), in which the framework provides a default configuration based on industry best practices. As one example, a class named `Musician` would be mapped by default to a

database table called **Musician**.

The conventional configuration is a timesaver, and in many cases it works well enough. It is also possible to customize your JPA configuration. As an example, you could use JPA's `@Table` annotation to specify the table where the `Musician` class should be stored.

Listing 5. JPA's @Table annotation

```
@Entity
@Table(name="musician")
public class Musician {
  // ..class body
}
```

Listing 5 tells JPA to persist the entity (`Musician` class) to the `musician` table.

Primary key

In JPA, the *primary key* is the field used to uniquely identify each object in the database. The primary key is useful for referencing and relating objects to other entities. Whenever you store an object in a table, you will also specify the field to use as its primary key.

In Listing 6, we tell JPA what field to use as `Musician`'s primary key.

Listing 6. Specifying the primary key

```
@Entity
public class Musician {
   @Id
   private Long id;
```

In this case, we've used JPA's `@Id` annotation to specify the `id` field as `Musician`'s primary key. By default, this configuration assumes the primary key will be set by the database--for instance, when the field is set to auto-increment on the table.

JPA supports other strategies for generating an object's primary key. It also has annotations for changing individual field names. In general, JPA is flexible enough to adapt to any persistence mapping you might need.

CRUD operations

Once you've mapped a class to a database table and established its primary key, you have everything you need to create, retrieve, delete, and update that class in the database. Calling `session.save()` will create or update the specified class, depending on whether the primary-key field is null or applies to en existing entity. Calling `entityManager.remove()` will delete the specified class.

Entity relationships in JPA

Simply persisting an object with a primitive field is only half the equation. JPA also has the capability to manage entities in relation to one another. Four kinds of entity relationships are possible in both tables and objects:

1. One-to-many
2. Many-to-one
3. Many-to-many
4. One-to-one

Each type of relationship describes how an entity relates to other entities. For example, the `Musician` entity could have a *one-to-many relationship* with `Performance`, an entity represented by a collection such as `List` or `Set`.

If the `Musician` included a `Band` field, the relationship between these entities could be *many-to-one*, implying collection of `Musician`s on the single `Band` class. (Assuming each musician only performs in a single band.)

If `Musician` included a `BandMates` field, that could represent a *many-to-many relationship* with other `Musician` entities.

Finally, `Musician` might have a *one-to-one relationship* with a `Quote` entity, used to represent a famous quote: `Quote famousQuote = new Quote()`.

Defining relationship types

Page 1 of 2 >