What is JPA? Introduction to Java persistence

<https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>

# Jakarta Persistence API

As a specification, the [Jakarta Persistence API](https://jcp.org/en/jsr/detail?id=338) (formerly 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 they are persisted in your Java applications.

By itself, JPA is not a tool or framework; rather, it defines a set of concepts that guide implementers. 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 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 3.

# JPA as opposed to [JDBC](https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html)

The core idea behind JPA as opposed to [JDBC](https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html), is that for the most part, JPA lets you avoid the need to “think relationally." In JPA, you define your persistence rules in the realm of Java code and objects, whereas JDBC requires you to manually translate from code to relational tables and back again.

# JPA 3 in Jakarta EE

The Java Persistence API was first released as a subset of the Enterprise JavaBeans 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)). JPA was adopted as an independent project of Jakarta EE in 2019. The current release as of this writing is [JPA 3.1](https://projects.eclipse.org/projects/ee4j.jpa).

Popular JPA implementations like Hibernate and EclipseLink now support JPA 3. Migrating from JPA 2 to JPA 3 involves some [namespace changes](https://thorben-janssen.com/migrating-jpa-2-x-to-3-0/), but otherwise the changes are under-the-hood performance gains.

# JPA and Hibernate

Because of their intertwined history, Hibernate and JPA are frequently conflated. However, like the [Java Servlet](https://www.infoworld.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 many JPA tools.

Developed by Gavin King and first 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. 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. The latest release as of this writing, [Hibernate ORM 6](https://hibernate.org/orm/releases/), implements JPA 2.2. Additional Hibernate tools include [Hibernate Search](https://hibernate.org/search), [Hibernate Validator](https://hibernate.org/validator/%22), 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 Enterprise JavaBeans (EJB) 3.0, but has since evolved as its own specification. [EJB is a specification with a different focus from JPA](https://www.infoworld.com/article/3432125/what-is-ejb-the-evolution-of-enterprise-javabeans.html), 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.

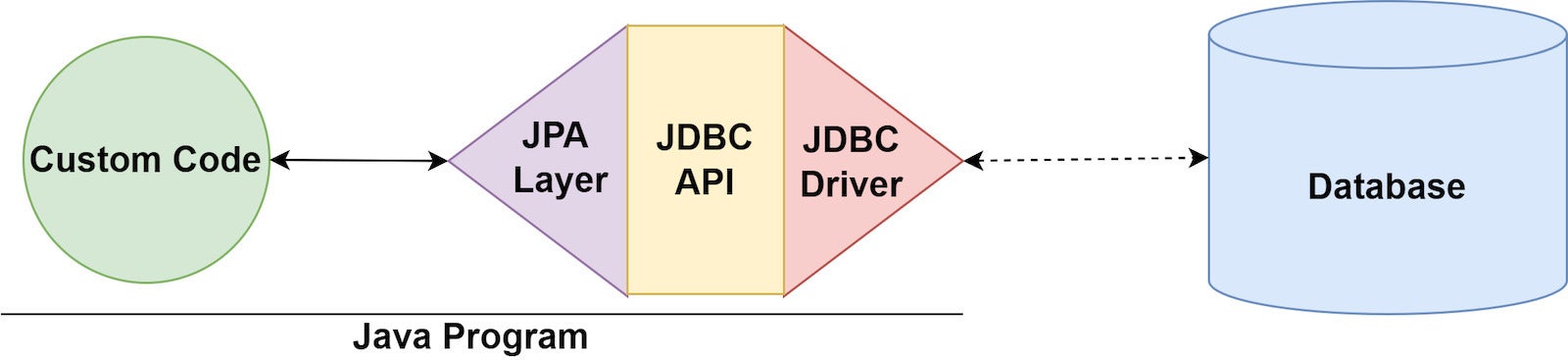
Object-relational mapping is a task–one that developers have good reason to avoid doing manually. A framework like Hibernate ORM or EclipseLink codifies that task into a library or framework, an ORM layer. As part of the application architecture, the ORM layer is responsible for managing the conversion 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, and by sticking with defaults, you avoid having to maintain configuration metadata.

# 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.

*Figure 1. JPA and the Java 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. We'll take a look at both manual and Spring-based JPA installation and setup shortly.

# Java Data Objects

Java Data Objects (JDO) 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 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.

As a specification, 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.