As a specification, the [Java Persistence API](https://jcp.org/en/jsr/detail?id=338) 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](http://hibernate.org/orm/), 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](https://www.eclipse.org/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](https://jcp.org/en/jsr/detail?id=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](https://jcp.org/en/jsr/detail?id=317)). As of this writing, JPA 2.2 has been adopted for [continuation as part of Jakarta EE](https://projects.eclipse.org/proposals/eclipse-project-jpa).

**JPA and Hibernate**

Because of their intertwined history, Hibernate and JPA are frequently conflated. However, like the [Java Servlet](https://www.javaworld.com/article/3313114/what-is-a-java-servlet-request-handling-for-java-web-applications.html) specification, JPA has spawned many compatible tools and frameworks; Hibernate is just one of them.

**[**[**Also on InfoWorld: JDK 16: What’s coming in Java 16**](https://www.infoworld.com/article/3569150/jdk-16-whats-coming-in-java-16.html)**]**

Developed by Gavin King and released in early 2002, [Hibernate](https://www.javaworld.com/article/2072999/get-started-with-hibernate.html) is an ORM library for Java. King developed Hibernate as an [alternative to entity beans for persistence](http://www.javaperformancetuning.com/news/interview041.shtml). 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](http://hibernate.org/) is one of the most mature JPA implementations, and still a popular option for ORM in Java. [Hibernate ORM 5.3.8](http://in.relation.to/2019/02/19/hibernate-orm-538-final-out/) (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](https://www.javaworld.com/article/2077880/data-storage-introduction-to-hibernate-search.html%20target=), [Hibernate Validator](https://www.javaworld.com/article/2137346/java-tip-hibernate-validation-in-a-standalone-implementation.html), and [Hibernate OGM](http://hibernate.org/ogm/), which supports domain-model persistence for NoSQL.

**JPA and EJB**

As noted earlier, [JPA was introduced as a subset of EJB 3.0](https://www.javaworld.com/article/2072086/java-web-development-simplify-enterprise-java-development-with-ejb-3-0-part-2.html), 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.

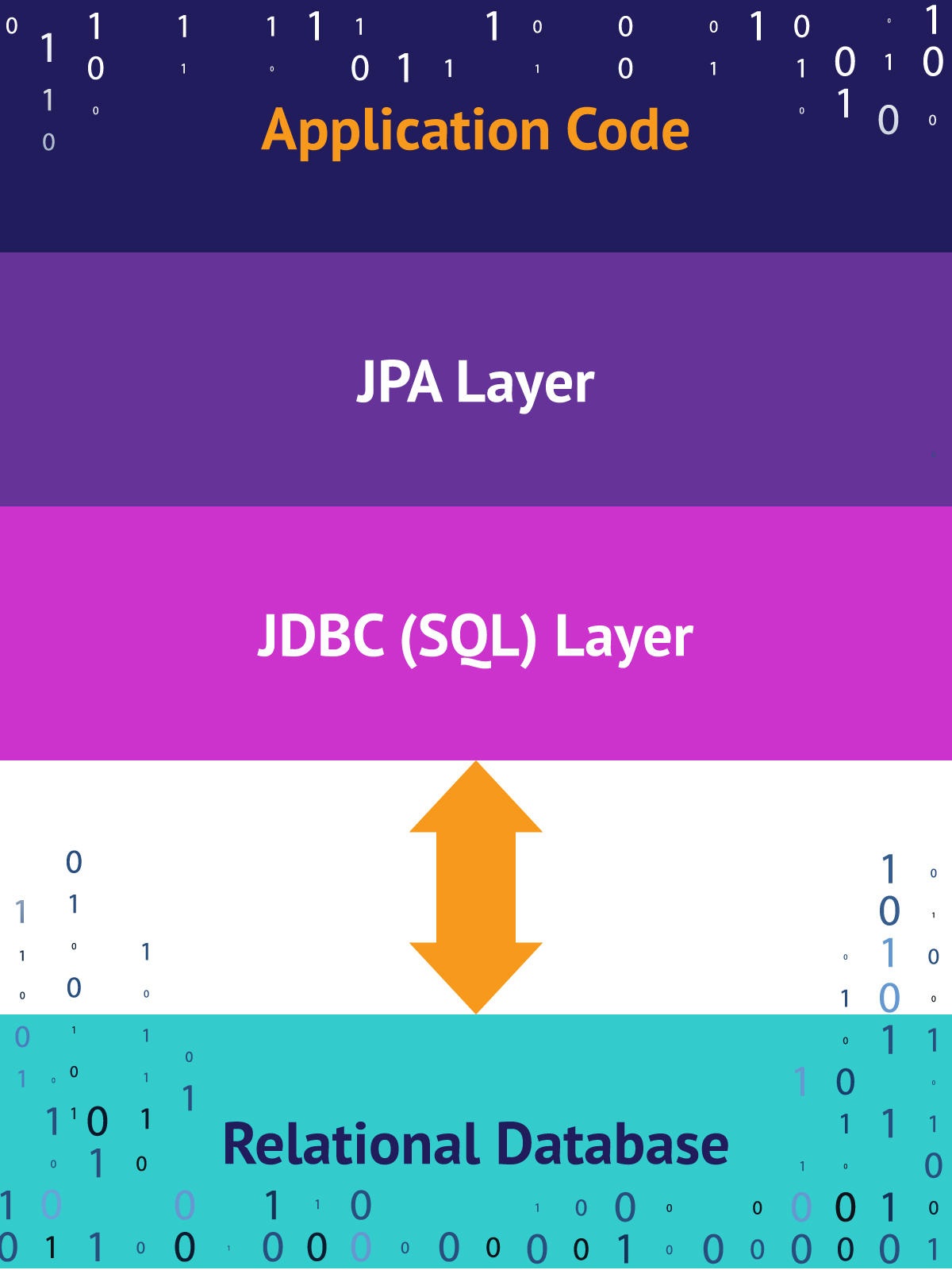
**[**[**Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!**](https://pluralsight.pxf.io/c/321564/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fpaths%2Fjava)**]**

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](http://hibernate.org/ogm/) and [EclipseLink](https://www.eclipse.org/eclipselink).

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

JavaWorld / IDG

*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](http://db.apache.org/jdo/why_jdo.html) 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](https://www.javaworld.com/article/3388036/what-is-jdbc-introduction-to-the-java-database-connectivity-api.html).

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](https://javabeat.net/eclipselink-jpa-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 Musicians 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**

JPA has annotations for each of of its relationship mapping types. Listing 7 shows how you might annotate the one-to-many relationship between Musician and Performances.

### Listing 7. Annotating a one-to-many relationship

**public** **class** **Musician** {

@OneToMany

@JoinColumn(name="musicianId")

**private** **List**<**Performance**> performances = **new** **ArrayList**<**Performance**>();

*//...*

}

One thing to notice is that the @JoinColumn tells JPA what column on the **Performance** table will map to the Musician entity. Each performance will be associated to a single Musician, which is tracked by this column. When JPA loads a Musician or a Performance into the database, it will use this information to reconstitute the object graph.

**Fetching strategies in JPA**

In addition to knowing *where* to place related entities in the database, JPA needs to know *how* you want them loaded. *Fetching strategies* tell JPA how to load related entities. When loading and saving objects, a JPA framework must provide the ability to finetune how object graphs are handled. For instance, if the Musician class has a bandMate field (as shown in Listing 7), loading george could cause the entire **Musician** table to be loaded from the database!

What is needed is the ability to define *lazy loading* of related entities--recognizing, of course, that [relationships in JPA can be eager or lazy](https://www.javaworld.com/article/2077819/understanding-jpa-part-2-relationships-the-jpa-way.html). You can use annotations to customize your fetching strategies, but JPA's default configuration often works out of the box, without changes:

1. One-to-many: Lazy
2. Many-to-one: Eager
3. Many-to-many: Lazy
4. One-to-one: Eager

**Transactions in JPA**

While outside the scope of this short introduction, *transactions* allow the developer to write to the database. In Listing 1, we achieved a simple transaction with the line: em.getTransaction().commit();. Transactions can be defined in a variety of ways, from explicit interactions via the API, to using annotations to define transactional boundaries, to using Spring AOP to define transactions.

**JPA installation and setup**

We'll conclude with a quick look at installing and setting up JPA for your Java applications. For this demonstration I'll use EclipseLink, the JPA reference implementation.

The common way to install JPA is to *include* a JPA provider into your project. Listing 8 shows how you would include EclipseLink as a dependency in your Maven pom.xml file.

**Listing 8. Include EclipseLink as a Maven dependency**

org.eclipse.persistence

eclipselink

2.5.0-RC1

You will also need to include the driver for your database, as shown in Listing 9.

**Listing 9. Maven dependency for a MySql connector**

mysql

mysql-connector-java

5.1.32

Next, you'll need to tell the system about your database and provider. This is done in the persistence.xml file, as shown in Listing 10.

**Listing 10. Persistence.xml**

http:*//xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">*

<persistence-unit name="MyUnit" transaction-type="RESOURCE\_LOCAL">

<properties>

<**property** name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/foo\_bar"/>

<**property** name="javax.persistence.jdbc.user" value=""/>

<**property** name="javax.persistence.jdbc.password" value=""/>

<**property** name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>

</properties>

</persistence-unit>

</persistence>

There are other ways to provide this information to the system, including programmatically. I recommend using the persistence.xml file because storing dependencies this way makes it very easy to update your application without modifying code.

**Spring configuration for JPA**

Using Spring will greatly ease the integration of JPA into your application. As an example, placing the @SpringBootApplication annotation in your application header instructs Spring to automatically scan for classes and inject the EntityManager as required, based on the configuration you've specified.

Listing 11 shows the dependencies to include if you want Spring's JPA support for your application.

**Listing 11. Adding Spring JPA support in Maven**

org.springframework.boot

spring-boot-starter

2.1.3.RELEASE

org.springframework.boot

spring-boot-starter-data-jpa

2.1.3.RELEASE

**Conclusion**

Every application that deals with a database should define an application layer whose sole purpose is to isolate persistence code. As you've seen in this article, the Java Persistence API introduces a range of capabilities and support for Java object persistence. Simple applications may not require all of JPA's capabilities, and in some cases the overhead of configuring the framework may not be merited. As an application grows, however, JPA's structure and encapsulation really earn their keep. Using JPA keeps your object code simple and provides a conventional framework for accessing data in Java applications.

**Learn more about the Java Persistence API and related technologies**

* [What is JDBC? Introduction to Java Database Connectivity](https://www.javaworld.com/article/3388036/what-is-jdbc-introduction-to-the-java-database-connectivity-api.html): Get to know Java's low-level API for connecting to a database, issuing SQL queries, and more.
* [Java persistence with JPA and Hibernate, Part 1: Entities and relationships](https://www.javaworld.com/article/3373652/java-persistence-with-jpa-and-hibernate-part-1-entities-and-relationships.html): Get started modeling entities and relationships with Java 8 and Hibernate ORM.
* [Java persistence with JPA and Hibernate, Part 2](https://www.javaworld.com/article/3387643/java-persistence-with-jpa-and-hibernate-part-2-many-to-many-relationships.html): Hands-on practice modeling many-to-many relationships and inheritance relationships in Java 8 and Hibernate ORM.
* [Implementing state-dependent behavior](https://www.javaworld.com/article/2077003/how-to-implement-state-dependent-behavior.html): A classic introduction to the state pattern and state dependency in Java.
* [Eager/Lazy Loading In Hibernate](https://www.baeldung.com/hibernate-lazy-eager-loading): How to apply eager and lazy loading in Hibernate.
* [JPA 2.2 brings some highly anticipated changes](https://www.infoq.com/articles/JPA-2.2-Brings-Highly-Anticipated-Changes): An overview of specification updates in JPA 2.2, including improvements to CDI, better alignment with the Date and Time API, and support for @Repeatable annotations.
* [Should you use JPA for your next project?](https://thoughts-on-java.org/use-jpa-next-project/): Key questions that could help you decide.

*This story, "What is JPA? Introduction to the Java Persistence API" was originally published by*[*JavaWorld*](http://www.javaworld.com/)*.*