Pro Jakarta Persistence in Jakarta EE 10

An In-Depth Guide to Persistence in Enterprise Java Development

—

Fourth Edition

—

Lukas Jungmann
Mike Keith
Merrick Schincariol
Massimo Nardone

Apress®

Pro Jakarta Persistence in Jakarta EE 10

An In-Depth Guide to Persistence in Enterprise Java Development

Fourth Edition

Lukas Jungmann
Mike Keith
Merrick Schincariol
Massimo Nardone

Apress®

Pro Jakarta Persistence in Jakarta EE 10: An In-Depth Guide to Persistence in Enterprise Java Development

Lukas Jungmann
Prague, Czech Republic

Mike Keith
Ottawa, ON, Canada

Merrick Schincariol
Almonte, ON, Canada

Massimo Nardone
HELSINKI, Finland

ISBN-13 (pbk): 978-1-4842-7442-2
ISBN-13 (electronic): 978-1-4842-7443-9
https://doi.org/10.1007/978-1-4842-7443-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: James Markham
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Steve Harvey on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484274422. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Bára, Tobiáš, Sofie and Mikuláš.
I love you.

—Lukáš

Table of Contents

About the Authors

Lukas Jungmann is the specification project lead for Jakarta Persistence and for a number of other Jakarta Specification projects including Jakarta Activation, Mail, XML Binding, SOAP with Attachments, and XML Web Services; contributor to Jakarta Platform, JSON Processing, and JSON Binding specification projects; lead for a number of implementation projects of various Jakarta specifications including EclipseLink, Eclipse Metro, and Eclipse Angus. He holds a bachelor's degree in Applied Informatics from the University of Finance and Administration in Prague, Czech Republic, and has over 15 years of experience working with Enterprise Java–related technologies. He has spoken at numerous conferences around the world. He is employed as a software developer at Oracle in Prague, Czech Republic.

Mike Keith was the co-specification lead for JPA 1.0 and a member of the JPA 2.0 and JPA 2.1 expert groups. He sits on a number of other Java Community Process expert groups and the Enterprise Expert Group (EEG) in the OSGi Alliance. He holds a master's degree in Computer Science from Carleton University and has over 20 years of experience in persistence and distributed systems research and practice. He has written papers and articles on JPA and spoken at numerous conferences around the world. He is employed as an architect at Oracle in Ottawa, Canada. He is married and has four kids and two dogs.

Merrick Schincariol is a consulting engineer at Oracle, specializing in middleware technologies. He has a Bachelor of Science degree in Computer Science from Lakehead University and has more than a decade of experience in enterprise software development. He spent some time consulting in the pre-Java enterprise and business intelligence fields before moving on to write Java and J2EE applications. His experience with large-scale systems and data warehouse design gave him a mature and practiced perspective on enterprise software, which later propelled him into doing Java EE container implementation work.

Massimo Nardone has more than 25 years of experience in security, web/mobile development, cloud, and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a CISO, CSO, security executive, IoT executive, project manager, software engineer, research engineer, chief security architect, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. His technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and more.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas). He is currently working for Cognizant as head of cyber security and CISO to help both internally and externally with clients in areas of information and cyber security, like strategy, planning, processes, policies, procedures, governance, awareness, and so forth. In June 2017 he became a permanent member of the ISACA Finland Board.

Massimo has reviewed more than 45 IT books for different publishing companies and is the co-author of *Pro Spring Security: Securing Spring Framework 5 and Boot 2-based Java Applications* (Apress, 2019), *Beginning EJB in Java EE 8* (Apress, 2018), *Pro JPA 2 in Java EE 8* (Apress, 2018), and *Pro Android Games* (Apress, 2015).

About the Technical Reviewer

Jan Beernink works for Google and is a contributor to several projects related to OmniFaces. Jan holds an MSc degree in Computer Science from the Vrije Universiteit of Amsterdam, the Netherlands.

Acknowledgments

Many thanks go to my awesome and beloved family - my wife Barbora, and my children Tobiáš, Sofie, and Mikuláš - for their endless patience and support while working on this book.

I also want to thank Steve Anglin for giving me the opportunity to work on this edition of this book. A special thanks goes, to Mark Powers for supporting me during the editorial process.

Finally, I want to thank Jan Beernink, the technical reviewer of this book, for helping me make the book better.

—Lukas Jungmann

CHAPTER 1

Introduction

Enterprise applications are defined by their need to collect, process, transform, and report on vast amounts of information. And, of course, that information has to be kept somewhere. Storing and retrieving data is a multibillion-dollar business, evidenced in part by the growth of the database market as well as the emergence of cloud-based storage services. Despite all the available technologies for data management, application designers still spend much of their time trying to efficiently move their data to and from storage.

Despite the success the Java platform has had in working with database systems, for a long time it suffered from the same problem that has plagued other object-oriented programming languages. Moving data back and forth between a database system and the object model of a Java application was a lot harder than it needed to be. Java developers either wrote lots of code to convert row and column data into objects or found themselves tied to proprietary frameworks that tried to hide the database from them. Fortunately, a standard solution, the Jakarta Persistence API, was introduced into the platform to bridge the gap between object-oriented domain models and relational database systems.

This book introduces version 3.1 of the Jakarta Persistence API as part of the Jakarta EE 10 and explores everything that it has to offer developers.

One of its strengths is that it can be slotted into whichever layer, tier, or framework an application needs it to be in. Whether you are building client-server applications to collect form data in a Swing application or building a website using the latest application framework, Jakarta Persistence can help you provide persistence more effectively.

To set the stage for Jakarta Persistence, this chapter first takes a step back to show where we've been and what problems we are trying to solve. From there, we will look at the history of the specification and give you a high-level view of what it has to offer.

L. Jungmann et al., *Pro Jakarta Persistence in Jakarta EE 10*, https://doi.org/10.1007/978-1-4842-7443-9_1

Relational Databases

Many ways of persisting data have come and gone over the years, and no concept has more staying power than the relational database. Even in the age of the cloud, when "Big Data" and "NoSQL" regularly steal the headlines, relational database services are in consistent demand to enable today's enterprise applications running in the cloud. While key-value and document-oriented NoSQL stores have their place, relational stores remain the most popular general-purpose databases in existence, and they are where the vast majority of the world's corporate data is stored. They are the starting point for every enterprise application and often have a lifespan that continues long after the application has faded away.

Understanding relational data is key to successful enterprise development. Developing applications to work well with database systems is a commonly acknowledged hurdle of software development. A good deal of Java's success can be attributed to its widespread adoption for building enterprise database systems. From consumer websites to automated gateways, Java applications are at the heart of enterprise application development. Figure 1-1 shows an example of a relational database of user to car.

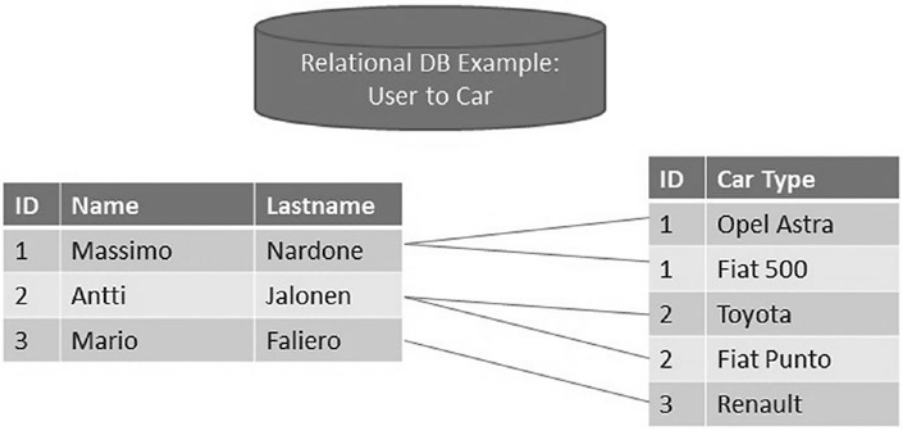

Figure 1-1. *User to car relational database*

Object-Relational Mapping

"The domain model has a class. The database has a table. They look pretty similar. It should be simple to convert one to the other automatically." This is a thought we've probably all had at one point or another while writing yet another data access object (DAO) to convert Java Database Connectivity (JDBC) result sets into something object-oriented. The domain model looks similar enough to the relational model of the database that it seems to cry out for a way to make the two models talk to each other.

The technique of bridging the gap between the object model and the relational model is known as object-relational mapping, often referred to as O-R mapping or simply ORM. The term comes from the idea that we are in some way mapping the concepts from one model onto another, with the goal of introducing a mediator to manage the automatic transformation of one to the other.

Before going into the specifics of object-relational mapping, let's define a brief manifesto of what the ideal solution should be:

- *Objects, not tables*: Applications should be written in terms of the domain model, not bound to the relational model. It must be possible to operate on and query against the domain model without having to express it in the relational language of tables, columns, and foreign keys.
- *Convenience, not ignorance*: Mapping tools should be used only by someone familiar with relational technology. O-R mapping is not meant to save developers from understanding mapping problems or to hide them altogether. It is meant for those who have an understanding of the issues and know what they need, but who don't want to have to write thousands of lines of code to deal with a problem that has already been solved.
- *Unobtrusive, not transparent*: It is unreasonable to expect that persistence be transparent because an application always needs to have control of the objects that it is persisting and be aware of the entity lifecycle. The persistence solution should not intrude on the domain model, however, and domain classes must not be required to extend classes or implement interfaces in order to be persistable.

- *Legacy data, new objects*: It is far more likely that an application will target an existing relational database schema than create a new one. Support for legacy schemas is one of the most relevant use cases that will arise, and it is quite possible that such databases will outlive every one of us.
- *Enough, but not too much*: Enterprise developers have problems to solve, and they need features sufficient to solve those problems. What they don't like is being forced to eat a heavyweight persistence model that introduces large overhead because it is solving problems that many do not even agree *are* problems.
- *Local, but mobile*: A persistent representation of data does not need to be modeled as a full-fledged remote object. Distribution is something that exists as part of the application, not part of the persistence layer. The entities that contain the persistent state, however, must be able to travel to whichever layer needs them so that if an application is distributed, then the entities will support and not inhibit a particular architecture.
- *Standard API, with pluggable implementations*: Large companies with sizable applications don't want to risk being coupled to product-specific libraries and interfaces. By depending only on defined standard interfaces, the application is decoupled from proprietary APIs and can switch implementations if another becomes more suitable.

This would appear to be a somewhat demanding set of requirements, but it is one born of both practical experience and necessity. Enterprise applications have very specific persistence needs, and this shopping list of items is a fairly specific representation of the experience of the enterprise community.

The Impedance Mismatch

Advocates of object-relational mapping often describe the difference between the object model and the relational model as the impedance mismatch between the two. This is an apt description because the challenge of mapping one to the other lies not in the similarities between the two, but in the concepts in each for which there is no logical equivalent in the other.

In the following sections, we present some basic object-oriented domain models and a variety of relational models to persist the same set of data. As you will see, the challenge in object-relational mapping is not so much the complexity of a single mapping but that there are so many possible mappings. The goal is not to explain how to get from one point to the other but to understand the roads that may have to be taken to arrive at an intended destination.

Class Representation

Let's begin this discussion with a simple class. Figure 1-2 shows an `Employee` class with four attributes: employee ID, employee name, start date, and current salary.

Figure 1-2. *The Employee class*

Now consider the relational model shown in Figure 1-3. The ideal representation of this class in the database corresponds to scenario (A). Each field in the class maps directly to a column in the table. The employee ID becomes the primary key. With the exception of some slight naming differences, this is a straightforward mapping.

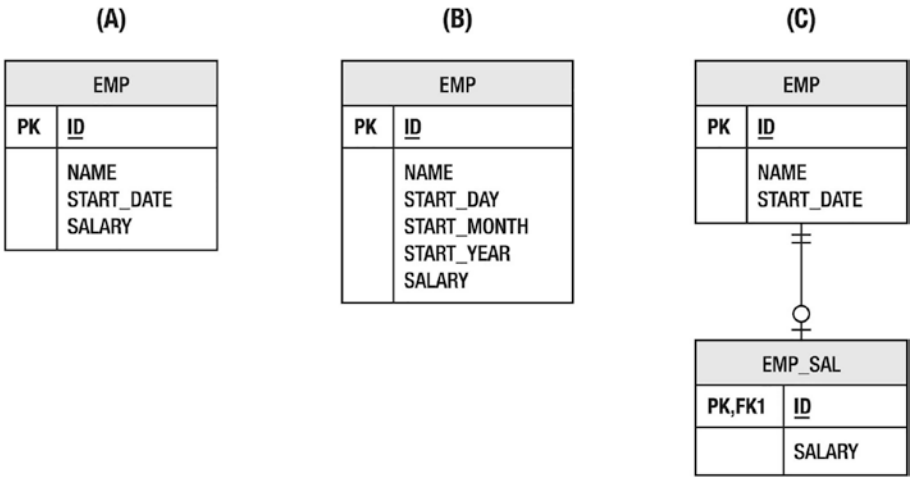

Figure 1-3. *Three scenarios for storing employee data*

In scenario (B), we see that the start date of the employee is actually stored as three separate columns, one each for the day, month, and year. Recall that the class used a `Date` object to represent this value. Because database schemas are much harder to change, should the class be forced to adopt the same storage strategy in order to remain consistent with the relational model? Also consider the inverse of the problem, in which the class had used three fields, and the table used a single date column. Even a single field becomes complex to map when the database and object model differ in representation.

Salary information is considered commercially sensitive, so it may be unwise to place the salary value directly in the `EMP` table, which may be used for a number of purposes. In scenario (C), the `EMP` table has been split so that the salary information is stored in a separate `EMP_SAL` table. This allows the database administrator to restrict `SELECT` access on salary information to those users who genuinely require it. With such a mapping, even a single store operation for the `Employee` class now requires inserts or updates to two different tables.

Clearly, even storing the data from a single class in a database can be a challenging exercise. We concern ourselves with these scenarios because real database schemas in production systems were never designed with object models in mind. The rule of thumb in enterprise applications is that the needs of the database trump the wants of the application. In fact, there are usually many applications, some object-oriented and some based on Structured Query Language (SQL), which retrieve from and store data into a single database. The dependency of multiple applications on the same database means that changing the database would affect every one of the applications, clearly an undesirable and potentially expensive option. It's up to the object model to adapt and find ways to work with the database schema without letting the physical design overpower the logical application model.

Relationships

Objects rarely exist in isolation. Just like relationships in a database, domain classes depend on and associate themselves with other domain classes. Consider the `Employee` class introduced in Figure 1-2. There are many domain concepts that could be associated with an employee, but for now let's introduce the `Address` domain class, for which an `Employee` may have at most one instance. We say in this case that `Employee` has a one-toone relationship with `Address`, represented in the Unified Modeling Language (UML) model by the `0..1` notation. Figure 1-4 demonstrates this relationship.

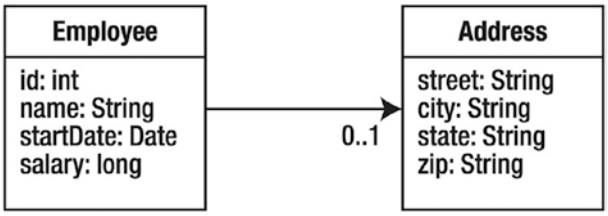

Figure 1-4. *The Employee and Address relationship*

We discussed different scenarios for representing the `Employee` state in the previous section, and likewise there are several approaches to representing a relationship in a database schema. Figure 1-5 demonstrates three different scenarios for a one-to-one relationship between an employee and an address.

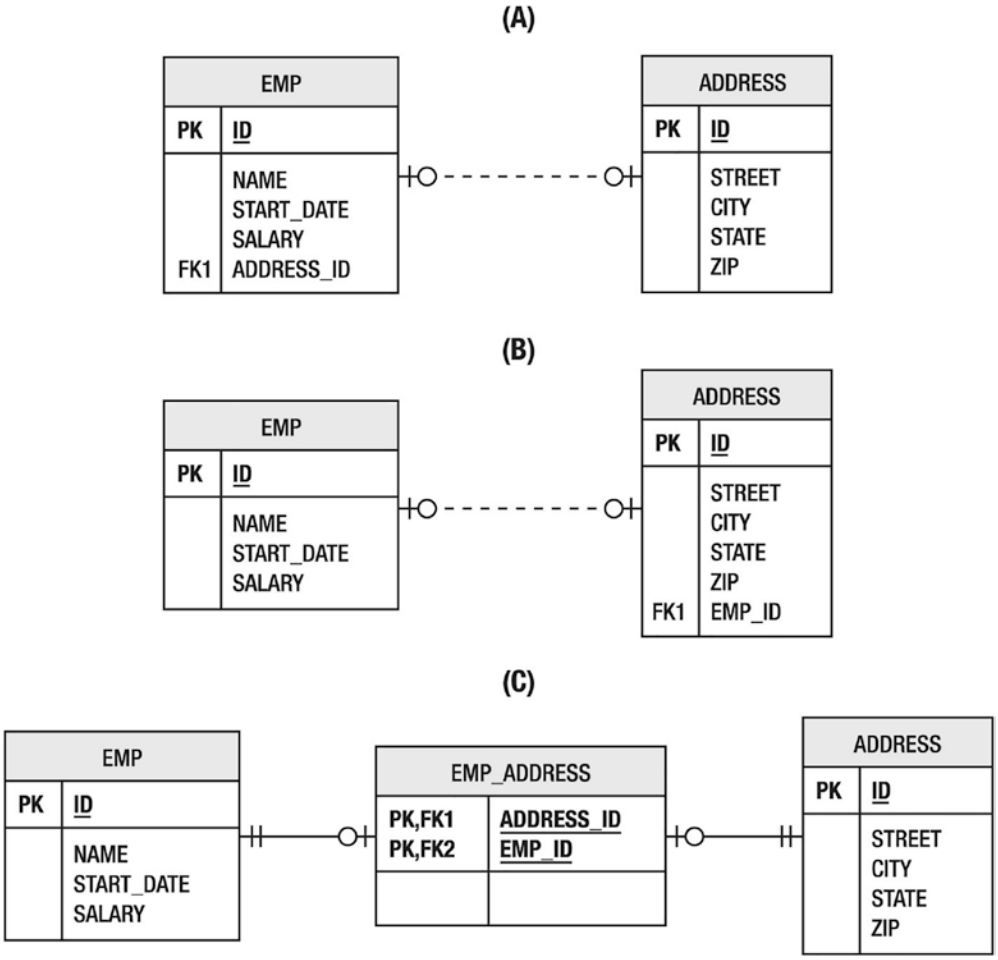

Figure 1-5. *Three scenarios for relating employee and address data*

The building block for relationships in the database is the foreign key. Each scenario involves foreign key relationships between the various tables, but in order for there to be a foreign key relationship, the target table must have a primary key. And so before we even get to associate employees and addresses with each other, we have a problem. The domain class `Address` does not have an identifier, yet the table that it would be stored in must have one if it is to be part of relationships. We could construct a primary key out of all of the columns in the `ADDRESS` table, but this is considered bad practice. Therefore, the `ID` column is introduced, and the object-relational mapping will have to adapt in some way.

Scenario (A) of Figure 1-5 shows the ideal mapping of this relationship. The `EMP` table has a foreign key to the `ADDRESS` table stored in the `ADDRESS_ID` column. If the `Employee` class holds onto an instance of the `Address` class, the primary key value for the address can be set during store operations when an `EMPLOYEE` row gets written.

And yet consider scenario (B), which is only slightly different yet suddenly much more complex. In the domain model, an `Address` instance did not hold onto the `Employee` instance that owned it, and yet the employee primary key must be stored in the `ADDRESS` table. Either the object-relational mapping must account for this mismatch between domain class and table or a reference back to the employee will have to be added for every address.

To make matters worse, scenario (C) introduces a join table to relate the `EMP` and `ADDRESS` tables. Instead of storing the foreign keys directly in one of the domain tables, the join table holds onto the pair of keys. Every database operation involving the two tables must now traverse the join table and keep it consistent. We could introduce an `EmployeeAddress` association class into the domain model to compensate, but that defeats the logical representation we are trying to achieve.

Relationships present a challenge in any object-relational mapping solution. This introduction covered only one-to-one relationships, and yet we have been faced with the need for primary keys not in the object model and the possibility of having to introduce extra relationships into the model or even associate classes to compensate for the database schema.

Inheritance

A defining element of an object-oriented domain model is the opportunity to introduce generalized relationships between like classes. Inheritance is the natural way to express these relationships and allows for polymorphism in the application. Let's revisit the `Employee` class shown in Figure 1-2 and imagine a company that needs to distinguish between full-time and part-time employees. Part-time employees work for an hourly rate, while full-time employees are assigned a salary. This is a good opportunity for inheritance, moving wage information to the `PartTimeEmployee` and `FullTimeEmployee` subclasses. Figure 1-6 shows this arrangement.

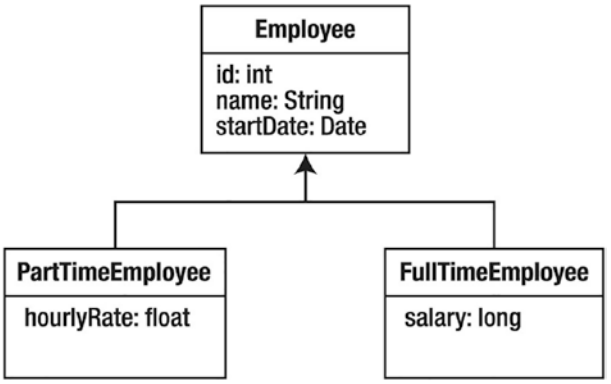

Figure 1-6. *Inheritance relationships between full-time and part-time employees*

Inheritance presents a genuine problem for object-relational mapping. We are no longer dealing with a situation in which there is a natural mapping from a class to a table. Consider the relational models shown in Figure 1-7. Once again, three different strategies for persisting the same set of data are demonstrated.