Mastering Java Persistence API (JPA)

Realize Java's Capabilities Spanning RDBMS, ORM, JDBC, Caching, Locking, Transaction Management, and JPQL

NISHA PARAMESWARAN KURUR

bpb

Mastering Java Persistence API (JPA)

Realize Java's Capabilities Spanning RDBMS, ORM, JDBC, Caching, Locking, Transaction Management, and JPQL

Nisha Parameswaran Kurur

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-55511-263

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

To View Complete BPB Publications Catalogue Scan the QR Code:

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Foreword

The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

– Robert Frost

The above lines from the poem "*Stopping by Woods on a Snowy Evening*" by Robert Frost always propel me. I think this is true for every professional, aspirant, and whoever wants to be a part of the industry. I believe learning is an endless journey for everyone; sometimes we learn from the experienced fellows the other time even a kid teaches us in a new way. The important point is learning and it does not matter from whom we are grabbing the skills. This book expounds on the above quote, and I must say, you will praise yourself once you read this book.

James Gosling, Mike Sheridan, and Patrick Naughton – a great team gave us a gift of live-long technology (language and framework). It is now more than two decades since the first stable release of Java in the year 1995. Java is still famous and popular among many industries. However, there are many new adaptions and additions we have now in Java but the base fragrance of Java is the same. Gradually, this language is coming with more and more features. You have to update yourself if you want to swim in the ocean. The current book is very helpful for you to build your basic blocks with the latest terminology, additions and/or updates in this language.

I appreciate the style of the author how she curated and assembled the book. This book meets its purpose as layout by the author. The author has built the basic blocks of the widely used topics in a detailed manner, and are:

- Object Relational Mapping.
- Database terminologies and their abstracts
- What's new in the language?
- Runtime objects

The author has started with theory to define the basic concepts and visualize them in the form of dynamic diagrams. In the mid of the book, you will know the terminologies that can be used even with the help of short/tiny examples. In the final phase of the book, the author presents the difference between old and new versions. These are defined in very simple examples and in the form of images/diagrams so that one can easily grab these concepts. Additionally, the author has provided an appendix where you can find the solutions to the problems. Apart from this, every chapter has assessment sections so that the readers can assess themselves after going through a chapter.

This book is built with basic blocks and can be a ready reckoner for any professional or student. You'll learn and understand them theoretically to implement various topics. After reading this book, you will realize that you are now ready to learn advanced things and if you know the advanced concepts already, then you will find yourself refreshed with the additions/updates that we have in the current version of the language.

I hope you appreciate this book as much as I did. And I hope you enjoy working on database analysis.

Enjoy!

Dr. Gaurav Aroraa
New Delhi, India

Dedicated to

My beloved parents
&
My cherished family

About the Author

Nisha Parameswaran Kurur is a technical architect with a wide range of interests, including topics such as databases, algorithms, data structures, operating systems, and networking principles. She has worked on numerous real-world application projects for large clients in various domains like networking, broadcast, retail, finance, healthcare, etc. with different companies including services as well as product sectors from the IT industry. She earned her Master's Degree in Computer Science from the Indian Institute of Technology Madras (IITM), Chennai, India and has published conference papers and white papers. She is also a specialized coding skills instructor for courses conducted by ASAP, Kerala Government. She is deeply interested in propagating her knowledge to wider audiences and hence is involved in authoring books in local languages for non-technical people (with a great focus on next-generation kids) who have the interest to learn computer subjects.

About the Reviewers

- **Gaurav Aroraa** is a tech enthusiast and technical consultant with more than 23 years of experience in the industry. He has a Doctorate in Computer Science. Gaurav is a Microsoft MVP award recipient. He is a lifetime member of the Computer Society of India (CSI), an advisory member and senior mentor at IndiaMentor, certified as a Scrum trainer and coach, ITIL-F certified, and PRINCE-F and PRINCE-P certified. Gaurav is an open-source developer and a contributor to the Microsoft TechNet community. He has authored books across the technologies, including Microservices by Examples Using .NET Core (BPB Publications).

  Blog links: **https://gaurav-arora.com/blog/**

  LinkedIn Profile: **https://www.linkedin.com/in/aroragaurav/**

- **Abhijeet Prakash** has 6 years of extensive experience in Artificial Intelligence, Machine Learning, NLP, Deep Learning and Full Stack Development using tools like Cookiecutter, KerasTuner, DVC, Google Cloud Video API, Selenium, BS, Google Colab, Apigee, FastAPI, AWS, Google Cloud Platform with programming languages like Ruby, Node, Python, etc. with a demonstrated history of working in information technology and product-based companies.

  Abhijeet has pursued MCA from the Department of Mathematical Science and Information Technology, B.U. He has worked with various banks, NBFCs, and Fin-Tech companies. He has worked as Machine Learning Engineer and is currently working as an Artificial Intelligence Engineer. Abhijeet has also written a book on Ethical Hacking.

Acknowledgement

There are quite a few people I want to thank for the continued and ongoing support they have given me. First and foremost, I would like to thank my parents for their vision and motivation, without which I would not have been what I am today. Next, I would like to thank my family for their whole-hearted and continuous encouragement and support, which has always been my pillar of strength. A special note of thanks to my kids for actively participating in my book writing activities by helping me in creating diagrams.

Words fall short to explain my appreciation to my friends and teachers, right from my school to my masters, who shaped me into the person I am now. I am grateful to all my colleagues and employers (current and previous) who gave me various opportunities to learn and grow.

My gratitude extends to the team at BPB Publications for handing over such a huge responsibility and being supportive enough to provide me with quite a long time to finish the book. Covid-19 pandemic and other personal interruptions kept me away from the book writing process for more time than expected. Kudos to the book coordinators at BPB, whose trust, patience, and persistence kept me going. I profoundly thank my book reviewers for doing an excellent job in a very short span of time. Overall, it was a tremendous experience that cannot be explained in words.

Preface

This book covers Java persistence API – starting from its birth to the recent advancements by including different kinds of databases that are non-relational. The key focus is on the JPA concepts and their usage in different JPA providers with a special focus on Hibernate and EclipseLink.

The book describes the basic concepts of a relational database that are required to understand the JPA concepts as well. It demonstrates various features of JPA with code snippets that include configuration using XML (old style) as well as annotations (new style). It also provides a good understanding of complex database concepts like locking, concurrency and transactions.

The book takes a practical approach with more focus on JPA provider implementations - it focuses on two widely used implementations namely, Hibernate and EclipseLink. This book also provides sample mavenized project which can be used as a starting step for any JPA-based project creation. It also provides important JPA questions and their solutions which helps the reader to be well prepared for any interview.

Chapter 1 will aim at introducing Java, and various strategies formed in Java for persisting data followed by a detailed explanation of various terms related to Object-relational model, mapping, and impedance mismatch and finally concluding with a discussion on various types of technologies that are/were used to solve the Java persistence problems.

Chapter 2 will concentrate on the most widely used persistence model namely relational databases. All the basic concepts like tables, rows, columns, cells, keys, types of data, conversions, and access types are detailed out with specific mention on how these are used in JPA. Here the main aim is to introduce the relational database design concepts and their usage in JPA. It is noteworthy that configurations are given in both XML as well as annotations as much as possible.

Chapter 3 will focus on operations on the relational database like identifiers and how to use them as keys. It also discusses sequencing strategies to generate identifiers, and other strategies to generate primary keys. This chapter touches upon inheritance and locking techniques with a special interest in various locking strategies like optimistic, pessimistic, etc.

Chapter 4 will discuss more on how the stored data are related to one another. This also provides a detailed discussion on various relations found in RDBMS and how they are defined in JPA as well as implemented in JPA providers.

The storage of data becomes relevant only if it can be retrieved later on for reference.

Chapter 5 will discuss more on how the stored data can be retrieved and what all querying mechanisms are supported in JPA. This also discusses the JPQL BNF which can be considered as a reference for those who want to go in-depth and understand the JPQL building blocks. Note that this JPQL BNF is optional and can be skipped if required. This will not cause any break in the flow of the contents.

Chapter 6 will explain the Entity Manager and its role in JPA. Persisting, caching and transaction management come under this section and is helpful in understanding the important role that the Entity Manager has in any JPA implementation. Various CRUD operations performed by RDBMS become the Entity Manager's responsibility.

Chapter 7 will describe the two most widely used JPA implementations – Hibernate and EclipseLink. Both the implementations will be explained in the same structure – history, features, architecture, and querying being the main topics. A few special topics like cache architecture and poly glot persistence will also be covered as part of this.

Appendix Part 1 will discuss advanced topics in JPA like events, views, stored procedures, replication, multi-tenancy, auditing, NoSQL, etc. There are many more topics that might raise interest in the reader to understand more about the relational and non-relational world of data and their object references.

Appendix Part 2 will provide a sample maven project with a JPA example which can be considered as a starting project for JPA. The desired output is also mentioned so that it can be verified by the reader. This also provides a list of sample questions along with solutions or hints to the solution which aims at making the reader aware of the type of questions to be expected during an interview. Note that, for this reason, every chapter is provided with a Q&A section and points to ponder which provides many interesting facts.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/2mrczls

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Mastering-Java-Persistence-API-JPA-**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

Section - I
Introduction

Chapter 1

Java Persistence API and Object-Relational Mapping

Introduction

In 1991, *James Gosling*, *Mike Sheridan* and *Patrick Naughton*, a team of *Sun engineers*, initiated a project to develop a simple, robust, portable, platform-independent, object-oriented, multi-threaded, high performance, interpreted language that can be used for small, embedded systems. Since the team was named *Green*, the language was initially named *Greentalk*. Considering its similarity in name to *Smalltalk*, it was renamed to *Oak*, as it is considered as a symbol of strength. As this name was already trademarked by *Oak technologies*, *James Gosling* chose the name Java while drinking a cup of coffee (Java is the brand name as well as the name of the island in Indonesia where the first coffee was produced).

The first release of Java was in 1995 by *Sun Microsystems* as a free, platform-independent, object-oriented, application programming language. This was considered as one among the 10 best products of 1995. However, the stable release of Java (*JDK 1.0*) happened in Jan 1996. Java's popularity grew as one of the best choices for web application programming, and this association of Java with "*Internet*" was due to the *Applet* feature that could be embedded onto a web page and run on a browser. Being free, the availability of source code enabled Java to be adopted by many companies, and hence its growth fostered from being client-side applets to the standard server-side language. **Java Enterprise Edition** (**JEE**) is an open enterprise software platform that provides a standard model for server applications that can

be run on almost all Java-compatible Enterprise products. Using JEE, developers can easily adapt to the increasing and greater demands in the IT industry without incurring extensive additional costs.

Few of the main technologies that make up the JEE are as follows:

- Java Servlets
- JavaServer Pages (JSP)
- Enterprise JavaBeans (EJB)
- Java Message Service (JMS)
- Java Naming and Directory Interface (JNDI)
- Java Database Connectivity (JDBC)
- JavaMail
- Java Transaction Service (JTS)
- Java Transaction API (JTA)
- J2EE Connector Architecture (J2EE-CA, or JCA)

Enterprise Java Beans (**EJB**) is a major part of the J2EE specification, and defines a model for building server-side, reusable components. There are three types of enterprise beans currently supported by J2EE—session beans, message-driven beans, and entity beans. Session beans are used for sharing information among clients of the application. Message-driven beans are used for asynchronous execution based on messages from message-oriented middleware. Entity beans are used to model persistent business objects particularly data in database.

The need to connect to databases in an object-oriented manner led to the popularity of JPA, which allows the programmer to seamlessly integrate any database with the application without worrying about the underlying database driver or query language details.

This chapter mostly discusses the different aspects of persistence, databases and reasons that led to the development and acceptance of JPA/ORM. This discussion is necessary in order to set the background for the rest of the book.

Structure

This chapter discusses the following:

- Java persistence mechanisms
- Types of databases

- Object-Relational definitions
  - Object-Relation Model
  - Object-Relational Mapping
  - Object-Relational Impedance Mismatch
- Java Persistence API
  - History of JPA
  - Other Java persistence specifications
  - Advantage of JPA/ORM
- Persistence products

Objectives

This chapter will bring out to the reader the Java persistence technologies, types of databases, and object-relational definitions, and its impact on program development. You will also learn about JPA and other persistence specifications in Java, and the JPA provider implementation comparison study.

Java Persistence

Persistence of data, in computer science, is describing data that outlives the process that created it. In Java, persistence is about storing any application-related data to any level and retrieving it when required. Java data includes strings, numbers, dates, byte arrays, images, XML, and Java objects.

There are many ways of data persistence that are commonly used in Java. A few of them are as follows:

File Input Output (**IO**): File IO is mainly used to persist data in file system of the underlying operating system as files. Mostly string data or primitive data is stored into files. All the file contents are read/written as sequence of data called **streams,** which consists of bytes. Java objects are stored into files using a technique called **serialization** which is explained as follows.

Serialization: Serialization is used to make the state of an object persistent. This means that the state of the object is converted to a stream of bytes and stored into a file. Whenever this is required, the reverse process (deserialization) is done and the object is recreated.

Java Database Connectivity (JDBC): JDBC is mainly used for connecting JEE applications to database. This is the basic underlying technology used by most of the Java applications to store and retrieve data from the **database** (**DB**).

Java Connector Architecture (**JCA**): Java Connector Architecture is a more generic standard for connecting to any **Enterprise Information System** (**EIS**). It can be a database or any other system like Siebel Systems, SAP AG, Oracle applications, and so on.

Java Data Objects (JDO): JDO is a general specification for Java object persistence. Applications that use JDO are independent of the underlying database and can be easily migrated from one type of database to another type of database. JDO implementations support many data stores like relational and object databases, XML, flat files, LDAP, ODF formats, MS-Excel formats, and so on. This was mainly part of the Sun specifications that was later donated to Apache foundation.

Service Data Objects (SDO): SDO is a specification to manage the processing of data in **service-oriented architecture** (**SOA**) from heterogeneous sources like XML documents, relational databases, web-services, and so on. This was originally developed in 2004 as a joint collaboration between Oracle (BEA) and IBM.

Enterprise Java Bean (**EJB**): EJB has many categories of beans out of which Entity beans are used for data persistence in Java applications. Two types of persistence are seen in Entity beans—**Bean Managed Persistence** (**BMP**) and **Container Managed Persistence** (**CMP**). BMP allows the entity bean to manage and implement all the operations directly with the help of persistence operations (ex: JDBC, JDO, SDO). However, as name suggests, with CMP, the EJB container implicitly manages the persistent state. The developer need not worry about the database access and related functions while using this kind of persistence. **Java Persistence API** (**JPA**) was defined as part of this kind of persistence.

Databases

In simple terms, a database is nothing but a program that stores data. This can be done in many ways and different kinds of data can be stored in these databases. Different types of databases are listed as follows:

Object databases: Object databases have their information represented as objects. Since Java is an object-oriented language, storing data and retrieving them as objects becomes a natural and common way to do. But these database management systems did not achieve much success when compared to relational databases.

XML databases: XML databases allow data to be specified, stored and retrieved in XML formats. The data can be queried, transformed, exported, and returned to another system in XML form. This is a kind of document-oriented database that is in turn a category of NoSQL databases.

NoSQL databases: NoSQL or non-SQL or non-relational database equips itself with storage and retrieval of data which is in a different format other than the tabular

relations. The data can be in any format like XML, JSON, etc. Additionally, NoSQL database does not require a structured schema definition for data storage.

Relational databases: Although there are many kinds of databases, relational databases remain to dominate the industry even though it is a relatively old technology. In this model, data is organized into tables of columns and rows. Each row is identified with a unique key. Rows are called **tuples** or **records,** whereas columns are called **attributes**. Each table represents an entity, whereas each row in the table is a particular instance of that entity, and columns describe the attributes of that particular instance. There had been many attempts to replace relational model first with object-oriented, then object-relational and finally XML databases, none of these models achieved much success and relational databases still remain overwhelmingly the dominant database model.

The data in the application is based on Java objects, whereas the data in the database is based on table-row-column relational model. **Java Persistence API** (**JPA**) is an attempt to help developers conquer this challenge with ease.

Object-Relational Definitions

With the growth of object-oriented languages for application development, programmers started thinking of mapping relational and object-oriented models for the ease and clarity of implementation and hence, **Object-Relational Model** as well as **Object-Relational Mapping** became frequently used terminologies. Also, the mismatch between these two models also posed a few limitations while using them. The following sections describe about the model, the mapping, and the mismatch.

Object-Relational Model

Object-Relational Model is a design that provides object-oriented features integrated to a relational database. This allows the developer to integrate their object-oriented code with datatypes and methods to the database with ease. The Nordic Object/Relational Database Design[1] is an example proposal for this kind of database. The objective of this kind of database is more power, greater flexibility, better performance, and greater data integrity than its predecessors (either relational or object-oriented as this is a hybrid technology).

Object-Relational Mapping

Object-Relational Mapping (**ORM**) is a programming technique that helps in converting data from different incompatible type systems to objects for the ease of manipulation by object-oriented languages. This reduces the burden of the developer to convert existing object values to simpler forms to be stored into the database.

With relational databases being used widely, table-row-column model makes the data manipulation tedious for the programmer. ORM is a boon, as the coding is also reduced and the programmer can increase productivity by applying the logic in an object-oriented way without worrying about the actual row-column format of the relational database.

In this text, the mapping technique will be discussed in further detail as this know-how will make the understanding of JPA evident.

Object-Relational Impedance Mismatch

An Object-Relational Impedance Mismatch[2] relates to the problems that occur while representing data from relational databases in object-oriented languages. This term is used as an analogy from electrical engineering, where impedance matching is important in any optimal circuit design to have maximum power flow[3]. Here, the impedance mismatch causes a difficulty in the flow of data in the application when accessing a relational database by an object-oriented language.

Basically, object-oriented concepts are based on software engineering fundamentals, whereas relational-model concepts are based on proven principles of mathematics. These theoretical differences form a non-ideal combination with a few hitches.

Conceptual differences

Many concepts in **object-oriented** (**OO**) are totally against the concepts in relational model. Encapsulation allows the object's internal representation to be kept hidden, whereas Object-Relational Model exposes the underlying content of an object to interact with an interface not specified by the object implementation itself.

Access specifiers in relational model is not as important as in object-oriented. Similarly, classifications and characteristics of the various attributes in these two models conflict in their relative as well as absolute natures.

Classes, inheritance, and polymorphism are not supported by relational model, whereas these are the basic principles of object-oriented design. Interfaces are the only access to the internals of an object in object-oriented paradigm. But, in relational world, views provide alternating perspectives.

Data type differences

The relational model has all the rows as immutable, which means it strictly prohibits by-reference attributes (only exception is in case of foreign keys where cross-reference columns are used). But in OO, it expects by-reference to be the standard behavior. Any attribute can have a reference to yet another object at any level.

Also, the types of scalar data differ between database and OO languages. For ex., VARCHAR (10) is a way to describe a string in a database column definition which strictly restricts the string size to 10. But in case of a string in Java, no length is mentioned and hence, this restriction in the column size goes unnoticed in the application code, if additional effort is not taken by the developer.

Another simple yet unnatural difference is the way data in the database system is treated by the querying mechanisms (SQL vs OO language). For example, SQL systems mostly ignore the trailing white spaces while comparing values, whereas OO language string comparison does not do that.

Data handling differences

Relational model has well-defined and relatively small set of primitive operations for the query and usage of data. In OO languages, data querying and manipulation are done via custom-built imperative operations that may result in changing the state of the entity itself.

Relational design is mainly based on unique tuples (or rows) that provide set-based operations, whereas OO languages deal with list-based/map-based operations making the data handling dissimilar.

Data Composition Differences

In OO, objects can be formed from many objects to a high degree or can be a specialization of a generic definition. Relations, however, are tuples (rows) with the same headers (column names), and do not have an ideal counterpart in OO languages.

Transactional differences

Transaction is the smallest unit of work performed by a database. Relational transactions are much larger than any other operation performed by classes in OO languages. Transactions can have multiple sets of data manipulations, whereas in OO languages they are constrained to individual assignments to primitive-typed fields. Isolation, durability, atomicity, and consistency are important to transactional data, whereas this is ensured only to the primitive-typed fields in case of OO transactions.

Integrity differences

The relational model requires declarative constraints on the scalar types, attributes, relational variables, and databases as a whole. In case of OO languages, no such constraints are explicitly declared but exceptions are raised while trying to operate on an encapsulated internal data.

Solution for Mismatch

When different systems are employed, mismatch is bound to happen. So, the first step would be to recognize the problem and either minimize or compensate it. Frameworks that use the minimization technique at runtime dynamically correlate a row in the dataset to an entity instance. While coding statically, the entity class is correlated to the relation in the dataset. This helps in achieving a few advantages like:

- smaller code size and faster compilation
- schema can be changed dynamically with changing the correlation in the code
- domain data can be accessed at higher levels in a straight-forward manner for presentation, transportation, as well as validation
- complex mapping avoidance
- constraint checking can be done

And also, a few disadvantages such as:

- possible performance cost of runtime construction and access
- cannot make use of OO principle like polymorphism

Another way is to use non-relational databases as this impedance mismatch, as the name suggests, occurs only when using with relational models. NoSQL databases or XML databases can be used as alternate databases to relational ones. However, migrating an existing relational database to a non-relational database is also a mammoth task.

Java Persistence API (JPA)

Java provides specifications to make sure that all who uses this specification feature should understand what will be available for the user. However, these specifications don't limit the implementer and give freedom on how this feature can be achieved. JPA is one such specification that is used for accessing, persisting, and managing data between Java objects/classes and a relational database. JPA was defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification.

JPA became so popular that it is now considered the standard industry approach for **Object to Relational Mapping** (**ORM**) in the Java Industry. Note that it is a set of definitions, or in java terms, a set of interfaces. It requires an implementation to perform actual persistence or data management. There are many JPA implementations, both

open-source as well as commercial, and any JEE application server should support its use since these implementations adhere to the Java specification.

Layers in JPA based Java app

The figure given below (*Figure 1.1*) depicts the layers that clearly indicate how JPA accomplishes the task of preparing the application to be independent of the database or the provider. This figure also provides a list of JPA providers as well as JDBC drivers available in the market.

The main attraction of JPA is that it empowers **Plain Old Java Objects** (**POJO**) to be easily persisted without any extra implementations or methods when compared to EJB 2 CMP. This allows an object to be mapped to a database table via Object-Relational Mappings described using standard annotations or XML configurations.

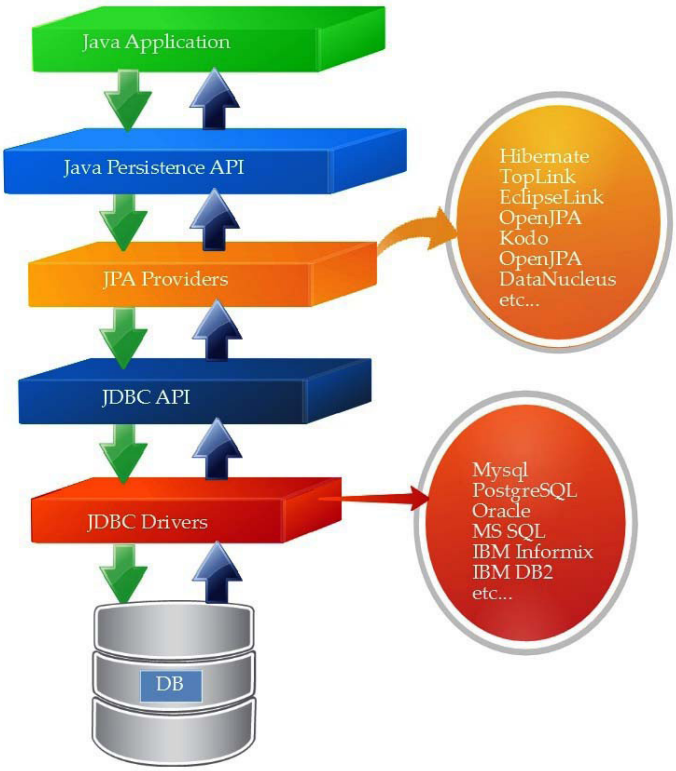

***Figure 1.1**: Various layers in a java application that uses JPA.*

A runtime entity manager API is specified by JPA that takes care of processing queries and transaction on the objects. JPA defines an object-level query language JPQL to query data from the database in form of objects.

History of JPA

JPA is the latest of several Java persistence specifications.

The first Java persistence specification was the **Object Management Group** (**OMG**) persistence service Java binding, which did not become successful. There are rarely any commercial products supporting this specification.

The next one was *EJB 1.0 CMP Entity Beans*, which was very successful in being adopted by the big Java EE providers (BEA, IBM), but the response against the specification requirements for Entity Beans was that they were overly complex and with a poor performance.

EJB 2.0 CMP tried to reduce some of the complexity of Entity Beans by local interfaces, but the majority of the complexity remained. EJB 2.0 also lacked portability as the deployment descriptors defined for the mapping were all proprietary. This influenced in creating another Java persistence specification, **Java Data Objects** (**JDO**).

JDO was adopted by several independent vendors such as Kodo JDO, and several open-source implementations, but never had much success with the big Java EE vendors.

Although, by now there were two competing Java persistence standards (EJB CMP and JDO) in the market, the majority of the users still preferred to continue using proprietary API solutions like TopLink, Hibernate, and so on. The TopLink product was acquired by Oracle from WebGain, booming its usage on the Java EE community.

EJB CMP complexity led to the EJB 3.0 specification with the main goal of reducing complexity and hence JPA was defined. This specification was meant to unify the EJB 2.0 CMP, JDO, Hibernate, and TopLink APIs, and products and have been successful in doing so.

Most of the persistence vendors have released JPA implementations confirming to its adoption by the industry and the users. Following are a few examples:

1. Hibernate[4] (acquired by JBoss which was later acquired by Red Hat)
2. TopLink[5] (acquired by Oracle)
3. EclipseLink[6] (Eclipse-based implementation of TopLink)
4. KodoJDO[7] (acquired by BEA which was later acquired by Oracle)
5. Cocobase[8] (owned by Thought Inc.)
6. Java Persistence Objects – JPOX[9] (available on sourceforge.net)
7. DataNucleus[10] (available on GitHub)

Even though JPA is considered as a widely accepted specification, there are many other persistence specifications as well that are being used. The next section describes them and their latest versions available in the market.

Other Persistence Specs

There are many specifications related to persistence in Java. They offer a wide variety of features and are uniquely different in many aspects.

Specification	Latest version	Year of last release
Java Database Connectivity (JDBC)[11]	4.3	2017
Java Data Objects (JDO)	3.0	2010
Java Persistence API (JPA)	2.1	2013
Java EE Connector Architecture (JCA)	1.6	2013
Service Data Objects (SDO)	2.1	2009
Enterprise Java Bean-Container Managed Persistence (EJB CMP)	3.0	2009

***Table 1.1**: Latest versions of persistence specifications available in Java*

Many of these specifications are used based on the requirements while developing the application. However, JPA specifications conform to a wide variety of application requirements and hence, become suitable for a large suite of applications.

Advantages of JPA/ORM

This is an intriguing question. There are many reasons to use an ORM framework or persistence product, and many reasons to use JPA in particular.

- Since the application development is in object-oriented language, ORM gives a seamless integration with the database without even knowing about the actual SQL queries used.
- JPA also avoids the basic nitty-gritty details on how to connect to a database while using other persistence specifications like JDBC.
- ORM makes the application database and schema independent as they are all managed by the ORM layer and not by the application itself.

- Most of the ORM/JPA products are open source and free. Moreover, many corporations provide support and service for these products. Hence, the usage of ORM/JPA becomes more luring.
- ORM also provides features like caching, complex database and query optimizations that are considered high-end performance specialties while accessing data.
- ORM leverages object-oriented programming and object model usage, whereas JPA is a standard and a part of EJB3 and JEE with a very usable and functional specification.
- JPA supports both standard and enterprise editions of Java (JSE & JEE) and is portable across application servers and persistence products (avoids vendor lock-in).

JPA needs an implementation to be usable while developing an application. There are many products available in the market that create confusion among the developers. The next section tries to enable the user to ask the right questions and seek the answers while selecting a provider from those available in the market.

Persistence Products

Although there are many persistence products in the market, the recommendation is to choose a product based on a widely used standard specification like JPA, which gives more flexibility to switch between different persistence providers or between different server platforms.

Apart from JPA support, there are a wide variety of factors to be considered while choosing a persistence product for an application. There are many products that support JPA, which makes it even more complicated. A few aspects to be taken care of and the questions to be answered are listed as follows:

- Whenever an application is in its initial phases, the deployment details are also planned and agreed upon. It is very important to know the types of persistence products that are supported or come integrated with the server/database platforms. Only then going further, the persistence layer of the application can be designed with ease.
    - Which products are supported by the chosen server platform?
    - Are there any which come integrated with the server?
    - Does the product integrate with the database platform?
- The cost of the persistence product is also another important decision factor. There are many components to be thought about in this tier.

  - Whether the product is free and open source? If so, can enterprise level be supported and services be purchased?
  - Is there any existing relationship with the company producing this product?

- A product having a large active user base is always a judicious choice. This helps in making sure that most of the features are provided as many have been using it actively. Also, in case of free and open-source products, the community would be very engaged and prompt.
  - Is the product being used actively?
  - Does the product enjoy a large user base?
  - Does the product have active and open forums? Do the questions posted in these forums receive useful responses?

- Once the product is picked, it is really hard to consider the change in product as that involves time and cost. So, it is important to evaluate the product's performance abilities before using it.
  - How does the product perform and scale?
  - Is the product JPA compliant?
  - Are there any other functionalities that the product offers apart from the JPA specification?

Product	JPA 1.0	JPA 2.0	JDO 2.0	JDO 3.0	CMP 2.1	Version	Year of release	Open source	App servers
Hibernate (Red Hat)	Yes	Yes				5.6	2021	Yes	JBoss[12]
EclipseLink (Eclipse)	Yes	Yes				3.0.2	2021	Yes	Oracle Weblogic (12c)[13], Glassfish (v3)[14]
TopLink (Oracle)	Yes	Yes			Yes	12c (12.2.1.4.0)	2019		Oracle Weblogic (12c), Oracle AS (10.1.3)[15]

Product	JPA 1.0	JPA 2.0	JDO 2.0	JDO 3.0	CMP 2.1	Version	Year of release	Open source	App servers
OpenJPA (Apache)	Yes	Yes				3.1.2	2020	Yes	Geronimo[16], Websphere Application Server (8.0)[17]
DataNucleus (DataNucleus)	Yes	Yes	Yes	Yes		5.2.1	2019	Yes	Apache Isis[18]
TopLink Essentials (java.net)	Yes					2.0	2007		Glassfish (v2), OracleAS (10.1.3)
Kodo (Oracle)	Yes		Yes			4.1.4	2007		Oracle Weblogic (10.3)

***Table 1.2**: A summary of various JPA compliant persistence products that are available in market.*

The most advanced, well-documented, open-source and highly active community that generates new features fast designates Hibernate among the top most used JPA providers. EclipseLink is the Oracle's version of JPA that comes from the official Java community and is designated as the reference implementation of JPA.

Conclusion

This chapter discussed about the concept of persistence in general by explaining different persistence methodologies and kinds of databases, and in particular about Java persistence by examining various Java technologies used. This chapter also introduced the technical concepts of JPA and ORM by elaborating its distinct advantages and disadvantages.

Through this chapter, the developer can understand the relevant points to be considered while evaluating a persistence product for an application. Also, a summary of diverse features supported by available JPA products in the market has been provided. This can help to decide on the JPA product as well as the server platform required for an application.

In the next chapter, the basic structure of relational databases–tables, will be explored.

Abbreviations

API: Application Programming Interface (defines interactions between multiple software intermediaries)

SQL: Structured Query Language (domain-specific language mostly used for data management in relational databases)

XML: Extensible Markup Language (file format that is used for storage and transport of data)

LDAP: Lightweight Directory Access Protocol (application protocol for accessing and maintaining distributed directory information services over Internet Protocol networks)

IBM: International Business Machines (American multinational IT and consulting firm)

BEA: the name of an American software company that is an acronym of the names of its three founders: *Bill Coleman*, *Ed Scott* and *Alfred Chuang*.

SAP AG: Systems, Applications, and Products (name of a German company in software systems)

Multiple Choice Questions

1. **Java language features include:**
    a) object oriented
    b) platform independent
    c) both a & b
    d) none of these

2. **JPA stands for:**
    a) Java Persistence Appendix
    b) Java Persistence Application Programming Interface
    c) Java Performance Applet
    d) none of these

3. **ORM stands for:**
    a) Object Relationship Mapping
    b) Object Relational Model

c) both a & b
d) none of these

4. The first Java persistence specification is:

a) OMG
b) JPA
c) CORBA
d) Entity beans

5. JPA was formed as part of:

a) JDBC
b) EJB 1.0 CMP
c) EJB 2.0 CMP
d) EJB 3.0 CMP

6. Serialization is the process of converting:

a) object to character stream
b) character stream to object
c) object to byte stream
d) byte stream to object

7. EJB consists of:

a) message-driven beans
b) session beans
c) entity beans
d) all of these

8. Relational databases store data in form of:

a) tuples
b) tables
c) objects
d) byte streams

Answers

1. **c**
2. **b**
3. **c**
4. **a**
5. **d**
6. **c**
7. **d**
8. **b**

Questions

1. What is persistence in computing terms?
2. Explain the various persistence mechanisms in Java.
3. Describe types of databases available. Find out examples for each type.
4. Elaborate on Object-Relational Impedance Mismatch.
5. Illustrate with a diagram how JPA is effective in making the application independent of the underlying provider as well as the DB.

Key Terms

- **JPA**: Java specification for persisting data and managing data using ORM techniques.
- **JPA Provider**: Actual implementation of the JPA definitions.
- **ORM**: Object-Relationship Mapping to avoid the complexity of relational tables in object-oriented programs and treat the data as objects themselves.

Points to Ponder

1. One persistence tool that got wide acceptance is *Apache iBATIS18*. This is neither based on ORM or JPA, but it relatively reduces the programmer's effort to convert the object data to SQL queries and vice versa. The mapping of SQL databases and Java objects are done in XML files and decoupled from the application code hence making the code relatively simpler to understand and maintain.

2. Ideally, JPA supports relational databases. However, now there are a few provider implementations of JPA that have extended their support to NoSQL databases as well. Some of them are *DataNucleus JPA, Hibernate* and *EclipseLink*. JDO is the specification that applies to both RDBMS and non-RDBMS.

References

1. Nordic Object/Relational Model: Article by Paul Nielsen **https://docs.microsoft.com/en-us/previous-versions/bb245675(v=msdn.10)?redirectedfrom=MSDN**
2. **https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch**
3. Impedance mismatch: Article by techopedia **https://www.techopedia.com/definition/32462/impedance-mismatch**
4. **http://hibernate.org/**
5. **https://www.oracle.com/technetwork/middleware/toplink/overview/index-086944.html**
6. **https://www.eclipse.org/eclipselink/**
7. **https://docs.oracle.com/cd/E13189_01/kodo/docs40/full/html/solarmetric_intro.html**
8. **http://thoughtinc.com/cber_index.html**
9. **https://sourceforge.net/projects/jpox/**
10. **http://www.datanucleus.org/**
11. Companies that support JDBC in their products: **https://www.oracle.com/technetwork/java/index-136695.html#**
12. **https://docs.jboss.org/jbossas/docs/Installation_Guide/4/html-single/index.html**
13. **https://www.oracle.com/middleware/technologies/weblogic.html**
14. **https://javaee.github.io/glassfish/**
15. **https://en.wikipedia.org/wiki/Oracle_Application_Server**
16. **http://geronimo.apache.org/**
17. **https://www.ibm.com/cloud/websphere-application-platform/**
18. **https://isis.apache.org/**
19. **https://en.wikipedia.org/wiki/Apache_iBATIS**

Chapter 2
Tables–Attributes and Embeddable Objects

Introduction

With the detailed descriptions on the types of databases in the previous chapter, it is clear that even though there are various types of databases, **Relational Database Management System** (**RDBMS**) is the most widely used database for application development[3].

Before the invent of RDBMS, flat files were used as data stores and it was very difficult to retrieve the contents selectively. However, the paper, *A Relational Model of Data for Large Shared Data Banks* written in June 1970 published by *Dr. Edgar Frank Codd* in **Association of Computer Machinery** (**ACM**) journal, created the concept of RDBMS based on the relational model.

By mid-1980s, *Dr. Codd* defined 13 rules (called Codd's 12 commandments) and numbered them as Rule-0 to Rule-12[4]. However, by then, there were a few DBMSs already available as "RDBMS" which did not completely adhere to the all these rules. These databases are referred to as **Pseudo-RDBMS** (**PRDBMS**), and those that totally adhere to all the rules are referred to as **Truly-RDBMS** (**TRDBMS**).

A relation is a mathematical concept based on the idea of sets where data is structured in grid-like mathematical structures consisting of columns and rows. Most of the mathematical operations in sets are also used in relations like join, intersection, union, etc.

In this chapter, tables and their mapping to objects are the major topics explained.

Structure

This chapter discusses the following with relevant examples in each section:

- Mapping access types
- Basic attributes
- Temporal, dates, times, timestamps and calendars
- Enums
- LOBs, BLOBs, CLOBs and serialization
- Column definition and schema generation
- Insertable, updateable, read only
- Conversion, types of custom
- Embeddables – Sharing, embedded IDs, nesting and querying
- Advanced mapping for multiple tables with foreign keys, joins, outer joins as well as tables with special characters

Objectives

This chapter will bring out to the reader RDBMS concept of tables, JPA mapping of object to a table, and their access modes. You will learn about the different types of table attributes and also about advanced mapping.

Tables

A table is the basic persist structure of a relational database. A table contains a list of columns or attributes, which define the table's structure, and a list of rows or tuples that define the table's data. Each column has a specific type called **domain** and generally a specific size. The following figure shows the terminologies that may be used interchangeably throughout this text. It is very important to note these terms, and hence highlighted in a picture.

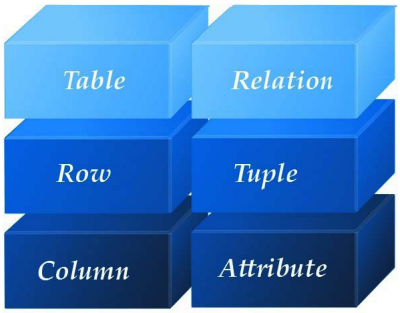

***Figure 2.1**: Terms that can be used vice-versa*

A table is a collection of values stored as rows and columns. As shown in *figure 2.2*, **`Student`** table contains columns named **`ID`**, **`Registration Number`**, and **`Name`** that are also known as **attributes**. A set of values for columns forms a row in the table that is known as **tuple** or **record**. If the values {ID=1, REGNO=11111, Name=ABC} are inserted into the **`Student`** table, then it can be considered as a tuple / record / row in the table.

ID	REGNO	NAME

***Figure 2.2**: Student table – an example*

A relation is formed as a table with various combinations of attribute values forming tuples. The combination of attributes or columns that make the tuple identifiable is called a **key**. There are mainly eight types of keys in DBMS. They are as follows:

1. **Super Key**: a group of single or multiple keys that identify rows in a table. A super key may have additional attributes that are not needed for unique identification. Consider the sample Student table given above in *figure 2.2*. Here, the student ID as well as registration number can be considered as unique keys. Hence, the super keys are **`{ID}`**, **`{REGNO}`**, **`{ID, NAME}`**, **`{ID, REGNO}`**, **`{REGNO, NAME}`**, and **`{ID, REGNO, NAME}`**

2. **Primary Key**: a column or group of columns in a table that uniquely identify every row in that table. The primary key cannot be a duplicate, meaning the same value can't appear more than once in the table. A table cannot have more than one primary key. As per the table above, either **`{ID}`** or **`{REGNO}`** can be considered as the primary key but not both.

3. **Alternate Key**: a column or group of columns in a table that uniquely identify every row in that table other than the primary key. A table can have multiple choices for a primary key but only one can be set as the primary key. All the keys that are not primary keys are called alternate keys. If the **`{ID}`** has been

declared as primary key in the table above, then **{REGNO}** can be considered as the alternate key, and vice-versa.

4. **Candidate Key**: a set of attributes that uniquely identify tuples in a table. Every table must have at least a single candidate key. A table can have multiple candidate keys, but only a single primary key. Primary key and alternate key together form the set of candidate key. Here they are **{ID}** and **{REGNO}**.
5. **Compound Key**: two or more attributes (mostly primary keys from other tables) that uniquely recognizes a specific record. It is possible that each column may not be unique by itself within the database.
6. **Composite Key**: a combination of two or more columns that uniquely identify rows in a table. The difference between compound and composite key is that any part of the compound key can be a foreign key, but the composite key may or may not be a part of the foreign key.
7. **Foreign Key**: a column that creates a relationship between two tables. The purpose of foreign keys is to maintain data integrity and allow navigation between two different tables. It acts as a cross-reference between two tables as it references the primary key of another table.
8. **Surrogate Key**: an artificial key that aims to uniquely identify each record is called a surrogate key. They do not lend any meaning to the data in the table. Mostly auto generated keys in DB are surrogate keys. They have no relation with the data being inserted.

The standard set of domains or attribute types are limited to basic types including numeric, character, date-time, and binary (although most modern databases have additional types and typing systems). A deep discussion on the various attribute types is done later in this chapter.

Tables can have constraints that define the rules which restrict the row data, such as primary key, foreign key, and unique constraints. Tables also have other artifacts such as indexes, partitions, and triggers. While performing the mapping of tables to objects, all the preceding factors have to be taken into consideration. Hence, mapping becomes an important part of JPA.

Mapping

The first thing to persist something in Java is to define how it is to be persisted. This is called the **mapping process**. Many object-relational mapping tools could generate an object model for a data model that included the mapping and persistence logic in it. ORM products also provided mapping tools to allow the mapping of an existing

object model to an existing data model and stored this mapping meta-data in flat files, database tables, XML, and finally annotations.

In JPA, mappings can either be stored through Java annotations, or in XML files. One significant aspect of JPA is that only the minimal amount of mapping is required. JPA implementations are required to provide defaults for almost all aspects of mapping an object. The minimum requirement to map an object in JPA is to define which objects can be persisted. This is done through either marking the class with the **`@Entity`** annotation, or adding an **`<entity>`** tag for the class in the persistence unit's ORM XML file. Also, the primary key, or unique identifier attribute(s), must be defined for the class. This is done through marking one of the class' fields or properties (get method) with the **`@Id`** annotation, or adding an **`<id>`** tag for the class' attribute in the ORM XML3 file.

The JPA implementation will default all other mapping information, including defaulting the table name, column names for all defined fields or properties, cardinality, and mapping of relationships, all SQL and persistence logic for accessing the objects. Most JPA implementations also provide the option of generating the database tables at runtime, so very little work is required by the developer to rapidly develop a persistent JPA application.

Entity class

The entity class must be annotated with the **`Entity`** annotation or denoted in the XML descriptor as an entity. A few properties of entity class[1] are as follows:

- The entity class must have a no-argument constructor which must be public or protected. The entity class may have other constructors as well.
- The entity class must be a top-level class. An enum or interface must not be designated as an entity.
- The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
- If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the serializable interface. Note that Serializable interface allows to persist the state of an object instance.
- Entities support inheritance, polymorphic associations, and polymorphic queries.
- Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes; and non-entity classes may extend entity classes.

- The persistent state of an entity is represented by instance variables. The state of the entity is available to clients only through the entity's methods—i.e., accessor methods (getter/setter methods) or other business methods. This holds up the object-oriented principle of encapsulation.

An example, **`Student`** table, is given in the *figure 2.3* with **`ID`** as its primary key.

ID	NAME	ADDRESS

***Figure 2.3**: Student table – another example*

The following code snippet provides a sample **`Entity`** class which maps the previously shown **`Student`** table:

```
@Entity
public class Student implements Serializable {
    @Id
    private Long id;
    private String name;

    public Student() {}
    public Long getId() {return id; }
    public void setId(Long id) {this.id = id;}
    public String getName() { return name; }
    public void setName(String name) {this.name = name;}
}
```

***Code 2.1**: Java mapping using JPA annotations for the Student table*

```
<?xml version="1.0"encoding="UTF-8"?>
<entity-mappings version="1.0">
<description>My First JPA XML Application</description>
<package>entity</package>
<entity class="entity.Student" name="Student">
<table name="STUDENT"/>
<attributes>
<id name="Id">
<generated-value strategy="TABLE"/>
</id>
```

```
<basic name="Name">
<column name="Name" length="50"/>
</basic>
<basic name="Address">
    <column name="Address" length="100"/>
</basic>
</attributes>
</entity>
</entity-mappings>
```

***Code 2.2**: Java mapping using JPA XML for the Student table*

The preceding figure shows another way of JPA mappings based on XML files. This was used before the annotations were supported in Java. However, now, annotations are used more than XML since it is easier to maintain and all the mappings and code are available in the same file. Otherwise, there would be multiple files, one for the XML mapping and other for the actual logic implementation which would be in Java.

Access modes

The mapping information of an entity should be available to the ORM provider at runtime, so that when the reading/writing of data happens, the provider must be able to map data into/from the entity instance. This mapping information is defined by the access type of the class. Normally, the class values are loaded/stored either through reflection, or through generated byte code, but depends on the JPA provider and configuration. There are two modes of access available in JPA:

Field Access Mode:

There are two ways to declare field access mode.

1. Explicitly annotate the entity with **`@Access(AccessType.FIELD)`**

```
@Entity
@Access(AccessType.FIELD)
public class Student implements Serializable {
private Long id;
      . . .
}
```

***Code 2.3**: Entity level annotation for declaring field access type*

2. Annotating the fields of the entity will cause the provider to use field access to get and set the state of the entity.

```
@Entity
public class Student implements Serializable {
    @Id
    private Long id;
    . . .
}
```

***Code 2.4**: Field level annotation for declaring access type*

All fields must be declared as either protected, package, or private. Public fields are disallowed because it would open up the state fields to access by any unprotected class in the JVM. This access type is normally safer, as it avoids any unwanted side-affect code that may occur in the application get/set methods.

Property Access Mode: There are two ways to declare property access mode as well.

1. Explicitly annotate the entity with **`@Access(AccessType.PROPERTY)`**

```
@Entity
@Access(AccessType.PROPERTY)
public class Student implements Serializable {
      private Long id;
      . . .
```

***Code 2.5**: Entity level annotation for declaring property access type*

2. Annotate the getter methods of the entity will cause the provider to use property access to get and set the state of the entity.

```
@Entity
public class Student implements Serializable {
      private Long id;
      @Id
      public Long getId() {return id;}
      public void setId(Long id) {this.id = id;}
```

***Code 2.6:** Method level annotation for declaring access type*

When property access mode is used there must be getter and setter methods for the persistent properties. The type of property is determined by the return type of the getter method and must be the same as the type of the single parameter passed into the setter method. Both methods must be either public or protected visibility. The mapping annotations for a property must be on the getter method. This access type has the advantage of allowing the application to perform conversion of the database value when storing it in the object. However, the user should be careful to not put any side-affects in the get/set methods that could interfere with persistence.

JPA does not define a default access type, so the default may depend on the JPA provider. The default is assumed to occur based on where the @Id annotation is placed, if placed on a field, then the class is using FIELD access, if placed on a get method, then the class is using PROPERTY access. The access type can also be defined through XML on the <entity> element using access XML attribute.

```
<?xml_version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0">
<description>My First JPA XML Application</description>
<package>entity</package>
<entity class="entity.Student" name="Student" access=”FIELD”>
<id name="Id">
<generated-value strategy="TABLE"/>
</id>
<basic name="Name">
<column name="Name" length="50"/>
</basic>
<basic name="Address">
```

Code 2.7: *XML based access declaration. Note the highlighted field is used to declare access type*

JPA allows to use mixed mode access i.e. class to use one default access mechanism, but for one attribute to use a different access type. This can be used for attributes that need to be converted through a set of database specific get, set, or allow a specific attribute to avoid side-affects in its get, set methods. This is done through specifying the @AccessType annotation accordingly on the mapped attribute.

```
@Entity
@Access(AccessType.FIELD)
public class Student implements Serializable {
private Long id;
private String name;
```

```
public Student() {}

public Long getId() { return id; }
public void setId(Long id) { this.id = id; }

@Access(AccessType.PROPERTY)
public String getName() { return "Hi " + name;       }
public void setName(String name) { this.name = name; }
}
```

***Code 2.8**: Example where mixed access mode is being used*

In the preceding example, the entity as a whole is defined to use FIELD access type whereas the attribute name is defined to use PROPERTY access type. Note that the name in the DB is not as such stored in the object instead is prefixed with a "Hi". This is just an example to show how the DB values can be tailored and stored in the object and vice-versa.

Attributes

A basic attribute is one where the attribute class is a simple type such as String, Number, Date or a primitive. A basic attribute's value can map directly to the column value in the database. The following table summarizes the basic types and the database types they map to.

Java Type	Database Type
String (char, char[])	VARCHAR (CHAR, VARCHAR2, CLOB, TEXT)
Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte)	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
int, long, float, double, short, byte	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
byte[]	VARBINARY (BINARY, BLOB)
boolean(Boolean)	BOOLEAN (BIT, SMALLINT, INT, NUMBER)
java.util.Date	TIMESTAMP (DATE, DATETIME)
java.sql.Date	DATE (TIMESTAMP, DATETIME)
java.sql.Time	TIME (TIMESTAMP, DATETIME)

java.sql.Timestamp	TIMESTAMP (DATETIME, DATE)
java.util.Calendar	TIMESTAMP (DATETIME, DATE)
java.lang.Enum	NUMERIC (VARCHAR, CHAR)
java.util.Serializable	VARBINARY (BINARY, BLOB)

***Table 2.1**: Basic attribute mapping java type to database type*

Technically, a basic attribute is mapped through the **@Basic** annotation or the **<basic>** element. The types and conversions supported depend on the JPA implementation and database platform. However, the commonly used way to map a basic attribute in JPA is to do nothing. Any attributes that have no other annotations and do not reference other entities will be automatically mapped as basic, and even serialized if not a basic type.

@Column annotation (or **<column>** element) is used to specify the column name for a basic attribute. The column annotation also allows for other information to be specified such as the database type, size, and some constraints. If the annotation is not provided, the column name is same as the attribute name, as uppercase.

All the non-persistent fields should be marked as **@Transient**(or **<transient>** element). This specifies the JPA provider not to map this field to any of the DB columns.

Date and time attributes

Date, time, timestamps are attributes that are available both in Java as well as in databases. At times, they are simple and the basic mapping can be used, but at certain instances, they become quite complex.

java.sql.Date can be directly mapped to **DATE** field in DB and **java.sql.Time** can be mapped to **TIME**. Similarly, **java.sql.Timestamp** can be mapped to **TIMESTAMP** in DB. However, if **java.util.Date** or **java.util.Calendar** needs to be mapped to **DATE** or **TIME** or TIMESTAMP, the JPA provider must be given an indication to perform certain sorts of conversions. **@Temporal** annotation (**<temporal>** element) is used to achieve this mapping.

```
@Entity
public class Student implements Serializable {
	@Id
	private Long id;

	@Transient
	private String nickname;
```

```
        @Column (name=”S_NAME”)
        private String name;

        @Basic
        @Temporal (DATE)
        private java.util.Date dob;
}
```

***Code 2.9**: Usage of @Column, @Transient and @Temporal*

```
<?xml version="1.0"encoding="UTF-8"?>
<entity-mappings version="1.0">
        <description>My First JPA XML Application</description>
        <package>entity</package>
        <entity class="entity.Student" name="Student">
                <attributes>
                        <id  name="Id"><generated-value  strategy="TABLE"/></
                id><transient name="nickName"/>
                        <basic name="Name">
                                <column name="S_NAME" length="50"/>
                        </basic>
                        <basic name="dob">
                                <temporal>DATE</temporal>
                        </basic>
                </attributes>
        </entity>
</entity-mappings>
```

***Code 2.10**: XML equivalent of @Transient, @Column and @Temporal respectively*

The example in *figure 2.11* shows the way of using **`@Transient`**, **`@Column`** and **`@Temporal`** annotations, while *figure 2.12* shows how the same configurations can be achieved using XML files.

Milliseconds: The precision of milliseconds vary based on temporal classes, database types, and on different databases. Many classes (**`java.util.Date`** and **`java.util.Calendar`**) support milliseconds, whereas some do not support them at all (**`java.sql.Date`** and **`java.sql.Time`**). However, there are classes that even support nanoseconds (**`java.sql.Timestamp`**).

Similarly, the support for milliseconds in databases also alter in a wide manner. Oracle prior to version 9 only had a DATE that supported a date and time without milliseconds. As Oracle 9 was released, it had TIMESTAMP that supported till nanoseconds. MySQL has DATE, TIME, and DATETIME types. DB2 has DATE, TIME, and TIMESTAMP (supports only till milliseconds). There are even databases where the milliseconds stored have precision problems, and it seems to store only an estimate of the milliseconds value.

Hence, it is always recommended to not use timestamp fields as primary key or for version locking as there can be compatibility issues while converting from DB to Java and vice-versa.

Timezones: Earlier versions of Java only supported timezones in **`java.util.Calendar`**. However, Java 8 has a set of Date / Time API (**`java.time.*`**), which makes the timezone handling simpler. Although some databases support timezones, most of the databases do not store the timezone as value. For ex: Oracle has two kinds of timestamps with timezones **`TIMESTAMPZ`** (timezone is stored) and **`TIMESTAMPLTZ`** (local timezone is used). JPA providers like TopLink and EclipseLink have extended support for the previously-mentioned different kinds of timestamps by using the annotation **`@TypeConverter`**. An example usage of the annotation is as follows:

```
@Entity
public class Student implements Serializable {

   @TypeConverter (name=”ageToYearOfBirth”, dataType=Integer.class,
   objectType=Date.class)
   @Column (name=”S_AGE”)

   @Convert (“ageToYearOfBirth”)
   private Date birthYear;
}
```

***Code 2.11**: Usage of @TypeConverter and @Convert*

The example shows how the age of a student stored in the **`S_AGE`** column of the **`STUDENT`** table in DB can be converted to the birth year of that student.

The preceding annotations can also be configured using XML files as follows:

```
<?xml version="1.0"encoding="UTF-8"?>
<entity-mappings version="1.0">
      <description>My First JPA XML Application</description>
      <package>entity</package>
      <type-converter   name=”ageToYear”   data-type=”Integer”   object-
type=”Date”/>
```

```
        <entity class="entity.Student"name="Student">
              <attributes>
                    <basic name="birthYear">
                          <column name="S_AGE" length="5"/>
                          <convert name=”ageToYear” />
                    </basic>
              </attributes>
        </entity>
</entity-mappings>
```

***Code 2.12**: XML equivalent usage for @TypeConverter and @Convert*

The type-converter is used typically to convert it from one data type to another in a few JPA implementations. Rather, it is a specific annotation for the more generic annotation **`@Converter`** (**`<converter>`** tag).

Enumerations

Enumerations are used as a list of constant values that can be used in an object model. For example, a student object may have gender as an enumeration type Gender (MALE, FEMALE). While converting this to database terms, it can be mapped to values like MALE-0 and FEMALE-1, respectively. However, this becomes difficult to manipulate as the object expects a String, whereas the values in database are integers. Hence, annotations are provided for storing/retrieving the values as String. The examples for both annotation as well as XML are as follows:

```
public enum Gender {
      MALE,FEMALE
}

@Entity
public class Student {

  @Enumerated (EnumType.STRING)
  private Gender;
      . . .
}
```

```
<?xml version="1.0"encoding="UTF-8"?>
<entity-mappings version="1.0">
<entity class="entity.Student"
name="Student">
<attributes>
<basic name="gender">
<enumerated>STRING</enumerated>
</basic>
</attributes>
</entity>
</entity-mappings>
```

***Table 2.2**: Different ways of configuring Java enum types (using @Enumerated as well as XML tag <enumerated>)*

Optional: A Basic attribute can be optional if its value is allowed to be null. By default, everything is assumed to be optional, except for an Id, which cannot be optional. Optional is basically a hint that applies to database schema generation to add a NOT NULL constraint to the column if false. Some JPA providers also perform validation of the object for optional attributes, and throw a validation error before writing to the database.

LOBs, BLOBs, CLOBs, and Serialization

Large objects (**LOB**) are set of data types that hold large data such as large binary (**Binary large object** or **BLOB**) values or string (**Character large object** or **CLOB**) values as the normal VARCHAR/VARBINARY database types typically have size limitations. A LOB is often stored as a locator in the database table, with the actual data stored outside of the table. In Java, a CLOB maps to a String, and a BLOB maps to byte array. In addition, a BLOB may also represent serialized objects.

By default, in JPA any serializable attribute that is not a relationship or a basic type

will be serialized to a BLOB field. JPA defines the **`@Lobannotation`** (in XML **`<lob>`**element) to define that attribute maps to a LOB type in the database, as LOBs may need to be persisted specially. Various databases and JDBC drivers have diverse limits for LOB sizes. Some JDBC drivers have issues beyond 4K, 32K or 1M. For example, Oracle thin JDBC drivers had a 4K limitation in some earlier versions for which a workaround was provided later.

```
@Entity
public class Student {

  @Basic (fetch=FetchType.LAZY)
  @Lob
  private Imagephoto;
.....
}
```

```
<?xml version="1.0"encoding="UTF-8"?>
<entity-mappings version="1.0">
<entity class="entity.Student"
name="Student" access="FIELD">
<attributes>
<basic name="photo" fetch="LAZY"><lob/>
</basic>
</attributes>
</entity>
</entity-mappings>
```

***Table 2.3**: Configuring LOB types with LAZY fetch (using @Lob as well as XML tag <lob>)*

Some JPA providers as well as JDBC drivers prefer using streams for reading LOBs. Generally, the entire LOB is read and written for the attribute. For very large LOBs, the fetch type could be set to lazy mode such that it is not read unless accessed for better performance. Note that this is optional in JPA, so some providers may not support this mode setting. A workaround to improve the performance is to store

the LOB in a separate table and define a one to one relation to the LOB object. Also, avoid mapping the LOB attribute if it is never desired to be read entirely. It may be possible to map the LOB to **`java.sql.Blob`** or **`java.sql.Clob`**.

Lazy fetching

While a mapping is defined in JPA, this gives the instruction to fetch that particular column value from the DB and place it in the mentioned variable in the Java object. By default, all basic mappings follow EAGER fetching strategy which means to populate the column value whenever the object is selected. However, there are conditions where many column values in the object are not needed to be fetched, unless required. This fetch strategy is known as the **LAZY** strategy, and the column values is not selected whenever the object is selected. Instead, the column value will be retrieved in a separate database select whenever the attribute is accessed. The support for LAZY fetch is an optional feature of JPA, and hence, some JPA providers may not support it.

In order to support lazy fetch on basic attributes, some form of byte code weaving,

or dynamic byte code generation will be required, which may have issues in certain environments or JVMs or may require pre-processing in the persistence unit jar.

Note that since each lazy marked attribute causes a separate database select, only attributes that are rarely accessed should be marked so. Otherwise, it can cause major performance issues.

Some JPA providers support fetch groups in general which allow more sophisticated control over what attributes are fetched per query. Fetch groups are sets of fields that load together in order to provide performance improvements over standard data fetching. Specifying fetch groups that allow for tuning of lazy loading and eager fetching behavior.

Column definition and schema generation

There are various attributes on **`@Columnannotation`** (as well as **`<column>`**element) for database schema generation. Note that many JPA providers allow the feature of auto-generation of the database schema, and the Java types of the object's attributes are mapped to the corresponding database type based on the DB platform being used. The configurations for schema generation in the DB with the JPA provider can be done via properties in **`persistence.xml`**. Note that **`persistence.xml`** is the deployment descriptor for persistence using JPA. If auto-generation/update of schema is not enabled as mentioned earlier, then none of the following attributes will affect the schema of the database.

The **`columnDefinition`** attribute of **`@Column`** can be used to override the default database type used, or enhance the type definition with constraints or other such DDL. The length, scale and precision can also be set to override defaults. It is always good to set these values to avoid data truncation.

The unique attribute can be used to define a unique constraint on the column.

However, most JPA providers automatically define primary key and foreign key constraints based on the Id and relationship mappings.

JPA does not define any options to define an index. Some JPA providers may provide extensions for this. However, in most cases, the indexes need to be created through native queries.

Insertable, updatable, read only fields

`@Columnannotation` (and of course its XML element counterpart) defines insertable and updatable options. These options can be used in following conditions:

- when this column, or the foreign key field are omitted from the SQL insert or update statement
- to be used if constraints on the table prevent insert or update operations
- if multiple attributes map to the same database column, such as a foreign key field through a ManyToOne and Id mapping.

Setting both **`insertable`** and **`updatable`** to false, effectively makes the attribute as read-only. **`Insertable`** and **`updatable`** can also be used in the database table defaults, or auto assigns values to the column on insert or update. Note that by doing so, the object's values will be out of synch with the database, unless it is refreshed.

For auto assigned ID columns, a generated value should normally be used, instead of setting **`insertable`** to false. Some JPA providers support returning auto assigned fields values from the database after insert or update operations. The cost of refreshing or returning fields back into the object can affect performance. So, it is normally better to initialize field values in the object model, not in the database.

Conversion

A common problem in storing values to the database is that the value desired in Java differs from the value used in the database. Common examples include using a Boolean in Java and a 0, 1 or a 'T', 'F' in the database. Other examples are using a String in Java and DATE in the database.

Date/Time translations have already been dealt in one of the previous sections. For generic conversions, some JPA providers support translation using **`@Convert`**,

`@Converter`, **`@ObjectTypeConverter`** and **`@TypeConverter`** annotations (and corresponding XML elements/attributes).

One way to accomplish this conversion is to translate the data through property get/set methods.

```
@Entity
public class Student implements Serializable {

  private boolean isActive;

  @Transient
  public boolean getIsActive () {return isActive;}
  public void setIsActive (boolean isAct) {this.isActive = isAct; }

  @Basic
  private String getIsActiveValue () {
    return (isActive)? "T": "F";
  }

  private void setIsActiveValue (String isAct) {
    this.isActive = "T".equals(isAct);
  }
}
```

***Code 2.13**: Translating via property get/set methods*

In the preceding piece of code, the **`isActive`** variable is used to store the Boolean version of the DB column value that is stored as a string ("T" or "F"). Hence, the variable is declared as transient, and it is not directly fetched from the DB. The value from the DB is accessed via the get and set methods annotated as **`@Basic`** (**`getIsActiveValue`** and **`setIsActiveValue`**). Once the value is obtained, the variable is accordingly populated and made available for other Java objects via **`getIsActive`** and **`setIsActive`** methods.

Custom types

JPA defines support for most common database types, however some databases and JDBC driver have additional types that may require additional support.

Some custom database types include:

- TIMESTAMPTZ, TIMESTAMPLTZ (Oracle)

- TIMESTAMP WITH TIMEZONE (Postgres)
- XMLTYPE (Oracle)
- XML (DB2)
- NCHAR, NVARCHAR, NCLOB (Oracle)
- Struct (STRUCT Oracle)
- Array (VARRAY Oracle)
- BINARY_INTEGER, DEC, INT, NATURAL, NATURALN, BOOLEAN (Oracle)
- POSITIVE, POSITIVEN, SIGNTYPE, PLS_INTEGER (Oracle)
- RECORD, TABLE (Oracle)
- SDO_GEOMETRY (Oracle)
- LOBs (Oracle thin driver

To handle persistence to custom database types, either custom hooks are required in the JPA provider, or there should be a mix of raw JDBC code along with JPA objects. Some JPA providers allow custom support for many custom database types, whereas, some even provide custom hooks for adding developer's own JDBC code to support a custom database type.

Embeddables

In any application object model, there are two types of objects:

1. Independent objects
2. Dependent parts of other objects

For example, a student object can contain elements like **`firstName`**, **`lastName`**, standard, division, and so on. The address can be a separate object and can be used to store the address of the student. Hence, when the student details are requested for, a composition of student object along with its corresponding address object is provided. Here, the student object is the independent one, whereas the address object is the dependent part. In a relational DB, this can be handled in two different ways.

1. Address can be in a separate table and have a foreign key which would be the student id.
2. Address can be embedded into the student table itself and hence, be part of the same table.

In JPA, a relationship where the target object's data is embedded in the source object's table is considered as an embedded relationship and the target object is considered as an embeddable object. Embeddable objects have different requirements and restrictions than Entity objects and are defined by **`@Embeddable`** annotation or **`<embeddable>`** element.

An embeddable object cannot be directly persisted, or queried, it can only be persisted or queried in the context of its parent. An embeddable object does not have an id or table. Although, JPA spec does not support embeddable objects having inheritance, some JPA providers may allow this feature.

In JPA 2.0, relationships from embeddable objects are supported and are defined through **`@Embedded`** annotation or **`<embedded>`** element. JPA 2.0 spec also supports collection relationships to embeddable objects.

```
@Embeddable
public class Address {
  @Column (name=”STREET_NAME”)
  private String streetName;

  @Column (name=”CITY”)
  private String city;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<embeddable      class="entity.Address"
access=”FIELD”>
<attributes>
<basic name="streetName">
<column name=”STREET_NAME”/>
</basic>
<basic name="city">
<column name=”CITY”/>
</basic>
</attributes>
</embeddable>
```

```
@Entity
public class Student {

  @Id
  private long id;

  @Column (name=”S_NAME”)
  private String studName;

  @Embedded
  private Address studAddress;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<entity  name="Student”  class=”entity.
Student" access=”FIELD”>
<attributes>
<id name="id"/>
<basic name="studName">
<column name=”S_NAME”/>
</basic>
<embedded name="studAddress"/>
</attributes>
</entity>
```

***Table 2.4**: Configuring Embeddable as well as Embedded objects using annotations (on the left) and XML (on the right)*

JPA embeddable objects can support features like sharing between multiple objects, relationships, collections, inheritance (in specific JPA implementations), and so on. There are also certain restrictions on querying embeddable objects. The upcoming sections will elaborate on all the previously-mentioned features and restrictions.

Sharing

An embeddable object can be shared by multiple objects. Consider the preceding embeddable object **`Address`**. This can be used to describe the address of the student or the staff of the school. If the address is not stored in a separate table but stored along with student/staff table, then the column names can be different. JPA embeddables support this by allowing each embedded mapping to override columns used by **`@AttributeOverride`** annotation or **`<attribute-override>`** element.

```
@Entity
public class Student {
  @Id
  private long id;

  @Column (name=”STUD_NAME”)
  private String studName;

  @Embedded
  @AttributeOverrides({
    @AttributeOverride(
     name=”streetName”,
     column=@Column(name=”STR_
NAME”)),
    @AttributeOverride(
     name=”city”,
     column=@Column(name=”CITY”))
    })
  private Address studAddress;
}
```

```
<?xml version="1.0"
encoding="UTF-8"?>
<entity-mappings version="1.0">
<entity class="entity.Student"
name="Student" access=”FIELD”>
<attributes>
<id name=”id”/>
<basic name="studName">
<column name=”STUD_NAME”/>
</basic>
<embedded name=”studAddress”>
<attribute-override
name=”streetName”>
<column name=”STR_NAME”/>
</attribute-override>
<attribute-override name=”city”>
<column name=”CITY”/>
</attribute-override>
</embedded>
</attributes>
</entity>
</entity-mappings>
```

```
@Entity
public class Staff {
  @Id
  private long id;

  @Column (name=”STAFF_NAME”)
  private String staffName;

  @Embedded
  @AttributeOverrides({
   @AttributeOverride(
    name=”strName”,
    column=@Column(name=”S_
NAME”)),
   @AttributeOverride(
    name=”city”,
    column=@Column(name=”S_
CITY”))
  })
  private Address staffAddress;
}
```

```
<?xml
version="1.0"encoding="UTF-8"?>
<entity-mappings version="1.0">
<entity class="entity.Staff"
name="Staff" access=”FIELD”>
<attributes>
<id name=”id”/>
<basic name="staffName">
<column name=”STAFF_NAME”/>
</basic>
<embedded name=”staffAddress”>
<attribute-override name=”strName”>
<column name=”S_NAME”/>
</attribute-override>
<attribute-override name=”city”>
<column name=”S_CITY”/>
</attribute-override>
</embedded>
</attributes>
</entity>
</entity-mappings>
```

***Table 2.5**: Sharing of Embeddable object Address using annotations (on the left) and XML element (on the right)*

The preceding example shows the usage of embeddable object **`Address`** in the **`Student`** as well as **`Staff`** entity object.

Embedded Id

There can be cases where the table has composite primary keys. In such cases, the primary key needs multiple columns to be mapped to, and hence can be defined using an **`EmbeddedId`**. This defines a separate embeddable Java class with **`@EmbeddedId`** annotation or **`<embedded-id>`** XML element, to contain all the columns defined as basic column mappings.

`EmbeddedId` is also used as the structure passed to the find as well as search implementations in the JPA. Also, some JPA products use it as a cache key to track

an object's identity. Hence, in most of the JPA products, it is required to implement an **`equals()`**and **`hashCode()`** method on the **`EmbeddedId`**.

```
@Entity
public class Student {
  @EmbeddedId
  private StudentPK id;
}

@Embeddable
public class StudentPK {
  @Basic
  private int studentID;
  @Basic
  private int standard;
  @Basic
  private String division;

  public StudentPK (long sId, int std,
                    String div) {
    this.studentID = sId;
    this.standard = std;
    this.division = div
  }
  . . .
  public boolean equals(Object object) {
    if (object instanceof StudentPK) {
      StudentPK pk = (StudentPK) object;
     return ((studentId == pk.studentId) &&
              (standard == pk.standard) &&
            (division.equals(pk.division)))
    } else {
      return false;
    }
  }
  public int hashCode() {
    return (studentId + standard
           + division.hashCode());
  }
}
```

```
<entity class="entity.
Student"
name="Student"
access="FIELD">
<embedded-id name="id"
class="entity.StudentPK"/>
</entity>
<entity class="entity.
StudentPK"
name="StudentPK"
access="FIELD">
<attributes>
<basic name="studentID"/>
<basic name="standard"/>
<basic name="division"/>
</attributes>
</embeddable>
```

***Table 2.6**: EmbeddedId for composite primary keys*

The **`equals()`** method makes sure that each part of the primary key uses appropriate methods for comparison between the attributes of **`EmbeddedId`**. The **`hashCode()`** method will return the same value for two equal objects.

Nesting

A nested embeddable is a relationship to an embeddable object from another embeddable. This was not supported in JPA 1.0 spec, as it only allows basic relationships in an embeddable object. However, from JPA 2.0 onwards, nested embeddables are supported as technically there is nothing preventing the usage of **`@Embedded`** annotation inside an embeddable object. TopLink/EclipseLink supports embedded mappings from embeddables. The existing **`@Embedded`** annotation or **`<embedded>`** element can be used.

```
@Embeddable
public class StudentDetails {
  private Address address;
  private int standard;
      . . .
  public StudentDetails() {
    this.address = new Address();
  }

  @Transient
  public Address getAddress(){
    return address;
  }
  @Basic
  public String getStreet(){
    return getAddress().getStreetName();
  }
  public void setStreet(String stName){
    getAddress().setStreetName(stName);
  }

  @Basic
  public String getCity(){
```

```
    return getAddress().getCity();
  }
  public void setCity(String cityName){
    getAddress().setCity (cityName);
  }
```

***Code 2.14**: Workaround for using a nested embeddable in JPA 1.0 based products*

A workaround to having a nested embeddable, and for embeddables in general, is to use property access, and add **`get`**/**`set`** methods for all of the attributes of the nested embeddable object.

Inheritance

Embeddable inheritance is when one embeddable class subclasses another embeddable class. JPA spec does not allow inheritance in embeddable objects, however some JPA products may support this. Inheritance, in embeddables, is always a single table as an embeddable must live within its' parent's table. Generally, attempting to mix inheritance between embeddables and entities is not a good idea, but may work in some cases.

Relationships

A relationship is when an embeddable has a mapping to an entity. JPA 1.0 spec only allows basic mappings in an embeddable object, so relationships from embeddables are not supported, however JPA 2.0 supports all relationship types from an embeddable object.

Relationships to embeddable objects from entities other than the embeddable's parent are ordinarily not a good idea, as an embeddable is a private dependent part of its parent. Generally, relationships should be to the embeddable's parent, not the embeddable. Otherwise, it would normally be a good idea to make the embeddable an independent entity with its own table. If an embeddable has a bi-directional relationship, such as a OneToMany that requires an inverse ManyToOne, the inverse relationship should be to the embeddable's parent.

```
@Entity
public class Student {
  private SchoolDetails schoolDetails;

  @Embedded
  public SchoolDetails getSchoolDetails() {
```

```
    return schoolDetails;
  }
  @OneToOne
  public SchoolDetails getSchoolAddress() {
    return getSchoolDetails().getAddress();
  }
  public void setSchoolAddress(Address address) {
    getSchoolDetails().setAddress(address);
  }
}
```

***Code 2.15**: Workaround for using a nested embeddable in JPA 1.0 based products*

A workaround to having a relationship from an embeddable is to define the relationship in the embeddable's parent, and define property get/set methods for the relationship that set the relationship into the embeddable.

Collections

A collection of embeddable objects is similar to a OneToMany relationship except the target objects are embeddables and have no ID. This allows for a OneToMany to be defined without an inverse ManyToOne, as the parent is responsible for storing the foreign key in the target object's table. JPA 1.0 did not support collections of embeddable objects, but JPA 2.0 does support collections of embeddable objects through the element collection mapping.

Normally, the primary key of the target table will be composed of the parent's primary key, and some unique field in the embeddable object. The embeddable should have a unique field within its parent's collection, but does not need to be unique for the entire class. It could still have a unique ID and still use sequencing; or if it has no unique fields, its ID could be composed of all of its fields.

The embeddable collection object will be different than a typical embeddable object as it will not be stored in the parent's table, but in its own table. Embeddables are strictly privately owned objects, deletion of the parent will cause deletion of the embeddables, and removal from the embeddable collection would cause the embeddable to be deleted. Embeddables cannot be queried directly, and are not independent objects as they have no identifier.

Querying

Embeddable objects cannot be queried directly, but they can be queried in the context of their parent. Typically, it is best to select the parent, and access the embeddable from the parent. This will ensure the embeddable is registered with the persistence context.

```
SELECT student.period from Student student where student.period.
endDate=:param
```

***Code 2.16**: Workaround for using a nested embeddable in JPA 1.0 based products*

If the embeddable is selected in a query, the resulting objects will be detached, and changes will not be tracked.

Mapping – Advanced

In this chapter, till now, various access modes and different attributes have been dealt with. This section will provide more insights to the scenarios that are not the ideal cases, and hence, would require more explanation. While mapping a table, the ideal case would be each class maps to a single table. However, this is not always possible, and here is the list of scenarios where exceptions would happen:

- **Multiple tables**: One class looks into two or more tables.
- **Sharing tables**: Two or more classes are stored in a single table.
- **Inheritance**: A class is involved in inheritance.
- **Views**: A class is mapped to views.
- **Stored procedures**: A class is mapped to a set of stored procedures.
- **Partitioning**: Some instances of a class mapped to one table, while other instances mapped to another table.
- **Replication**: The data of a class is replicated to multiple tables.
- **History**: A class has historical data.

These are advanced cases, and some are handled by JPA spec, while others are not. The following sections investigate on how and up to what level the JPA supports each one of these and how to bypass the limitations.

Multiple tables

A class spanning over to multiple tables typically occurs on legacy or existing data models, where the object model and data model do not match. It can also occur in

inheritance when subclass data is stored in additional tables. Multiple tables may also be used for performance, partitioning, or security reasons.

JPA allows multiple tables to be assigned to a single class by using annotation **`@SecondaryTable`** (or **`<secondary-table>`** element). By default, **`@Id`** column(s) are assumed to be in both tables, such that the secondary table's **`@Id`** column(s) are the primary key of the secondary table and a foreign key to the first table. If the first table's **`@Id`** column(s) are not named the same, the annotation **`@PrimaryKeyJoinColumn`** (or **`<primary-key-join-column>`** element) can be used to define the foreign key join condition.

In a multiple table entity, each mapping must define which table the mapping's columns are from. This is done using the table attribute of the **`@Column`** or **`@JoinColumn`** annotations (or with available appropriate XML elements). By default, the primary table of the class is used, so it is only required to set the table for secondary tables. For inheritance, the default table is the primary table of the subclass being mapped.

With the **`@PrimaryKeyJoinColumn`**, the name refers to the foreign key column in the secondary table and the **`referencedColumnName`** refers to the primary key column in the first table. If there are multiple secondary tables, they must always refer to the first table. When defining the table's schema, define the join columns in the secondary table as the primary key of the table, and a foreign key to the first table. The order of tables can be important as that order will match the one that will be used by the JPA implementation to insert into the tables. Hence, it is very important to ensure that the table order matches the constraint dependencies. JPA does not allow having a foreign key map to a table other than the target object's primary table. Normally, this will not be an issue as foreign keys almost always map to the ID/primary key of the primary table, but in some advanced scenarios this may be an issue. Some JPA products allow the column or join column to use the qualified name of the column to allow this type of relationship i.e., **`@JoinColumn(referenceColumnName="STUDENT_DATA.STUD_NUM")`**.

Multiple tables with foreign keys

Normally, a foreign key is mentioned from a secondary table to a primary table. But there are cases where the secondary table is referenced through a foreign key from the primary table to secondary table. The JPA spec does not cover this scenario directly and hence, if possible, it is always better to change the data model to stay within the boundary of the JPA spec. Changing the data model is not so easily possible and hence, the best solution is to change the object model by defining a single class for each table and implement the logic accordingly.

Swapping of primary and secondary tables can solve the issue in certain cases. However, this will have side-effects and may have querying and mapping implications. This also cannot be used if there are more than two tables.

Another option is to just use the foreign key column and thus will be technically backward. This is not compatible with the JPA spec but might work with some JPA implementations. However, the inserts into the table may not follow the order and hence constraints need to be removed or deferred.

The third option is to create a database view and map the class to this view. In certain databases, views are read-only, but many other databases allow writes or triggers to handle writes into views.

Multiple table joins

Occasionally, the data model and object model do not get along very well. The database could be a legacy model and not fit very well with the new application model. In these cases, advanced multiple table joins might be required. Examples of these include having two tables related not by their primary or foreign keys, but through some constant or computation.

Consider having a **`STUDENT`** table and **`ADDRESS`** table. The **`ADDRESS`** table can have the **`STUDENT_ID`** as foreign key and can have multiple addresses (like permanent, present, etc.). If the permanent address is only required, then there should be a join where the foreign key matches and the address type matches to permanent. Such scenarios should be handled by either redesigning the data or object model or by using a view.

Another problem of multiple table mapping is when the secondary table may or may not have a row defined for the object. Then, an outer join should be made and the object should be read-only as there is no guarantee that the object exists or not. This is not directly supported by JPA, and it is always best to reconsider the data model or object model design.

Tables with special characters and mixed case

Some JPA providers may have issues with table and column names with special characters, such as spaces. In general, it is best to use standard characters, no spaces, and all upper-case names.

It may be required to "*quote*" table and column names with special characters. For example, if the table name had a space, it could be defined as the following:

```
@Table("\"Employee Data\"")
```

Conclusion

In this chapter, the basic structure of relational database table, how Java objects are mapped to the tables, and the different kinds of attributes that are supported via JPA have been explained in detail. This brings us to the end of this introductory session.

In the next session, the activities that are performed and how the tables are used will be dealt with in detail. The next chapter is on the various operations available in DB, and how they can be achieved via JPA.

Abbreviations

RDBMS: Relational Database Management System

Multiple choice questions

1. **Relational Database model was proposed by:**
   a) E F Codd
   b) C. J Date
   c) Avi Silberschatz
   d) Henry F Korth

2. **Relational databases store data in form of:**
   a) tuples
   b) tables
   c) objects
   d) byte streams

3. **Each row in the table of a relational databases is also known as:**
   a) attribute
   b) object
   c) tuple
   d) column

4. **Each column in the table of a relational databases is also known as:**
   a) tuple
   b) attribute
   c) object
   d) column

5. **A column in the table of a relational database that creates a relationship between two tables is known as:**
    a) primary key
    b) composite key
    c) compound key
    d) foreign key

Answers

1. **a**
2. **b**
3. **c**
4. **b**
5. **d**

Questions

1. Explain the significance of keys used in relational databases.
2. Explain two scenarios where a single class cannot be ideally used to denote a table.
3. Explain the terms:
    - Lazy fetching
    - Embeddable
    - Access modes
    - Attributes
    - Custom types

Points to ponder

1. In field access mode, the provider ignores the getter and setter methods. All the data access happens through fields.
2. **TopLink/EclipseLink**: It does not require the implementation of **`equals()`** or **`hashCode()`** in the ID class. Note that in any Java class implementation, if **`equals()`** method is overridden, then it is a must to override **`hashCode()`** method as well.

3. An embeddable object's data is contained in several columns in its parent table. Since there is no single field value, there is no way to know if a parent's reference to the embeddable is null. The only assumption that can be made is that if every field value of the embeddable is null then the reference should also be null. JPA does not allow embeddables to be null, but some JPA providers may support this.

4. **TopLink/EclipseLink**: It supports an embedded reference being null. This is set using a Descriptor Customizer and **`AggregateObjectMapping.setIsNullAllowed`** API.

5. **TopLink/EclipseLink**: It supports inheritance with embeddables using a Descriptor Customizer, and the InheritancePolicy.

6. **TopLinkTopLink/EclipseLink**: It provides proprietary API for its mapping model Class Descriptor. The method **`addForeignKeyFieldNameForMultipleTable()`** allows for arbitrary complex foreign key relationships to be defined among the secondary tables.

7. **TopLinkTopLink/EclipseLink**: It provides proprietary API for mapping **`DescriptorQueryManager`**. The method **`setMultipleTableJoinExpression()`** allows for arbitrary complex multiple table joins to be defined.

References

1. Java Persistence Wikibooks: **https://en.wikibooks.org/wiki/Java_Persistence**
2. Java persistence specifications: **https://jcp.org/en/jsr/detail?id=317**
3. Stack overflow Developer Survey: **https://insights.stackoverflow.com/survey/2021# section-most-popular-technologies-databases**
4. Codd's 12 rules: **https://database.guide/codds-12-rules/**

Section - II

Operations and Relationships

Chapter 3

Operations – Identity, Sequencing and Locking

Introduction

Till now, the discussion was mostly on the type of data that can be stored in a relational database and how they can be represented in the Java world. Any application is usable only till the data utilized in it is accurate and helpful. It is very important to have the database activities designed in such a way that it enables the user to achieve legitimate path to the data repository. It is also equally important that the data at rest is not being hampered or contaminated internally or through the application without the user's knowledge. In this chapter, the discussion takes a step ahead and brings out what all operations are required for the reliable storage and access of data.

Structure

This chapter discusses the following with relevant examples in each section:

- Identifiers and their common problems.
- Sequencing strategies – table sequencing, identity sequencing, sequencing object, advanced sequencing
- Primary keys through events, triggers
- Inheritance, single table and joined multiple table inheritance

- Locking techniques
  - Optimistic locking
  - Pessimistic locking
  - Other advanced locking

Objectives

This chapter will bring out to the reader the important operations on the database that are mainly used by JPA. Here, the issues like ordering, sequencing and locking are considered. You will learn about operations such as identity operation and sequencing operation. You will also learn about the concepts of *locking* and *inheritance*.

Operations

Database operations are basically the tasks that can be performed on a database. It is a key differentiator on what kind of operations and up-to which level these are supported in a database. Any database should support the basic operations like create, read, update, and delete of an entry (CRUD operations). Apart from these, there are a few more important operations that help in performing or enhancing these basic operations.

Identity

One of the most important aspects of data storage is to uniquely identify the stored data. This can be done by an identifier and hence, the work done by the identifier can be termed as identity operation. As discussed in the previous chapter, there are various kinds of keys in a relational database system. However, the primary key is the most important one, which attributes the unique identification of each record in the database. This unique identifier is very important while persisting an object as this helps in performing various operations like querying, defining relationships, updating, and deleting that object.

The object identifier in a database context is its primary key. In JPA, this primary key must be defined on the entity class that is the root of the entity hierarchy. It may also be declared on a mapped superclass that is a (direct or indirect) superclass of all entity classes in the entity hierarchy. Note that the primary key must be defined exactly once in an entity hierarchy. The primary key can have one or more fields of properties or attributes of the entity class.

- A simple (i.e., non-composite) primary key must correspond to a single persistent field or property of the entity class. The **`@Id`** annotation or **`<id>`** element must be used to denote a simple primary key.

- A composite primary key must correspond to either a single persistent field or property or to a set of such fields or properties described as follows. A primary key class must be defined to represent a composite primary key. Composite primary keys typically arise while mapping from legacy databases, when the database key comprises several columns. The **`@EmbeddedId`** or **`@IdClass`** annotation is used to denote a composite primary key.

@EmbeddedId vs @IdClass

As mentioned in *Chapter 2, Tables – Attributes and Embeddable Objects*, **`@EmbeddedId`** is used to denote the composite primary keys (refer to *Table 2.6* of *Chapter 2, Tables – Attributes and Embeddable Objects*). Instead of using **`@EmbeddedId`**, **`@IdClass`** can also be used to denote the composite primary key. The sample code provided earlier would be changed as given below with the usage of the **`@IdClass`** annotation.

```
public class StudentPK implements Serializable {
 private long studentID;
 private int standard;
 private String division;
//default constructor
 public StudentPK (long sId, int std, String div) {
   this.studentID = sId;
   this.standard = std;
   this.division = div
 }
 // equals and hashcode should be implemented
}
@Entity
@IdClass(StudentPK.class)
public class Student {
 @Id
 private long studId;
 @Id
 private int studStd;
 @Id
 private String studDiv;
}
```

***Code 3.1**: Usage of @IdClass*

The differences are as follows:

1. With **`@IdClass`**, the columns are to be specified twice – once in StudentPK and again in Student. But with **`@EmbeddedId,`** this is not required.
2. JPQL queries are also different for both cases. In case of **`@IdClass,`** the query is a bit simpler but in case of **`@EmbeddedId`** an extra traversal is required.
   - **`@IdClass: SELECT student.studId FROM Student`**
   - **`@EmbeddedId: SELECT student.id.studentId FROM Student student`**
3. If the different parts of the composite key are accessed individually, then **`@IdClass`** is preferred. In cases where the complete identifier is used as an object, **`@EmbeddedId`** is preferred.

However, in both cases, in order to define the composite primary keys, these rules should be followed:

- The composite primary key class must be public.
- It must have a no-arg constructor.
- It must define **`equals ()`** and **`hashCode ()`** methods.
- It must be serializable.

In the next section, the ways of determining the object ID are being discussed.

Sequencing

An object ID can either be a natural ID or a generated one. A natural ID is one that occurs in the object and has some meaning in the application. Examples of natural IDs include user IDs, email addresses, phone numbers, and social insurance numbers. A generated ID is a sequence ID generated by the system, and automatically assigned to new objects. The benefits of using sequence numbers are that they are guaranteed to be unique, allow all other data of the object to change, are efficient values for querying and indexes, and can be efficiently assigned. The main issue with natural ID is that everything always changes at some point; even a person's social insurance number can change. Natural IDs can also make querying, foreign keys, and indexing less efficient in the database.

In JPA, an **`@Id`** can be easily assigned a generated sequence number through the @**`GeneratedValue`** annotation, or **`<generated-value>`** element.

```
@Entity
public class Student {

      @Id
      @GeneratedValue
      private long studentId;
       . . .
}
```

```
<?xml_version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0">
  <entity class="entity.Student"
          name="Student">
    <attributes>
      <id name="studentId">
        <generated-value/>
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

***Table 3.1**: Generated sequence number configuration using annotation (left) as well as XML element (right)*

There are several strategies for generating unique IDs. Some strategies are database agnostic, and others make use of built-in databases support. JPA provides support for several strategies for id generation defined through the GenerationType enum values: TABLE, SEQUENCE, and IDENTITY. The choice of which sequence strategy to use is important as it affects performance, concurrency and portability.

Table sequencing

Table sequencing uses a table in the database to generate and store the unique IDs. The table consists of two columns, one stores the name of the sequence, the other stores the last ID value that was assigned. There is a row in the sequence table for each sequence object. Each time a new ID is required the row for that sequence is incremented and the new ID value is passed back to the application to be assigned to an object.

SEQ_NAME	SEQ_COUNT
STUD_SEQ	123
SUB_SEQ	50

***Table 3.2**: Sample sequence table named SEQ_TABLE*

The preceding table shows sequence numbers for a database where it stores students and subjects with the table-sequencing method. Whenever a row is added to students/subjects, the **`SEQ_COUNT`** of that particular row (in case of students, **`STUD_SEQ`** row is accessed, and in case of subjects, **`SUB_SEQ`** row will be considered) is incremented and taken from this table as the next sequence ID.

Table sequencing is the most portable solution as it just uses a regular database table and can be used on any database. Table sequencing also provides good performance because it allows for sequence pre-allocation, which is extremely important to insert performance, but can have potential concurrency issues.

In JPA, **@TableGenerator** annotation or **<table-generator>** element is used to define a sequence table. TableGenerator defines a **pkColumnName** for the column used to store the name of the sequence, **valueColumnName** for the column used to store the last id allocated, and **pkColumnValue** for the value to store in the name column (normally the sequence name).

```
@Entity
public class Student {
  @Id
  @TableGenerator (
    name=”TABLE_GEN”,
    table=”SEQ_TABLE”,
    pkColumnName=”SEQ_NAME”,
    valueColumnName=”SEQ_COUNT”,
    pkColumnValue=”STUD_SEQ”)
  @GeneratedValue (
    strategy=GenerationType.TABLE,
    generator=”TABLE_GEN” )
  private long studentId;
    . . .
}
```

```
<entity class="entity.Student"
        name="Student">
 <attributes>
  <id name="studentId">
   <generated-value strategy=”TABLE”
        generator=”TABLE_SEQ”/>
      <table-generator  name=”TABLE_
SEQ”
        table=”SEQ_TABLE”
        pk-column-name=”SEQ_NAME”
       value-column-name=”SEQ_COUNT”
        pk-column-value=”STUD_SEQ”/>
  </id>
 </attributes>
</entity>
```

***Table 3.3**: JPA configuration for using the SEQ_TABLE to generate the sequence values.*

The preceding table shows the annotation based as well as the XML based configuration of the table-based sequence generation in JPA. On the left side, the annotation based is shown, whereas on the right side, XML based configuration is shown.

While using this kind of sequencing technique, the existence of the sequence table in the DB and the corresponding name used in the JPA code should be verified. The table expects to have a starting value, i.e., the initial sequence number should be given for every sequence field entry in the table. This initialization is to avoid “*sequence not found*” errors. The names used in JPA as well as the names in the table should be cross-checked, so that there are no “*table not found*” or “*invalid column*” errors.

Sequencing objects

Sequence objects use special database objects to generate IDs and these objects are not supported in all databases. These sequence objects are only supported in databases like Oracle, DB2, and Postgres. Usually, a sequence object has a name **SEQUENCE**, an increment value **INCREMENT**, and other database object settings. To obtain the sequence value, each time **SEQUENCE.NEXTVAL** is selected and the sequence is incremented by **INCREMENT**.

Sequence objects provide the optimal sequencing option, as they are the most efficient and have the best concurrency, however, they are the least portable as most databases do not support them. Sequence objects support sequence pre-allocation through setting the **INCREMENT** on the database sequence object to the sequence pre-allocation size.

The DDL to create a sequence object depends on the database. For example, in Oracle it is

```
CREATE SEQUENCE STUD_SEQ INCREMENT BY 1 START WITH 100.
```

Here, **STUD_SEQ** is the sequence and 1 is the increment and the starting value is 100.

In JPA, the **@SequenceGenerator** annotation or **<sequence-generator>** element is used to define a sequence object.

```
@Entity
public class Student {
  @Id
  @GeneratedValue (
  strategy=GenerationType.SEQUENCE,
  generator=”SEQ_OBJ” )

  @SequenceGenerator (
    name=”SEQ_OBJ”,
    sequenceName=”STUD_SEQ”,
    allocationSize=100)
  private long studentId;
    . . .
}
```

```
<entity class="entity.Student"
        name="Student">
 <attributes>
  <id name="studentId">
                  <generated-value
strategy=”SEQUENCE”
       generator=”SEQ_OBJ”/>
    <sequence-generator name=”SEQ_
OBJ”
       sequence-name=”STUD_SEQ”
       allocation-size=100/>
  </id>
 </attributes>
</entity>
```

***Table 3.4**: JPA configuration for using the STUD_SEQ to generate the sequence values*

Errors such as "*sequence not found*" can occur if the **`SEQUENCE`** object is not defined in the DB. Also, ensure that the sequence object's **`INCREMENT`** value matches the **`allocationSize`** in JPA. If there is a mismatch in these values, it can result to invalid, duplicate or negative numbers. There can be cases where the **`allocationSize`** is greater than the **`INCREMENT,`** and hence, the JPA would duplicate or use negative numbers to fill in the rest values.

Identity sequencing

Identity sequencing uses special **`IDENTITY`** columns in the DB to allow the DB itself to automatically assign an ID to the object, whenever a row is inserted. Identity columns are supported in many databases, such as MySQL, DB2, SQL Server, Sybase, and PostgreSQL. Oracle does not support **`IDENTITY`** columns, but it is possible to simulate them using sequence objects and triggers.

Although identity sequencing seems like the easiest method to assign an id, they have two major issues.

1. One is that since the id is not assigned by the database until the row is inserted, the id cannot be obtained in the object until after commit or after a flush call.
2. Another is that identity sequencing also does not allow for sequence pre-allocation, potentially causing a major performance problem, and hence is not recommended.

In JPA, there is no annotation or element for identity sequencing as there is no additional information to specify. Only the strategy attribute for the corresponding annotation or XML element needs to be set to **`IDENTITY`**.

```
@Entity
public class Student {
  @Id
  @GeneratedValue (
  strategy=GenerationType.IDENTITY)
  private long studentId;
    . . .
}
```

```
<entity class="entity.Student"
        name="Student">
 <attributes>
  <id name="studentId">
    <generated-value
         strategy="IDENTITY"/>
  </id>
 </attributes>
</entity>
```

***Table 3.5**: JPA configuration for using the IDENTITY to generate the sequence values*

There can be errors like inserting null values or error on insert if the configurations provided are not correct. It can also happen if the DB used does not support identity sequencing. Another reason for this error could be if the table does not set the primary key column to be an identity type.

Identity sequencing requires the insert to occur first and only then, the ID can be assigned a value. In earlier cases, the ID is assigned on persist, but in this type of sequencing, ID assigning happens only after commit or flush. So, in order to obtain the ID, a select after each insert is essential, which doubles the insertion cost.

This type of sequencing does not support sequence pre-allocation. In case of MySQL DB, a server restart causes auto increment counter to be lost on restart. Hence, if a row has been inserted and MySQL restarted, the same ID might be re-used, thus violating the uniqueness feature of primary key.

A very common error found in this kind of sequencing is that the child's ID is not assigned to the parent on persist. This is because the ID is only obtained after the persist phase for identity sequencing. This can be avoided by first persisting the parent, then call flush before persisting the child.

Advanced sequencing

The following section talks about the various issues that can come up while using sequencing techniques discussed previously.

Concurrency and deadlocks

One issue with table sequencing is that the sequence table can become a concurrency bottleneck, even causing deadlocks. If the sequence ids are allocated in the same transaction as the insert, this can cause poor concurrency, as the sequence row will be locked for the duration of the transaction, preventing any other transaction that needs to allocate a sequence id. In some cases, the entire sequence table or the table page could be locked causing even transactions allocating other sequences to wait or even deadlock. If a large sequence pre-allocation size is used, this becomes less of an issue, because the sequence table is rarely accessed. Some JPA providers use a separate (non-JTA) connection to allocate the sequence ids, avoiding or limiting this issue. In this case, if a JTA data-source connection is used, it is important to also include a non-JTA data-source connection in the **`persistence.xml`**.

Guaranteeing sequential Ids

Sequence and identity sequencing are non-transactional and typically cache values on the database. This is desired to have a good performance but there can be large gaps in the ids that are allocated. By setting the allocation size of the sequence to 1 and ensuring the sequence ids are allocated in the same transaction of the insert, there can be sequence ids without gaps. Generally, it is much better to live with gaps

and have better performance. However, if performance and concurrency are less of a concern, and true sequential ids are desired, then a table sequence can be used.

Customizing

JPA supports three different strategies for generating ids, however, there are many other methods. Some JPA products provide additional sequencing, id generation options, and configuration hooks.

Moreover, there are several other ways to integrate customized id generation strategies. The simplest is to just define the id as a normal id and have the application assign the id value when the object is created. There can be application specific strategy for generating ids, such as prefixing ids with the country code, or branch number.

Running out of numbers

One paranoid delusional fear that programmers frequently have is running out of sequence numbers. Since most sequence strategies just keep incrementing a number, it is unavoidable that the ids eventually run out. The following table shows the duration of each numeric precision to run out of sequence ids.

ID column definition	Number of IDs	Duration
NUMBER (5)	99,999	Almost 70 days (one id per minute)
NUMBER (10)	9,999,999,999	Almost 300 years (one id each second)
NUMBER (20)	99,999,999,999,999,999,999	3,000,000,000 years (one id per millisecond)

***Table 3.6**: ID column numeric precision and its duration to run out*

From the preceding table, it is clear that as long as a large enough numeric precision is used to store the sequence id, this need not be concerned as an issue.

Primary keys for sequencing

Primary keys are IDs that can uniquely identify a row in a table. It can be a single column or group of several columns in a table. Hence, primary keys have an important role in sequencing the rows in a particular table, since they can uniquely identify each object in the table.

Composite primary keys

A composite primary key is one that is made up of several columns in the table. It can be used if no single column in the table is unique. It is generally more efficient and simpler to have a one-column primary key, such as a generated sequence number, but sometimes a composite primary key is desirable and unavoidable.

There are two methods of declaring a composite primary key in JPA–**`IdClass`** and **`EmbeddedId`** (which has been discussed in detail earlier).

Primary keys through triggers

A database table can be defined as having a trigger that automatically assigns its primary key. Generally, this is not a good idea, and it is better to use a JPA provider generated sequence id, or assign the id in the application. The main issue with the id being assigned in a trigger is that the application and object require this value back. For non-primary key values assigned through triggers, it is possible to refresh the object after committing or flushing the object to obtain the values back. However, this is not possible for the id, as the id is required to refresh an object.

If there are any alternate ways to obtain the id generated by the trigger, then the select SQL can be used after the insert to obtain the id and set it back to the object. This select can be performed in a JPA **`@PostPersist`** event. However, this kind of implementation is based on the JPA provider being used. Some JPA providers:

- may not allow a query execution during an event.
- may not pick up a change to an object during an event callback.
- may not allow the primary key to be un-assigned/null when not using a **`GeneratedValue.`**

However, some JPA providers have built-in support for returning values assigned in a trigger (or stored procedure) back into the object.

Primary keys through events

If the application generates its own id instead of using a JPA **`GeneratedValue`**, it is sometimes desirable to perform this id generation in a JPA event, instead of the application code having to generate and set the id. In JPA, this can be done through the **`@PrePersist`** event.

In case if there is no primary key, the best solution is normally to add a generated id to the object/table. If this option is not feasible, sometimes, there is a column or set of columns in the table that make up a unique value and can be hence used as the id in JPA. The JPA Id does not always have to match the database table primary key constraint, nor is a primary key or a unique constraint required. If the table

truly has no unique columns, then use all of the columns as the id. Typically, when this occurs, the data is read-only. So, even if the table allows duplicate rows with the same values, the objects will be the same anyway; so it does not matter that JPA thinks they are the same object. The issue with allowing updates and deletes is that there is no way to uniquely identify the object's row, so all of the matching rows will be updated or deleted.

Inheritance

Inheritance is a fundamental concept of any object-oriented programming language. According to this concept, an object picks up all the properties and behaviors of its parent object. This is one of the most important concepts being used in Java. However, RDBMS have no concept of inheritance, and hence, there is no standard way of implementing this. Thus, the hardest part of persisting inheritance is to adopt a representation of inheritance in RDBMS.

JPA uses the **`@Inheritance`** annotation (or the **`<inheritance>`** element) for defining the inheritance strategy on the root entity class. There are three inheritance strategies defined— **`SINGLE_TABLE, TABLE_PER_CLASS`** and **`JOINED`**.

Single table inheritance is the default, and table per class is an optional feature of the JPA spec, so not all providers may support it. JPA also defines a mapped superclass concept defined though the **`@MappedSuperclass`** annotation, or the **`<mapped-superclass>`** element. A mapped superclass is not a persistent class, but allow common mappings to be defined for its subclasses.

Single table inheritance

Single table inheritance is the simplest and typically the best performing solution. In single table inheritance, a single table is used to store all of the instances of the entire inheritance hierarchy. The table will have a column for every attribute of every class in the hierarchy. A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value.

ID	TYPE	NAME	BUDGET
1	L	Accounting	50000
2	S	Legal	Null

***Table 3.7**: Single table inheritance example Table Project*

For example, consider the preceding table named **`PROJECT`** that is being used for single table inheritance. The JPA annotations (and equivalent XML configurations) for the preceding table are as follows:

```
@Entity
@Inheritance
@DiscriminatorColumn(name="TYPE")
@Table(name="PROJECT")
public abstract class Project {
  @Id
  private long id;
    . . .
}
@Entity
@DiscriminatorValue("L")
public class LargeProject extends
Project {

  private BigDecimal budget;
    . . .
}
@Entity
@DiscriminatorValue("S")
public class SmallProject extends
Project {
      . . .
}
```

```
<entity class="entity.Project"
name="Project">
  <table name="PROJECT"/>
  <inheritance />
 <discriminator-column name="TYPE"/>
  <attributes>
    <id name="id"/>
    …
   </attributes>
</entity>

<entity class="entity.LargeProject"
name="LargeProject">
   <discriminator-value>L</
discriminator-value>
</entity>

<entity class="entity.SmallProject"
name="SmallProject">
   <discriminator-value>S</
discriminator-value>
</entity>
```

***Table 3.8**: Single table inheritance JPA annotations and XML configurations for the previously given table named Project*

The following are a few common problems:

- **No class discriminator column**

  If an existing database schema is being mapped to, then the table may not have a class discriminator column. However, sometimes, the inherited value can be computed from several columns, but not a one-to-one mapping from value to class. Some JPA providers use this kind of support. Another option is to create a database view that manufactures the discriminator column, and map the hierarchy to this view instead of the table. In general, the best solution is just to add a discriminator column to the table.

- **Non-nullable attributes**

  Every subclass will have a set of attributes that are unique from its parent as well as sibling classes (i.e. other classes that inherit from the same parent class). Hence, defining attributes as non-null, in a table that is represented by inheritance in JPA, would not be feasible as the other sibling classes might have to insert null into those columns that are specific to a particular subclass. Instead of defining non-null constraints on the column, a table constraint to check the discriminator value can be used as a workaround to achieve this. In general, it is always better to not constraint the columns in a table and just have null entries.

Joined and multiple table inheritance

Joined inheritance is the most logical inheritance solution because it mirrors the object model in the data model. In joined inheritance, a table is defined for each class in the inheritance hierarchy to store only the local attributes of that class. Each table in the hierarchy must also store the object's id (primary key), which is only defined in the root class. All classes in the hierarchy must share the same id attribute. A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value. For example, consider the following tables:

PROJECT

ID	TYPE	NAME
1	L	Accounting
2	S	Legal

SMALLPROJECT

ID
2

LARGEPROJECT

ID	BUDGET
1	50000

***Table 3.9**: Joined multiple table inheritance example with following tables Project, SmallProject and LargeProject*

```
@Entity
@Inheritance (strategy=Inheritance.
JOINED)
@DiscriminatorColumn(name="TYPE")
@Table(name="PROJECT")
public abstract class Project {
  @Id
  private long id;
    . . .
}
@Entity
@DiscriminatorValue("L")
@Table(name="LARGEPROJECT")
public class LargeProject extends
Project {
  private BigDecimal budget;
}
@Entity
@DiscriminatorValue("S")
@Table(name="SMALLPROJECT")
public class SmallProject extends
Project {
}
```

```
<entity class="entity.Project"
name="Project">
  <table name="PROJECT"/>
  <inheritance strategy="JOINED"/>
  <discriminator-column
name="TYPE"/>
  <attributes>
    <id name="id"/>
    ...
 </attributes>
</entity>
<entity class="entity.
LargeProject"
name="LargeProject">
   <table name="LARGEPROJECT"/>
   <discriminator-value>L</
discriminator-value>
</entity>
<entity class="entity.
SmallProject"
name="SmallProject">
   <table name="SMALLPROJECT"/>
   <discriminator-value>S</
discriminator-value>
</entity>
```

***Table 3.10**: Joined multiple table inheritance JPA annotations and XML configurations*

The advantage of joined inheritance strategy is that it does not waste database space as in single table strategy. On the other hand, because of multiple joins involved for every insertion and retrieval, performance becomes an issue when inheritance hierarchies become wide and deep.

- **Poor performance**

  As mentioned previously, the performance becomes an issue when the query is to the root or branch classes. If such queries are averted, then this performance issue can be diluted. The two ways to query root or branch classes that are used by JPA providers are:

1. All the subclass tables are joined using an outer join and then query is executed.
2. Query the root table and then, query only the required subclass table.

The former has the advantage of having only one query, while the latter has the privilege of avoiding outer joins that causes performance issues in databases.

Practical scenarios - midway of single and joined hierarchies

Most real-world inheritance hierarchies are somewhere in between, having joined tables in some sub classes and not in others. Hence, in practical scenarios, it does not fit with either the joined or the single table inheritance strategy completely. Unfortunately, JPA does not directly support this. A possible workaround is to map inheritance hierarchy as single table, and then add the additional tables in the subclasses, either through defining a Table or SecondaryTable in each subclass as required. However, this depends on the JPA provider, and might have to enquire whether the provider has some specific solution to solve this issue. If there are no such solutions, then decide on which strategy would be more relevant in that particular case, and choose to have either a single table or one per subclass.

No class discriminator column

If an existing database schema is being mapped, table may not have a class discriminator column. Possible workarounds for this issue are:

1. Some JPA providers do not require a class discriminator when using a joined inheritance strategy, so this may be one solution.
2. Different techniques are used to determine the class for a row.
    a) Sometimes the inherited value can be computed from several columns, or there is a discriminator but not a one-to-one mapping from value to class. Some JPA providers provide extended support for this.
    b) Another option is to create a database view that manufactures the discriminator column, and then maps the hierarchy to this view instead of the table.

The preceding mechanisms for inheritance are provided in the JPA spec to be mandatory. However, there are few optional inheritance strategies as well, which will be discussed in the next section.

Advanced inheritance strategies

Consider the following tables. Based on this example, the different advanced inheritance strategies are explained.

LARGEPROJECT

ID	NAME	BUDGET
1	Accounting	50000

SMALLPROJECT

ID	NAME
2	Legal

***Table 3.11**: Example tables LargeProject and SmallProject for explaining advanced inheritance strategies.*

In the preceding example, there is no inheritance descriptor column, and hence the inheritance strategies used are slightly different. They are as follows:

1. table per class
2. mapped super classes

Table per class: This is used in the object model, even when it does not exist in the data model, i.e., there is no inheritance data that is available in the database. In this method, a table is defined for each concrete class and it stores all the attributes of that class and all its super classes.

```
@Entity
@Inheritance
(strategy=Inheritance.TABLE_PER_
CLASS)
public abstract class Project {
  @Id
  private long id;
}
@Entity
@Table(name="LARGEPROJECT")
public class LargeProject extends
Project {
  private BigDecimal budget;
}
@Entity
@Table(name="SMALLPROJECT")
public class SmallProject extends
Project {
}
```

```
<entity class="entity.Project"
name="Project">
  <inheritance strategy="TABLE_PER_
CLASS"/>
  <attributes>
    <id name="id"/>
    ...
 </attributes>
</entity>
<entity class="entity.LargeProject"
name="LargeProject">
   <table name="LARGEPROJECT"/>
</entity>
<entity class="entity.SmallProject"
name="SmallProject">
   <table name="SMALLPROJECT"/>
</entity>
```

***Table 3.12**: Table per class inheritance JPA annotations and XML configurations.*

This strategy has to be used very cautiously as this may not be supported by all the JPA providers (since it is optional in JPA spec).

The main disadvantage of table per class model is queries or relationships to the root or branch classes become expensive. To query any of the branch classes or the root class, it requires multiple queries, or unions.

1. One solution is to use single table inheritance, and this is good only if the classes have a lot of attributes in common. If the hierarchy is big and has very few common attributes, this solution is not desirable.

2. Another solution is to use **`MappedSuperClass`**, but then there can be no queries or relationships to the class.

Mapped superclass: This is very much similar to table per class inheritance, but does not allow querying, persisting, or relationships to the superclass. All the mapping information are to be inherited by the subclasses. The subclasses are bound to define the table, id and other information, and can modify any of the inherited mappings. A mapped super class is an abstract class, hence not an Entity, and instead is defined using **`@MappedSuperclass`** annotation or the **`<mapped-superclass>`** element.

```
@MappedSuperclass
public abstract class Project {
  @Id
  private long id;
  @Column(name=”NAME”)
  private String name;
  . . .
}
@Entity
@Table(name=“LARGEPROJECT”)
@AttributeOverride(name="name",
column=@Column(name="NAME"))
public class LargeProject extends
Project {
  private BigDecimal budget;
}

@Entity
@Table(name=“SMALLPROJECT”)
public class SmallProject extends
Project {
}
```

```
<mapped-superclass  class="entity.
Project">
  <attributes>
    <id name="id"/>
    <basic name="name">
       <column name="NAME"/>
     </basic>
 </attributes>
</mapped-superclass>

<entity class="entity.
LargeProject" name="LargeProject">
  <table name="LARGEPROJECT"/>
  <attribute-override>
     <column name="NAME"/>
  </attribute-override>
</entity>

<entity class="entity.
SmallProject" name="SmallProject">
   <table name="SMALLPROJECT"/>
</entity>
```

***Table 3.13**: Mapped superclass JPA annotations and XML configurations.*

Locking

Locking and concurrency is a critical issue for most applications that should be considered in the development life cycle itself. However, most applications tend to ignore about these during the initial days, and finally end up integrating some locking mechanism just before going to production. This probably could be one of the reasons why a large percentage of software projects fail or get cancelled.

Locking strategy is very decisive in an application where concurrent writes to same objects are performed. Data corruption can be prevented only if this is well thought out and implemented. There are two strategies for the same:

1. optimistic locking
2. pessimistic locking

JPA has support for version optimistic locking, but some JPA providers support other methods of optimistic locking, as well as pessimistic locking. Correctly implementing locking in the application typically involves more than setting some JPA or database configuration option. Locking also involves application level changes, and makes sure that other applications accessing the database also take proper steps with respect to the locking policy being used.

Optimistic locking

In optimistic locking, the object is not locked when it is accessed for the first time in the transaction, instead, its state is saved. When other transactions that are accessing the same object try to modify the state of the object, the present state and the saved state are compared. If the states differ, then it's a clear indication of a conflicting update, and the transaction will be rolled back.

JPA supports optimistic locking using version field, which can be a number or timestamp, that gets updated on each update. A numeric value is suggested as it is more precise, portable, efficient, and easier to deal with than a timestamp. The **`@Version`** annotation or **`<version>`** element is used to define this optimistic lock field.

```
@Entity
public class Student {
  @Id
  private long id;
  @Version
  private long version;
}
```

```
<entity class="entity.Student"
name="Student">
  <attributes>
     <id name="id"/>
     <version name="version"/>
   ……
  </attributes>
</entity>
```

***Table 3.14**: Optimistic locking using version JPA annotations and XML configurations.*

As shown previously, the version annotation is used similar to that of an Id annotation. This version attribute is automatically updated by the JPA provider and not by the application. Normally, the Entity Manger **`merge()`** API will make sure that the merging of the version is also done. However, if the merging is done manually, then the application should take care of the merging of version as well.

Whenever a locking conflict happens, an **`OptimisticLockException`** will be thrown which could be wrapped inside any other exceptions, but the cause of the exception should be set properly. Only then the application can report the error to the user and allow them to determine what has to be done next.

Pessimistic locking

In pessimistic locking, the object is locked when it is initially accessed for the first time in a given transaction. The lock then is released only when the transaction completes; the object is not accessible for any other transactions during the transaction. Hence, pessimistic locking ensures that no other transactions can modify or delete the reserved data.

Version 1.0 of the JPA specification, which offered just the fundamental features of an **object relational mapping** (**ORM**) framework, supported only optimistic locking. JPA 2 added pessimistic locking support, bringing it more in line with the locking features in the Hibernate ORM framework. All the locking modes available in JPA are as follows:

- **`READ (JPA1)`**/**`OPTMISTIC (JPA2)`**
- **`WRITE (JPA1)`**/ **`OPTIMISTIC_FORCE_INCREMENT (WRITE in JPA2)`**
- **`PESSIMISTIC_READ (JPA2)`**
- **`PESSIMISTIC_WRITE (JPA2)`**
- **`PESSIMISTIC_FORCE_INCREMENT (JPA2)`**
- **`NONE`** – only lock mode provided outside a transaction

While **`PESSIMISTIC_READ`** mode generally represents a shared lock, **`PESSIMISTIC_WRITE`** represents an exclusive lock. In the former mode, the lock is obtained on an entity as soon as the transaction begins. It is best when the data accessed is not frequently modified, as it allows other transactions to read the entity. While, the latter mode is preferably the best when there is a high probability of update failure due to multiple transactions accessing the object.

`PESSIMISTIC_FORCE_INCREMENT` mode locks an entity when a transaction reads the entity. The version number is incremented towards the end of the transaction, irrespective of whether the entity was updated or not. This option is made for occasions where the combination of pessimistic and optimistic mechanisms is to be used.

Transaction isolation levels: Transaction isolation has to ensure the consistency of the system. There are four levels of transaction isolation as follows:

1. `READ_UNCOMMITTED`
2. `READ_COMMITTED` (protecting against dirty reads)
3. `REPEATABLE_READ` (protecting against dirty and non-repeatable reads)
4. `SERIALIZABLE` (protecting against dirty, non-repeatable reads, and phantom reads)

Most databases default to `READ_COMMITTED` transaction isolation. This means that there will be no uncommitted data.

Common mistakes or problems

Applying the lock: The most common mistake made while locking in general is locking the wrong section of code. This is true in both the cases of locking, whether it be optimistic or pessimistic. The basic scenario is:

1. User requests some data, the server reads the data from the database and sends it to the user in their client [doesn't matter if the client is html, **remote method invocation** (**RMI**) or web service].
2. The user edits the data in the client.
3. The user submits the data back to the server.
4. The server begins a transaction, reads the object, merges the data, and commits the transaction.

The issue is that the original data was read in Step 1, but the lock was not obtained until Step 4. So, any changes made to the object in between steps 1 and 4 will not result in a conflict. Hence, there is no point in locking as the data in hand when obtaining the lock would not be the actual one available in the database.

In case of pessimistic locking, the database locks will only occur in Step 4, and any conflicts will not be detected. Hence, this becomes the main reason for not using database locking where scalable web applications are considered. For the locking to be valid, the database transaction must be started at Step 1 and not committed until Step 4. This means that a live database connection and transaction must be held open while waiting for the web client, along with the locks, and there is no guarantee that the web client will not sit on the data for hours holding database resources and locking data for all other users that is highly undesirable. Hence, pessimistic locking should not be considered in such scenarios.

For optimistic locking, the solution is simple. The version must be sent to the client along with the data (or kept in the http session). When the user submits the data

back, the original version must be updated into the object read from the database, to ensure that any changes made between steps 1 and 4 will be detected.

Optimistic lock exception handling: The first issue that comes up when using optimistic locking is what to do when an **`OptimisticLockException`** occurs. The typical response is to automatically handle the exception (create a new transaction, refresh the object and merge data back and re-commit), which actually defeats the whole point of locking. **`OptimisticLockException`** should not be auto-handled, and the conflict should be reported to the user. Either convey that there is an edit conflict or refresh the object and present the user with the current data and the data that they submitted, and help them merge the two if appropriate.

Overemphasis of locking: Locking can prevent most concurrency issues, but be careful of going overboard in over analyzing every possible hypothetical occurrence. To maintain flawless concurrency, code developers provide locks on every possible related object, which would probably be very expensive, and more importantly raise possible conflicts every time data is access/modified, and hence would be entirely useless. So, be careful of being too paranoid, such that the usability of the system is sacrificed.

Other applications accessing the same data: Any form of locking that is going to work requires that all applications accessing the same data follow the same rules. If optimistic locking is used in one application, but no locking in another accessing the same data, they will still conflict. The old application can read data (without locking), then update the data after the new application reads, locks, and updates the same data, overwriting its changes.

Concurrent transactions overwriting data: Consider the following transactions:

1. Transaction A reads row x.
2. Transaction B reads row x.
3. Transaction A writes row x.
4. Transaction B writes row x (and overwrites A's changes).
5. Both commit successfully.

Serializable transaction isolation would prevent the preceding issue and is explained in detail in the upcoming section.

Merge an object that was deleted by another user: Ideally, the merge should trigger an **`OptimisticLockException`** because the object has a version that is not null and greater than 0, and the object does not exist. But this is probably JPA provider specific, or throw a different exception. In case, if persist is being called, the object might get re-inserted.

Table without a version column: The best solution is just to add one. Field locking is another solution as well, which will be described in detail in the upcoming section.

Version in each table for inheritance: There should be only version information in the root table and not in all the multiple tables provided for the inheritance.

Advanced locking

There are various other advanced locking techniques that are supported by JPA.

Timestamp locking

Timestamp version locking is supported by JPA with the attribute type as **`java.sql.Timestamp`** or other date/time type. However, the configurations used are the same as that of the numeric version locking. This kind of locking is frequently used in tables that already have a last updated column, and can be more useful since it includes the relevant information on when the object was last updated.

However, timestamp locking should be used cautiously as there are different levels of timestamp precision in various databases. Some of them do not store timestamp's milliseconds, while some other do not store them precisely. Hence, in general, time stamp locking is less efficient than numeric-version locking.

Cascaded locking

Locking objects is more than locking rows in the database. An object can be besides a simple row; an object can span multiple tables, have inheritance, have relationships, and even have dependent objects. So, determining when an object has *changed* and needs to update its' version can be further difficult than determining when a row has changed.

JPA defines that when any of the object's tables changes, the version is updated. However, it is less clear on relationships. Relationships like Basic, Embedded, or a foreign key are clear on changes, as the version will be updated uniquely. But this is not very clear when the relationship has multiple children for a single parent row. Consider the following example:

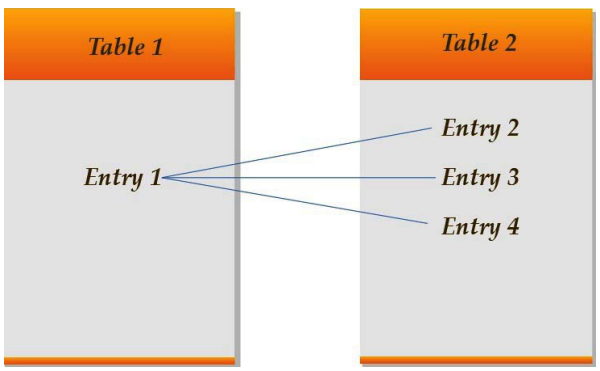

***Figure 3.1**: A single entry is related to multiple entries in another table*

In the preceding scenario, the version of **Entry1** is updated whenever there is a change in either **Entry2**, **Entry3** or **Entry4**. i.e. **Entry2**, **Entry3** and **Entry4** may be on a lower version number than **Entry1** as it gets incremented on every update being carried out for every update on **Table2** entries. Hence, the version update for relations is not so clear, and hence depends on how the JPA provider implements the same.

Note that JPA does not have a cascade option for locking, and has no direct concept of dependent objects, so this is not an option. One way to simulate this is to use write locking and define a version only in the root parent object (here **Table1**), and whenever a child is changed, the lock API is called on the parent as well and the version updated.

If the application considers one user updating one dependent part of an object, and another user updating another part of the object to be a locking contention, then cascading locking is to be used. One of the advantages of cascaded locking is only fewer version fields to maintain, and only the update to the root object needs to check the version. However, this can make a difference in optimizations such as batch writing, as the dependent objects may not be able to be batched if they have their own version that must be checked.

Field locking

Field locking involves comparing certain fields in the object while updating, and if those fields have changed, then the update fails. This can be used when a finer level of locking is desired. For example, if one user changes the object's name and another changes the object's address, these updates should not conflict, and only desire optimistic lock errors when users change the same fields. Only these conflicts should be concerned and any other field conflicts should not be of issue in this scenario. This is also the solution for tables where the version field is missing. For example, legacy schemas can use this when a new column cannot be added for locking.

There are several types of field locking:

- **All fields compared in the update**: This can lead to an extremely big `WHERE` clause, but will detect any conflicts.
- **Selected fields compared in the update**: This is useful if conflicts in only certain fields are desired.
- **Changed fields compared in the update**: This is useful if only changes to the same fields are considered to be conflicts.

JPA does not support field locking by default. However, there are a few JPA providers that support this. If JPA provider does not support field locking, it is difficult to simulate, as it requires change to update the SQL. The JPA provider may allow overriding the updated SQL, in which case, All or Selected field locking may be

possible. But changed field locking is more difficult because the update must be dynamic.

Another way to simulate field locking is to flush the changes, then refresh the object using a separate Entity Manager, and connection and compare the current values with the original object. While using field locking, it is important to keep the original object that was read.

Read and write locking

JPA supports read and write locks by means of EntityManager's **`lock()`** API. The L**`ockModeType`** argument can either be **`READ`** or **`WRITE`**. A **`READ`** lock will assure that the object's state does not change on commit. A **`WRITE`** lock will prevent any other transaction from reading, writing, or deleting the object. Typically, the **`READ`** lock checks the optimistic version field, while the **`WRITE`** checks and increments it.

Normally, when making a change to one object that is based on the state of another object, it is preferred to ensure that the other object represents the current state of the database at the point of the commit. Optimistic read and write locking allow this requirement to be met explanatorily and favorably (and without deadlock, concurrency, and open transaction issues).

No locking (Ostrich locking)

This is probably the most common form of locking in use, although it seems to be very ridiculous. Most prototypes or small applications frequently do not have the requirement, or do not have the need for locking. Handling what to do when a locking conflict occurs is beyond the scope of the application, so it is best to just ignore the issue.

Locking - Best practices

The main advantage of pessimistic locking is that once the lock is obtained, it is fairly certain that the edit will be successful. This can also be termed as an issue as it uses database resources, and require a database transaction and connection to be held open for the entire duration of the activity done by the user. Hence, this becomes highly undesirable for interactive web applications. Another potential problem with pessimistic locking is concurrency issues and causing deadlocks.

It is perhaps best to enable optimistic locking always in JPA, as it is fairly simple to do, at least in concept. However, in highly concurrent applications, pessimistic locking can be more successful as optimistic locking can cause too many locking errors.

Conclusion

In this chapter, the primary operations like identity, sequencing, and locking have been discussed. To bring clarity to these, a few more topics were also discussed like inheritance.

In the next session, different relationships that can be formed using these operations are detailed out. It also mentions the various strategies that are found in JPA.

Multiple choice questions

1. **An identity column in a table can be denoted with which annotation:**
   a. @Id
   b. @EmbeddedId
   c. @IdClass
   d. all of the above

2. **The most commonly used locking technique is:**
   a. ostrich locking
   b. pessimistic locking
   c. optimistic locking
   d. cascaded locking

3. **The default transaction isolation provided by most of the JPA providers are:**
   a. READ_COMMITTED
   b. READ_UNCOMMITTED
   c. REPEATABLE_READ
   d. SERIALIZABLE

4. **Which of the following statements are true?**
   i) Timestamp locking is not supported by JPA.
   ii) Cascade locking is supported by JPA.
   iii) Field locking is not supported by JPA.
   iv) Optimistic locking is supported by JPA.

a. i, and ii only
b. ii, iii and iv only
c. iii and iv only
d. None of the above

5. **Which locking technique was introduced in JPA2?**
   a. pessimistic locking
   b. field locking
   c. cascade locking
   d. read locking

Answers

1. **d**
2. **a**
3. **b**
4. **c**
5. **a**

Questions

1. Explain the transaction isolation values and their usage.
2. Describe the common issues or doubts that crop up while implementing locks.
3. Explain various sequencing strategies along with examples.

Points to ponder

1. ID should never be changed for an object. This can result in errors or strange behavior depending on the JPA provider.
2. **TopLink/EclipseLink**: It provides a ReturningPolicy that allows for any field values including primary key to be returned from the database after an insert or update. This is defined through the `@ReturnInsert`, `@ReturnUpdate` annotations, or the `<return-insert>`, `<return-update>` XML elements in the `eclipselink-orm.xml`.

3. **TopLink/EclipseLink**: A UnaryTableSequence allows a single column table to be used. A QuerySequence allows for custom SQL or stored procedures to be used. An API also exists to allow a user to supply their own code for allocating ids.

4. **Hibernate**: A GUID id generation option is provided through the **`@GenericGenerator`** annotation.

5. Composite primary keys are common in legacy database schemas, where cascaded keys can sometimes be used. Other common usages of composite primary keys include many-to-many relationships or aggregate one-to-many relationships.

6. If object does not have an id, but its table does, then the object can be made an Embeddable object; embeddable objects do not have ids. An Entity that contains this Embeddable to persist and query it can also be created.

7. TopLink and EclipseLink support both the querying mechanisms that are used in joined multiple table inheritance. The multiple query mechanism is used by default. Outer joins can be used instead through using a **`DescriptorCustomizer`** and the ClassDescriptor's InheritancePolicy's **`setShouldOuterJoinSubclasses()`** method.

8. TopLink and EclipseLink support computing the inheritance discriminator through Java code. This can be done through using a Descriptor Customizer and the Class Descriptor's Inheritance Policy's **`setClassExtractor()`** method.

9. In Hibernate, the inheritance discriminator column's purpose can be accomplished through using the Hibernate **`@DiscriminationFormalua`**. This also allows DB specific SQL functions to be used to compute the discriminator value.

10. The timestamp value in timestamp version locking can either come from the database, or from Java (mid-tier). JPA does not allow this to be configured, however some JPA providers may provide this option. Using the database's current timestamp can be very expensive, as it requires a database call to the server.

11. TopLink and EclipseLink support cascaded locking through annotations (and corresponding XML elements of course) like **`@OptimisticLocking`** and **`@PrivateOwned`**. They also support field locking using **`@OptimisticLocking`** annotation.

12. The popular myth is that the ostrich buries its head in sand when it senses danger. Hence, the no-locking strategy is also called ostrich strategy as it ignores the locking issue completely, although it is aware of it.

13. Some JPA providers support joined inheritance with or without a discriminator column. Some require the discriminator column, and some do not support the discriminator column. So. joined inheritance does not seem to be fully standardized yet.

14. JPA 2.0 has two new standard query hints which can be passed to any Query, NamedQuery, or **`find()`**, **`lock()`** or **`refresh()`** operation.

    • **`javax.persistence.lock.timeout`**—Number of milliseconds to wait on the lock before giving up and throwing a **`PessimisticLockException`**.

    • **`javax.persistence.lock.scope`**—The valid scopes are defined in **`PessimisticLock- Scope`**, that can have values **`NORMAL`** or **`EXTENDED`**

Chapter 4
Relationships – Types and Strategies

Introduction

The previous chapter discussed more about how data can be populated into the tables via operations. It is equally relevant to know how each set of data can be related to one another. Data level relationships become important in any application as they portray most of the business-level relationships, and hence help in forming the required information for the application user. What data should be related, how should they be related, and till what level, all these determine the various aspects of the relevance of the data for that particular application. Hence, relationships form the crux of any RDBMS based application.

Structure

This chapter discusses the following with relevant examples in each section:

- Types of relationships:
  - OneToOne
  - OneToMany / ManyToOne
  - ManyToMany

- Element collections
- Maps
  - Map key columns (JPA 2.0 feature)
- Nested collections
- Query optimization – Fetching techniques
  - Lazy, Join, Batch
- Variable and heterogenous relationships
- Cascading
  - Orphan removal (JPA 2.0 feature)
- Target entity, nested joins
- Collections
  - Order column (JPA 2.0 feature)

Objectives

This chapter goes into the details of the various types of relationships that are available in JPA, and how they work in the relational DB. Also, based on the relationships, query optimization techniques like join fetching and batch fetching are discussed. This chapter also enables the reader to understand the new features available as part of JPA 2.0 like orphan removal, order column, and so on.

Relationships

A relationship is simply a reference from one object to another. In relational databases, keys (primary and foreign) mainly define the relationship. The primary key of a source row becomes the foreign key in the target row (or vice versa). Thus, relationships are described in such a way that the inverse query always exists and it is essential to have the key information to retrieve the objects of the relationship. Hence, the relational databases always have bidirectional relationships.

In Java, these relationships are represented through object references (pointers) from one object to another. However, all the object reference pointers cannot be determined as relationships. The references to data attributes (which themselves are objects like String, Date etc.) are considered as part of the object itself and references to other persistent objects are treated as relationships. If a relationship maps to a collection of objects, a collection or array type is used to hold the contents of the relationship. Hence, all relationships in Java/JPA are unidirectional, i.e., if a source object holds a relation to a target object, there is no guarantee that the target object also has a

relation with the source. In order to handle bidirectional DB relationships using JPA, it becomes important to associate the source with the target in both directions.

JPA relationship types

This section discusses the majority of the types of relationships that exist in most object models. Each type of relationship can be implemented in different ways.

- **OneToOne**: a unique affiliation from one object to another. Its inverse is also a OneToOne relation.
- **ManyToOne**: a reference from many objects to a unique object. Its inverse is a OneToMany relation.
- **OneToMany**: a collection or map of objects, inverse of a ManyToOne relation.
- **ManyToMany**: a collection or map of objects, inverse of which is also a ManyToMany relation.
- **Embedded**: a reference to an object that shares the same table of the parent.
- **ElementCollection**: JPA 2.0, a collection or map of basic or embeddable objects, stored in a separate table.

In general, JPA relations can be classified broadly as bidirectional and unidirectional. In bidirectional relationship, the navigational access as well as cascading options are provided in both directions, and hence no explicit query is required. In case of unidirectional relationships, navigation is possible only in one direction and based on which part manages the foreign key, the direction is decided. For example, in a ManyToOne relationship, the access is from the child to parent since the child table will have the foreign key. However, in case of OneToMany relationships, the access is from parent to child since parent manages the foreign key.

OneToOne relationship

As the name suggests, one record in a table is related uniquely to another record in a different table. As per RDBMS nomenclature, this relationship is shown by having unique foreign keys in one table. For example, consider a scenario where passengers

are allocated seats and no two passengers can have the same seat. This has been diagrammatically shown as follows:

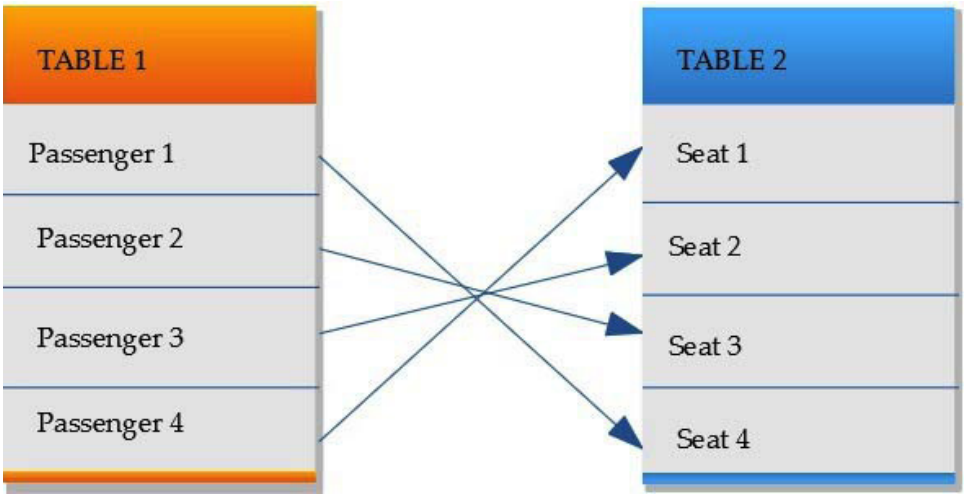

***Figure 4.1**: One to One relationship between the passenger table and seat table*

Here, **TABLE1** has the foreign key (i.e. the primary key of **TABLE2**) which uniquely identifies the seat being used by the passenger. In JPA, the foreign key column name in **TABLE1** is formed as the concatenation of the column name of TABLE1_Primary key column name of **TABLE2**. The foreign key column of **TABLE1** has the same type as that of the primary key of **TABLE2** and a unique constraint is formed on this column. This assures that all the entries in the foreign key column are unique and hence the relationship becomes OneToOne.

Bidirectional OneToOne relationships need declarations on the code from both sides, whereas unidirectional relationships only require declarations to be at the relationship owner's side. Consider the example above being represented in JPA.

```
@Entity
public class Passenger {
 private Seat seat;

 @OneToOne
 public Seat getSeat () { return seat; }
 public void setSeat (String passSeat) { this.seat = passSeat; }
}

@Entity
```

```
public class Seat {
 private Passenger passenger;

 @OneToOne(mappedBy=”seat”)
 public Passenger getPassenger () { return passenger; }
 public void setPassenger (Passenger pass) { this.passenger = pass; }
}
```

***Code 4.1**: Bidirectional OneToOne relationship in JPA*

It is obvious that from both directions the other entity object can be retrieved from the above code, i.e., from **Passenger** object, the seat object can be obtained and from the **Seat** object, the **Passenger** object. This makes it a complete bidirectional mapping.

Now, consider the following code that also works well for OneToOne relationships. Here, it is pretty evident that the **Passenger** has access to the **Seat,** but vice versa is not possible. Therefore, it is a unidirectional case of OneToOne mapping in JPA.

```
@Entity
public class Passenger {
 private Seat seat;
 @OneToOne
 public Seat getSeat () { return seat;}
 public void setSeat(String passSeat){  this.seat = passSeat; }
}

@Entity
public class Seat {
. . .
}
```

***Code 4.2**: Unidirectional OneToOne relationship in JPA*

In unidirectional relations mapped by attribute is not used. Due to the absence of this information, there is a requirement of an additional join table of foreign keys which hampers performance. So, by using bidirectional relations, there may be an increase in performance, but it can cause an impact on security at JPA level since both the objects are accessible from either side. There are many real-world scenarios where both these kinds of relationships are suited, and hence should be used wisely by the JPA developer.

OneToMany/ManyToOne relationship

OneToMany relationship is a single record in one table being mapped to multiple records in a different table. In a bidirectional OneToMany relationship, the inverse relationship becomes ManyToOne. For example, consider a school with many classes and its students. Each standard will have multiple students, but none of the students can be assigned to more than one class. So, from the standard to student, it is a OneToMany relationship but in reverse direction it is a ManyToOne relationship.

```
@Entity
public class Standard {
  private Collection<Student> students;

  @OneToMany(mappedBy="std")
  private Collection<Student> getStudents(){
    return students;
  }
  private void setStudents(Collection<Student> studs){
    this.students = studs;
  }
}

@Entity
public class Student {
  private Standard std;

  @ManyToOne
  public Standard getStandard() {  return std; }
  public void setStandard(Standard studStd){  this.std = studStd; }
}
```

***Code 4.3**: Bi-directional OneToMany/ManyToOne relationship in JPA*

In this case, the list of students can be accessed from the **`Standard`** entity as well as the standard of a student can be access via the Student entity. **`Student`** is considered to be the owner of this relationship. Also, while storing in DB, the foreign key in the **`Student`** table is stored as **`STANDARD_<Primary key of Standard table>`**. Hence, for bidirectional OneToMany relations, there is no need of an additional join table to get the additional details. But there can be cases where every relation need not

be bidirectional. In case of the example above, suppose there can be a OneToMany relationship between the student and the grades obtained in each subject. A student can have multiple grades and it is not necessary that from the grade object the details of the students need to be accessed.

```
@Entity
public class Student {
  private Collection<Grade> grades;

  @OneToMany
  private Collection<Grade> getStudGrades (){  return grades; }
 private void setGrades(Collection<Grade> sGrads){ this.grades = sGrads; }
}

@Entity
public class Grade {
……
}
```

***Code 4.4** : Uni-directional OneToMany relationship in JPA*

In the example above, one can only trace the grades from the student object. It is not possible to get it done from the grades object since it does not store any reference to the student details. Hence, this is a typical unidirectional OneToMany relationship. To achieve this behavior, JPA uses an additional join table named **`Student_Grade,`** with two foreign key columns. One foreign key column would be **`STUDENT_<Primary key of STUDENT>`** and of the same type as that of the Primary key of Student table, and the other foreign key column would be **`GRADE_<Primary key of GRADE>`** with the type as that of the primary key of **`Grade`** table.

Consider when every **`Student`** entity provides the details of the class teacher. But it is not necessary to have the list of students that needs to be accessed by the teacher who is assigned as the class teacher for a particular standard. It is only necessary to know which standard has been assigned to a particular teacher to get to know the list of students for whom that teacher would be assigned to as the class teacher.

```
@Entity
public class Student {
 private Teacher cTeacher;
```

```
 @ManyToOne
 public Teacher getClassTeacher () {return cTeacher;}
 public void setClassTeacher (Teacher tcr) {this.cTeacher = tcr;}
}

@Entity
public class Teacher { … }
```

***Code 4.5**: Uni-directional ManyToOne relationship in JPA*

There are cases where only unidirectional ManyToOne relationships are required.

In such cases, similar to bidirectional OneToMany/ManyToOne relationships, the foreign key is stored in the table which defines the relationship. Note that no additional join table is required for a unidirectional ManyToOne relationship.

ManyToMany relationship

In this relationship, one record in a table can be related to multiple records in a different table and the inverse is also true. For example, consider the use case of students taking up online courses. Each course can have multiple students and there are no restrictions that the student should only take a single course. Hence, this becomes a ManyToMany relationship bidirectionally.

```
@Entity
public class Student {
 private Collection<Course> courses;

 @ManyToMany
 public Collection<Course> getCourses () {return courses;}
  public void setCourses (Collection<Course> sCourses) {this.courses =
sCourses;}
}

@Entity
public class Course {
  private Collection<Student> students;
```

```
  @ManyToMany(mappedBy=”courses”)
  private Collection<Student> getStudents () {return students;}
  private void setStudents (Collection<Student> studs) {this.students =
studs;}
}
```

Code 4.6: *Bi-directional ManyToMany relationship in JPA*

In order to implement bidirectional ManyToMany relationship, a new join table is created by JPA with table name as **Table1_Table2,** where **Table1** and **Table2** are the entities in the relation and **Table1** is the owner of the relation. This join table will have two columns, one with the primary key of **Table1** and the other with the primary key of **Table2**. As per the above example, the join table will be named as **STUDENT_COURSE** since **STUDENT** table is the owner of this ManyToMany relation. The join columns will be **STUDENT_<PrimaryKey of STUDENT>** and **COURSE_<PrimaryKey of COURSE>.**

There are cases where only one directional ManyToMany relation is of interest. In such cases, unidirectional ManyToMany mapping is used in JPA. However, in this case as well, a join table is created as mentioned above.

```
@Entity
public class Student {
 private Collection<Publication> publications;

 @ManyToMany
 public Collection<Publication> getPublications() {return publications;}
 public void setPublications(Collection< Publication > pubs){
   this.publications = pubs;
 }
}

@Entity
public class Publication {}
```

Code 4.7*: Uni-directional ManyToMany in JPA*

In the example above, there can be many students involved in a publication and vice versa, a student can have multiple publications. However, the publications can be retrieved via the **Student** entity only. This is because only the uni-directional relationship is considered. Similar to the bidirectional relationship, a join table is

created by JPA to store the ManyToMany mapping keys.

Mapping a join table with additional columns: A frequent problem is that two classes have a ManyToMany relationship, but the relational join table has additional data. Suppose that **`Employee`** table has ManyToMany relation with Project, however, the join table also has an additional column that mentions whether the employee is a lead or not (column named as **`IS_PROJECT_LEAD`**). The ideal solution for this kind of models would be to create a class for the join table. The class can have a many to one relation with **`Employee`** as well as **`Project`** tables along with the attributes for additional data.

There are a few JPA providers with additional support for mapping to join tables with additional data. But this mapping creates more complications since the primary key is composite i.e., the Employee ID as well as Project ID has to be considered as the primary key and hence they require an **`IdClass`** instead of an ID.

Tip: It is always better to add a generated ID attribute in the association class and hence avoid the usage of a composite primary key and usage of IdClass.

There can be cases where the relationship between two objects are mapped using a key which is another object unrelated to the above mapped two objects. So, in that case, the association object can have the map key as an attribute in the class used to model the join relationship (as per JPA specifications). In case, if the additional data is only required in the DB and not by the Java program, there can be triggers set to update the DB automatically.

Embedded relationship

An entity can be represented as using other entity classes which otherwise does not have an existence of their own. This kind of relationship is called an embedded relationship and has been discussed in detail in *Chapter 2, Tables – Attributes and Embeddable Objects* (section named Embeddables).

As discussed earlier, embeddable objects strictly belong to the owning entity and cannot be shared with other persistence entities. However, an embeddable class can have a relationship to an entity or a collection of entities. This means that there can be a relationship between an entity and the entity that owns the embeddable object.

Element collections

This type of collections are only available in JPA 2.0 and is expected to handle non-standard relationships. An element collection is defined by the annotation **`@ElementCollection`** (**`<element-collection>`** in XML) and can be used to describe a OneToMany relationship to either an **`Embeddable`** object or a **`Basic`** object. It can also be used along with a Map to define a relationship where key can be of any type

and values can be either an **`Embeddable`** object or a **`Basic`** object.

Tip: ElementCollections are always stored in a separate table and this table is defined using the annotation `@CollectionTable` (`<collection-table>` in XML). In `@CollectionTable`, name of the table and the join column(s) are defined (`@JoinColumns` are used when there is a composite primary key).

When **`ElementCollection`** mapping is used to define a collection of **`Embeddable`** objects, it is similar to a OneToMany, however, the target object is an Embeddable instead of an Entity. In this way, simple objects are defined without defining an Id or ManyToOne inverse mapping. Yet, this relation has a limitation that the target objects cannot be independently queried, persisted, or merged. They are always operated along with the parent object i.e., persisted, merged, and removed along with their parent. Although cascade option is not available, the fetch type defaults to LAZY as other collection mappings.

```
@Entity
public class Student {
 @Id
 @Column(name=”STUD_ID”)
 private long studentId;

 @ElementCollection
 @CollectionTable(name=”PUBS”, joinColumn=@JoinColumn(name=”PUB_ID”))
 private List<Publication> publications;
}

@Embeddable
public class Publication {

 @Column(name=”PUB_ID”)
 private String id

 @Column(name=”TITLE”)
 private String title;

 @Column(name=”CONF”)
 private String acceptedConf;

 @Column(name=”DATE”)
 private Date dateOfPub;
}
```

***Code 4.8**: ElementCollection using Embeddable objects*

An **ElementCollection** can also be used to define a collection of **Basic** objects as well. In the example above, the table has multiple columns like **TITLE**, **CONF**, **DATE**, and so on. Hence, the **Embeddable** object is being used. Instead, if the table had only an id and title of publication, a basic object is enough.

```
@Entity
public class Student {

 @Id
 @Column(name="STUD_ID")
 private long studentId;

 @ElementCollection
 @CollectionTable(name="PUBS", joinColumn=@JoinColumn(name="PUB_ID"))
 private List<String> pubTitles;
}
```

***Code 4.9**: ElementCollection using Basic objects*

Here, the publication titles are read as String (Basic object) since there are no other columns in the table **PUBS**.

Maps

In Java context, **java.util.Map** is a kind of collection which stores a key value pair where the value can be an object or yet another collection. Although few JPA providers support Map implementations, in broader sense JPA expects the usage of Map interface as an attribute type for the collection types like OneToMany, ManyToMany or ElementCollection. In JPA 1.0, map key is defined as the attribute taken from the target object which specifies the relationship with the target object.

The annotation **@MapKey** (in XML **<map-key>**) is used to achieve the definition above. If not provided, the target object's primary key is used as the map's key value.

```
@Entity
public class Person {
 @Id
 private long id;
```

```
 @OneToMany(mappedBy=”owner”)
 @MapKey(name=”type”)
 private Map<String, PhoneNumber> phoneNumbers;
}

@Entity
public class PhoneNumber {
 @Id
 private long id;

 @Basic
 private String type;

 @ManyToOne
 private Person owner;
}
```

***Code 4.10**: MapKey example in JPA 1.0*

The preceding example shows the object model of corresponding two DB tables, **`Person`** and **`PhoneNumber`**. A person can have multiple phone numbers based on the type (i.e., work, home, mobile, landline etc.) and hence there is a OneToMany relationship from **`Person`** to **`PhoneNumber`**. Here, the key of the map representing the OneToMany relation is the type of the phone. That is why the type becomes the **`mapkey`** attribute which is part of the **`PhoneNumber`** table and hence specifies the target object's (here **`PhoneNumber`**) column name. If **`@MapKey`** does not provide a column name, then the map would have used the Id of the **`PhoneNumber`** (i.e. the primary key of the table) and hence it would have been difficult to understand what type of phone number it is unless the object inside the map is retrieved and accessed. Hence, **`@MapKey`** enables the user to logically relate to elements based on requirements.

Another annotation used is **`@MapKeyClass`** to specify what class is being used as the key for the map that denotes the relationship. An example usage of this annotation would be like **`@MapKeyClass(String.class)`**. Here, the key object type is defined as String. However, if generics are used, then it is not required to provide this annotation, i.e. if the map is defined as **`Map<String,Object>`**, then there is no need to provide this annotation. Also, note that **`@MapKeyClass`** is never used when **`@MapKey`** is used and vice-versa.

The **@MapKey** in JPA 1.0 had a limitation that the key should be part of the target object. However, in JPA 2.0, the limitation was elevated with the key of the map being either one of the given objects:

- a basic value, stored in the target table or a join table
- an embedded object, stored in the target table or a join table
- a foreign key to another entity, stored in the target table or a join table

The definition above helped in greater flexibility and complexity in the various models that could be mapped in JPA 2.0. The type of the mapping is determined by both the key and the value of the map. Thus, if key is basic and value is an entity, a OneToMany mapping can be used, but if key is an entity and value is basic, an **ElementCollection** mapping is used. Based on the type of keys, different annotations are used.

Annotation	XML tag	Key value
@MapKeyColumn	**<map-key-column>**	Basic
@MapKeyEnumerated	**<map-key-enumerated>**	Enum
@MapKeyTemporal	**<map-key-temporal>**	Calendar
@MapKeyJoinColumn	**<map-key-join-column>**	Entity (primary / foreign key)
@MapKeyJoinColumns	**<map-key-join-columns>**	Composite foreign keys
@MapKeyClass	**<map-key-class>**	Embeddable

***Table 4.1**: Various annotations for mapping key attributes in a relationship*

The table above provides a snapshot view of how to provide the annotations based on the type of relationships being defined in JPA 2.0. Below, a few of the above-mentioned annotations are explained via examples.

```
@Entity
public class Person {
 @Id
 private long id;

 @OneToMany(mappedBy=”owner”)
 @MapKeyColumn(name=”PHONE_TYPE”)
 private Map<String, Phone> phoneNumbers;
}
@Entity
```

```
public class Phone {
 @Id
 private long id;
 @ManyToOne
 private Person owner;
}
```

***Code 4.11**: MapKeyColumn example in JPA 2.0*

In the example above, there are two tables–**Person** and **Phone**. Since, in this case, the key used is a Basic value (**PHONE_TYPE** column from **Phone** table which is a String) OneToMany relationship is defined.

```
@Entity
public class Person {
 @Id
 private long id;

 @OneToMany(mappedBy=”owner”)
 @MapKeyJoinColumn(name=”PHONE_TYPE_ID”)
 private Map<PhoneType, Phone> phoneNumbers;
}
@Entity
public class Phone {
 @Id
 private long id;
 @ManyToOne
 private Person owner;
}
@Entity
public class PhoneType {
 @Id
 private long id;
 @Basic
 private String type;
}
```

***Code 4.12** MapKeyJoinColumn example in JPA 2.0*

In the example above, there are three tables – **Person**, **Phone** and **PhoneType**. Since the foreign key value (**PHONE_TYPE_ID** column from **Phone** table which is the primary key of **PHONE_TYPE** table) is being used, OneToMany mapping is being used.

```
@Entity
public class Person {
 @Id
 private long id;

 @OneToMany
 @MapKeyClass(PhoneType.class)
 private Map<PhoneType, Phone> phoneNumbers;
}
@Entity
public class Phone {
 @Id
 private long id;
}
@Embeddable
public class PhoneType {
 @Basic
 private String type;
}
```

***Code 4.13**: MapKeyClass example in JPA 2.0*

In the example above, there are two tables – **Person** and **Phone**. Here, an **Embeddable** object is defined (a join table **Person_Phone** in DB) and to denote this as the key value **@MapKeyClass** annotation is used.

Nested collections

In an object model, it is very natural to have complex collection relationships like List of Lists or a Map of Maps or a Map of Lists, etc. But it is very difficult to map this kind of objects into a relational database. Hence, JPA does not support such nested collection relations and hence it is always advisable to have an object model that makes querying and persisting easier via JPA.

However, in cases where these nested relations cannot be avoided, objects that wrap such relations can be created and used.

```
public class Student {
 private long id;
 private Map<String, List<Project>> projects;
}

@Entity
public class Student {
 @Id
 @GeneratedValue
 private long id;

 @OneToMany(mappedBy=”student”)
 @MapKey(name=”subject”)
 private Map<String,ProjectType> projects;
}
@Entity
public class ProjectType {
 @Id
 @GeneratedValue
 private long id;

 @ManyToOne
 private Student student;

 @Column(name=”SUBJECT”)
 private String subject;

 @ManyToMany
 private List<Project> projects;
}
```

***Code 4.14**: Nested Class being wrapped into a different class*

`Student` class has a list of projects based on the subject which is a nested collection. It is basically a map of projects where the subject name is the key and the list of projects done by the student for that particular subject as its value. Since the value is a **`List`**,

a wrapper class **`ProjectType`** is written, which has a ManyToMany relationship with the projects. It also contains the subject name that is used as the key in the nested collection.

Query optimization techniques

In this section, various techniques that are being used in JPA for optimizing the query executions are being discussed. These techniques help in achieving better performance while querying for multiple objects using joins.

Join fetching

This is a query optimization technique in which multiple objects can be read in a single database query. The tables in which multiple objects reside are first joined, and then the required objects are selected from the joined data. This is widely used in OneToOne relationships, and can also be used in other relationships like OneToMany and ManyToMany. This is a solution for the classic ORM n+1 performance problem.

Consider the selection of n Student objects and access their corresponding addresses, which is stored in a different table. In basic ORM (including JPA), one DB select is first done to get the **`Student`** objects and then n selects are done to access each student's address. By using join fetching, the above n+1 selects can be reduced to one select by selecting both the Student and their address using the following query:

```
SELECT stud FROM Student stud JOIN FETCH stud.address
```

The JPQL join fetch syntax above performs a normal inner join. However, the result will not have the list of Students who do not have an address. If the relationship allows null or empty collections, then the outer join fetch below should be done using the **`LEFT`** syntax.

```
SELECT stud FROM Student stud LEFT JOIN FETCH stud.address
```

JPA does not specify whether a join fetch has to be used always for a relationship. It is always mentioned at the query level as join fetch may not be required always in a particular relationship. There is a different support in JPA mappings called **`EAGER`** option for loading the entire relationship in the beginning itself. But this does not mean that the tables are joined. In certain JPA providers, **`EAGER`** is interpreted as join fetch, whereas some other providers support a separate option for always having a join fetch for a relationship.

Eager join fetching

A common illusion regarding **`EAGER`** fetching is that the relationship should be join fetched, i.e., fetched in same **`SQL SELECT`** statement as the source object.

A few JPA providers do implement eager in the aforementioned way. However, just because an object is desired to be loaded, it does not mean that it should be join fetched. In certain cases, the object might have been already read and existing in the persistence context. In those cases, no join fetch is required, and just the reference to the already-existing object needs to be returned. For example, consider Student-Marks relationship. In this case, almost always, the Student is loaded before the Marks instance and hence, the student reference in the Marks' object is made **`EAGER`**. This does not mean that the **`EAGER`** relationship between the **`Marks`** and **`Student`** need to have a join fetch as the Student instance is already available in the persistence context.

Lazy fetching

In real life scenarios, the cost of creating objects becomes more than the cost of executing a fetch. So, it is important to make sure that only required objects are built and thus avoid the cost of creating unwanted objects. Lazy fetching can be considered as a cost optimization technique that allows to delay the fetch of objects in a relationship until they are accessed. The example for **`LAZY`** fetch is given in *Table 2.3* in *Chapter 2 Tables, Attributes and Embeddable Objects*.

In JPA, the fetch attribute on any relationship can be set as **`LAZY`** or **`EAGER`** enums defined in FetchType enum. The default is LAZY, for all relationships except for OneToOne and ManyToOne.

Tip: In general, it is a good idea to make every relationship `LAZY`.

The magic of deferred loading of relationships is done by the JPA using its own proxy implementation of Collection, List, Set or Map implementations. Whenever any method accesses this proxy, the real collection is loaded and returned to the method. Hence, JPA requires to use any of the collection interfaces to define its collection relationships.

In case of OneToOne and ManyToOne relationships, JPA does byte code manipulation of the entity class which allows access to the field or its corresponding get/set methods. When a method accesses the field or invokes the get/set method, it is diverted to first retrieve the relationship and then return the value. For the above-mentioned byte code manipulations, JPA requires an agent or a post-processor and has to be ensured that it is correctly used.

Serialization and Detaching

When an object is serialized or detached, any lazy relationship that was not instantiated before serialization will not be available and will throw an error (or return null depending on the JPA provider). Thus, ensuring that the lazy relationship is available even after serialization or the detaching of an object is a major issue.

The simplest solution to the problem above would be making all the relationships EAGER. However, keeping in mind the cost of object building, there can be steps taken to make sure that all the relationships that are required after serialization should be instantiated upfront. Hence, by marking only the required relationships as EAGER, the cost of building can also be taken care of.

The second solution is to keep the relationships LAZY itself and objectify the relationships before serialization by invoking any method specific to that relation. This approach has an advantage that relationships required can be use case specific i.e., for a certain case one relationship might be required whereas for another case yet another relationship might be required. Both the relationships can be maintained as LAZY and based on the requirement can be instantiated before serialization, and hence avoid the objectification of the case-specific unwanted relations.

The third solution is to use the **`JPQL JOIN FETCH`** while querying the objects for the relationship. This will make sure that the relationship has been instantiated. However, when used on multiple collection relationships, it forms a *n2* join on the database which makes the query inefficient.

In case of detached objects also, if accessed after the end of the transaction, there can be similar issues as that of serialization mentioned above. There are some JPA providers that allow lazy relationships after the end of transaction or after the close of Entity Manager. However, there are JPA providers that don't do the same. In those cases, extra care should be taken by the user to make sure that the lazy relationships are instantiated before detaching the object (either end of transaction or close of entity manager).

Batch fetching

In case of join fetching, a single query was used to solve the ORM *n+1* performance problem. However, in certain cases, the result can be inefficient causing *n2* data sets. When the DB has large data, this can be considered worse than the normal performance (i.e. executing n+1 queries). Hence, a new query optimization technique was considered where a finite set of queries are executed instead of a single query and thus provide a better performance. This query optimization technique is called **batch fetching**. In batch fetching, similar to the normal fetching, the first query is done to get the root objects. Then to fetch the related objects, the original query is joined with the query for related objects and all the related objects are fetched in a single database query. Hence, there are two queries made (in contrast with the join fetching, where only one query is made) which still reduces the number of queries as compared with the normal fetch (*n+1* queries).

Consider the same example that was considered for join fetching in the earlier section. In case of *n* Student objects and their addresses (stored in a different table), the normal fetch takes one query for **`Student`** object and *n* queries for their

corresponding addresses. While using batch fetching, the first query for **`Student`** object remains the same. However, the next query is a join with the **`Address`** table with the selected **`student`** objects. This ensures optimal efficiency especially for reading collection relationships and multiple relationships since no selection of duplicate data as in join fetching.

Tip: JPA does not mandate the support for batch reading, but some JPA providers have this support. TopLink and EclipseLink supports `@BatchFetch` annotation and also an "`eclipselink.batch`" query hint to enable batch reading. `JOIN`, `EXISTS`, and `IN` are the three forms of batch fetching supported.

Variable and heterogeneous relationships

In real world scenarios, there can be cases where a relationship can be defined among several unrelated, heterogeneous values. The related values may share a common interface or may share nothing in common. The relationship can also be any basic value, or an Embeddable.

JPA does not support heterogenous relationships in general. However, some JPA providers support variable relationships and heterogeneous relations as well.

Tips

1. **The most common and easy workaround for providing this relationship support is to have a common superclass for the related values.**
2. **Another way to define this kind of relationship is to provide virtual attributes by get/set methods, and mark these heterogeneous get/set methods as transient.**
3. **Yet another solution is to serialize the value to a binary or to a String from which the value can be restored.**

Cascading

When an operation is performed on an entity, there can be reasons to perform similar operations on the entities related to it. So, it is important to have an option to provide this facility in the relationship mappings. Cascade option in relationship mappings allows the common operations to be performed on the entities related through these mappings as well.

```
@Entity
public class Student {
 @Id
 private long id;
```

```
 @OneToOne(cascade={CascadeType.REMOVE})
 @JoinColumn(name="ADDR_ID")
 private Address address;
}
```

***Code 4.15**: Cascade option for OneToOne relationship*

Consider **`Student`** entity which is related by OneToOne relationship mapping to **`Address`** entity. Whenever, student entity is removed, it doesn't make sense to retain the address entity, and so that should also be removed. Hence, marking the cascade option on delete for the OneToOne relationship makes sure that whenever the student entity is deleted, its corresponding address entity is also deleted.

The common operations that can be cascaded are defined in the **`CascadeType`** enum and they are as given below:

- **PERSIST**: When this option is provided, whenever the parent entity is being persisted, its related entities are also persisted. Note that if this option is not given in the relationship and there is a new object related to the parent, an exception will occur since the new object has not been persisted and hence the relationship cannot be completed. There are not a lot of problems using PERSIST cascade option always other than an impact on the performance.

  Normally, whenever a commit occurs, a relationship with cascade persist option stores both the entity and its related objects. However, if there is a need of any generated IDs from the new related object to be persisted in the parent object, it is always better to call the persist explicitly before the commit so that the parent object gets access to the generated IDs as well.

- **REMOVE**: This option makes sure that the child is also removed when the parent is removed. Note that this should be used only for dependent relationships. In case of a OneToMany relationship, the dependent object will not be removed by removing the parent object. Instead, the parent id will be set as Null in the dependent object. JPA requires an explicit call to remove the dependent object. However, some provides support an option to have the objects removed. JPA 2.0 also defines an option for this.

- **MERGE**: This option allows the child also to be merged whenever the parent object is merged. The relationship reference itself will always be merged and can be an issue when transient variables are used. (Transient variables are used normally to limit serialization). In such cases, either manual merging should be done or transient relationships should be reset.

- **REFRESH**: The refresh option on parent will attempt a refresh option on child as well. This option should be used carefully as it could cause a reset of changes made to other objects as well.
- **ALL**: All the above operations (Persist, Merge, Refresh and Remove) are cascaded with this single option.

Orphan removal (JPA 2.0)

In case of dependent relationships, the related objects cannot exist without the source. So, it is normally desired to have them deleted whenever the source is deleted. However, in JPA 1.0 whenever cascade remove option is set on a OneToMany or OneToOne relationship, the dependent object is not related, instead its parent id column is set to NULL. So, it is required to explicitly call the remove on the dependent object. However, JPA 2.0 provides orphan removal option which ensures that any object no longer referenced from the relationship is deleted from the database.

```
@Entity
public class Person {
 @Id
 private long id;

 @OneToMany(orphanRemoval=true, cascade={CascadeType.ALL})
 private List<PhoneNumber> phoneNumbers;
}

@Entity
public class PhoneNumber {
 @Id
 private long id;

 @Basic
 private String type;

 @ManyToOne
 private Person owner;
}
```

***Code 4.16**: Orphan Removal option set for OneToMany relationship*

In the above example, whenever a **Person** entity is deleted, if **orphanRemoval** option is not set, all the corresponding **PhoneNumber** entities will have the owner object set to NULL. These entries will have to be explicitly removed by executing a remove operation by the Entity Manager. However, by setting **orphanRemoval** to true, all the corresponding **PhoneNumber** entities will be removed instead of setting the owner to NULL. So, no explicit removal of dependent objects required.

Target entity

TargetEntity attribute allows the reference class of the relationship mapping to be specified. This is normally used when the field uses an interface type but the mapping needs to be specified on an implementation class. Also, this is used if the field is a superclass type and the relationship requires to be mapped to its subclass.

Normally, this is not used as it is set from the field type, get-method-return type, and so on.

```
@Entity
public class Person {
 @Id
 private long id;
 @OneToMany(targetEntity=PhoneNumber.class)
 @JoinColumn(name=”OWNER_ID”)
 private List phoneNumbers;
}
```

***Code 4.17**: TargetEntity option set for OneToMany relationship*

In the preceding example, the **List** is an interface and generics is not used to specify the kind of objects in the list. Hence, the **targetEntity** is provided to clarify that the list is of **PhoneNumber** objects.

Nested Joins

Although nested joins are supported by some JPA providers, in general, nested joins are not allowed in JPA using JPQL. Join fetch of multiple relationships is provided in JPA but not nested relationships.

```
SELECT per FROM Person per LEFT JOIN FETCH per.address LEFT JOIN FETCH
per.phoneNumbers
```

Given above is an example JPQL for multiple join fetch. Here, the **Person** entity has relationship with **Address** entity as well as the **PhoneNumber** entity. Both the relationships are fetched using multiple joins.

Duplicate Data and Huge Joins

Duplicate data is one major issue in join fetching. Consider the above join query with multiple joins. For a person, there can be multiple addresses (present address, permanent address, office address, and so on) and also there can be multiple phone numbers (home, work, mobile, and so on). Let the **`Person`** entity have 3 addresses and 3 phone numbers. On fetching this join, there would be 9 entries for each **`Person`** entity (3*3 combinations). Hence, if there are n **`Person`** records, it would create *n**9 records in the join. As the *n* grows, this can be an issue. Normally, whenever the number of queries is reduced, the chances of having duplicate data increases, which can be a serious performance issue in huge table joins. To avoid such issues, an alternate technique is to use batch fetching which has been explained earlier in this chapter.

Filtering, complex joins

There can be instances where a relationship may not be always dependent on the foreign key but on some other conditions. Consider a student relationship to list of projects. The student might have done many projects, but only few of them might be active at present. The actual relationship between **`Student`** and **`Projects`** is through their IDs, but it requires more execution to figure out the list of currently active projects.

However, JPA does not support these types of relationship mappings as it only approves mappings defined by foreign keys and not based on any other constant values, columns, or functions. A simple workaround can be to map the relationship via foreign key and filter the results in the **`get`**/**`set`** methods. Another way is to provide a query for the results instead of defining a relationship.

Collections

Java has its own collection interfaces: Collection, List, Set, or Map. For JPA to provide the relationship mappings that are collections (OneToMany, ManyToOne, ManyToMany and ElementCollection), it requires only the above collection interfaces to define the relationship field. This is important as JPA assumes that only the collections above will be used for relationship mapping as there are many actions provided based on this. Lazy fetching is one such action provided based on this assumption.

While defining the field for a collection relationship, care should be taken not to define a collection implementation as such. The field should always be an interface that is part of the Java collections framework (Collection, List, Set or Map). For example, the relationship field can be a **`List`** but not an **`ArrayList`**. **`ArrayList`** can be used as the instance value. Some JPA providers support the usage of collection implementations as well.

In database, generally duplicates are not supported. However, in a **`JoinTable`** there can be duplicates and JPA does not support duplicates essentially. Hence, there should be provision to support this kind of duplicates. This is done by creating a class denoting the join table itself.

For example, consider a **`Student`** having a ManyToMany relationship with **`Project`**. In the join table, there can be additional information like, whether the **`Student`** is the Project leader or not. i.e. in the Join table which has the **`Project_ID`**, **`Student_ID`** for storing the ManyToMany mapping can also have the additional information like **`Is_Lead`**, **`Project_Duration`**, and so on.

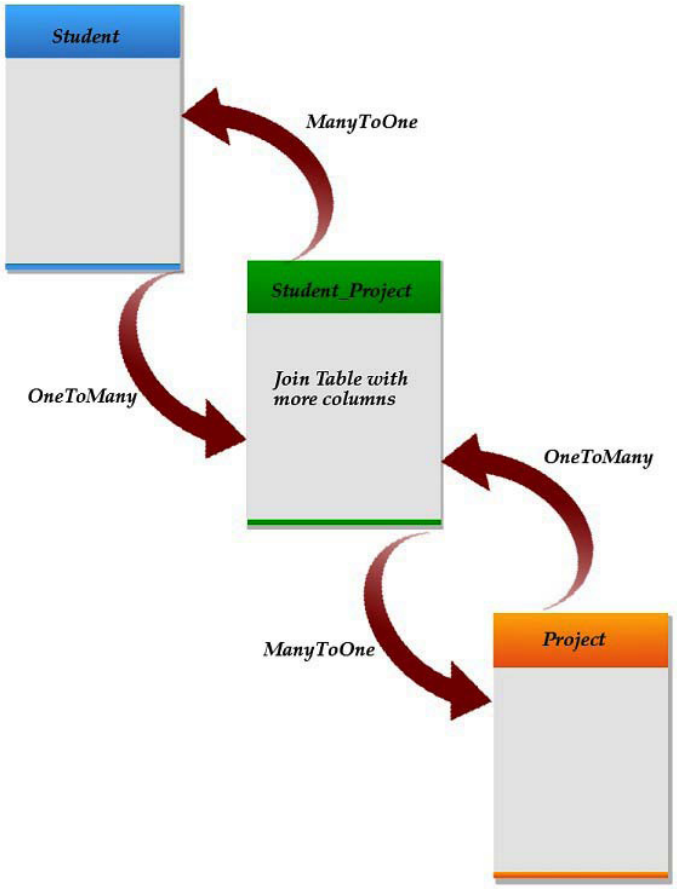

***Figure 4.2**: ManyToMany relationship with additional columns in the join table.*

In the picture above, the **`Student_Project`** table is the join table created by JPA to maintain the ManyToMany mapping. Consider there are additional fields like **`Is_Lead`**, **`Project_Duration`** in the join table. Hence, it would be always better to have an additional class created for the join table. In this scenario, the ID can become complex with multiple fields and only the entire combination remains unique. It

is always better to use a generated ID value for the same so that the ID remains no longer complicated but becomes simpler and there will be no duplicated fields in the ID as it is a generated value.

Ordering

When the collection values are retrieved from the database, there is an order in which they are obtained. This order can be set via the JPA annotations. The annotation used is **`@OrderBy (<order-by> XML element)`**. When this annotation is provided, essentially the **`ORDER BY`** string is added to the JPQL query string.

The default value assumed for ordering is the ID value of the target objects. JPA1.0 supports ordered lists only by adding an index (attribute to the object and column to the table). However, in JPA2.0 extended support for an ordered List is achieved using **`OrderColumn`**. Note that using **`@OrderBy`** does not ensure that the collection is ordered in memory. **`SortedSet`** interface and **`TreeSet`** collection maintains order and hence a few JPA providers use these collection types to maintain the correct ordering.

```
@Entity
public class Person {
 @Id
 private long id;

 @OneToMany(targetEntity=PhoneNumber.class)
 @OrderBy(”areaCode”)
 private List phoneNumbers;
}
```

***Code 4.18**: @OrderBy annotation*

Here, **`areaCode`** is the column based on which the ordering is done whenever the values are retrieved from the database.

Order column (JPA 2.0)

O**`rderColumn`** is a feature introduced in JPA 2.0 and defines an order of collection mapping. It is defined as **`@OrderColumn`** annotation (in XML, **`<order-column>`** element) and is based on the type of mapping. In case of a OneToMany mapping or ElementCollection mapping, the column should be from the target object, whereas for a OneToMany mapping with join table or ManyToMany mapping, the column should be from the join table.

```
@Entity
public class Person {
 @Id
 private long id;

 @OneToMany
 @OrderColumn(name="indexNum")
 private List phoneNumbers;
}
```

***Code 4.19**: @OrderColumn annotation*

Here the **`OrderColumn`** is mentioned as **`indexNum`** which is a column in the **`Person`** table. This column is used to store the id of the phone numbers from the **`PhoneNumber`** table.

Common problems

In this section, the focus in on various common problems that arise while using various techniques that were mentioned throughout this chapter. It includes functional issues like object corruption, lazy loading serialization issues to performance issues like excessive queries, and poor performance due to EAGER loading of many huge tables.

Object corruption

This is a common problem with bi-directional relationships where one side of the relationship is updated which does not consequently update the other side. Thus, the relation becomes out of sync and hence the object gets corrupt.

In JPA, it is the responsibility of either the application or the object model to maintain relationships. There are two different methods to solve this issue: Either use the setter from one side and make the other side protected or add it to both sides and make sure that the relationship is maintained without any looping issues.

```
@Entity
public class Person {
  private List phoneNumbers;
  public void addPhone(Phone phone) {
    this.phoneNumbers.add(phone);
    if(phone.getOwner() != this) phone.setOwner(this);
}}
```

```
public class Phone {
  private Person owner;
  …
  public void setOwner(Person person) {
    this.owner = person;
   if(!person.getPhones().contains(this)) person.getPhones().add(this));
  }}
```

***Code 4.20**: Object update handles the relationship update as well*

Here, when a phone number gets added to a **`Person`** object, the **`Phone`** object is also set with the appropriate owner. This makes sure that the bidirectional relationship is maintained. There is a common expectation that the JPA provider automatically maintains the relationships. This expectation is not true and becomes an issue if the objects are serialized or detached or used outside the scope of JPA. However, there are a few providers that support automatic relationship maintenance. In general, it is always better to make sure that the object model takes care of the relationship irrespective of the support from the JPA provider.

Tip: In order to avoid the instantiation of large collection while adding a child object, the bi-directional relationship is not mapped and instead queried for as required.

Poor performance

This issue is mostly due to the usage of EAGER fetch in relationships. In certain cases, this type of fetch can result in reading all the objects as related objects for a particular source object. For example, if there is a **`Staff`** table with a relationship **`managerOf`** which is EAGER type and if the **`Principal`** is the source object, then the EAGER fetch of **`managerOf`** will in turn fetch all the **`Staff`** objects as the principal manages the entire staff of the school.

The simplest and easiest way is to make all relationships LAZY. By default, OneToMany and ManyToMany are LAZY relationships. However, OneToOne and ManyToOne are not and hence needs to be configured accordingly.

Excessive queries

While trying to access the address from a **`Person`** object, and LAZY fetch being used for **`staysAt`** relationship (this can be either a OneToOne or ManyToOne), an additional query is being used. So, when there is a case of accessing all the **`Person`** addresses, there would be one query for accessing the list of **`Persons,`** and

then *n* queries to get the addresses of *n* Person objects. This can also lead to poor performance. The solution for this is to use **Join Fetching** and **Batch Reading**.

Lazy loading – not functioning as expected:

The JPA providers use various techniques to provide the magic of LAZY loading and make the objects appear as if they exist. The widely used approaches are build-time bytecode instrumentation, run-time bytecode instrumentation, and run-time proxies. Build-time bytecode generates the entity changes after compilation but before the run time. This is the best performance way but can cause serialization problems in Java. In case of run-time bytecode technique, an agent is used which allows the byte-code weaving dynamically during run-time. So, based on the provider chosen, proper configurations should be provided (either for static byte code generation or agent for dynamic byte code generation).

Also, it is important to ensure that the relationship shouldn't be accessed when inappropriate. For example: if there is a property access (i.e., the annotations are provided on the getter/setter methods) and in the **`set`** method, if the related lazy value is accessed, then it negates the lazy loading effect. So, it is important to remove the side-effects of the set method or use the field access.

Normally, if lazy relationships are not instantiated before serialization, then the object will not get serialized and hence will cause an error or provide null values if accessed after deserialization.

Relationship target is an interface

If an interface is used as a target entity, and it has multiple implementations, then JPA does not support it directly. Instead, the interface has to be changed to an abstract class and use **`@Inheritance`** annotation. More details on this annotation are already explained in *Chapter 3 Operations – Identity, Sequencing and Locking*.

Dependent object removal from OneToMany collection

When an object is removed from a collection, JPA 1.0 does not mandate that it should be removed from the database. A **`remove()`** call should be made explicitly to remove the object from the database. However, some providers support options to remove the object from the database once it gets removed from the collection.

In JPA 2.0, this option has been provided and hence, the object removal from collection triggers the object deletion from the database.

Bidirectional relationships

In a bi-directional ManyToMany relationship, there should be sufficient care taken to ensure that the relationship is established via both sides. Otherwise, the object will not be available in the collection even after refresh.

Also, if **`mappedBy`** is not used on one side of the relationship, then it shall be assumed as multiple relationships and have duplicate rows inserted into the join table.

Primary keys in CollectionTable

In JPA 2.0, there is no specific mechanism to define an Id in an Embeddable. This makes the JPA assume that the combination of all fields in the Embeddable along with the foreign keys provide the unique key which is required for a delete/update operation. This could be useless and unreasonable if the Embeddable is big, or complex. However, some JPA providers have defined ways to specify the Id for an Embeddable.

Conclusion

In this chapter, the main focus has been on relationships and various types of relations that are found in the relational database world. Moving further, how these real-world database relationships can be mapped to the Java based relationships have also been dealt with in a detailed manner.

In the next chapter, how the database queries are formed and what all steps are taken by the provider to get the actual data out of the DB to the object is being described in a more explicit manner.

Multiple choice questions

1. **The reference class for a relationship is provided by the annotation:**
    a) refClass
    b) targetEntity
    c) refObject
    d) all of the above

2. **The @MapKey in JPA 2.0 had a limitation that the key should be part of the target object:**
    a) true
    b) false
    c) not applicable
    d) none of these

3. **The various values that @MapKey in JPA 2.0 can assume are:**
   a) a basic value
   b) an embedded object
   c) a foreign key to another entity
   d) all of these

4. **Orphan removal helps in:**
   a) removing dependent objects from database when they are no longer required in a relationship
   b) removing objects from the database in a customized manner
   c) removing foreign key objects only from the database
   d) all of these

5. OrderBy annotation is used for:
   a) ordering the collection while retrieved from the database
   b) ordering the relationships while storing it to the memory
   c) ordering the collection before removing them from the database
   d) all of these

Answers

1. **b**
2. **b**
3. **d**
4. **a**
5. **a**

Questions

1. Explain different relationships found in JPA. Compare it with the database relationships.
2. Describe various query optimization techniques available with suitable examples.
3. How are Maps used in the JPA context? Explain.
4. Figure out the various problems seen while using relations and explain any two with suitable examples.

Points to ponder

1. An **`OUTER`** join in SQL is one that does not filter absent rows on the join, but instead joins a row of all null values. Note that **`OUTER`** joins can be less efficient on some databases, so avoid using an **`OUTER`** if it is not required.

2. TopLink and EclipseLink supports a **`@JoinFetch`** annotation and XML on a mapping to define that the relationship always be join fetched.

3. It may be desirable to mark all relationships as EAGER as everything is desired to be loaded, but join fetching everything in one huge select could result in an inefficient, overly complex, or invalid join on the database.

4. Technically in JPA, LAZY is just a hint, and a JPA provider is not required to support it. However, in reality, all main JPA providers support it, and they would be pretty useless if they did not.

5. The EAGER default for OneToOne and ManyToOne is for implementation reasons (more difficult to implement), not because it is a good idea.

6. TopLink and EclipseLink supports variable relationships through **`@VariableOneToOne`** annotation (corresponding XML as well). Mapping to and querying interfaces are also supported through their **`ClassDescriptor's InterfacePolicy`** API.

7. TopLink and EclipseLink support filtering and complex relationships using **`DescriptorCustomizer`** interface to define a **`selectionCriteria`** using Expression criteria API or define SQL or stored procedure call using **`selectionQuery`**. This allows any condition to be applied including constants, functions, or complex joins.

8. The JPA 1.0 spec does not allow an Id to be used on a ManyToOne.

Section - III

Runtime Access and Process Objects

Chapter 5
Query Infrastructure

Introduction

The storage of data becomes meaningful only if it can be later retrieved in various ways for different purposes. The data that can only be stored is of no use, and hence is wasted without proper retrieval mechanisms. Till now, the discussions were mostly on how to store the data in DB. This chapter discusses the various methodologies that are used for querying the data from the DB that are supported in JPA.

Structure

The structure of this chapter is as given as follows:

- Querying
    - Named queries
    - Dynamic queries
    - Query results
    - Common query samples
    - Optimization
    - Advanced topics

        - Update/Delete
        - Flush mode
        - Pagination
        - Native SQL
        - Raw JDBC

- JPQL BNF
    - Select
    - Update
    - Delete
    - Literals
    - New in JPA 2.0

Objectives

This chapter focusses on the various retrieving mechanisms that are used in JPA. This also provides many sample queries that are commonly used. Various optimization techniques are also discussed so that the performance of the retrieval process can be improved. This chapter also tries to explain the **Backus-Naur Form** (**BNF**) definitions that are used to define **Java Persistence Query Language** (**JPQL**). Moreover, this section can help in understanding JPQL further in depth.

Tip: The JPQL BNF section is more informatory and hence not mandatory and can be skipped if required. That would not affect the understanding of subsequent chapters.

Querying

As per the definition, querying is the process of asking questions to clarify doubt or request information. Hence, querying is the most important part while storing data. If stored data cannot be retrieved on demand or as per custom requirements, then that data can be considered useless. Query languages can be broadly classified as:

- **Database query languages**: These languages are mainly used to extract factual data based on certain criteria (i.e., select the various values for a particular field where another field matches some value). SQL is the most common query language used in relational databases.
- **Information retrieval query languages**: These languages are mainly used to extract documents/records that contain the information inquired. CQL is a

formal information retrieval language used while dealing with web indexes, bibliographic catalogs, museum collections, and so on.

Since JPA mostly deals with databases, it mainly provides querying mechanisms that are more similar to SQL. However, JPA provides mainly three different mechanisms for querying data.

1. **JPQL**: JPQL is a very powerful query definition language based on BNF. This allows to define the database queries based on the entity model that has been defined as part of the application development. This feature of JPQL makes it very appealing to Java developers since they are more comfortable using the entity model names rather than the database table names. However, databases can only understand SQL statements and it becomes the JPA implementation's responsibility to convert the JPQL to SQL. This is the easiest way to implement due to its simplicity. But it is difficult to create dynamic queries using JPQL.

2. **JPA Criteria API**: This is a feature added in JPA 2.0 to provide type safe way of writing queries using the Java program. This is mostly used in case of dynamic queries. A static, instantiated, canonical metamodel class is generated for each entity model, and hence, that can be used for type safe checking. Thus, the API provides various features like:

   a) Checking for query correction and preventing creation of queries that are syntactically incorrect.

   b) Raising compilation error after refactoring the code, if there are any deviations from the requirements of the entity model.

   c) Supporting autocompletion as there is a generated class for each entity model, i.e., the various method names that are part of the entity model will be given in the auto completion feature of the IDE being used for application development.

3. **Native SQL Queries**: These are SQL queries that use the actual database object names and can be directly executed on the DB via the client. JPA provides annotation **`@NamedNativeQuery`** (*Code 5.1*) where a native query can be defined by a name and can be used across the code by referring to it using the name. JPA defines the **`Query`** interface as the API for using the native queries. This interface provides various ways of retrieving the result from the database i.e., it has method to return the result as a single result, result list or only the first result based on the requirement. In short, the developer who is familiar with native SQL can easily declare named native queries and use it throughout the code by passing the name of the actual query to the **`Query`** interface, and then expect a response from this interface.

```
@Entity(name=”PassengerEntity”)
@Table(name=”passenger”)
@NamedNativeQueries({
    @NamedNativeQuery(
        name = «getAllPassengers",
        query = «SELECT id, firstName, lastName, email, passenger.seat,
date” +
                «FROM passenger, seat», resultClass=PassengerEntity.
class ),
    @NamedNativeQuery(
        name = «getAllPassengersBySeat",
        query = «SELECT id, firstName, lastName, email, date, passenger.
seat " +
                    «FROM passenger, seat» + «WHERE passenger.seat = ?",
                    resultClass=PassengerEntity.class)
})
public class PassengerEntity implements Serializable{
 //implementation here
}
```

***Code 5.1**: NamedNativeQuery example*

There are many other querying languages and frameworks available for the same reason. Some of them are listed as follows:

- EJBQL
- JDOQL
- EclipseLink Query Language (EQL)
- Query By Example (QBE)
- TopLink Expressions
- Hibernate Criteria
- Object Query Language (OQL)
- Query DSL

In general, JPA needs to create static-named queries or dynamic queries. The various techniques used for creating named-static queries and dynamic queries are as follows.

Named queries

Mostly, named queries are for static queries that are used repeatedly in the application. The first benefit of having this kind of query is that it needs to be defined only once and can be reused throughout the application. This makes it attractive not only while development, but also while maintaining the application as the logic change in the query needs to be modified only at a single place.

The second benefit of these queries is that they are more optimized than dynamic queries. This is because most of the JPA providers pre-compile/pre-parse the named queries since they are available during the compile time itself.

The third benefit of these queries is that they can be optimized or overridden via the ORM configurations since they are part of the persistence metadata. i.e., assume a particular named query has been defined in the application with a certain name (say query1). If the application developer needs to make change in the query1, either the code can be changed or another query with name as query1 can be defined in orm.xml. So, in cases where the actual code is not available for the user, this technique helps in overriding the queries if their respective names are known.

Whether to use the annotation-based configuration or XML-based JPA configuration for named queries is a matter of choice for any particular application. However, there are a few scenarios where an XML-based JPA configuration in general would be preferred.

1. Unsupported or regular updates – Older applications that do not support annotation or applications that need regular updates prefer the usage of XML based configuration since it is more easier to change and does not involve any code changes.

2. Externalize JPA related information from the application–This is an influential reason why most of the applications still depend on XML-based configuration. This allows the JPA information to be controlled from an external file, which need not be compiled along with the application source code.

3. Large number of entities in different packages–This is more important in case of named queries. If any named query implementation has to be changed, the entire set of packages are to be searched for to find out the package/class in which that particular named query is defined. If this had been defined in the **`orm.xml`**, there is only one file to be checked for.

A single named query is defined using **`@NamedQuery (<named-query>)`** annotation, whereas if there are multiple queries, each one is defined using the above annotation and the entire list is defined using **`@NamedQueries (<named-queries>)`**. A similar example of named native query is given in the preceding *Code 5.1*. Note that any class

with **@Entity** can be used for defining the named queries. But it is always better to define the queries on the same **Entity** class that they query. Also, the name used must be unique for the entire persistence unit. **EntityManager** is used for accessing the named queries and **Query** interface is used for executing the queries. More details of this will be dealt with in *Chapter 6, Entity Manager – Persisting, Caching and Transaction*.

Since named queries are static queries, they are mostly defined in such a way that they can receive parameters and execute the query based on the provided parameters. In JPQL, named parameters and positional parameters can be used, and are denoted using the **:<name>** syntax for named parameters, and "?" (question mark) for positional parameters.

Query hint

In DB terminology, query hints are additions to a query that instructs the DB engine on how to execute the query. Usage of database query hints should be done judiciously as it can alter the normal behavior of the DB query execution engine.

However, the query hints supported by JPA are not at all related to the preceding DB query hints. Instead, they are nothing but JPA provider customization preferences. It uses the **setHint** method of the **Query** interface to set the hints as and when required. The most common query hints supported by JPA are timeout, fetchgraph, loadgraph, etc.

`@NamedQuery(`
`name=”findAllEmpInCity”,`
`query=”select emp from Employee emp where emp.address.city = :city”`
`hints={@QueryHint(name=”jpa.batch”, value=”emp.address”)}`
`)`
`public class Employee {`
`…`
`}`
	`<entity-mappings>`
	`<entity name="Employee" class="org.acme.Employee" access="FIELD">`
	`<named-query name="findAllEmployeesInCity">`
	`<query>Select emp from Employee emp where emp.address.city = :city</query>`
	`<hint name="jpa.batch" value="emp.address"/>`
	`</named-query>`
	`<attributes><id name=”id"/></attributes>`
	`</entity>`
	`</entity-mappings>`

***Table 5.1**: NamedQuery example using annotations and XML with QueryHints*

```
EntityManager em = getEntityManager();
Query query = em.createNamedQue“y("findAllEmpInCity");
query.setParameter("city", "Toronto");
List<Employee> employees = query.getResultList();
```

***Code 5.2**: Sample code to invoke named query*

The preceding Java code (*Code 5.2*) demonstrates the usage of a named query (defined in *Table 5.1*). The parameter for the named query here is 'city' and is being set in the query using the **`setParameter`** method. Hence, this query returns the list of all employees with their address with city value being "**`Toronto`**". The preceding query is not only a named query, but also dynamic, as it accepts a parameter (here city) and can dynamically change the actual query being executed.

Dynamic queries

The example given in *Table 5.1* and Code *5.23* is a typical dynamic query where it changes according to the parameter set in the code during runtime. Here, the query depends on the context as well as the parameters being set while the actual execution happens. JPQA and Criteria API are two options in JPA for creating dynamic queries. Similar to named queries, dynamic queries can also have parameters, query hints, and so on accessed through EntityManager's **`createQuery`** API and executed via Query interface.

Parameters

There are different ways of passing parameters in JPQL. The most common way is to use the ":" syntax in the query and set the parameter using the name used along with the syntax. This is called **named parameter**.

For example, **`select emp from Employee emp where emp.address.city=:city`**

In the preceding example, the parameter name is 'city' and is used for passing the value using **`setParameter("city", <value>)`**.

Another way of defining the parameter is using "?" in native SQL queries. This can also be used as "?"**`<int>`**. This is called positional parameter and the parameters are set according to the int value provided.

For example, **`select * from Employee where name=? and age=?`**

In the preceding example, the name is the first parameter and age the second. Hence, **`setParameter(1, "Rina") and setParameter(2, "25")`** are used to set the values for the query. Positional parameters always start with 1.

Temporal parameters can also be passed as date, time, or timestamp. Even objects can be passed as parameters. Note that for all types of queries, whether it is JPQL, Criteria, Native SQL or Named Query, the parameters are always set on the query.

Query results

The query results always return the **`Entity`** objects managed by the persistence context. The changes made can be further tracked as part of the current transaction. There are three ways of obtaining the results after executing a query:

1. **`getResultList`**: returns a list of objects (entity objects or arrays).

Query	Output
`select s from Student s`	List of Student Entity objects: the objects in this list are managed.
`select s.fName from Student s`	List of String values: here, the data is not managed since they are not entity objects.
`select s.fName, s.lName from Student s`	List of Object<String,String>: in this case, also data is not managed.
`select s, s.address from Student s`	List of Object<Student,Address>: in this case, the data is managed since both are Entity objects.

***Table 5.2**: Output for various queries.*

2. **`getSingleResult`**: returns a single object. The preceding rules hold good for this as well. If an entity object is being returned, then the data would be managed. If the query returns nothing, then depending on the JPA provider implementation, it can either throw an exception or return a null. Similarly, if the query returns more than a single result, depending on the JPA provider implementation, it can either throw an exception or just return the first result.
3. **`executeUpdate`**: returns the number of rows affected by this query, normally used for update/delete queries.

In the next section, explanation of various common ways of writing queries is being done.

Common query samples

To retrieve data, there are multiple ways of writing queries. In this section, different query conditions are discussed along with the ways of writing it in JPQL as well as in **`Criteria`** API. Normally, the basic steps to create a criteria query are as follows:

- **`EntityManager`** is used to get the **`CriteriaBuilder`** object.
- **`CriteriaQuery`** is used to create the query object, **`CriteriaQuery.from`** method sets the query root and **`CriteriaQuery.select`** sets the result list type.
- **`TypedQuery<T>`** prepares the query for execution by specifying query result type.
- **`TypedQuery.getResultList`** executes and returns the entity collection as a **`List`**.

In the following examples, only the first two steps are considered in the **`Criteria`** API query samples.

Join

This is mainly done for querying values from a OneToMany relationship (discussed in detail about join in *Chapter 4, Relationships – Types and Strategies*).

For example, consider querying all the students studying a particular subject (say Economics). Considering the relation to be a unidirectional one, (i.e., the relation is from the student to subject only), JPQL query would be as follows:

```
SELECT stud FROM Student stud JOIN stud.subjects sub where sub.name =
“Economics”
```

Criteria would be as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Student> query = cb.createQuery(Student.class);

Root<Student> stud = query.from(Student.class);

Join<Subject> sub = stud.join("subjects");

query.where(cb.equal(sub.get("name"), "Economics"));
```

First, the builder instance is retrieved from the entity manager and the criteria query is created mentioning the entity class. **`Root`** represents the target entity i.e., the entity that is being expected as the result of this query. Here, it is the student entity. The next statement specifies the **`Join`** condition i.e., the entity on which the join has to take place (here **`Subject`** entity given as the expecting result) and the attribute of the **`Root`** entity on which the join has to be executed (here, subjects attribute of the **`Student`** entity). The preceding **`join`** method creates an inner-join as it only provides the attribute name of the target entity. The criteria builder checks for the subject name (here, the name being checked for is “**`Economics`**”) and thus forms the final query in the last line.

Sub-select

This is mainly done for querying all or subset of values from a ManyToMany relationship.

For example, consider a query which has to query all students who are being taught by a particular teacher (say here "**`Teacher1`**"). Here, the relationship is OneToMany from both sides, i.e., a student is taught by many teachers and a teacher teaches many students. Hence, it is a ManyToMany relationship.

JPQL

```
SELECT stud FROM Student stud JOIN stud.taughtBy teacher where EXISTS
(SELECT t FROM Teacher t where teacher = t and t.name=”Teacher1”)
```

Criteria would be as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Student> query = cb.createQuery(Student.class);

Root<Student> stud = query.from(Student.class);

Join<Teacher> teacher = stud.join("taughtBy");


Subquery<Teacher> subquery = query.subquery(Teacher.class);

Root<Teacher> subTeacher = query.from(Teacher.class);

subquery.where(cb.equal(teacher, subTeacher), cb.equal(subTeacher.
get("name"), " Teacher1"));

query.where(cb.exists(subquery);
```

Here, the initial four lines are similar to the earlier query, and the only difference is the **`Join`** entity and the attribute name of the target entity (**`Teacher`** and **`taughtBy`** respectively). The next part creates the subquery with the **`Entity`** class that is the **`target`** class (here, it is **`Teacher`**). In the where clause of the subquery, first, a check is made to see whether the object in the subquery is similar to the object in the main query and then the check is done to see whether the name of the object is as expected (here, "**`Teacher1`**"). Once the subquery is framed, then the main query is executed with the subquery in its where clause.

Join fetch

This is a technique used for query optimization (discussed in *Chapter 4, Relationships – Types and Strategies*).

For example, consider a query to select all the students and their addresses. This can be done in a single query using join fetch. Without using join fetch, the students would be available in the first fetch and for each student's address, an additional

query would be made. Hence, if there are *n* students altogether, the join fetch would get students and their addresses in a single fetch, whereas a normal join would get students in a single fetch. Then, it would require *n* additional accesses to get all the addresses.

In short,

Total number of queries for join fetch = 1

Total number of queries for join = 1+n

JPQL:

1. This will join fetch the addresses.

```
SELECT stud FROM Student stud JOIN FETCH stud.address
```

2. This will fetch both the address and the phone numbers.

```
SELECT stud FROM Student stud JOIN FETCH stud.address JOIN FETCH
stud.phone
```

3. This will avoid null and empty values by giving outer join explicitly.

```
SELECT stud FROM Student stud LEFT OUTER JOIN FETCH stud.address
LEFT OUTER JOIN FETCH stud.phone
```

4. This will fetch all students whose phone number area code is "123".

```
SELECT stud FROM Student stud LEFT OUTER JOIN FETCH stud.address
LEFT OUTER JOIN FETCH stud.phone where stud.phone.areacode =
"123"
```

Criteria would be as follows:

1.
```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Student> query = cb.createQuery(Student.class);
Root<Student> stud = query.from(Student.class);
Fetch<Address> a = stud.fetch("address");
query.select(stud);
```

2.
```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Student> query = cb.createQuery(Student.class);
Root<Student> stud = query.from(Student.class);
Fetch<Address> address = stud.fetch("address");
Fetch<Phone> phone = stud.fetch("phone");
query.select(stud);
```

3.
```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Student> query = cb.createQuery(Student.class);
Root<Student> stud = query.from(Student.class);
Fetch<Address> address = stud.fetch("address", JoinType.LEFT);
Fetch<Phone> phone = stud.fetch("phone", JoinType.LEFT);
query.select(stud);
```

4.
```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Student> query = cb.createQuery(Student.class);
Root<Student> stud = query.from(Student.class);
Fetch<Address> address = stud.fetch("address", JoinType.LEFT);
Join<Phone> phone = (Join<Phone>) stud.fetch("phone", JoinType.
LEFT);
query.select(stud).where(cb.equals(phone.get(“areaCode”),”123”));
```

The first query is to fetch the addresses as part of the join fetch, while the second fetches phone in addition to address. The third one is to explicitly mark the join fetch to be a left outer join, so that there are no invalid entries. All the three queries directly use the **`Fetch`** interface. However, in the fourth query, a **`where`** clause is added to the join fetch and hence, the **`Fetch`** interface cannot be used since it does not support the usage of expression in the **`where`** clause. The workaround is to typecast the **`Fetch`** interface to **`Join`** interface (both are sub-interfaces of the same parent **`FetchParent`**). But the outcome of this code depends on how the JPA implementation has been done (Note that this approach works on Hibernate and EclipseLink). To overcome this shortfall, entity graphs have been introduced in JPA 2.1.

Query based on relation

There can be cases where values are not fetched as desired due to the type of the relation and its manner, i.e., the relation may be unidirectional or bidirectional. Based on this, the query has to change.

For example, consider a query to get all the students studying a given subject. The unidirectional case for this query has been considered previously. Assume, if this was a bidirectional relationship, the query would have been slightly different and as follows:

```
SELECT sub.students FROM Subject sub where sub.name = “Economics”
```

Criteria would be as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Subject> query = cb.createQuery(Subject.class);
```

```
Root<Subject> sub = query.from(Subject.class);
query.select(sub.get("students")).where(cb.equal(sub.get("name"),
"Economics"));
```

Here, the target entity is considered as **`Subject`** and not **`Students`**. Since the relation has a bi-directional manner, it should work in the same way from both sides. Hence, there is no difference whether the target entity is **`Subject`** or **`Student,`** although the query is to get the list of students.

Simulate casting

Consider a scenario where an entity has a relationship to another entity with subclass. It is not possible to query the entity based on any of the subclass attributes as JPA does not define a cast operation. However, it is possible to simulate this by adding a secondary **`JOIN`** to the subclass query.

For example, consider all the students who are part of internship projects that have a budget higher than a particular amount (say 10K).

```
SELECT s from Student s JOIN s.proj p, LargeProject lproj
where p = lproj and lproj.budget > 10000
```

Here, the projects are being joined with the subclass **`LargeProject,`** where the budget is also mentioned. Note that **`LargeProject`** is a subclass of **`Project`**.

More queries

Consider a query to select the first element in the resulting collection. This can be done in many different ways:

- **`setMaxResults`** method in **`Query`** API

  ```
  Query query = em.createQuery(
            "SELECT s.proj FROM Student s WHERE s.id =:id");
  query.setMaxResults(1);
  ```

- Straight JPQL query

  ```
  SELECT p FROM Project p where p.id = (SELECT MAX(pr.id) FROM
  Student stud JOIN stud.proj pr where stud.id = :id)
  ```

- **`INDEX`** function if the collection is an indexed list (i.e., by using **`@OrderByColumn`** annotations as discussed in previous chapter. Note that this can be done only from JPA 2.0 onwards.

  ```
  SELECT p from Student st JOIN st.proj p, where st.id = :id and
  INDEX(p) = 1.
  ```

Consider a query to order the values based on the size of a collection. This can be done by using:

- **`SIZE`** function

```
SELECT s, SIZE(s.proj) FROM Student s ORDER BY SIZE(s.proj) DESC
```

- **`GROUP BY`** and **`COUNT`**

```
SELECT s, COUNT(p) FROM Student s JOIN s.proj p GROUP BY s ORDER
BY COUNT(p) DESC
```

- **`GROUP BY`** and alias for **`COUNT`**

```
SELECT s, COUNT(p) as pcount FROM Student s JOIN s.proj p GROUP
BY s ORDER BY pcount DESC
```

Optimization

Normally, a query first reads an object and then fetches its related objects by reading them one by one. This happens to be the main drawback, and the performance can be improved by reading multiple objects at a time instead of individual records. The important approaches in handling such queries are discussed in detail under the section *Query optimization techniques* of the previous chapter.

Another approach to optimization is to provide various JDBC options while executing a query. In general, these options need not be exposed by the JPA provider, but certain JPA providers allow query hints to configure such options.

Some such options are as follows:

- **Fetch size**: This allows to configure the number of rows fetched from the database. Larger the size, better for large queries.
- **Timeout**: This sets the maximum execution time for a query and notifies the database to cancel the query if it takes too long.
- **Batch writing**: This improved the DB performance by sending a group of statements in a single transaction.
- **Data format**: A few JPA providers optimize the data as per the format that the application requires. In cases where there are format conflicts, this can lead to issues and hence can be turned off.

Advanced topics

U**`pdate`**/**`Delete`**: Ideally in JPA, the update to an object should be done via **`set`** methods and delete should be done by removing the object using the Entity Manager's **`remove`** method. However, JPQL provides **`UPDATE`** and **`DELETE`** queries

for batch operations. Similar to **`SELECT`** queries, these queries also have a **`WHERE`** clause and can be used for the same purpose. **`executeUpdate`** method of the **`Query`** interface can be made use of for these queries.

For example

```
UPDATE Student s SET s.phone.areaCode = "123" WHERE s.address.city = :city
```

The preceding query updates phone number area code for all students who stay in a particular city.

For example

```
DELETE FROM Student s WHERE s.address.city = :city
```

The preceding query deletes all students who stay in a particular city.

Tip: The execution of `UPDATE/DELETE` queries can affect the objects that are already registered in the Entity Manger's context. So, it is always good to either clear the context before executing these types of queries or use a new transaction for these queries.

Flush Mode: When changes are made in JPA, these are only made in the memory within Java, and will be written to the DB only while the transaction is committed. Any queries to the DB may not see the changes made within the transaction. Hence, JPA needs to instruct the provider to push all changes to the DB before any query operation.

Flush operation can be expensive and can affect performance and concurrency. If the application changes are not yet in a state to be pushed to the DB, a flush can be an overkill. It becomes a side effect of a query operation and can hold up the locks and other resources for the duration of the transaction.

JPA supports two flush modes as follows:

- **`AUTO`** – executes flush before every query execution (default mode).
- **`COMMIT`** – flush operation needs to be done only before a transaction commit.

This can be set using **`setFlushMode()`** method either at **`Query`** level or at EntityManager level to affect all queries executed with it.

Tip: Whenever a flush is desired, `EntityManager.flush()` can be directly invoked. This forces the flush operation to be performed irrespective of the flush mode configured.

Pagination: When the query result is large, it is ideal to provide the result as pages with each containing n results. This is a very common requirement, especially when a web user views the query result and would like to navigate through the results, page by page, on clicking *previous*/*next* to go to respective pages. If the query fetches the entire result, then each page request would cause a re-query, which can be a performance issue.

A solution to this problem is to have a stateful session bean and handle the paging. Although, the initial page load would take long time, the subsequent page navigations would be faster since the session bean would control the results without re-querying. Another solution is to have JPA providers that support caching of query results. Every page navigation would execute the query to provide the cached results.

In case of larger query results, **`setFirstResult()`** and **`setMaxResults()`** methods from **`Query`** API can be used for paging support. JDBC supports **`setMaxResults`**, and so do most JDBC drivers. Hence, it should work for most of the JPA providers as well as databases. Since there is no standard SQL for pagination, support for **`setFirstResult`** depends on the DB and JPA provider.

Tip: While enabling pagination, it is always important to order the result. If the query does not provide an order, the results would be in a different order each time and hence would cause issues.

Native SQL Queries: Customarily, queries in JPA are defined using JPQL as it supports object model which allows data abstraction, and DB schema and platform independence. Although JPQL supports almost all constructs of SQL syntax, there can be some aspects of SQL or DB specific extensions/functions that may not be possible via JPQL. Hence, native queries are preferred in such scenarios. Native queries can also be used for executing stored procedures or executing DML/DDL operations. They are defined using **`@NamedNativeQuery`** as shown in *Code 5.1*.

Native queries can be generated dynamically using **`createNativeQuery()`** method of **`EntityManager`** API. The result of a native query can be a specific class or raw data or complex result. In case of a specific class, the **`resultClass`** attribute must be set. In case of a complex result, result set mappings can be used.

When the query result is complex, and returns data from multiple objects, then **`@SqlResultSetMapping`** annotation must be used (as shown in *Code 5.3*). This annotation contains an array of **`@EntityResult`** and **`@ColumnResult`**.

`EntityResult` – Specifies the JPA entity and provides a list of **`@FieldResult`** that maps the actual column name (in the native query) to the entity column name.

`ColumnResult` – Specifies the column name (in the native query) and the type.

```
@NamedNativeQuery (
  name="findAllStudentsInCity",
  query="SELECT S.*, A.* FROM student S, address A WHERE S.id = A.stud_
id AND A.city = ”?”,
  resultSetMapping = ”student-address”)
@SqlResultSetMapping (
   name="student-address",
```

```
   entities = {
     @EntityResult (entityClass=Student.class,
       fields = {
           @FieldResult (name=”id”, column=s.id),
           ……
         }),
     @EntityResult (entityClass=Address.class)
   }
)
public class Student {
 //implementation here
}
```

***Code 5.3**: Using result set mapping for native queries with complex results*

In the preceding **`@FieldResult`**, the name attribute provides the JPA entity column name and the column attribute provides the actual native query column name.

Raw JDBC: There can be scenarios where JDBC driver has specific features that can be accessed only via its code or integrate other applications that use JDBC instead of JPA. Hence, it would be vital to use the JDBC code along with JPA code.

Access to JDBC connection can be made either through the Java enterprise server DataSource or direct connection to DriverManager or third-party connection pool frameworks. The challenge is to have JDBC connection and JPA application share the same transaction context. If that is required, then JTA DataSource is to be used by JPA and share the same global transaction with the JDBC connection. Some JPA providers implement an API to access raw JDBC connection from their internal connection pool that helps to get both the JDBC connection and JPA in the same context.

In JPA 2.0, there is an unwrap API to access the connection from **`entityManager`**.

```
java.sql.Connection con = entityManager.unwrap(java.sql.Connection.
class);
```

The connection obtained as given here can be used for raw JDBC access. However, this connection should not be closed after being used since it is the EntityManager's connection and resources will be released. This can lead to malfunctioning or uncertainty in the application.

Stored procedures: A stored procedure is a set of DB operations written in a language (similar to SQL) and stored in the DB itself. They are useful in many ways:

- to minimize the data being sent to/from the DB to the application
- to perform batch processing
- to access specific functions/values that can be only done from the DB server
- to avoid giving user access to raw tables and hence make a stricter policy

However, there are a few disadvantages as well for stored procedures. They are as follows:

- Written in a different language from the application and hence more effort for maintenance is required.
- Uses limited procedural programming language and hence, becomes less flexible and more difficult to develop/debug

Since the code is within the DB itself, there is a misinterpretation that stored procedures are faster. In fact, usage of stored procedures reduces the dynamic ability of the persistence layer to optimize data retrieval. Stored procedures improve performance only if SQLs used are optimal than those used in the application. So, it is always better to minimize the use of stored procedures and allow applications to have optimal queries using prepared statements.

JPA does not have direct support for stored procedures. However, it can be achieved to an extent using Native SQL queries. But this support can be only be done for queries that return nothing or a DB result set. JPA does not support the usage of OUTPUT or INOUT parameters in the stored procedures. However, there are few JPA providers which extensively support stored procedures by even overriding CRUD operations of an Entity with stored procedures/custom SQL.

JPQL BNF

The Backus Naur Form is a way of representing context free grammars. In simple words, context free grammar can be considered as a set of recursive rules which can be used to generate different patterns of strings. So basically, BNF is used to define these set of rules.

In this section, the definition of JPQL using BNF is being shown. The simplest way of representing the query language statement is as follows:

```
QL_statement::= select_statement
                | update_statement
                | delete_statement
```

The preceding statement can be explained as any query language statement in JPQL, either be a select statement or update statement or delete statement. The term

select_statement, **update_statement** and **delete_statement** represents the abovementioned statements respectively. The '|' represents the OR operator.

In further sections, each statement is represented as a group of words along with the operators.

```
select_statement::= select_clause from_clause
[where_clause] [groupby_clause]
[having_clause] [orderby_clause]
```

The preceding statement says that the **select_clause** and **from_clause** are mandatory for the **select_statement**. However, the clauses within the [] are optional.

```
update_statement::= update_clause [where_clause]
```

The preceding statement is for the **update_statement,** which makes the **update_clause** mandatory and **where_clause** optional.

```
delete_statement::= delete_clause [where_clause]
```

The preceding statement is for the **delete_statement,** which makes the **delete_clause** mandatory and **where_clause** optional.

Now, the query language and its statements have been defined, the further rules that define the clauses are to be considered. Here, the **select** clause is much more complicated than the **delete**/**update** clauses. This is because select query provides more flexibility, while querying than the **update**/**delete** queries.

```
update_clause::= UPDATE abstract_schema_name [[AS]
identification_variable]
SET update_item {, update_item}*

delete_clause::= DELETE FROM abstract_schema_name [[AS]
identification_variable]
```

The following rules are specific to select clause as it allows the user to write select in many different ways. So, all those conditions should be considered while defining the rule for **select** clause.

```
select_clause::= SELECT [DISTINCT] select_expression
{, select_expression}*
```

This clause can have single select expression or multiple select expressions based on the query. But, at least one **select** expression is a must. Hence, the preceding clause is defined as one **select_expression** first (which is mandatory) and then

zero or more occurrences of **`select`** expressions which is denoted with curly braces {} followed by an asterix symbol (*).

Before explaining the **`select`** clause with its relevant factors, it is imperative to define few definitions which will be used widely in many other clauses as well. They are as follows:

```
single_valued_path_expression::= state_field_path_expression
| single_valued_association_path_expression

state_field_path_expression::= {identification_variable
| single_valued_association_path_expression}.state_field

state_field::= {embedded_class_state_field.}*simple_state_field

single_valued_association_path_expression::= identification_variable.
{single_valued_association_field.}*single_valued_association_field

collection_valued_path_expression::=
identification_variable.{single_valued_association_field.}*collection_
valued_association_field

constructor_expression::= NEW constructor_name
                  (constructor_item {, constructor_item}*)

constructor_item::= single_valued_path_expression
                     | aggregate_expression

aggregate_expression::= {AVG | MAX | MIN | SUM}
                      ([DISTINCT] state_field_path_expression)
                       | COUNT ([DISTINCT] identification_variable
                       | state_field_path_expression
                       | single_valued_association_path_expression)
```

The **`select`** expression can be defined as follows:

```
select_expression::= single_valued_path_expression
                     | aggregate_expression
                     | identification_variable
```

```
| OBJECT (identification_variable)
| constructor_expression
```

The preceding expressions can be used for querying normal objects. However, there can be embedded objects for which these queries may not work. In this case, first the parent object has to be queried and the embedded object should be accessed in the parent's context.

Many new features have been introduced in BNF as part of JPA 2.0. Syntax words like **`KEY`**, **`VALUE`**, **`ENTRY`**, **`TYPE`**, etc. have been introduced. Functions like **`CASE`**, **`COALESCE`**, **`NULLIF`**, **`SUBSTRING`**, **`CONCAT`**, etc. have been introduced. Few usages like IN for collection parameters and AS for select option have also been introduced in this version.

Conclusion

This chapter throws light on how queries are formed, the various kinds of queries, the different query samples and the common optimization techniques. This chapter also provides a few insights to the JPA BNF definitions.

The next chapter focuses on the run-time applications that helps JPA in providing more details. It also tries to include caching and transaction management activities done by the JPA layer.

Multiple choice questions

1. **JPQL stands for:**
    a) Java Persistence Query Language
    b) Java Persistence Query Logic
    c) Java Property Question Limit
    d) none of these

2. **CQL, SQL are types of:**
    a) Database query language, Information retrieval query language, respectively
    b) Information retrieval query language, Database query language, respectively
    c) Database query languages
    d) Information retrieval query languages

3. **Query hint is used for:**
    a) retrieving information from linked tables
    b) storing information into linked tables
    c) instructing the DB on how to execute the query
    d) executing the query based on previous queries

4. **Advantage(s) of stored procedures are:**
    i. less effort for development and debugging
    ii. stricter access to raw tables
    iii. maintenance friendly
    iv. provides batch processing
    v. minimize the data sent from DB to application
        a) i, ii, iii, iv only
        b) i, ii, iv, v only
        c) ii, iv, v only
        d) ii, iii, iv, v only
        e) all of these

5. **Native SQL queries are defined with the annotation:**
    a) @NativeQuery
    b) @SQLQuery
    c) @NativeNamedQuery
    d) @NamedNativeQuery
    e) none of these

Answers

1. **a**
2. **b**
3. **c**
4. **c**
5. **d**

Questions

1. What is a Query hint? Elaborate on how can it be used for query optimization in JPA.
2. Explain the three different query results and how they are used in various situations.
3. What is flush mode? What are the various types of flush modes?
4. How is pagination handled in JPA?
5. Explain BNF grammar with a simple example from JPA context.

Chapter 6

Entity Manager – Persisting, Caching, and Transaction

Introduction

While the previous chapter described the languages used for querying the data from the DB, this chapter details the techniques that are used to manage and support the queries. **Entity Manager** (**EM**) plays a very important role in accessing the DB entities by providing a single interface which encapsulates all the operations that can be performed on the DB. In short, all the operations—**Create, Read, Update, Delete** (**CRUD**)—on a DB entity are done via EM.

Structure

In this chapter, we will cover the following topics:

- Persisting
  - Persist, merge, remove
  - Advanced operations
- Transactions
  - Transaction management
    - Resource local
    - JTA

  - Join transactions
  - Transaction failures
  - Nested transactions
- Caching
  - Object cache/data cache/query cache
  - Cache types
  - Stale data
  - Clustered caching
  - Cache transaction isolation

Objectives

This chapter introduces various functionalities that are done by the EM to ensure that data handling is done accurately. EM uses many techniques that are used in general for storing and retrieving data. Here, these techniques are explained with respect to the DB context.

Persisting

Whenever an application needs to communicate to the DB via JPA, EM API is used at runtime. This API is responsible for all the DB interactions with respect to the application. It is very important to know about the persistence context while trying to understand the EM API.

Persistence context is the set of unique entity instances on which the operations are being performed. Being placed between the actual DB and the application, it works like a first level cache where the application accesses and saves them. It is responsible for managing the instance lifecycle. This makes sure that at any given point of time, each entity has a unique instance.

The persistence context can be of two types—transaction-scoped or extended-scoped. Transaction-scoped context is limited to a single transaction, whereas the extended-scoped context can stretch over multiple transactions.

An EM instance is related to a specific persistence context. EM API is used to create and remove the entities, search entities based on primary key, perform queries on entities in this linked context. In short, EM API is the interface to access persistence context entities. It includes the following operations:

- persist
- merge
- remove
- find

Since EM has an object-oriented API, there is no explicit update operation given. Whenever a commit happens on the transaction, it is the EM's duty to find out which objects in its context have changed and update them accordingly.

Hence, it is difficult to find a mapping between the API usage and database SQL/DML operations.

Persist

`persist` is used to insert a new object into the DB. This registers the new object in the persistence context corresponding to the EM. The actual insertion of the entity into the DB happens in the commit or flush phase of the transaction.

`persist` can be called only on new entity objects. It may also be invoked on objects that are available in the persistence context just to make sure that any of its related objects are new and require to be persisted (cascading persist). If this method invocation is done on existing objects that are not in the persistence context, it can cause an exception to be thrown or it may even get inserted into the DB based on the JPA provider implementation. This can further cause a database constraint violation exception if there are any constraints defined or else might even get inserted as a duplicate entity to the table. Hence, it is very important to understand when and how the **`persist`** method call should be performed.

If a new object is added and related to an object that already exists in the persistence context with cascading **`persist`** relationship, then the transaction commit or the context flush will be enough to insert the new object to the DB. An explicit persist call to the new object is not required in this case.

`persist` cannot be called on embeddable objects, collections or any other kind of non-persistent objects. Embeddable object will be persisted once the owning entity gets persisted.

Another important point to note is that persist can be invoked only within a transaction, otherwise out of transaction exception will be thrown. The primary key of the entity should be set in the object before invocation of persist if it is not a generated one.

When an object uses a generated ID, it may not be possible to set the ID before calling **persist**. Also, note that the ID may not be generated as soon as the persist method is invoked. To understand when and how can the generated ID be retrieved, it is necessary to grasp the entity lifecycle and ID generation mechanisms provided by JPA.

The four lifecycle states for an entity instance are – *new*, *managed*, *detached* and *removed*. When an object is created, then the entity instance is in new state. This means that the EM is not aware of the object being created as it is not in its persistence context. As soon as the EM persist call is made, the instance moves to managed state. This is when the EM becomes aware that this object is able to handle the generated IDs as per the definition. Generated IDs can be either of the following two categories:

- pre-allocated and are available to EM before commit
- allocated only after transaction commit

Hence, JPA has four strategies out of which three of them fall in the first category, whereas one falls in the second category.

- **`GenerationType.AUTO`**
- **`GenerationType.IDENTITY`**
- **`GenerationType.SEQUENCE`**
- **`GenerationType.TABLE`**

`AUTO` is the default strategy where JPA uses the underlying DB's strategy for generating IDs. **`IDENTITY`** is based on the database's auto-increment column and hence, the ID can be generated only once the entity is inserted into the database. **`SEQUENCE`** is based on a sequence that has been created in the DB. When an entity is persisted, JPA ensures that the ID value is set in the entity before the commit. **`TABLE`** is based on the ID values stored in one of the DB tables. In this case as well, JPA inserts the ID value as soon as the entity is persisted.

```
EntityManager em = getEntityManager();
em.getTransaction().begin();
Student stud = new Student();
stud.setFirstName(“Harry”)
Address = new Address();
address.setCity("London");
stud.setAddress(address);
em.persist(stud);
em.getTransaction().commit();
```

***Code 6.1**: Sample java code to persist a Student object in DB*

In the preceding example, first the EM is retrieved and the transaction is started. Note that only the **`stud`** object is persisted and that there is a new object called A**`ddress`** which is related to **`Student`** object. However, persisting the **`stud`** object will be sufficient since the relationship between these objects are cascade persist. Hence, the commit will take care of inserting both the **`student`** and **`address`** entities to the DB.

Cascading persist

When persist is invoked on an object, the same operation will be performed on every object related by cascade persist. This is a very important relationship among objects which can be very crucial while deciding the performance of the application. Consider an object A is related to object B and is not using cascade persist. If the instance of object B is new, one has to make sure that the new instance of B is persisted before persisting A provided A is the owner of the relation. If not done in that order, an exception may be thrown. So, normally programmers consider to mark every relationship as cascade persist in order to avoid worrying about invoking persist on every object. However, that can also lead to many issues.

One important issue is the performance. Whenever an object is persisted, all the related objects need to be examined whether they refer to any new objects. This can go deeper when there are multiple levels of object relations with cascade persist. Consider a large set of objects related to each other. If the object relationship can be expressed as a graph of n objects, and only the root object is persisted, then it may not be a problem as there would be only n persist operations performed. But if every object is persisted, it can cause n^2 persist operations that causes a huge performance issue. JPA specification only defines to apply persist to all objects whether new, existing or already persisted.

Another issue is that even after removing an object, and invoke persist on the related object, the removed object might get persisted if the relation is cascade persist. This may be desired in certain cases but not always. So, whenever an object is removed, its reference from all the related objects with cascade persist relationship should also be removed for the desired outcome. Otherwise, the remove will not be taken into consideration and hence the removed object will be persisted back. To understand this better, it would be good to have a look at the various state transitions of an entity within the persistence context. The transition diagram is as follows.

Whenever an entity object is newly created by the application, it is not known to the EM and hence not managed by it. JPA enforces the entities to be moved to the persistence context so that it can be managed by the EM and make use of the flexibilities provided by the EM for easier application development.

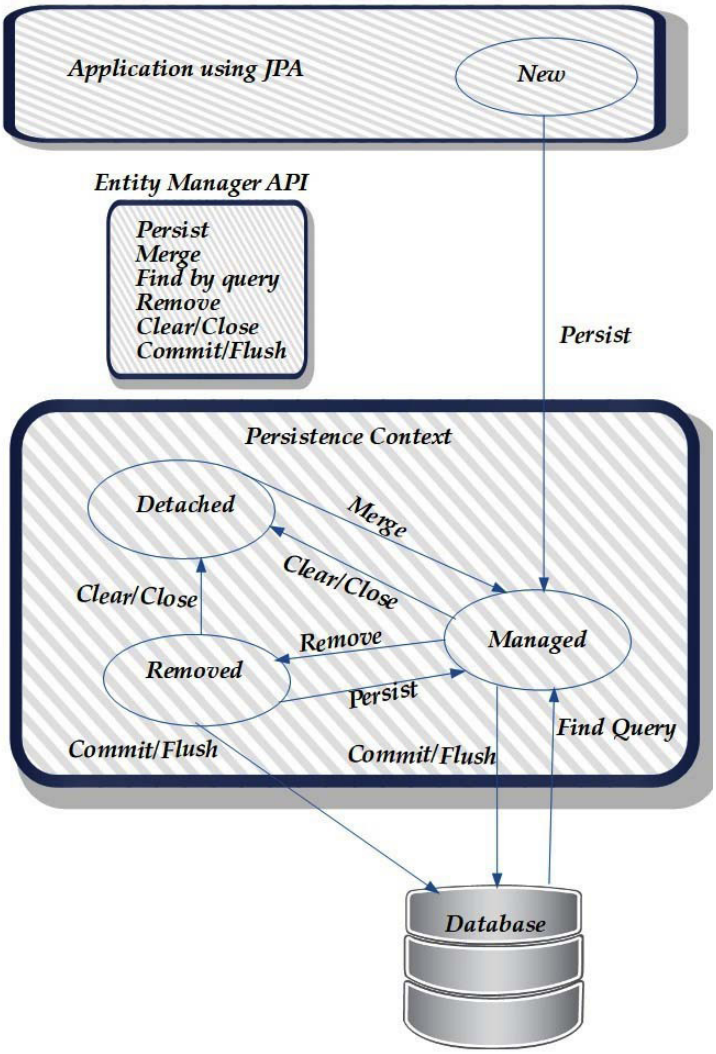

***Figure 6.1**: Entity life cycle states*

Consider object A in the managed state being removed. So, it moves from managed state to removed state as shown in the figure above. However, the reference of A was not removed from object B, which is related to A via cascade persist. Now, while invoking a persist on B will cause the persist on A to be invoked since the reference still exists and hence, this object gets back to managed state by overlooking the remove operation.

Now, consider if the reference of A was removed from B before calling persist on B. Then, on invoking persist on B will not cause an invocation of persist on A and it remains in the removed state itself. And finally, when the transaction gets committed, the entity A will be deleted from the DB since it has been in the removed state. Thus, if the relationship has cascade on persist, make sure that removing an object and its references are equally important to get the object deleted from the DB.

The ideal recommendation for indicating relationships as cascade persist are those that are composite or those that are privately owned by the object.

Merge

When there are changes made in a detached object that needs to be stored into the DB, a **`merge`** operation is used. This operation does not update the DB object directly, instead it moves the entity state from detached to managed and on transaction commit or context flush, the object gets updated in the DB.

`merge` operation is mostly confused with the update operation. However, there is a huge difference. Note that merge is always applied on a detached object, i.e., an object that was either cloned or serialized or read in a different transaction of the EM or read by a different EM itself. **`merge`** operation behaves differently in different scenarios as follows:

- If the object is in managed state, nothing will be done other than cascade merges, if there are any (i.e., if any related object has been in the detached state and the relation has cascade merge, then it would be moved to managed state).
- If the object is in new state, a copy of the object is moved to managed state and this copy is returned as persisted object.
- If the object is detached, then a copy of the object moved to managed state and returned as persisted object.
- If the object is detached but another object with the same ID exists in managed state, then the state of the detached object is copied to the managed object and returned.
- If the object is removed, then it throws **`IllegalArgument`** exception as per the JPA specifications.

Similar to **`persist`**, **`merge`** operation also needs to be invoked within a transaction. If invoked otherwise, an exception will be thrown. Also, this operation can be invoked only on **`Entity`** objects. Embeddable objects will be taken care automatically while merging the owning **`Entity`** objects. A sample Java code that shows how merge can be done to a detached object is given in the *figure 6.3.*

```
EntityManager em = createEntityManager();

Student studOld = em.find(Student.class, id);
```

```
em.close();
...
em = createEntityManager();
em.getTransaction().begin();
Student studNew = em.merge( studOld );
em.getTransaction().commit();
```

***Code 6.2**: Sample java code to merge an object in DB*

Here, the first student object named **`studOld`** is created by an EM and then the EM is closed. Then, the next call to **`createEntityManager()`** will provide an entirely new EM instance. Now, if the changes made in **`studOld`** object have to be restored, a merge operation can be performed as this object was created by another EM and thus **`studOld`** object would be in detached state. The final commit will get the changes in **`studNew`** to be stored into the DB which would reflect the changes made in **`studOld`** object as well.

Cascading merge

Cascade merge performs in the same way as that of cascade persist. All the related objects marked as cascade merge in an object will be checked for further merge and hence it is quite normal for the developers to think that cascade merge can be marked on every relationship. However, as seen in cascade persist, it mainly causes performance issues while trying to pass through a large collection of objects.

Another issue is that if the detached object was corrupted, it would create data inconsistency problem while trying to merge the detached corrupt object with the already existing managed instance. There can be cases where another user made changes to an object which remains detached in this context and merge would either cause the changes of that other user to be removed or throw an **`OptimisticLockException`** based on the locking policy of the DB.

Similar to **`persist`**, **`merge`** also considers the same ideal recommendation for cascade merge.

Transient variables

Java has transient variables, i.e., variables that are not serialized while transferring from one resource to another. It is very important in certain cases to use transient variables as there may not be any requirement to get the entire huge object from the DB and only fetch the required fields. Hence, extra care should be provided while merging a transient variable in JPA.

If an object is serialized with a few fields marked as transient, then this object will contain those transient fields as null. Since it is a serialized object, it is in the detached state and a merge on this object or any other object invoking this object via cascade

merge, will cause the null values of the transient fields to be updated to the managed object and further updated in the next commit/flush to the DB. If the null values are updated in the non-nullable columns, then an exception would be thrown either during commit/flush or during the merge itself.

In case of entity objects, it is highly recommended to use LAZY loading instead of marking the objects as transient. Please refer earlier chapters which provide more details on LAZY loading. Even though JPA provides the specification for merge, many JPA providers have extended these operations allowing a shallow or deep merging. There are few which even allow merging without reference merging.

Remove

`remove` operation is used to mark the object to be deleted in the DB. Consider this operation has been done on an object. This means that this object would be deleted from the DB on transaction commit/context flush. Similar to all the previously mentioned operations, **`remove`** can be invoked only within a transaction and that too only on managed **`Entity`** objects. It can be neither invoked on detached objects nor on embeddable objects, collections or non-persistent objects. A general rule to follow while using this operation is to first find the object and then remove it.

```
EntityManager em = getEntityManager();
em.getTransaction().begin();
Student stud = em.find(Student.class, id);
em.remove(stud);
em.getTransaction().commit();
```

***Code 6.3**: Sample java code to remove an object from DB*

Here, first the **`student`** object to be removed is queried using the find operation. Then, it is removed in order to make sure that the object on which the remove is done is actually a managed one. This also makes sure that this is not a detached or any other non-persistent object.

Cascading remove

Cascade remove operation is similar to that of other cascade operations. However, if an object relation is marked as cascade remove, and the object is removed from a collection or the reference of the object is removed, it will not cause the object to be actually moved to the state removed. Consider an object A which has a OneToMany relationship with object B. This means that the entity A will have a collection of B objects in its definition. If this relationship is marked as cascade remove, and if object A is removed, then it will only remove the references inside the collection. However, the actual B objects will not be moved to remove state and hence, will not

be removed from the DB. In general, it can be understood that for most of the cases where the relationship is OneToOne and the relation is marked as cascade remove, then that related object will also be removed. But if the relationship is marked as OneToMany or ManyToOne, the related objects will not be removed, instead they might get dereferenced, which leads to inconsistent DB data.

However, all the previously discussed points are mostly based on the JPA provider implementations. Also, as part of JPA 2.0, orphan removal option has been included which will take care of removing such dereferenced entities. *Chapter 4, Relationships – Types and Strategies* already explains orphan removal in detail including an example as well.

Reincarnation

In certain cases, it may be required to change the type of the object as part of inheritance. For example, consider a **`Person`** entity becoming a **`Student`** entity after joining a **`University`**. Also, consider that the **`Student`** entity extends **`Person`** entity and the ID for both the tables is the mobile phone number. So, when an existing **`Person`** entity joins the **`University`**, it should be removed from the **`Person`** table and added to the **`Student`** with the same ID as that of the **`Person`**.

Since this is to change the type of the entity, this is not supported by JPA and it can be considered as a bad object model design. However, this can be achieved if the first entity (**`Person`**) is removed in one transaction and the second entity (**`Student`**) is persisted in another transaction. But it can be only treated as two different entities altogether and not the same entity changing its type. Performing both these operations in a single transaction can lead to many more complexities as it mainly depends on the JPA provider implementation on how the same ID with different objects should be handled. In most of the cases, this will not work as expected.

Usage of native SQL queries can be done to make the updates according to the business logic. However, this should not be promoted and other techniques should be adopted to avoid reincarnation.

Advanced operations

Apart from the operations above, there are many advanced operations available with EM API. Those operations are—Refresh, Flush, Clear, Close, Get Reference, Get Delegate, Unwrap. Out of these operations, unwrap is only available from JPA 2.0 onwards.

Refresh

As the name suggests, the refresh operation revives the state of the object from the DB. This can be done only on managed **`Entity`** objects which has to be within an active transaction. Note that if there are any new changes in the instance of the object within the persistence context of the current transaction, those will be reverted and

the object state will be matching the current state of the object in the DB. Hence, it is always very important to make sure that refresh is not called without any genuine reason as this might cause the changes in the persistence context to be ignored. Or it should be made sure that the context changes are flushed to the DB before the refresh is being called.

Since refresh operation works only on managed objects, it is very important to first find the object with the active EM. This makes sure that the object is in managed state and refresh operation can be performed. Also, this action can be cascaded to any relationships marked accordingly. However, there is a dependency on the fetch type to access the contents from the DB. If the fetch type is LAZY, then the refresh will also do the same kind of fetch.

In general, refresh is used in two scenarios:

1. Revert changes in the current managed object.
2. Refresh the stale cached data if the JPA provider supports caching.

From JPA 2.0 onwards, a set of query hints for refresh has been defined. This will be dealt with more details in the upcoming section *Caching*.

Flush

The **flush** operation can be considered as a reverse to the refresh operation. This updates the database with all the current changes in the active EM even before a commit to the transaction is done. By default, the changes from the persistence context will be written back to the DB only on a transaction commit. This action can be used to update the object state in the DB even before the commit is done. Flush has several usages:

- When a query execution depends on the previous add/update/delete operations in the same transaction, flush can be used to return the latest changes made in the persistence unit.
- When the insert operation needs to return the ID of the inserted record which uses identity sequencing for generating ID.
- Store the current state of the persistence context as soon as any DB errors occur and before moving to error handling.
- Supports batch processing in a single transaction so that the changes from each batch are flushed to the DB and the consecutive batch starts execution.

The following code depicts the usage of **flush** operator after the persist operation is completed.

```
public long createStudent(Student stud) throws StudCreateException {
  EntityManager em = getEntityManager();
  em.persist( stud );
  try { em.flush(); }
```

```
  catch (PersistenceException exception)
    { throw new  StudCreateException (exception); }
  return stud.getId();
}
```

***Code 6.4**: Sample java code which uses flush to update the DB and provide the latest generated ID*

This means that the changes in the **`student`** object gets updated to the DB and the consecutive new transactions can have the latest **`Student`** object even before the current transaction is committed.

Clear

`clear` operation, as the name suggests, clears all objects from the persistence context. This means that all the objects in the persistence context becomes detached and are no longer tracked by the EM. Whenever a transaction commit fails, roll back is performed which abandons all the changes and restarts the persistence context. This is very much similar to a clear operation. However, the rollback can happen only at the end of the transaction on commit failure. But clear operation can be executed in between the transaction.

When the EM is being managed by the application, it would be generally using the same EM throughout the application. So, invoking a **`clear`** operation in this case becomes an application design consideration as it can have all the objects in the same EM from the start of the application. For JTA managed EM, at the end of each JTA transaction the **`clear`** operation is invoked as part of housekeeping activities.

Clearing the context becomes crucial in long running batch operations. The huge batch operations can be split into smaller chunks and after the completion of each chunk, a **`clear`** operation can be invoked so that the context does not keep growing causing performance issues.

Close

When the EM is application managed, then **`close`** operation is used to release its resources. The life expectancy of an EM can vary as long as either a transaction, or a request or a user's session. Normally, EM is active per request and gets closed at the end of the request. Once closed, all the objects in the EM are detached. So, it is very important to make sure that all the LAZY relationship values in the objects are loaded before the EM is closed.

Get reference

`getReference` operation provides the reference to an object without actually loading the object. The returned object will be a kind of proxy object with only the primary key values loaded. The other values can be lazily loaded and hence, can avoid

loading the entire object in the first go. However, the behavior of this operation is dependent on the underlying JPA provider implementation.

This is mainly used when an object's ID is required as a related object reference in an insert or update operation. Hence, the entire object need not be loaded and only the reference can be used to complete the operation.

It is very important to understand that this operation will not verify whether the referenced object actually exists in the DB or not. If the referenced object was removed from the DB due to some reason, then the subsequent operations using the reference object can cause exceptions or foreign key constraint violations.

```
EntityManager em = getEntityManager();
Teacher classTeacher = em.getReference(teacherId);
Student stud =  new Student();
……
em.persist( stud );
stud.setTeacher(classTeacher);
em.commit();
```

***Code 6.5**: Sample java code which uses getReference operator*

In the preceding example, the **`Teacher`** object is the proxy object which only has the ID value set. After the **`Student`** object is created and persisted, while using the **`setTeacher`** method, it only requires the ID of **`Teacher`** object as the foreign key and hence, the proxy object is sufficient.

Get delegate

One way to access the JPA provider's underlying EM implementation in a **Java Platform Enterprise Edition** (**JEE**) managed scenario is to use the **`getDelegate`** operation. The returned object will be a proxy EM instance that forwards the requests to the active EM of the current **Java Transaction API** (**JTA**) transaction. Note that it is very important to have this operation within the JTA transaction or else, the proxy EM will not be able to identify the active EM of the current transaction. This can lead to undefined results.

This operation also depends on the server and can vary accordingly. i.e., if a code block was used in one type of JEE server instance for a JPA provider, it may not be completely reusable in another JEE server instance.

The following code is used to get the Hibernate **`Session`** object from Tomcat EE implementation:

```
EntityManagerImpl hibernateEMImpl = (EntityManagerImpl)em.getDelegate();
Session sess = hibernateEMImpl.getSession();
```

When it is being moved to JBoss, the session is obtained in a single step as follows:

```
org.hibernate.Session sess = (Session) em.getDelegate();
```

Hence, the code using **`getDelegate`** of Hibernate in Tomcat server might require changes when migrating it to be used in JBoss.

Another option to retrieve the underlying implementation is to use the **`EntityManagerFactory`**. Note that this may be available only in particular servers and not in all of them.

In JPA 2.0, unwrap operation has been introduced, which is a more generic way of accessing the underlying EM implementation.

The code above used to retrieve the **`Session`** in JBoss using **`getDelegate`** operation gets changed to a more generic call in JPA 2.0 using unwrap operator as follows:

```
Session sess = em.unwrap(org.hibernate.Session.class);
```

Transactions

A transaction is a group action that contains either a single operation or multiple operations of the same/different type that is considered as a block and the entire set of action results collectively is taken as the transaction result, i.e., if a set of operations are performed, and any one of them fail, then the entire transaction is considered as failed. It is a must for a transaction that all the individual operations within the transaction should succeed. Hence, there is only the entire unit's success or failure.

Transactions are considered as a crucial part of persistence since it maintains data integrity, either changed according to the requirement or remain unchanged. It makes sure that there would not be a state where the data might be changed half way of the transaction and then aborted. If the transaction fails half way, it reverts the changes made till then and goes back to the state where it was before starting the transaction. This reverting is called **transaction rollback**.

Broadly transactions are divided into two –**Container Managed Transactions** (**CMT**) and **Bean Managed Transactions** (**BMT**). In CMT, the boundaries of the transactions are set by the EJB container. It simplifies the development by avoiding the boundary demarcation code in the bean, and hence makes it easier. Typically, when the bean method is invoked, a transaction is started and ended as it exits the method.

Whenever, the transactions should be managed with more flexibility, then it should be managed by the code itself and hence BMT is used. BMT uses the following operators to mark the transaction boundaries—**`begin`** to mark the start of the transaction, **`commit`** to mark the success condition and **`rollback`** to mark the failure condition.

Transaction management

There are two types of persistence context and each one of them have different mechanisms for transaction management:

- Container-managed persistence context
- Application-managed persistence context

In a container-managed context, it is the responsibility of the container to be aware of the ongoing transactions and propagate the context to distinct enterprise elements. This can be only used with JTA for both CMT and BMT.

```
@Stateless
public class StudentBean{



@PersistenceContext
private EntityManager em;




public void studTest(){
  Student stud = new Student();
  stud.setName("Mary");
  em.persist(stud);
 }
}
```

```
@Stateless
@TransactionManagement
         (TransactionManagementType.
BEAN)
public class StudentBean{

@PersistenceContext
private EntityManager em;

@Resource
private UserTransaction userTx;

public void studTest(){
  Student stud = new Student();
  stud.setName("Mary");
  userTx.begin();
  em.persist(stud);
  userTx.commit();
 }}
```

***Code 6.6**: Container managed persistence context using JTA for CMT (left) and BMT (right)*

In the preceding code, the existing EM is injected to the bean using **`@PersistenceContext`** in both cases (CMT as well as BMT). The default **`TransactionManagement`** type is container and that can be specified as **`BEAN`** to make it bean managed. However, in case of BMT, the user transaction is injected to the bean using **`@Resource`** in order to set the boundaries of the transaction. Hence, the transaction begin and commit can be seen in the bean code for BMT.

In application-managed context, it is the sole responsibility of the application to create and dispose the context. An **`EntityManagerFactory`** is injected into the

enterprise components and from that, an EM instance is fetched and used to interact with the persistence context. This can use both **RESOURCE_LOCAL** as well as JTA.

The following code shows how the above container managed persistence context bean code (*Figure 6.7*) can be converted to application managed context using **RESOURCE_LOCAL** transaction type.

```
@Stateless
public class StudentBean{

@PersistenceContext
private EntityManagerFactory emf;

public void studTest(){
  Student stud = new Student();
  stud.setName("Mary");
  EntityManager em = emf.createEntityManager();
  EntityTransaction et = em.getTransaction();
  et.begin();
  em.persist(stud);
  et.commit();
  em.close();
 }
}
```

***Code 6.7**: Application managed persistence context using Resource local type*

In the preceding case, the entity manager factory is injected. The EM and the entity transaction instances are obtained in the bean code. The transaction is maintained by the application itself by setting its boundaries in the code. Another way to use transactions in application managed persistence context is to depend on JTA itself. Once JTA comes into picture, it can be either CMT or BMT. The following code shows how the previous code (*Figure 6.8*) can be converted to JTA transaction type.

```
@Stateless
public class StudentBean{

@PersistenceContext
private EntityManagerFactory
emf;

public void studTest(){
  Student stud = new Student();
  stud.setName("Mary");
  EntityManager em
     = emf.
createEntityManager();
  em.persist(stud);
  em.close();
 }}
```

```
@Stateless
@TransactionManagement
         (TransactionManagementType.
BEAN)
public class StudentBean{

@PersistenceContext
private EntityManagerFactory emf;

@Resource
private UserTransaction userTx;

public void studTest(){
  Student stud = new Student();
  stud.setName("Mary");
  userTx.begin();
  EntityManager em = emf.
createEntityManager();
  em.persist(stud);
  userTx.commit();
  em.close();}
}
```

***Code 6.8**: Application managed persistence context using JTA for CMT (left) and BMT (right)*

Similar to the container managed persistence context using JTA, the code remains more or less similar apart from the fact that the injected object is not the EM but its factory class. Being application managed, the EM is created within the bean and closed when the persist operation is completed.

From the preceding examples, it is clear that JPA allows to handle transactions using two different mechanisms:

- Resource local transactions
- JTA integration

The first one is used mostly for **Java Platform Standard Edition** (**JSE**) or for application managed transactions in JEE. However, the second one is mostly used in container-managed transactions. Note that in JPA all changes made to persistent context objects are part of the transaction and hence transactions are at object level.

Resource local transactions

Resource local transactions are used either in standard Java, or in non-managed mode in enterprise Java. The following XML configuration should be present in the **`persistence.xml`** in order to use resource local transactions.

```
<persistence-unit name="test" transaction-type="RESOURCE_LOCAL">
<non-jta-data-source>jdbc/test</non-jta-data-source>
</persistence-unit>
```

Note that the **`<non-jta-datasource>`** element should be a data source server which is not JTA managed. When an EM is created, technically a transaction is also started. But unless the begin is performed explicitly, operations like **`persist`**, **`merge`** and **`remove`** cannot be invoked. However, this depends on the underlying JPA provider implementation on how the object changes would be handled if the transaction is not started explicitly by a **`begin`** call.

It is recommended to create an EM for each transaction so that the persistence context does not have any stale objects remaining and previously managed objects can be marked for garbage collection. When the transaction commits, the EM should be cleared/closed so that the unwanted objects can be garbage collected and no stale data remains. However, if the transaction commit fails, the objects are marked as detached. Only the changes in the database are rolled back (if any). The object changes are ignored and since they are detached, it may not be used further.

JTA transactions

This type of transaction can be used only in enterprise Java where it is managed by the container itself **Entity Java Bean** (**EJB**). The following XML configuration should be present in the **`persistence.xml`** in order to use JTA.

```
<persistence-unit name="test" transaction-type="JTA">
<jta-data-source>jdbc/test</jta-data-source>
</persistence-unit>
```

If JTA transactions are used, then the **`<jta-datasource>`** element should be a data source reference server configured as JTA managed. JTA transactions are defined in two different ways:

- JTA user transaction class
- Implicitly through session bean usage/methods.

`UserTransaction` is used for defining JTA transactions. In a **`SessionBean`**, each bean method invocation gets assigned to a JTA transaction. **`UserTransaction`** can be retrieved using JNDI lookup in most application servers or from the EJB context of a **`SessionBean`**.

Join transaction

Consider the example given in *figure 6.7* on the usage of JTA transaction on BMT for an application managed persistence context. In case of BMT, first the JTA transaction is started and then the EM is created. In that case, since the EM was created in the JTA context, it will naturally join the JTA and the changes will be updated on the JTA commit.

Otherwise, the EM transaction will not join the JTA automatically and will need explicit joining on the transaction as shown in the *code 6.9*

```
public void studTest(){
  Student stud = new Student();
  stud.setName("Mary");
  EntityManager em = emf.createEntityManager();
  userTx.begin();
  em.joinTransaction(); // to join the active JTA transaction.
  em.persist(stud);
  userTx.commit();
  em.close();
 }
}
```

***Code 6.9**: EM joining the active JTA transaction while using BMT*

Here, the EM creation happened before the **`userTx.begin()`** statement. This means that the EM was created outside the JTA transaction scope. Hence, for the JTA commit to actually perform the DB update, an explicit EM's join transaction is to be added before the commit happens.

This kind of code is normally used in stateful architecture where an EM with extended scope is used. In this stateful mechanism, the server stores the information on a client connection until the client's session is over. The advantage of using this is that the objects can be read using one transaction and committed using a new one without worrying about merging those objects. However, extra care should be taken to make sure that the EM instance creation happens in the absence of any active JTA transaction. Otherwise, the EM would naturally join the active JTA transaction.

Transaction failures

In order to deal with commit failures, there should be sufficient know-how about the application and understanding of various transaction states and how the fix can be

done when the failure happens in each state. However, JPA spec does not state any direct ways of handling errors or commit failures.

Whenever a transaction commits fails, it is rolled back consistently, and all managed objects are detached by clearing the persistence context. It is not advisable to keep the persistence context alive even after rollback as the state of the objects in the context may also be inconsistent due operations (like lock version increments etc.). However, few JPA provider implementations allow extended API for error handling of queries or commit failures.

When the persistence context is application managed and the transaction type is **`RESOURCE_LOCAL`**, then it is easier to implement error handling. Whenever a commit fails, merge all the managed objects of the failed transactions to a new EM and then commit the new EM. Note that in this case, all the IDs and optimistic lock versions that were assigned/incremented might be checked for and reverted.

Another involved way is to work with a non-transactional EM. Whenever it is time to commit, all the non-transactional objects are merged into a new EM and committed. If this fails, only the new EM gets affected. The actual EM remains unaffected. This helps in understanding the actual error and getting it persisted using another new transaction.

Nested transactions

A nested transaction is one that has other child/sub transactions within itself. If the parent transaction is rolled back, irrespective of whatever operations are done in the child transactions, all of them would be rolled back. JPA and JTA both have no support for nested transactions.

A few rules while using nested transactions are as follows:

- When the child transaction is active, the parent transaction can only either commit or abort or create more child transactions. No other operation is allowed.
- Whenever a child transaction is committed, it has no effect on the parent transaction. However, the parent transaction can view the changes made by the committed child transaction. When the parent commits, only then these changes will be visible to all the other child transactions.
- If the parent transaction gets committed or aborted when it has active child transactions, the child transactions will also perform the same operation as that of the parent, i.e., if parent commits its changes, then all the child transactions commit their changes to the DB. Else, if parent aborts, then all the child transactions will also get aborted.

Any visible changes made by the nested transactions are not visible in DB unless the parent commits. Similarly, the locks held by a nested transaction are not given back till the parent commits. The only limit to the depth of nesting is based on the memory size.

Caching

JPA has two levels of caching. The first level is the persistence context itself, which has been discussed so far. The second level can be considered as L2 cache and is being discussed in this section. Caching is the way of storing copies in a temporary storage for faster access and it is considered as one of the most relevant performance optimization mechanisms. A cache can store anything–objects, data, meta-data, meta-data relationships, query results, database statements, database connections, etc. The main interest here is for caching of objects or their data. It is necessary that the object identity should be preserved within the transaction or within an extended persistence context. However, it is not required to have support to caching span across transactions or persistence contexts.

JPA 1.0 does not have any formalization for shared object cache. It is totally based on the provider implementation whether a shared object cache should be supported or not. However, in JPA 2.0 **`@Cacheable`** annotation can be used to allow caching on a particular class. This can also be done with equivalent XML configuration.

```
@Entity
@Cacheable
public class Student
{ …… }
```

***Code 6.10**: Usage of @Cacheable*

The default mode of caching can be configured via XML or via code as shown in *figure 6.10*. In code, the **`EntityManagerFactory`** properties are set accordingly to reflect the caching strategy via **`SharedCacheMode enum`**. In XML, **`<shared-cached-mode>`** element is used to define the same.

```
Properties props = new Properties().add(
   "javax.persistence.sharedCache.mode",javax.persistence.
SharedCachedMode.NONE);
EntityManagerFactor emf = Persistence.createEntityManagerFactory(“PU”, props);
```

```
<persistence-unit name="ACME">
    <shared-cache-mode>NONE</shared-cache-mode>
</persistence-unit>
```

***Table 6.1**: Caching strategy for a persistence unit via code (top) or XML (bottom)*

There are different values for the previously mentioned **`SharedCacheMode enum`**.

- **`ALL`**: All the entities, entity-related state and data are cached.
- **`NONE`**: Disable caching for this persistence unit.
- **`ENABLE_SELECTIVE`**: Enable caching for all entities which have Cacheable value as true.
- **`DISABLE_SELECTIVE`**: Enable caching for all entities except those that Cacheable false.
- **`UNSPECIFIED`**: Caching depends on the provider default specifications.

The caching done can be of three types–Object caching where the entire object along with its data, structure and relationships are cached or data caching where the database row data is cached or query caching where the entire query results are cached.

Object cache

This type of cache stores the entities as such and in the same format as it is in Java. Whenever a cache-hit happens, it can be easily retrieved without any conversions. In certain cases, transient data may also be cached automatically. So, all these scenarios should be taken care of while using the objects from the cache. The main usage of this kind of cache is to store read-only data as it is extremely efficient and the major problem while using cache (updates and stale data) is not an issue in this case.

Object identity

Whenever objects are being used, their identity becomes an important matter. Object identity in Java means that if two objects refer to the same reference, then they are identical. Consider A and B as two objects and if they refer to the same logical object (i.e., point to the same memory location), then A == B returns true. In case of application managed EM, the object identity is maintained throughout the EM. However, in case of JEE managed EM, this is only within a transaction.

```
Student student1 = em.find(Student.class, 1);
Student student2 = em.find(Student.class, 1);
if (student1 == student2){
      System.out.println(“Both are same”);
}
```

***Code 6.11**: Object identity example1*

In the given code, both the objects **`student1`** and **`student2`** will be the same local object. Hence, on executing the code above, the statement will be printed. Note that

even if the objects are accessed in any way, it remains the same. However, when different EMs are being used, the object references change and hence their identity.

```
EntityManager em1 = factory.createEntityManager();
EntityManager em2 = factory.createEntityManager();
Student student1 = em1.find(Student.class, 1);
Student student2 = em2.find(Student.class, 1);
if (student1 != student2){
       System.out.println("Different objects");
}
```

***Code 6.12**: Object identity example2*

In the example above, two different persistence contexts are created via **`em1`** and **`em2`**. Hence, on executing the code above, the statement gets printed. This concept of object identity helps to maintain different copies of the same object within each transaction making it easy to isolate the changes from other users of the application. Within the same EM, if multiple copies of the same object are used, then it would be difficult to maintain data integrity as it will have to update multiple copies while an object update happens in the transaction.

Data cache

In data cache, only the data is stored not the object. It is basically the database row corresponding to the object instance. The main advantage is it is easy to implement and does not have to do any complex memory management for storing relationships or object identities. The e-data and the relationships should be retrieved from the database. In order to make this easier, some provider implementations along with a data cache, supports relationship cache or query cache.

Caching relationships

This is normally done only for OneToMany and ManyToMany relationships. OneToOne and ManyToOne normally have the reference ID as the object ID. Yet, in a reverse OneToOne mapping, it is the foreign key and not the primary key that will be available as the reference ID and hence, the disadvantage is that when there is a cache hit, the object created from this needs to be cached. So, in a relationship cache only the object ID and its relationship name is stored. No object or data is stored so as to stay away from duplicate and stale data issues.

Some data cache implementations store the structure instead of the database row. In such cases, the relationship cache is also maintained as part of this data cache. When there is a cache hit, the related objects in the data cache are taken one after the other. The likely outcome is that if the related object is missing in the cache, it would be

fetched from the DB, which can lead to a very poor performance. In order to lighten this issue, some implementations support batch queries for select operations.

Query cache

In this kind of cache, neither the objects nor the data is cached. Instead, the query results are cached. This is very useful when the query is not based on ID and the result expects multiple objects. The key of the cache is based on the query name and parameters. If a `NamedQuery` is being frequently executed, then its results can be added to the query cache just by executing it for the very first time. However, it is very important to make sure that the query results are not stale data. Hence, it interacts with the object cache and assures that it is up to date as much as the object cache. Similar to object caches, query caches also have invalidation options.

Cache types

There are various diverse caching types and most of them are based on the cache eviction policies, i.e., the way an element is removed from the cache when there is a requirement for a cache storage area. The list of cache types include:

- **LRU cache eviction**: Here X number of most recently used objects are kept in the cache and least recently used are discarded when the cache becomes full.
- **Full or no eviction**: It is forever caching of what all has been read till date. This is not at all good idea for databases that are large.
- **Soft**: When the memory is low, it uses Java garbage collection hints to release objects from the cache.
- **Weak**: Every active object in use is placed in the object cache.
- **L1**: It is the transactional cache of the EM which is not a shared cache.
- **L2**: Cache is stored in the `EntityManagerFactory` so that it can be shared among all Entity Managers.
- **Data cache**: Database rows or the data is cached.
- **Object cache**: Objects are cached directly.
- **Relationship cache**: Relationships are cached.
- **Query cache**: Query result sets are cached.
- **Read-only**: It is a cache that only stores, or only allows read-only objects.
- **Read-write**: It is a cache that can handle inserts, updates and deletes.

- **Transactional**: It is a cache that can handle inserts, updates and deletes, and adheres to transactional ACID properties.
- **Clustered**: Whenever an object is updated or deleted, these objects go stale and it has to be informed through the broadcast of invalidation message.
- **Replicated**: It refers to a cache that uses mechanism to broadcast objects to all servers when read into any of the server's cache.
- **Distributed**: It refers to a cache that spans across several servers in a cluster.

Stale data

While caching, it is a copy of the data in the DB gets stored for easier and faster access and hence, there can be possibilities that the copy version gets out of sync from the original value. This is called **stale** or **out of sync data**. This is not at all an issue for read-only data but it can be a huge issue for read-write data. There are different techniques to deal with this kind of data.

First level cache

In JPA, EM's persistence context itself acts as the first level cache. The object is cached for the duration of a transaction or a request based on the context configurations. So, if the object is accessed via the JPA, then it would have the latest object with the in-memory changes. However, in cases where native queries are used, it would be directly fetched from the DB and hence the in-memory changes would not be available. Even though the object changes will be put back to the DB only at the transaction commit, it is possible to push the changes to the DB in between the transaction using a flush operation. Hence, it is always important for the application developer to understand where and how to use the native queries and if required what all housekeeping activities have to be done in order to achieve data consistency.

In fact, some JPA providers even support the cached changes being merged along with the DB query results to maintain a consistent view of the data to the application. However, this may even become complex if the query is not a simple one.

The first level cache, if used for long, have chances of having stale data and can be cleared using the `clear` operation. There is a `refresh` operation as well, which would refresh the objects from the DB by overwriting the cached ones. Note that the refresh operation would overwrite all the changes in the cached object and have the DB changes updated.

Second level cache

The second level cache is not a requirement of JPA, but is an optional one. It covers different transactions and can be used to share objects among different EMs. Although most of the JPA providers support second level cache, its definition

and implementation are provider specific and may vary. In certain cases, second level cache is supported by default, whereas in other cases, it can be a third-party implementation plugged into the provider through configurations.

Stale data is a main issue in second-level cache. When a single application accesses the DB server, and whenever a native query (mostly insert/update/delete) is executed directly on the database, the data in the second level cache becomes out of sync. This can be handled by automatically invalidating the second level cache whenever a **Data Manipulation Language** (**DML**) query is executed. This should be however, taken care by the JPA provider.

In scenarios where multiple applications access the same DB server, then it becomes tough to deal with stale data in second level cache. When a transaction fetches an object and stores it in the cache, there can be yet another transaction deleting this particular object from the DB. Since there are multiple applications accessing the DB server, chances are high for the condition above to occur. Mostly, second level cache is used for find operations and if the condition above occurs, the data returned may not be actually available in the database and hence, data consistency is lost. Also, there can be cases where the DB got updated and the cache still has the old stale data. To avoid many such conditions, optimistic locking can help to a certain extent. However, locking should be used with proper understanding otherwise it would lead to performance issues and in worst case, to deadlock.

Solving stale data problem

Since stale data is one of the biggest issues in caching, various methods used for overcoming this to a certain extent will be discussed in the upcoming sections.

Refreshing

A trivial solution for stale data in cache is refresh the cache. Whenever fresh data is required, refresh should be performed to get the latest data from the data source. This is very commonly used in Internet browsers where the pages are cached locally and the user is aware that it might be the cached page and uses a refresh to make sure that the latest copy is loaded onto the browser.

JPA also provides support for refresh. Whenever a query hits the cache, it checks for the version in the cache and its corresponding version in the DB. If the DB version is newer, then the DB row values are obtained and the cache object updated. Otherwise, the cache value is returned since it is the latest version. This option provides optimal caching. In certain cases, it is always better to not have a 2nd level cache at all, since it becomes very complicated to deal with the stale data. Or else the applications should have a level of tolerance to the stale data.

Starting from JPA 2.0, a set of standard query hints are provided to allow refreshing or bypassing cache. **`CacheRetrieveMode`** and **`CacheStoreMode`** are the **`enum`** classes on which these query hints are defined.

1. **`javax.persistence.CacheRetrieveMode`**
    - **BYPASS**: Build the object directly from the database result and ignore cache.
    - **USE**: If the object/data is already in the cache, it will be used.
2. **`javax.persistence.CacheStoreMode`**
    - **BYPASS**: Do not cache the database results.
    - **REFRESH**: If the object/data is already in the cache, then refresh/replace it with the database results.
    - **USE**: Cache the objects/data returned from the query.

A sample usage of the query hint is as follows:

```
Query query = em.createQuery("Select stud from Student s");

query.setHint("javax.persistence.CacheStoreMode", "REFRESH");
```

***Code 6.13**: Query hint for Cache Store Mode example*

The query above on execution forces the cache to be refreshed since the query hint is set to **`REFRESH`** value.

JPA 2.0 also provides a Cache interface that can be used to manually evict or invalidate entities within the cache. This interface can be retrieved from the **`EntityManagerFactory`** using get cache operation. Some JPA providers extend this interface and provide additional functionalities as well.

```
Cache cache = entityManagerFactory.getCache();

cache.evict(Student.class,123);
```

***Code 6.14**: Cache Eviction example in JPA 2.0*

In the code above, the **`Student`** object with ID=123 will be evicted from the cache if it exists. If the evict method only takes in the class (here **`Student.class`**), then it removes all the entities of that class present in the cache.

Cache invalidation

Another natural way of dealing with stale data is to invalidate the cache. It can be either based on a time interval or at a particular instant of time. One way of invalidation can be by providing each entity, a **time to live** (**TTL**). If the entity crosses that time interval, then it is invalidated. This will always make sure that the data being read is not older than this amount of time.

Another way is to invalidate at a time of day. Normally, this would be during the low traffic hours and it makes sure that none of the cached data is older than a day's

data. This is most commonly used when any batch jobs are scheduled to be executed at a particular time and soon after that, the cache is invalidated.

Cache invalidation is also supported through API, and can be used in a cluster to invalidate objects changed on other machines.

Clustered caching

When caches are introduced into a clustered environment, it becomes more difficult to handle since each machine can update the database without updating the other machine caches. Therefore, it is very important to check whether caching is required in the cluster, and if so, how to configure it in the clustered environment.

Caching can mostly be used for read-only objects. Here, there should be some way to avoid stale data. While DB writes, optimistic locking can avoid writes on stale data. Certain JPA providers automatically refresh or invalidate the cache entry when there is an optimistic lock exception. This will allow to write if re-tried, since the first exception will invalidate the entity and the next try would fetch from the DB and do the necessary writes and store it back to the DB. Hence, in the second try it works as expected. Setting TTL is another way of reducing stale data.

There can be larger issues like returning the stale old data to the user who had just updated the same object via DML native queries. This can be solved by using a session that is used through the same machine in the cluster. Otherwise, this issue is bound to happen again. The application can also refresh the objects where up-to-date entities are more important, like use cache for read-only queries and refresh while a DML query is executed.

It is not of much benefit of allowing caching on objects which are write-mostly.

Hence, it is best to disable the cache for such objects. However, if the object is very complex and has lot of sophisticated relationships, and only a part of the object is being updated, then it will still be worth to have a cache fetching a complex object along with relationships. It can cause a huge performance hit if done always from DB.

Cache coordination

In a clustered environment, there should be some coordination between the caches in different machines in the cluster. One solution to cache coordination is to use a messaging framework in a clustered environment. JMS or JGroups along with JPA or application-based events, can be used to announce messages to invalidate the caches on other machines when an update occurs.

Distributed caching

If the cache is distributed over machines in a cluster, then it is a distributed cache.

Note that an object will be alive only on certain number of machines. This helps in reducing stale data as the object is always retrieved from the same location. This solution works well on all the machines in a cluster that are connected together on a high-speed network and the DB machine is under load or not connected well. The main difficulty in this setup is that the cache requires a network access. However, since this kind of cache reduces the DB access, the scaling of the application can be done without the DB becoming the bottleneck. There are even distributed cache providers who support local cache and then provide cache coordination mechanisms between the caches.

Cache transaction isolation

JPA provides different levels of transaction isolation in caches as well. They are broadly classified as–transactional and non-transactional.

In a transactional cache, the changes from a transaction are committed to the cache as a single atomic unit. This means the objects/data are first locked in the cache (preventing other threads/users from accessing the objects/data), then updated in the cache, then the locks are released. Ideally, the locks are obtained before committing the database transaction, to ensure consistency with the database.

In a non-transactional cache, the objects/data are updated one by one without any locking. This means there will be a brief period where the data in the cache is not consistent with the database. This may or may not be an issue, and is a complex issue to think about and discuss, and gets into the issue of locking, and the application's isolation requirements.

Optimistic locking is another important consideration in cache isolation. If optimistic locking is used, the cache should avoid replacing new data, with older data. This is important when reading, and when updating the cache.

Although the defaults should normally be used, it can be important to understand how the usage of caching is affecting your transaction isolation, as well as your performance and concurrency.

Conclusion

In this chapter, the main focus was on how the entities are stored and retrieved using the EM. This gives us a wider picture of why, when, where and how can a database row be accessed. In certain cases, it would be preferable to use the JPA object instead of the query. However, in some other cases, native query would be efficient.

The next chapter will introduce two most widely used JPA provider implementations and discuss about their various features.

Multiple choice questions

1. **Extended scope persistence context can extend over:**
   a) multiple queries
   b) multiple transactions
   c) multiple EMs
   d) all of the above

2. **Object Identity means if both the object instances point to the same location then both the instances are identical:**
   a) true
   b) false
   c) not applicable
   d) none of these

3. **The types of relationships that are normally cached:**
   a) OneToMany and ManyToMany
   b) ManyToOne and ManyToMany
   c) OneToOne, OneToMany and ManyToMany
   d) none of these

4. **Cache mode DISABLE_SELECTIVE means:**
   a) disable caching on all entities for which Cacheable is set as true.
   b) enable caching on all entities for which Cacheable is set as true.
   c) enable caching on all entities except those on which Cacheable is set as false.
   d) none of these

5. **getDelegate operation in JPA 1.0 has been changed to _____ in JPA 2.0.**
   a) getReference
   b) getWrapper
   c) getCache
   d) unwrap

Answers

1. **b**
2. **a**

3. **c**
4. **c**
5. **d**

Questions

1. Explain transaction management in various persistence context management.
2. Describe the entity life cycle with a diagram.
3. Discuss about various types of caches and their usages.
4. Write short notes on:
   - refresh
   - flush
   - clear
   - close
   - getCache

Points to ponder

1. TopLink/EclipseLink: Define a query hint "**`eclipselink.refresh`**" to allow refreshing to be enabled on a query.
2. Some JPA providers allow LAZY relationships to be accessed after EM close operation is performed.
3. TopLink / EclipseLink: Support object cache by default. Can be configured globally or selectively enabled per class. Property to enable/disable default object cache is "**`eclipselink.cache.shared.default`**".

   TopLink / EclipseLink: Read-only queries are supported through the "**`eclipselink.read-only`**" query hint, or by using the **`@ReadOnly`** annotation.
4. Hibernate: Caching is not enabled by default and supports third-party cache integration. Ehcache is one example which can be used along with hibernate.
5. TopLink/EclipseLink: Property to support query cache enabled through the query hint.
6. **`eclipselink.query-results-cache`**: Several configuration options including invalidation are supported.

7. TopLink/EclipseLink: Provides an extended Cache interface JpaCache which provides additional API for cache operations like invalidation, query, access and clear.

8. TopLink/EclipseLink: Provides support for time to live and time of day cache invalidation using **`@Cache`** annotation (**`<cache>`** element in XML configuration).

9. TopLink/EclipseLink: Support cache coordination using **Java Messaging Service** (**JMS**) and/or Remote Method Invocation **(RMI**).

10. TopLink: Supports integration with the Oracle Coherence distributed cache.

Section - IV

JPA Provider Implementations

Chapter 7
Hibernate and EclipseLink

Introduction

All the previous chapters were about the JPA features in general. However, it is also important to understand the most widely used JPA provider implementations and the features that make them more popular than the rest.

Structure

This chapter narrates the following topics:

- JPA Providers
  - Migration – JPA 2.X to 3.0
- Hibernate
  - History
  - Features
  - Architecture
    - Configuration
  - Querying in Hibernate
    - Hibernate query language

        - Hibernate criteria API
        - Native SQL
- EclipseLink
    - History
    - Features
    - Architecture
        - Configuration
    - Querying
        - Types of queries
    - Cache Architecture
        - Persistence unit cache
        - Persistence context cache
        - Cache isolation levels
- Weak reference
        - Types of caches
    - Polyglot persistence
        - SQL or Relational
        - NoSQL

Objectives

The motive of this chapter is to introduce two widely used JPA provider implementations with overview of their distinct features and various examples. This will help the reader to understand how and when to use each of these providers. As part of this chapter, the reader will be ready to start using these provider implementations and will also be familiar with how to migrate from JPA 2.X to JPA 3.0.

JPA providers

In this chapter, the two widely used JPA provider implementations are being discussed. Before the implementations are discussed, it is important to know that the latest version of JPA currently available in the market is JPA 3.0.

Owing to the technology transfer of the brand Java to Oracle in 2019, it was decided to move away from the name Java in open-source products and JEE was handed over to Eclipse foundation for further enhancements. Hence, Java Enterprise Edition

became Jakarta Enterprise Edition and so did the persistence APIs. Now onwards JPA is Jakarta Persistence API and the latest version will have no reference to the brand 'Java' or 'Javax' in it. This migration started from JPA 2.2 onwards and has been completed by JPA 3.0. Both the providers explained in this chapter support JPA 3.0 as well. Hibernate 5.5 and EclipseLink 3.0 are the versions that are fully compatible with JPA 3.0.

Migration from JPA 2.X to 3.0

Since there are no feature additions and only naming changes, the main change for migration would be to rename all the instances of Java to Jakarta. This would be applicable in:

- Package names
  - Import statements with prefix **`javax.persistence`** will be changed to have the prefix as **`jakarta.persistence.`**
- Configuration property names
  - Configuration properties with prefix **`javax.persistence`** will be changed to **`jakarta.persistence`**. These can be either in custom configuration files or in **`persistence.xml`**.
- XML namespaces – persistence.xml to be updated as follows:
  - **http://xmlns.jcp.org/xml/ns/persistence** changes to **http://jakarta.ee/xml/ns/persistence**.
  - **http://xmlns.jcp.org/xml/ns/persistence/orm** changes to **http://jakarta.ee/xml/ns/persistence/orm.**

Apart from the preceding changes, there should not be any other code change required as part of the migration activity.

Hibernate

Hibernate is an open-source persistence framework licensed under GNU-LGPL (GNU is Not Unix-Lesser General Public License). It is a lightweight, free to download **Object Relational Mapping** (**ORM**) tool based on JPA specifications. It simplifies the development of Java applications that interacts with databases.

Although created mainly for relational databases, Hibernate has been so popular that the non-relational version of this has also been developed for various kinds of databases like document stores (ex. MongoDB), key-value databases (ex. Infinispan), graph databases (ex. Neo4J) etc. It is called **Hibernate OGM** (Object Grid Mapper). This support for both relational and NoSQL DBs makes Hibernate widely accepted since the migration from one DB to another becomes very easy even if it is from a relational DB to a NoSQL or vice-versa.

History

Hibernate was initiated in 2001 by *Gavin King* while working for an Australian firm Cirrus Technologies, as a fix for the common problems faced while using Entity Beans (EJB2). Quoting Gavin King when asked for what led to build Hibernate:

"I was developing J2EE applications and was very frustrated by my lack of productivity, and by my inability to apply object modelling techniques to the business problem. I ended up spending more time thinking about persistence than I spent thinking about the user's problem. That's always wrong."

The initial version only had better persistence capabilities by simplifying the complexities and supplementing certain missing features.

In 2003, Hibernate 2.0 was released with significant improvements. Later, JBoss Inc (now part of RedHat) hired the lead developers of this team in order to keep up the momentum of development.

In 2005, Hibernate 3.0 was released with features like interceptor/call back architecture, JDK5.0 annotations, user defined filters, and so on. By 2010, this became a certified implementation of JPA 2.0 specifications.

In 2011, Hibernate Core 4.0 was released with multiple new features like multi-tenancy support, better session start, improved integration and auto discovery, internationalization support, and a clear distinction between the API, SPI and implementation classes. The current stable version of Hibernate is 5.6 with support for JPA 3.0.

Features

The most important features of Hibernate are as follows. Many of the features are interdependent and interrelated.

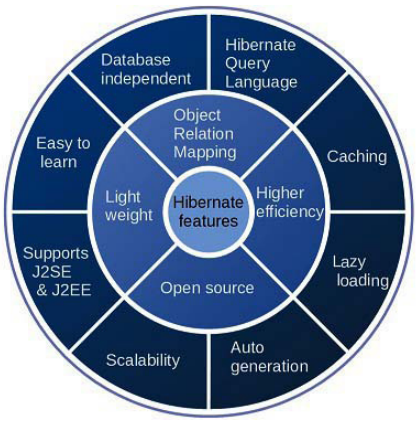

***Figure 7.1**: Hibernate features*

Object-relation mapping: This allows the interactions between the Java classes and the table rows seamlessly without the user to think more about the persistence layer. As per the creator's aim, Hibernate is here to solve the mismatch between the object-oriented and the relation-oriented concepts.

Database independent: By using database dialects, Hibernate achieves ability to switch between multiple databases without any code change. Only the database configuration changes and the rest of the code remains untouched. Moreover, Hibernate has also come up with OGM tools to switch between the NoSQL databases.

Hibernate Query Language (**HQL**): This can be considered as a query language for persistence objects very similar to SQL. This is not based on the underlying DB and hence, helps in achieving DB independence.

Light weight: No additional features are available other than the functionalities required for object to relation mapping. Also, any **Plain Old Java Object** (**POJO**) class can be converted to the persistence class by implementing Serializable (which is only for transfer of data). There is no requirement to extend or implement a particular class/interface in order to make the POJOs usable via Hibernate.

Easy to learn: In addition to the entry creations, Hibernate also automatically creates tables, if configured accordingly. This makes the user's task easier to create the entire set of tables required for an application. Moreover, since it has not much additional interfaces/classes that define the rules, it is easy to learn and implement.

J2SE/J2EE support: This makes it convenient to use Hibernate in standard java as well even without any enterprise features enabled. This is possible due to the lightweight nature of this framework.

Open source: Hibernate source code is available for free to download and use without any cost. This helps in growing the framework faster and in a more collaborative way with better outreach to every category of software industry.

Scalability: It is highly scalable and can be used in small-scale as well as large-scale applications.

Auto generation: Hibernate supports auto generation of the database schema. It also supports generation of schema via DB schema management tools like Liquibase, FlyWay, etc.

High efficiency: There are various fetching techniques used by Hibernate to improve performance like caching, lazy initialization, and so on.

Caching: Hibernate supports two levels of caching – First level and second level. This improves the speed of data access.

Lazy loading: This is to load only the necessary objects to perform the operation. This has been explained in detail in section Query optimization techniques in *Chapter 4, Relationships – Types and Strategies.*

Architecture

The Hibernate framework structure can be considered as the three main layers - Application layer, Hibernate core layer and Hibernate DB layer.

Application layer mainly consists of the entity objects that are to be used to transfer data to or from the DB.

Hibernate core layer is the heart of the framework that understands the configuration property values provided in the configuration file, loads the respective DB dialects, understands the mapping between the object and the table, manages the DB session and the query results appropriately. Any DB operation can be considered as one among the **Create, Read, Update, Delete** (**CRUD**) operations and can be converted to an appropriate query. Here, the core layer is responsible to first create a connection to the corresponding DB, create a query based on the operation mentioned and send the connection and query for execution to the next layer, i.e., the DB layer. It is also accountable to convert the query results from the DB layer back to the entity object notation to be consumed by the application layer.

The DB layer is in charge of performing the actual DB connectivity based on the configurations, execute the query passed on from the core layer and provide the results back to the core layer.

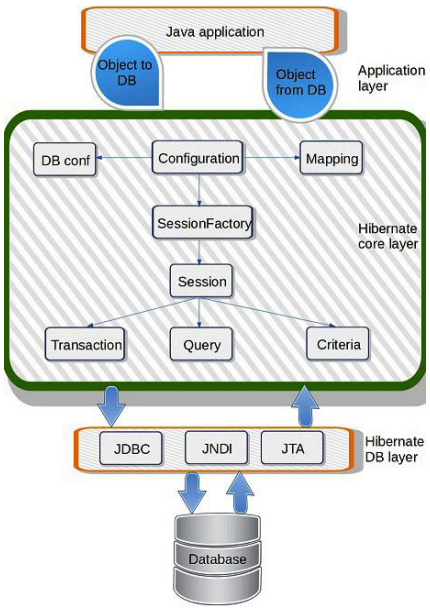

***Figure 7.2**: Hibernate architecture*

The preceding figure also depicts the core layer components along with their interactions and order of execution. Configuration class is used as an initialization time object and is only invoked when the application wants to invoke the Hibernate framework. This class reads both the mapping and configuration files. Based on the DB configurations, a single, thread-safe instance of **`SessionFactory`** is obtained from the **`Configuration`** object. This factory in turn, creates lightweight **`Session`** objects to perform the CRUD operations via the DB layer. Note that these session objects are not thread safe and hence need to be managed according to the requirement. Each session can have either a **`Transaction`** or a **`Query`** or a **`Criteria`** object that performs the actual DB operation via the DB layer.

Configuration

This is the most important part of Hibernate in which the developer needs to understand the relevance of each and every property. This would help in utilizing all the available features and fine tuning the performance of the framework. As part of the configuration, there are two types of files – Configuration and Mapping.

Configuration files can be either XML files or properties. JPA related configurations can be done in **`persistence.xml`** and Hibernate proprietary configurations can be done in **`hibernate.cfg.xml`** or **`hibernate.properties`**. Whenever, only JPA related configurations are being done, it is always better to use the **`persistence.xml`**.

```
<hibernate-configuration>
   <session-factory>
      <property name = "hibernate.dialect">
         org.hibernate.dialect.MySQLDialect
      </property>
      <property name = "hibernate.connection.driver_class">
         com.mysql.jdbc.Driver
      </property>
      <property name = "hibernate.connection.url">
         jdbc:mysql://localhost/test
      </property>
   </session-factory>
</hibernate-configuration>
```

***Code 7.1a**: Sample hibernate.cfg.xml*

```
hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/test
```

***Code7.1b:** Sample hibernate.properties file*

Tip: Whenever both properties file and hibernate configuration xml have the same properties configured, the xml file property values will be used by the application, i.e., the property file configurations can be overridden by the hibernate configuration xml.

The code for invoking different configuration files is also slightly different.

```
Configuration cfg = new Configuration().configure();
```

The preceding code will read the xml file and configure the properties accordingly. However, the following code can be used to read the properties or rather, even set the properties programmatically. Consider that the username and password for the DB is provided at run time, the following code can make use of the variable values.

```
Configuration cfg = new Configuration();
cfg.setProperty("hibernate.connection.username",userName);
cfg.setProperty("hibernate.connection.password",password);
```

Mapping files are those files in which each **`Entity`** object mapping is provided. Normally, the files are named **`<EntityName>.hbm.xml`**. However, now, most of the applications have started using JPA annotations instead of the mapping files.

```
<hibernate-mapping>
   <class name = "Student" table = "STUDENT">
      <meta attribute = "class-description">
         This class contains the student detail.
      </meta>
      <id name = "id" type = "int" column = "id">
         <generator class="native"/>
      </id>
      <property name = "firstName" column = "first_name" type = "string"/>
      <property name = "lastName" column = "last_name" type = "string"/>
      <property name = "std" column = "standard" type = "string"/>
   </class>
</hibernate-mapping>
```

***Code 7.2**: Hibernate mapping xml*

Here, the entity object is **`Student`** which is being mapped to the table **`STUDENT`**. The class variables **`firstName`**, **`lastName`** and **`std`** correspond to the table columns **`first_name`**, **`last_name`** and **`standard`** respectively. This information is being stored in the mapping file. However, with annotations, the mapping file (*Code 7.2*) can be removed and these details can be provided in the class file itself (*Code 7.3*).

```
@Entity
@Table(name="STUDENT")
public class Student {
@Id
@GeneratedValue(strategy=GenerationType.AUTO_INCREMENT)
@Column(name = "id")
private int id;

@Column(name = "first_name")
private String firstName;

@Column(name = "last_name")
private String lastName;

@Column(name = "standard")
private String std;
.....
}
```

***Code 7.3**: Hibernate mapping via annotations*

The advantage of using annotations for mapping is that it is easier to understand the relationship and need not have multiple files to actually figure out the object representation of the table row.

Querying in hibernate

There are two types of queries in Hibernate – complete entity read and partial entity read. While reading the complete entity, the entire object is populated and hence the result is given as list of the entity objects as follows (*Figure 7.3*).

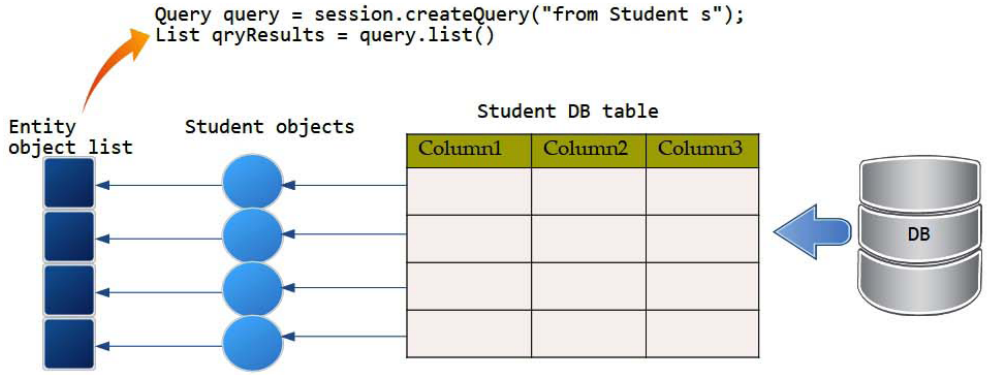

***Figure 7.3**: Reading from the DB and converting it to entity objects*

Here, the entire table (here **`Student`** table) is being read, each row converted to the corresponding entity object and list of these objects are created and returned.

Most of the times, all the properties (in case of tables: all columns) may not be required and hence, the query would be to fetch only a subset of the properties. In that case, only those columns (here **Name** and **Age**) are retrieved from the DB, converted to objects and placed in an array. Hence, the list returned in this case would be a list of object arrays as follows (*Figure 7.4*).

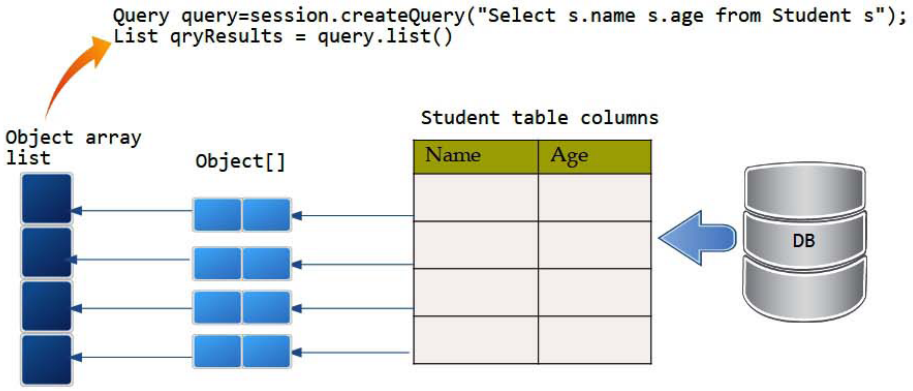

***Figure 7.4**: Reading from the DB and converting it to Object array*

DB query can be written in various ways in Hibernate. The commonly used techniques are **Hibernate Query Language** (**HQL**), Hibernate Criteria API and Native SQL.

HQL is normally used in most of the situations where the query is complex. They are defined as static strings and hence not type safe. i.e., the results have to be type casted accordingly and have higher chances of having run time exceptions.

On the other hand, criteria API can be used in situations where queries are dynamic and has to be created based on the run-time value of certain parameters. Since they are defined as Java objects and not by pure strings, the errors can be detected at compile time and are type-safe as well.

Native SQL queries are used to take the advantage of vendor specific optimized SQL features.

Hibernate Query Language (HQL)

HQL is a query language similar to SQL but instead of operating on table rows and columns, it works on objects and its attributes. At run time, these HQL queries generate the corresponding SQL queries. This helps in generating database independent query and hence, enables ease in porting from one DB to another.

In this language, the key words are case insensitive but the properties are case sensitive, i.e., words like 'select' can be written as SELECT or seLeCT but entity

name Student cannot be written as student or StuDent. If complete entities are to be read, then the query starts with **`FROM`** key word. If only few properties are required, then the query starts with **`SELECT`** key word. Sample HQL queries are used in *figure 7.3* and *figure 7.4*.

Similar to SQL queries, HQL also uses aggregate functions, joins, where clause, group by, order by and expressions while writing queries. It also allows to use positional parameters (as in JDBC) as well as named parameters.

```
Query query = session.createQuery("from Student s where s.deptNumber=?");
query.setParameter(0,300);
List qryResults = query.list();
```

Code 7.4a: *Sample java code for Positional parameter*

```
Query query = session.createQuery("from Student s where s.deptNumber=:num");
query.setParameter(num,300);
List qryResults = query.list();
```

Code 7.4b: *Java code for Named parameter*

Aggregate functions used are **`sum`**, **`min`**, **`max`**, **`avg`**, **`count`**, **`distinct`**, clauses like **`where`**, **`group by`**, **`order by`** and expressions are almost similar to SQL.

Explicit join is done using the keyword join in the query.

```
select add.city from Student as s inner join s.address as add
```

The query uses the inner join keyword for joining the **`Student`** object with the **`address`** object. Hence, it is an explicit join. The preceding query can be rewritten in HQL as follows:

```
select s.address.city from Student s
```

In the preceding HQL query statement, the **`Student`** table has to be joined with the **`Address`** table, and then have to select the city from the **`address`** table. Hence, the join happens inevitably in the underlying SQL but no explicit join key word is used in the HQL query. Hence, it is an implicit join.

Pagination support by HQL makes it a very suitable choice for web developers as it is very convenient to load a small chunk from a huge set of results and display it as pages as per the user request.

```
Query query = session.createQuery("from Student s");
query.setFirstResult(1);
Query.setMaxResults(5);
List qryResults = query.list();
```

Code 7.5: *Sample java code – Pagination support*

The preceding query internally fetches all the student records, assigns each row with a number starting from 0 onwards (i.e., the first row will have 0 as its row number and second row will have 1 as its row number and so on). Then, it sets the first result to row number 1 (which means the second row in the result set). The number of results returned will be limited by the count of 5. In short, the preceding query will return the second to sixth row of the actual result set.

Hibernate Criteria API

This technique uses a more programming-oriented way of creating and executing queries. Instead of passing a query string into the query object and executing it, a criteria query object is built through the code using various objects available as part of the **`Criteria`** API.

```
Criteria cr = session.createCriteria(Student.class);
List qryResults = cr.list();
```

Code 7.6*: List all Student entities using Hibernate criteria API*

The query in *figure 7.3* (in HQL) can be written as given above by using the **`Criteria`** API. However, the Hibernate Criteria API has been marked as deprecated from version 5.2 onwards and instead, it has started supporting JPA Criteria API. Hence, it is always better to use the JPA criteria API. So, the preceding query will be transformed as shown in the following code using JPA criteria API.

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Student> cr = cb.createQuery(Student.class);
Root<Student>root = cr.from(Student.class);
cr.select(root);
Query<Student> query = session.createQuery(cr);
List<Student> results = query.getResultList();
```

Code 7.7*: List all Student entities using JPA criteria API*

Hibernate criteria API is easier to use as it uses a straight forward method to create the different parts of the query and to execute them. On the other hand, JPA depends highly on the criteria builder interface to build the query. Hibernate allows to define

parts of the query as native SQL snippets. This makes it flexible to use DB specific features even if Hibernate doesn't provide direct support for them.

Native SQL

Native SQL are mainly used to make use of the DB specific features which enhance the overall performance in certain cases. However, the disadvantage is that it becomes dependent on the underlying DB, reducing the ease of portability. Hibernate 3.x allows to specify native SQL, including stored procedures for all operations.

```
SQLQuery query = session.createSQLQuery("select * from Student");
query.addEntity(Student.class);
List<Student> qryResults = query.list();
```

***Code 7.8**: Native SQL using Entity*

The preceding query is to select all the values in the Student table using native SQL. If the entity as a whole is retrieved, then it can use **`addEntity`** method to set the entity class. But in most of the cases, only a few scalar values are required and hence the entity class cannot be used. Instead, the result set has to be transformed using Hibernate's inbuilt result transformers or by custom result transformers.

```
SQLQuery query = session.createSQLQuery("select name, age from Student");
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List<Map> qryResults = query.list();
```

***Code 7.9:** Native SQL for scalar values using inbuilt result transformer*

Here, each row is considered as a **`Map`** where key is the column name and value is the actual value. So, in the preceding query (*Code 7.8*), each row can be represented as a **`Map`** as follows:

```
{"name": <name-value>, "age": <age-value>}
```

Result transformers were commonly used in Hibernate 4, but got deprecated in Hibernate 5. However, this feature is being re-introduced in Hibernate 6, but

in a slightly modified way such that it becomes more powerful than its previous incarnation in Hibernate 4.

Cache architecture

Hibernate provides multi-level cache mechanisms in order to reduce the database hits and hence, improve performance.

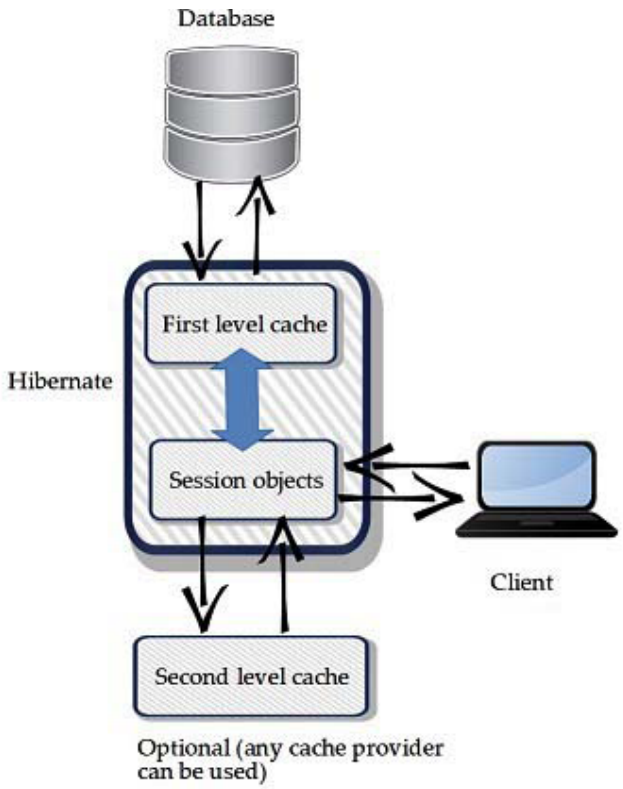

***Figure 7.5**: Hibernate cache architecture*

First-level cache

This is a mandatory cache and all requests are forced to pass through this cache. The session has its own copy of the object which is updated during the transaction and written back to the data source via the cache when the transaction is committed. Note that if the session is closed, all objects in the cache are lost and the changes are written (create/update) into the database.

This cache is used by default and no extra configuration is required for this.

Second-level cache

This is an optional cache and will be checked only after the first-level cache is consulted to locate an object. This can be configured on class level or collection level based on the caching to work across sessions. Third-party cache providers are mostly used with a handle provided in **`org.hibernate.cache.CacheProvider`** interface implementation.

For configuring second-level cache, two steps are to be done. First, decide on the concurrency strategy and then configure the cache provider along with the cache expiry and cache attributes that will be used by the cache provider.

Concurrency strategies: Transactional, read-write, non-strict read-write and read-only are the different concurrency strategies used in Hibernate. When it is critical to prevent stale data in the concurrent transactions and to be performed in a synchronous way, transactional strategy is applied. If the critical stale data prevention can be carried out in an asynchronous manner, then read-write strategy is used. When no guarantee of consistency is needed, then non-strict read-write can be used and finally read-only can be used in case where only the objects can be read.

Cache providers: EHCache, OSCache, SwarmCache, JBoss Cache

Third party cache-providers with description	Concurrency strategy			
	Read-only	Non-strict read-write	Read-write	Transactional
EHCache: Supports cache in memory or in disk and supports query result cache	✔	✔	✔	
OSCache: Supports cache in memory or in disk in a single JVM with many expiration policies as well as query result cache support	✔	✔	✔	
SwarmCache: Cluster cache that supports cluster invalidation but no support for query result cache	✔	✔		
JBoss Cache: Supports replication, invalidation, optimistic and pessimistic locking, as well as sync and async communication. Query result cache is also supported.	✔			✔

***Table 7.1**: Hibernate cache architecture*

The preceding table shows all the strategies that are supported by various third-party cache implementations.

Query-level cache

This is an optional cache for caching query results and is integrated closely with the second level cache.

EclipseLink

EclipseLink is an extensive open-source Java persistence solution which caters to different kinds of databases like relational, NoSQL, XML, JSON. It also allows a mix and match of all these databases, and hence supports polyglot persistence. This means that there would be multiple persistence units in a single application and an entity can be made up of multiple properties partially taken from one source of data and remaining properties from other sources of data. EclipseLink also provides a good support for creating database web services (REST as well as SOAP) that can be used to access relational database operations via web services.

Apart from JPA, EclipseLink also supports other standards like **Jakarta XML Binding** (**JAXB**), **Jakarta Connector Architecture** (**JCA**), **Service Data Objects** (**SDO**), and so on.

History

EclipseLink is based on the *Oracle TopLink* product. Originally, *TopLink* was developed in *SmallTalk* by a software company called *The Object* people based in Ottawa, Canada. It was ported to Java in 1998 and was called TopLink for Java. However, in 2002 TopLink was acquired by Oracle and developed as part of its Fusion Middleware product. It was part of the *OracleAS*, WebLogic and OC4J servers. In 2007, the TopLink code was donated to the Eclipse Foundation and thus EclipseLink was born. In 2008, *Sun Microsystems* selected EclipseLink as the reference implementation for JPA 2.0. EclipseLink is distributed under Eclipse Distribution License and this project is actively being supported by member companies like IBM and Oracle.

Features

Being the reference implementation of JPA, EclipseLink provides compliance to JPA 2.0. It supports almost all the features mentioned in JPA 2.0 and also additional proprietary features. Note that the portability reduces if proprietary features are used in the application development.

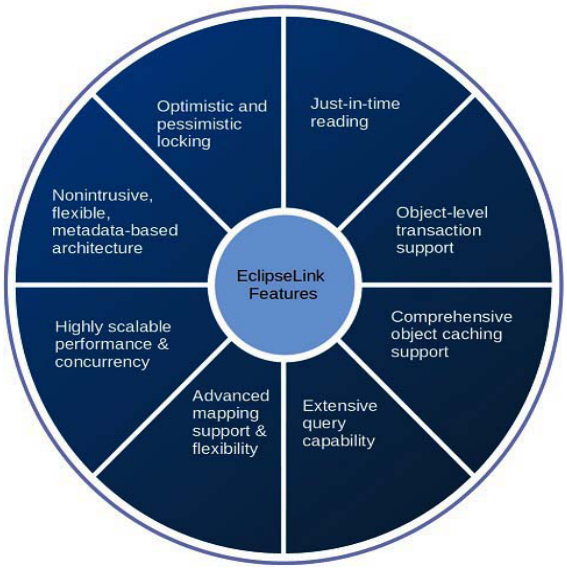

***Figure 7.6**: EclipseLink major features*

The main features of EclipseLink are:

- **Locking**-Optimistic as well as pessimistic locking: Refer to section *Locking* in *Chapter 3, Operations – Identity, Sequencing and Locking* for more details.
- **Just in time reading**: This is equivalent to the lazy loading concept in Hibernate.
- **Non-intrusive, flexible, metadata-based architecture**: It allows to use POJOs without any interfaces to extend makes it non-intrusive. Usage of JPA annotation/XML based mapping makes it metadata-based. The preceding two features make it very flexible enough that neither the database model nor the object model needs to consider any particular specifications.
- **Object level transaction support**: This helps in mapping the changes made at object level in different transactions and during commit merge the differences, if any, to the object in the cache. This reduces the stale objects in the cache and they remain in sync with their corresponding database entries.
- **Comprehensive object caching support**: The previous point mentions one among the various object caching techniques to reduce the cache refresh operations. It also allows to configure cache size and type on a per entity basis. Different cache types are WEAK, SOFT, SOFT_WEAK, HARD_WEAK, FULL.
- **Extensive query capability**: There are many types of queries in EclipseLink which will be detailed out in the upcoming sections.

- **Advanced mapping support and flexibility**: EclipseLink provides support for mapping java objects to relational/XML/JSON/SDO. This wide variety support for mapping provides the flexibility of using the same layer to retrieve data from a wide variety of data sources.
- **Highly scalable performance and concurrency**: EclipseLink provides the caching mechanism to support even server clusters. There are a few configurations to be done to support this kind of caching mechanism.

Apart from the preceding features, EclipseLink also supports proprietary features such as:

- Handling of database change events
- Composite persistence units to map entities to tables in multiple databases
- Support for multi-tenancy

Architecture

EclipseLink can be considered as an extensible framework for Java persistence as it provides support for JPA, JAXB, JSON binding, SDO and several customizations for different other mechanisms. The main components in this framework are:

EclipseLink– Object-Relational Mapping (ORM): This supports JPA that can be configured via XML or via custom annotations. It also provides features like robust caching including clustered support, usage of advanced database specific capabilities, extensive performance tuning and management options.

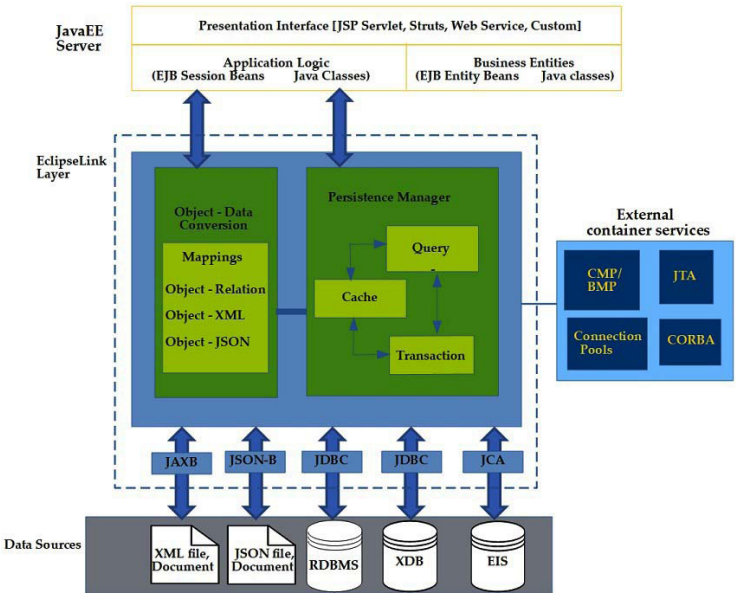

***Figure 7.7**: Eclipse Link architecture*

- **EclipseLink– Object-XML Mapping (OXM)**: This supports serialization services through **Java API for XML Binding** (**JAXB**) with extensions to provide mappings and critical performance optimizations.
- **EclipseLink– JavaScript Object Notation (JSON)**: This provides a reference implementation of Java API for **Java Specification Request** (**JSR**) 367 which denotes the JSON Binding specifications. This is a JSON equivalent of the OXM module and provides all its features for JSON mappings.
- **EclipseLink– Service Data Object (SDO)**: This provides JAVA API implementation to represent any Java object as an SDO and use all the capabilities available by any SDO implementation.
- **EclipseLink– Database Web Services/JPA-Rest Services (DBWS/JPA-RS)**: This allows to expose the underlying RDBMS as webservices by providing easy and efficient web service development capabilities to the developers.
- EclipseLink– **Enterprise Information Systems** (**EIS**): This allows mapping of Java POJOs to non-relational data stores using **Java Connecter Architecture** (**JCA**) API.

Configuration

Similar to Hibernate, EclipseLink also provides a wide range of functional and performance parameters that can be configured in the **`persistence.xml`** file.

```
<persistence-unit name=”default” transaction-type=”RESOURCE_LOCAL”>
   <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
   <!-- Default mapping file is orm.xml-->
   <mapping-file>META-INF/mappings.xml</mapping-file>
   <jar-file>my-application.jar</jar-file>
  <!-- Enables auto discovery of persistent classes -->
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
        <property name="eclipselink.jdbc.url" value="jdbc:oracle:thin:@
localhost:1521:ORCL"/>
      <property name="eclipselink.jdbc.user" value="root"/>
      <property name="eclipselink.jdbc.password" value="root"/>
      <property name="eclipselink.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
      <property name="eclipselink.target-database" value="Oracle"/>
     <property name="eclipse.logging.level" value="FINE"/>
     <property name="eclipse.logging.thread" value="false"/>
```

```
      <property name="eclipse.logging.session" value="false"/>
  </properties>
</persistence-unit>

eclipselink.jdbc.url="jdbc:oracle:thin:@localhost:1521:ORCL”
eclipselink.jdbc.user="root”
eclipselink.jdbc.password ="root”
eclipselink.jdbc.driver ="oracle.jdbc.OracleDriver”
eclipselink.target-database="Oracle”
eclipse.logging.level="FINE”
eclipse.logging.thread="false”
eclipse.logging.session="false"
```

***Code 7.10**: Eclipse Link persistence.xml configurations(top) and configuration via properties file (below)*

Apart from the **persistence.xml** file, there is a mapping file which provides the entity mapping details. The mapping file name is provided in the **persistence.xml** using the tag **<mapping-file>**. By default, **orm.xml** will be checked for the mapping details. However, if such configuration is given, then it would be considered. The mapping classes can also be provided via jar files using the tag **<jar-file>**. Note that it can be given as addition to the mapping file as well. In certain cases, if there are some files in the jar file that are annotated as **@Entity** but need not be considered in this deployment, those classes can be mentioned in tag **<exclude-unlisted-classes>**. Normally, in **Java Enterprise Edition** (**JEE**) environment, the application server itself will discover the local classes. However, in **Java Standard Edition** (**JSE**) environment, the EclipseLink provider has to do this functionality and hence, this tag is set to false to enable the discovery process.

The following figure (*Code 7.11*) depicts an example mapping file which maps the class to a table in the DB.

```
<entity-mappings>
   <description> XML Mapping file</description>
   <entity class="Student">
      <table name="STUDENT"/>
      <attributes>
         <id name="ID"><generated-value strategy="TABLE"/></id>
         <basic name="name">
            <column name="STUDENT_NAME" length="100"/>
         </basic>
```

```
        <basic name="standard"> </basic>
        <basic name="age"></basic>
      </attributes>
   </entity>
</entity-mappings>
```

Code 7.11: *Eclipse Link orm.xml configurations*

From the preceding example, it is clear that EclipseLink uses the same format as that of the standard JPA. It also supports annotations-based mapping and hence this mapping file can be avoided if the entity beans are annotated accordingly.

Querying

In general, querying function can be considered as performing the following functionalities:

- Specify the action to be done such that it is understood by the source which is being queried for.
- Execute this action in a controlled way.
- If there are results for the execution above, view / manage the results obtained.

Note that when there is a cache involved, the effect of the querying on the cache should also be considered to make the querying functionality complete. In this section, the various kinds of queries and caches provided in EclipseLink will be discussed.

Types of queries

There are various types of queries in Eclipse Link that are discussed in this section.

1. **Session queries**

   These are queries that are constructed and executed using a **`Session`** object. This consists of the most common database operations based on the type of session chosen for the query. The various session types are – **`UnitOfWork`**, **`Server`**, **`ClientSession`** and **`DatabaseSession`**. Server session is mainly to obtain the client / database session available with the particular server. So, it is not used in normal DB operations. However, the rest of the sessions can be used for DB operations based on the requirements. It is always recommended to use **`UnitOfWork`** as it provides the most efficient way to manage transactions, concurrency and referential constraints.

2. **Database queries**

   These queries are created using the **`DatabaseQuery`** object. The **`executeQuery`** method of **`Session`** object consumes different types of database query objects as input and executes the query on either the object or the data. The different types of database query objects are–Object-level read query, Data-level read query, Object-level modify query, Data-level modify query and Report query. As the name specifies, object-level queries are used for object-level access or modification. This can be reading a particular object, or all the objects or a set of objects. Similarly, data-level queries provide access to collection of results which can be either at the object level (row wise entire columns) or column level (single/subset of columns) or cell level.

3. **Named queries**

   Named queries are those that are used again and again as they are prepared once and reused many times. This is very useful especially while using complex queries that are very frequently used in a session. If the query is global to a project, then it should be configured at the session level. In session level, the query can be added to the **`Session`** object and can later be executed by providing its name. If the query is only used within a class, then it should be defined at the descriptor level. In this case, **`DescriptorQueryManager`** object provides API to store and retrieve named queries.

4. **Call queries**

   EclipseLink provides a variety of **`Call`** objects that wraps an operation or action on a data source. The different types are based on **Structured Query Language** (**SQL**), **Java Persistence Query Language** (**JPQL**) and **Extensible Markup Language** (**XML**). Based on the **`Call`** objects, queries can be created either directly via **`Session`** objects or indirectly using the **`DatabaseQuery`** context.

   In direct call queries, it can either access the fields of a relational database via SQL calls or interact with an **Enterprise Information System** (**EIS**) through **Java Enterprise Edition Connector Architecture** (**JCA**).

   SQL call queries are further classified as **`SQLCall`**, **`StoredProcedureCall`** and **`StoredFunctionCall`**. Using the preceding **`Call`** API, the input, output, and input-output parameters can be specified and their values can be assigned.

   EIS interactions can be further divided as **`IndexedInteraction`**, **`MappedInteraction`**, **`XMLInteraction`**, **`XQueryInteraction`**, and **`QueryStringInteraction`**. As the name denotes, indexed interactions can be used for JCA interactions using indexed records, whereas mapped interactions can be used in case of mapped records. Similarly, JCA interactions

based on XML, XQuery and Query string can use XML interactions, XQuery interactions and Query string interactions respectively.

5. **Redirect queries**

   Redirect queries help in achieving complex operations that otherwise would not be supported by the querying framework. The actual query implementation is a static method and when the query is invoked, the control is passed on to this static method. Redirect query helps in:

   - dynamically configuring query options based on the arguments.
   - dynamically defining selection criteria based on the arguments.
   - passing objects or expressions as arguments.
   - post-processing the query results.
   - performing multiple queries or special operations.

6. **Historical queries**

   A session only provides the latest version of objects. However, there can be cases where the query should be made time aware and should be able to select the version from the past. Historical queries help in doing so. While defining EclipseLink history policy, sessions can be configured to maintain historical versions of objects and this can be made use of in these queries. **`Session`** method **`acquireHistoricalSession`** can be made use with an **`AsOfClause`** that specifies the point of time.

```
AsOfClause astime = new AsOfClause (
                    System.currentTimeMillis() – 10*24*60*60*1000)
Session historicalSession = session.acquireSessionAsOf(astime);
Student pastStudent = (Student) historicalSession.readObject(Student.
class);
Address pastAddress = pastStudent.getAddress();
List pastCourses = pastStudent.getCourses();
historicalSession.release();
```

***Code 7.12**: Eclipse Link historical query example*

The preceding code is to get the details of the students who would have been in the system 10 days ago. Then, from the past student object, their addresses and the list of courses are retrieved.

1. **Interface and inheritance queries**

   EclipseLink supports creating descriptors on interfaces as well as its inheritance relationships in order to help querying. Based on the implementation of the interfaces, the following mechanisms are used.

   - When an interface has a single implementor, EclipseLink interface query returns the concrete class instance.
   - When there are multiple implementors, the query returns instances of all the implementing classes.

In case of inheritance hierarchies, configurations of the descriptors are made use of to define the way it should work.

- The default configuration is to read the subclasses and, in this case, the query returns the instance of the class and its subclasses.
- If the configuration is not to read the subclasses, then the query returns only the instance of the queried class.
- If the configuration is to include outer-join subclasses, then the query returns the instance of the class and its subclasses.
- If none of the preceding conditions are applicable, then the class should be a leaf class and hence only the instance of the class is returned by the query.

2. **Descriptor query manager queries**

   In EclipseLink, descriptors are used to store information that describes how an instance of a particular class can be represented by a data source. Descriptors have association mappings that relate class instance variables with data source and transformation routines that are used to store and retrieve values. Hence, there exists an instance of query manager for each descriptor and it can be found as `DescriptorQueryManager` object. This can be used to configure named queries, make default query implementations, as well as build additional join expressions.

Cache architecture

EclipseLink, like Hibernate, has two types of cache:

- **Shared persistence unit cache (L2 cache)**: maintains objects obtained from and written to the data source.
- **Isolated persistence context cache (L1 cache)**: maintains objects that participate in transactions.

Internally, EclipseLink maintains the persistence unit cache on the EclipseLink session and the persistence context cache on the EclipseLink unit of work. These caches work together with the data source to manage the objects in an application using EclipseLink as described in the following figure:

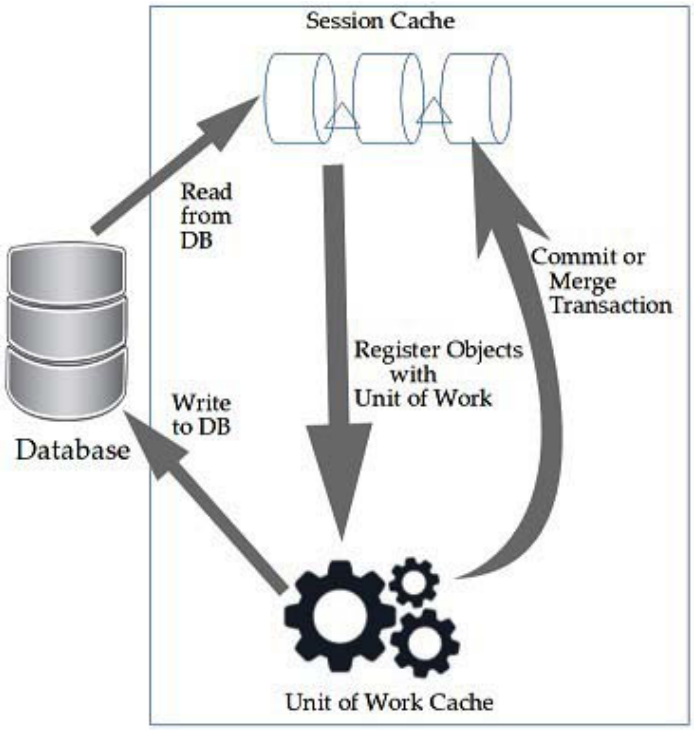

***Figure 7.8**: EclipseLink Session with cache*

From the preceding figure, it is clear that the session cache acts as the L2 cache, whereas the **Unit of Work Cache** acts as the L1 cache. The L1 cache, ideally is created for each transaction and is written back to the DB whenever a transaction is committed. However, in this case, the objects in L1 are also synced with their copies in L2, if any, in order to maintain sync and reduce the database access by the L2 cache. Note that the difference between Hibernate and EclipseLink is that Hibernate only implements L1 cache and it supports third party caching mechanisms for L2 cache. But in case of EclipseLink, it implements both L1 and L2 cache.

Persistence unit cache

This is a shared cache that helps in obtaining objects for all clients attached to a particular persistence unit. Whenever an object is read from the data source, EclipseLink creates a copy of the object in this cache and makes it available for all the other processes accessing the same persistence unit. This cache gets updated in the following scenarios:

- when a read object operation is performed on the data source.
- when a transaction is successfully committed by the persistence context cache.

Every unique persistence unit name will have a separate persistence unit cache. This cache is conceptually stored in the **`EntityManagerFactory`** and hence every persistence unit will have a single **`EntityManagerFactory`** and its cache. If an application has two factories with the same persistence unit name, then the cache will be shared (essentially the persistence unit is also the same instance). Properties of persistence unit are used to create the persistence unit and hence can affect the uniqueness quality. So, there can be instances where the application actually expects a shared cache but gets a separate cache. In order to configure this behavior, **`eclipselink.session.name`** property can be used to force two persistence units to resolve to the same instance and share the cache.

Persistence context cache

This is an isolated cache that helps in maintaining objects for the operations within the EntityManager. It insulates the object from the persistence unit cache, maintains it throughout the transaction, and writes it back to the persistence unit cache once the transaction is successfully committed to the data source. Note that only committed transaction changes are merged to the shared persistence unit cache. Flush or other such operations do not merge the changes to the shared cache. The life cycle of context cache depends on the type of persistence context being used in the application and differs accordingly.

Application managed persistence context is based on the application and is created from the **`EntityManagerFactory`**. This cache will remain till the EntityManager is either cleared or closed and hence this should have short-lived context caches or proper mechanism to clear the cache in a regular manner so as to avoid the cache from growing too big or out of sync with the persistence unit cache and/or the data source. Ideally, the best way to deal with this is to create an EntityManager for each transaction or request.

In container-managed persistence context, the container injects the context into a managed object (say **`SessionBean`** or so). Here, the context cache is maintained only for the duration of a transaction and after the transaction the objects are detached.

Cache isolation levels

Cache isolation levels are used to define how each entity will be cached in the persistence unit as well as persistence context caches. The three isolation levels are:

- **Isolated**: These entities are only stored in the persistence unit cache. They are not cached in the persistence context.

- **Shared**: These entities are stored in both persistence unit cache as well as persistence context cache. However, the read-only entities are shared and cached only in the persistence unit.
- **Protected**: Same like in shared, the entities can have relationships with both shared as well as isolated entities. So, the relationship with shared are stored in the persistence unit as well as copied to the isolated cache and stored along with the isolated relations in persistence context cache.

Weak reference mode: Normally, the persistence context is short-lived and it gets created per new request/transaction. This ensures that the persistence context does not cause any memory and performance issues. This also makes sure that the objects in the cache are not stale or out of sync with the data source.

However, there are cases where the application requires the persistence-context to be long-lived. In such cases, the EntityManager may not be created as per new request/transaction. EclipseLink offers a weak reference mode which makes a weak reference to the objects in the persistence context and allows the garbage collector to dispose the objects if they are not referenced by the application. Thus, it ensures that the context does not cause any memory or performance issues. The following options can be used:

- **HARD:** Default value where the persistence context grows until cleared or closed.
- **WEAK**: Unchanged and non-referenced objects will be marked for garbage collection. If the objects are tracked for changes, then they will not be eligible for garbage collection.
- **FORCE_WEAK**: Same as WEAK above. Here, even the tracked objects with changes but non-referenced, will be eligible for garbage collection. Hence there are chances for changes to get lost.

A weak reference mode configuration is done by the persistence unit property `eclipselink.persistence-context.reference-mode`.

Types of cache

In EclipseLink, cache type and size are configurable based on the application's memory requirements. This basically depends on the possibility of stale data, the JVM memory available, the machine memory availability, cost of garbage collection and the amount of data in the DB.

Types of cache		Caching	Guaranteed identity	Memory use
FULL	Objects are only removed by explicit delete.	Yes	Yes	Very High
WEAK	Objects are garbage collected whenever they are not referenced by the application. Referenced objects only cached.	Yes	Yes	Low
SOFT	Same like weak, but the object is removed only when the memory is low. Contains both referenced/non-referenced based on the memory availability.	Yes	Yes	High
SOFT_WEAK / HARD_WEAK	Same like weak, but maintains a sub cache of most frequently used with soft/hard references. Frequently used objects are maintained and the others are garbage collected when they are no longer referenced in the application.	Yes	Yes	Medium-high
NONE	No objects are cached	No	No	NA
CACHE	Fixed number of objects in the least recently used method.	Yes	No	Fixed

***Table 7.2**: EclipseLink types of cache*

By default, EclipseLink uses a **`SOFT_WEAK`** cache of size 100 objects. The size and type of cache can be configured by **`@Cache`** annotation's size and type attributes respectively. In addition to that, the query result caching type can also be configured by the property **`eclipselink.query-results-cache.type`**

Polyglot persistence

Using this feature of EclipseLink, a single application can access multiple types of databases without any data access object layers. The same JPA layer can be made use of accessing these data sources and provide the appropriate information to the application. This is a very important feature especially in today's scenario as most of the applications have more varieties of data sources like relational, NoSQL, XML/JSON etc.

For entities reading from the NoSQL databases, **`@NoSql`** annotation should be used. The following example is based on a **`Student`** record available in MongoDB. This

stores data as JSON (key-value pair). Here, the key name is **firstName**, **lastName** and **std** and their corresponding values will be read into those variables in the entity object.

```
@Entity
@NoSql(dataFormat=DataFormatType.MAPPED)
public class Student implements Serializable{
   @Id
   @GeneratedValue
   @Field(name = "_id")
   private String id;

   @Basic
   private String firstName;

   @Basic
   private String lastName;

   @Basic
   private String std;
   ……
 }
```

***Code 7.13**: EclipseLink usage for NoSQL student entity*

This object when compared with the relational DB entity looks similar, and hence the JPA layer can totally obfuscate the origin of the information to the application.

Conclusion

In this chapter, the main focus was on the two widely used JPA provider implementations and their various features. However, there has been studies to compare and contrast these implementations. In general, while considering various provider implementations, following few points can prove helpful:

- **Project maturity**: How long this product has been in the market and how well is the documentation?
- **Examples as sub projects**: Are there any sub-projects provided in the project that can be used as the starting point for the new application to be developed?

- **Community support**: Is there any kind of support from the JPA provider developer community such that the developer can get help while facing a critical bug at any phase of the application development?
- **Benchmarking**: Has this implementation been measured against other implementations? JPA performance benchmarking is done and available for reference at their site[5].
- **Application requirements**: Does the implementation have any features (JPA or proprietary) that are well suited for the new application?

In short, the points above will help in determining which provider implementation has to be chosen for developing an application using JPA.

Multiple choice questions

1. **Hibernate was developed initially:**
   a) as an implementation for JPA 1.0
   b) to solve the generic issues while using entity beans
   c) as a competitor for Oracle's TopLink
   d) all of the above

2. **EclipseLink is maintained by Oracle and provided under GPL.**
   a) false
   b) true
   c) partially true
   d) none of these

3. **Default cache type and size of Eclipse link is:**
   a) weak, 100 objects
   b) full, 1000 objects
   c) soft weak, 100 objects
   d) hard weak, 100 objects

4. **Caching in Hibernate is based on:**
   a) concurrency strategy and query strategy.
   b) query strategy and cache provider
   c) concurrency strategy and cache provider
   d) none of these

5. **Weak reference in EclipseLink is based on the following property:**
   a) eclipselink.persistence-context.ref-mode
   b) eclipselink.persistence-context.ref.mode
   c) eclipselink.persistence-context.reference.mode
   d) eclipselink.persistence-context.reference-mode

Answers

1. **b**
2. **a**
3. **c**
4. **c**
5. **d**

Questions

1. Explain the caching architecture in Hibernate and the role of third-party cache providers.
2. Describe the Eclipselink architecture in your own words.
3. Which among the above JPA provider implementation do you like? Explain with examples to prove your liking.
4. Write short notes on:
   a) Hibernate query techniques
   b) EclipseLink cache architecture
   c) Types of cache in any one of the provider implementations
   d) Polyglot persistence

Points to ponder

1. TopLink/EclipseLink: defines a query hint "**`eclipselink.refresh`**" to allow refreshing to be enabled on a query.
2. TopLink/EclipseLink: provides support for time to live and time of day cache invalidation using **`@Cache`** annotation (**`<cache>`** element in XML configuration).
3. EclipseLink: has many database-specific features for Oracle DB. They are hints, hierarchical queries, flashback queries and stored functions.

4. Hints: These are additional specifications that can influence the way the DB server SQL optimizer works. In EclipseLink, **`setHintString`** method of the **`DatabaseQuery`** object allows to set hints for an SQL query.

5. Hierarchical queries: Oracle DB allows to read the rows of the database in a hierarchical way. For example, in a family database, the data can be read as parents, then their children and then the grandchildren and so on. This can be done by using **`setHierarchicalQueryClause`** method of **`ReadAllQuery`** object.

6. Flashback queries: From Oracle9i onwards, EclipseLink helps in extracting a historical session from the past and this can be made use of understanding how the objects are changing over time.

7. Stored functions: This is an Oracle DB capability that enhances a stored procedure capability by executing all the steps and in addition to that return a value. EclipseLink supports this with a **`StoredFunctionCall`** object.

References

1. Articles on various Hibernate/JPA features: **https://thorben-janssen.com/tutorials/**

2. Tips on Hibernate usage: **https://thorben-janssen.com/tips/**

3. YouTube channel on Hibernate videos: **https://www.youtube.com/channel/UCYeDPubBiFCZXIOgGYoyADw**

4. Performance Comparison of Hibernate and EclipseLink Technologies for mapping an object-oriented model to RDBMS - **https://epublications.regis.edu/cgi/viewcontent.cgi?article=1904&context=theses**

5. JPA Performance Benchmark – Hibernate & MySQL vs EclipseLink & MySQL - **https://www.jpab.org/Hibernate/MySQL/server/EclipseLink/MySQL/server.html**

6. Understanding EclipseLink2.4 - **https://www.eclipse.org/eclipselink/documentation/3.0/concepts/toc.htm**

Section - V
Appendix

Part 1
JPA Advanced Topics

Introduction

This section is to introduce the reader to the advanced topics in JPA. This is not a mandatory section and is always good to know more about these advanced topics.

Structure

- Events
- Views
- Stored procedures
- Structured object data types
- XML data types
- History
- Filters
- Logical deletes
- Auditing
- Replication

- Partitioning
- NoSQL
- Multi-tenancy

Events

Any system can be considered as an ordered set of processes. An event, in the JPA context, can be considered as an activity to be performed in connection to any of these processes. For example, if there is an action A to be performed, the event can be a small activity that has to be either performed before A or after A. This helps in extending, debugging, monitoring, and customizing the ordered set of actions in any system.

JPA events are defined either through annotations or by XML configurations. The persistent class can have methods that can be annotated with an event annotation as per the requirement and this will be executed accordingly for all instances of this class. The actions on the entity are taken into consideration and JPA events are defined accordingly.

- **`PostLoad`**: occurs soon after the entity is loaded (or refreshed) to the persistence context.
- **`PrePersist`**: occurs just before the persist operation is invoked on an entity. This can occur even in cases of merging on new instances (which is similar to persist) as well as on cascade persist.
- **`PostPersist`**: occurs just after the database **`INSERT`** is completed but before the transaction is committed. This normally happens during the commit/flush operation. Note that if the object generates primary keys, then it will be available in this method only. It will not be available in the **`PrePersist`** event.
- **`PreUpdate`**: occurs only if there is a change in the data and the EntityManager has identified it as a modified record. This takes place during a flush or commit operation before the database **`UPDATE`** has occurred.
- **`PostUpdate`**: occurs when an instance is updated in the database during a flush or commit operation after the database **`UPDATE`** has occurred, but before the transaction is committed. Note that **`PreUpdate`** and **`PostUpdate`** events are not executed during the merge operation.
- **`PreRemove`**: occurs before the remove operation is performed on an entity. This gets invoked for cascade remove as well as orphan removal also.

- **PostRemove**: occurs during a flush or commit operation after the database **DELETE** has occurred, but before the transaction is committed. Note that this event is not executed during the remove operation.

An important point to take into consideration is that there cannot be two listeners for the same event in the same entity or same-entity listener.

```
@Entity
@Table(name="STUDENT")
public class Student {

  @PrePersist
  public void onPrePersistEvent() {
   log.info ("Student record getting
persisted");}

  @PreUpdate
  public void onPreUpdateEvent() {
   log.info ("Student record getting
updated");}
  ..... }
```

```
<entity name="Student"
     class="com.example.Student"
     access="FIELD">
  <pre-persist
  method-name="onPrePersistEvent"
/>
  <pre-update
   method-name="onPreUpdateEvent"
/>
</entity>
```

***Code A1.1**: Event definition within Entity using annotation (left) and using XML (right)*

Entity listeners

Events can also be configured externally using the @**EntityListeners** annotation or the **<entity-listeners>** XML element. This is a class-level annotation and is external to the entity definition. Note that even in the listener class no interface implementation is mandated, only the method annotations are required.

```
@Entity
@EntityListeners(value=StudentListener.class)
@Table(name="STUDENT")
public class Student {
.....
}
```

```
public class StudentListener {
  @PrePersist
  public void onPrePersistEvent() {
      log.info("Student record getting persisted");
  }
  @PreUpdate
  public void onPreUpdateEvent() {
      log.info("Student record getting updated");
  }
  .....
}
```

```
<entity name="Student" class="com.example.Student" access="FIELD">
<entity-listeners>
<entity-listener class="com.example.StudentListener">
        <pre-persist method-name="onPrePersistEvent" />
        <pre-update method-name="onPreUpdateEvent" />
</entity-listener>
</entity-listeners>
</entity>
```

***Code A1.2**: Event definition with Entity Listeners using annotation (left) and using XML (right)*

Default entity listeners

These are entity listeners configured at the persistence unit level which considers all the entity classes. This can be done only via XML. If a particular entity wants to disable this default listener, it can use **`@ExcludeDefaultListener`** annotation or **`<exclude-default-listeners>`** XML element.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.0" xmlns=http://java.sun.com/xml/ns/persistence/orm xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm/orm_2_0.xsd">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="org.acme.ACMEEventListener">
```

```
            <pre-persist method-name="prePersist"/>
            <pre-update method-name="preUpdate"/>
          </entity-listener>
        </entity-listeners>
      </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>
```

***Code A1.3**: Default Entity Listeners defined at the persistence unit level*

Event inheritance

As in the case of any class attributes or methods, entity listeners are also inherited from parent class to subclass. If the subclass does not prefer to have this behavior, then it should use **`@ExcludeSuperclassListeners`** annotation (**`<exclude-superclass-listeners>`** XML element).

It is now clear that the events can be defined at method level internal to the entity, at class level (using entity listeners) external to the entity as well as inherited from parent classes. So, there should be an order in which these events are to be executed when they are defined in multiple hierarchies. The order is decided based on the following rules:

- External listeners are always executed first and only then the entity level call backs are executed.
- Default listeners are executed first, then the listeners from the super classes based on the inheritance hierarchy. If there are multiple listeners in the same level of the hierarchy, then they are executed based on the order of definition.
- After all the entity listeners from the persistence unit as well as the super classes, entity listener defined for the actual entity class gets invoked.
- After the execution of all the external listeners, the internal call backs are executed starting from the topmost entity class in the inheritance hierarchy.
- Finally, the call back methods in the actual entity class are executed.

Views

A database view is the result of a query and can be searched similar to a table. Hence, views are considered as a virtual table that hides the complexity of a table and shows only the subset of data as per the requirement. Since a view can be considered as a table itself, in JPA, it can be mapped using **`@Table`** annotation. The columns in the view can be related to the object attributes.

Mostly, views are read-only and are only updated when the underlying table data gets updated, i.e., database triggers are executed to update the view data. However, modern RDBMS support insertable and updatable views, and hence it would not be an issue to use JPA views to insert/update a record in the view.

Stored procedure

A stored procedure is a set of instructions that resides on the database and are typically written in DB-specific language like SQL. For example, PL/SQL is used to write stored procedures in Oracle.

In JPA, any SQL can be executed using native queries. The SQL can either return nothing or return a database result set. However, the stored procedure execution depends on the DB. Some databases such as DB2, Sybase, and SQL Server returns result-sets for stored procedures, but Oracle only returns the cursor type as an output value.

```
EntityManager em = getEntityManager();
Query query = em.createNativeQuery("BEGIN VALIDATE_EMP(P_EMP_ID=>?);
END;");
query.setParameter(1, empId);
query.executeUpdate();
```

***Code A1.4**: Sample Oracle stored procedure in JPA*

JPA does not support stored procedures that use OUTPUT or INPUT parameters. However, some JPA providers have extended support for stored procedures, like support for cursor output parameters.

Structured object-relational data types

There was a trend of adding object-oriented concepts to relational databases and name these hybrid databases as object-relational databases. This went on further adding support in SQL as well as JDBC API. However, this concept did not get wide acceptance and standard relational data is still preferred over object-relational data due to its complexities. Some common object-relational database features include:

- Object types (structures)
- Arrays and array types
- Nested tables
- Inheritance
- Object ids (OIDs)
- Refs

A few databases that support the above data types are Oracle, DB2, PostgreSQL, and so on.

Ideally, JPA does not support object-relational data types, but some JPA providers may offer limited support. EclipseLink supports object-relational data-types through **`@Struct`**, **`@Structure`**, **`@StructConverter`**, **`@Array`** annotations or by using **`ObjectRelationalDataTypeDescriptor`** class and its mapping classes.

XML data types

As XML databases were introduced, many relational databases enhanced its XML support by making it aware of the XML data. The main advantage of this is that it allows querying using XPath or XQuery syntax. Some databases with XML support include Oracle (XDB), DB2, PostgreSQL, etc.

JPA does not mandate any support for XML data, although it is possible to store an XML into the database, as a string (mapped as Basic). Some JPA providers like EclipseLink, offer extended XML data support like query extensions, or allow mapping an XML DOM. In general, JAXB specification can be used to map XML data to objects.

Filters

In some conditions, it becomes useful if certain contents are removed from the query results from all the tables. This normally happens when a table is being shared by multiple applications or organizations or tenants. For example, a student database can have all the current students as well as alumni which can be differentiated with a Boolean variable in the table. So, when the application is developed based on this student database, report checking and things like that should be enabled only for current students. There can be other operations that are specific to the alumni.

One way of performing is to create separate views for each type of data. However, the type of data will be available only during run time, and this forces to create separate views for all the types of data which may not be always possible.

Another way is to add a criteria part to all the queries that are being executed for the application. This criteria part can be easily appended to the actual query using the type of data available at run-time.

Certain JPA providers provide support for filtering. TopLink / EclipseLink supports filtering data by annotation **`@AdditionalCriteria`**. This allows a custom additional arbitrary JPQL fragment to be appended to all queries while trying to query for that entity.

History

Past events occurring in a database are very important as they can be used for tracking and auditing purposes. It can also help us to derive useful insights about the application and its information.

Certain databases have auditing support and allow up to a certain level of tracking of the changes made. For example, the flashback feature of Oracle allows to store and track the changes made and also extends querying support on the past versions of data.

By defining the start and end timestamp columns, one can achieve the storage of the past events up to a certain extent. The start time stamp is updated whenever the row is created. When there is an update to the row, a new row is created with the current time and the old row's End timestamp is updated with the current time stamp to show that this was the timeframe when that particular row of data was relevant. Note that all the current rows will have the End timestamp as null. The table above can be queried like any other table to get the historical data as well.

Certain databases support Historical tables in particular like Oracle's flashback feature. Hence, some JPA providers also support historical queries. In EclipseLink this can be done by using the query hint "**`eclipselink.history.as-of`**" or Expression queries.

Logical deletes

Logical delete or soft delete is a mechanism to identify a data as deleted without wiping it out from the datastore. The most common technique used for this is to have a specific column that is set to indicate whether the row is deleted or not. No physical deletes of rows happen and this column value specifies the logical deletion of the data.

```
@Entity
@Table(name=”Student”)
@SQLDelete(sql=”update Student set isActive=0 where id=?”)
@Where(clause=”isActive = 1”)
public class Student {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Long id;
  private String name;
  private Integer age;
```

```
  private Boolean isActive=Boolean.FALSE;
  ……
}
```

***Code A1.5**: Sample java code using Hibernate JPA provider to implement logical delete*

In the code above, **`@SQLDelete`** annotation is provided to specify the operation to be performed instead of a delete operation on this entity. **`@Where`** annotation is used to provide the where condition while selecting the data for this particular entity. Hence, it makes sure that only active records are fetched by the application and no soft deleted records are included in the select queries. This is a type of filtering mechanism as mentioned in the section above on filters.

Auditing

Auditing is tracking of entity, its operations, and logging them for future references, in short, entity versioning. By default, JPA does not provide any API for auditing as such. However, it can be achieved by using the call back JPA events for providing a handler whenever there is a change in the entity. For example, events like **`@PreUpdate`**, **`@PreRemove`**, **`@PrePersist`** can be used for tracking and logging the changes. Audit entity listeners can be further implemented if multiple entity classes need to be tracked using the same mechanism.

Most of the JPA providers have their own way of providing auditing features. For example, Hibernate provides an artifact for this purpose (jar named **`hibernate-envers`**). This provides the facility of using **`@Audited`** annotation at the entity level or at the column level. If an audited entity is part of a relationship that need not be audited, then **`@NotAudited`** annotation can be used so that the relationship level changes are not audited and it strictly does the auditing for the entity alone.

In the case of Spring data JPA, the auditing has to be first enabled using **`@EnableJpaAuditing`** on the configuration class. Then, every entity to be audited is annotated with the **`@EntityListener`** annotation with the **`AuditingEntityListener`** class as its parameter.

Replication

Database replication is the process of copying data from one database to another database, mostly located in a different physical location in order to increase the performance and fault-tolerant behavior. The replication of data mostly results in distributed databases which further helps in load balancing and scaling of the database. So, in such cases, changes are to be written to multiple databases that should be taken care of either by the application layer, or the JPA layer, or the database layer itself.

Although most of the enterprise databases support some form of automatic backup or replication, JPA does not provide any specific support for replication in general. However, a few JPA providers do support replication-specific features. EclipseLink supports replication and load-balancing by **`@ReplicationPartitioning`**, **`@RoundRobinPartitioning`** annotations.

In the case of databases where replication is not supported, it can be managed by providing multiple persistence units and managing object persistence to both databases.

Partitioning

Data partitioning is performed when all the data set are not replicated into different databases, instead, a particular set of data is moved to a different database. This is normally used for easier scaling across multiple locations. Partitioning split the data across each database node, either vertically or horizontally. In vertical split, certain tables are in one database, whereas certain others are in another database. This makes it easier to implement but it should be considered that the entities should be partitioned in such a way that there are no two entities that have a relationship in two different partitions. This can be achieved by multiple persistence units that are different and can be used as separate vertical partitions.

In horizontal partitioning, the same set of tables remain throughout the database nodes. Only the data will be specific to each node. For example, a version of an application based on a location need not be storing the data of another location. It will be moved into a different database node. Note that all the tables will be the same since the same feature should be made available in both geographies. Horizontal partitioning is based on the data values and hence can be based on range, value, hash, or even round-robin.

Certain JPA providers support both vertical and horizontal partitioning. EclipseLink has support for both horizontal and vertical data partitioning using the following annotations:

```
@Partitioning, @HashPartitioning, @RangePartitioning,
@ValuePartitioning, @PinnedPartitioning and @Partitioned
```

Non-relational data

By default, JPA only depicts definitions for relational databases and their data. However, since these days non-relational databases like NoSQL, EIS, XML, and various other technologies are being used widely, to remain relevant in the data persistence layer, JPA providers have started supporting these through non-standard meta-data. In fact, one layer down, JDBC providers are also adapting SQL as well as

JDBC API to support non-relational databases which make the work easier for the JPA provider as such.

An example of how to use EclipseLink for NoSQL DB is given in section *Polyglot Persistence* of *Chapter 7, Hibernate and EclipseLink*. EclipseLink also provides support for EIS, XML, and various other non-relational data (refer to Chapter 7 for more details).

While using a combination of both relational and non-relational databases in an application, JPA has the advantage that it allows the application to consider both relational and non-relational data in the same way i.e., as Java objects, and hence reduce the complexity of data persistence and retrieval. However, using JPA for non-relational data can be tricky while switching between providers as their support is non-standardized.

The issue with support for non-relational databases is that these databases are not standardized based on the type of data and different databases support different data formats. For instance, consider various NoSQL databases. Based on the data formats stored, they are broadly classified into four types – key-value store, document-based store, column-based store, and graph-based store. Hence, this shows how vivid and different each non-relational database can be.

Multi-tenancy

Multi-tenancy is a way of software architecture where a single instance of the application provides services for separate multiple clients or tenants[1]. The tenants are given the facility to customize certain parts of the software like the color of the UI, dashboards to be created (if any), localization like time zone, language, etc. However, the application code cannot be modified by the tenants and remains the same for all the clients. This is a significant trend in the industry as it makes the services affordable due to cheaper installation costs.

- **Separate databases**: As the name suggests, each tenant has their own database.
- **Separate schemas**: Tenants use the same database but each tenant has their own set of tables or schema.
- **Shared schema**: Tenants share a common database, as well as a common schema and are determined by a tenant discriminator column.

The support for multi-tenancy is based on the JPA provider implementation. Hibernate uses tenant ID during session creation, and accordingly, the session will be created as shown as follows:

```
Session session = sessionFactory.withOptions()
                  .tenantIdentifier(tenantId)
                  .openSession();
```

Additional to the above passing of tenant ID, property **`hibernate.multiTenancy`** should be set with anyone among the following values–NONE (no multi-tenancy), SCHEMA (separate schema) or DATABASE (separate database). Based on the multi-tenancy style configured, a MultiTenantConnectionProvider should be configured that returns the session based on the tenant ID. This can either be implemented by the application developer or be defaulted to the standard implementation provided based on certain assumptions.

References

1. Multi-tenancy support in hibernate - **https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html# multitenacy**

PART 2

Sample JPA Application and Questions

Introduction

This section provides a sample application being developed by using JPA provider implementations and also explains how it helps in switching among data sources as well as provider implementations. This section also provides a few interesting questions that can help the reader to understand the topic in-depth with the implementation view point.

Sample1: Simple hibernate application

This is a sample Java application using Hibernate as the JPA provider and connecting to MySQL DB. Maven is used for dependency management. The user is assumed to have basic know-how about Java, DB queries, and maven.

Initially, the database does not have the table **`Student`**, which is created on running the application using Hibernate's property **`hibernate.hbm2ddl.auto`**.

```
mysql> select * from Student;
ERROR 1146 (42S02): Table 'jpa.student' doesn't exist
mysql>
```

***Figure A2.1**: MySQL prompt showing that the table does not exist in the DB named JPA*

The code is first compiled using maven to generate an executable jar. Then, the jar is executed to finally create the table, insert data, and query a few values from the table.

```
mysql>
mysql> select * from Student;
+------+-------+-------+------+
| id   | fName | lName | std  |
+------+-------+-------+------+
| 1013 | ABC   | DEF   | 5    |
| 1014 | MNO   | PQR   | 5    |
| 1015 | UVW   | XYZ   | 5    |
+------+-------+-------+------+
3 rows in set (0.00 sec)

mysql>
```

***Figure A2.2**: MySQL prompt showing that the table populated with data after execution*

```
mysql> select * from student_id;
+----------+
| next_val |
+----------+
|     1016 |
+----------+
1 row in set (0.00 sec)
```

***Figure A2.3**: MySQL prompt showing the table that stores the next ID in sequence*

```
Students with first name as 'ABC' is [First name = ABC Last Name = DEF Standard = 5]
Students with last name as 'XYZ' is [First name = UVW Last Name = XYZ Standard = 5]
Students with Standard as '5':
[First name = ABC Last Name = DEF Standard = 5, First name = MNO Last Name = PQR Standard = 5, First name = UVW Last Name = XYZ Standard = 5]
```

***Figure A2.4**: Output the values based on the queries executed*

Sample interview questions[6]

Almost all the textual questions are covered in the text. Sample references are provided for coding questions.

1. Describe JPA orphan removal. (refer Chapter 4)
2. Explain the persistence life cycle of an object. (refer Chapter 6)
3. What are the different types of identifier generation? (refer Chapter 3)
4. What is an entity? (refer Chapter 2)
5. What are the properties of an entity? (refer Chapter 2)
6. What is the role of the Entity Manager in JPA? (refer Chapter 6)

7. What are the constraints on an entity class? (refer Chapter 2)
8. How to handle @OneToMany relationship in the best possible way in JPA?[1]

Sample coding examples

1. Create a spring boot JPA application and change the JPA provider to EclipseLink.[2]
2. Create a JPA application that can be used against multiple data sources.[3,4]
3. Create a JPA application that uses a NoSQL database.[5]

References

1. The best way to handle One to Many relationships with JPA - **https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/**
2. A Guide to EclipseLink with Spring: **https://www.baeldung.com/spring-eclipselink**
3. Spring JPA Multiple Databases - **https://www.baeldung.com/spring-data-jpa-multiple-databases**
4. Connect to multiple databases in Hibernate - **https://stackoverflow.com/questions/1921865/how-to-connect-to-multiple-databases-in-hibernate**
5. Spring Boot and MongoDB in REST App - **https://www.dineshonjava.com/spring-boot-and-mongodb-in-rest-application/**
6. JPA Interview questions and answers for experienced - **https://codingcompiler.com/jpa-interview-questions-answers/**

Index