UNDERSTANDING JAVA PERSISTENCE API

Applications are made up of business logic, interaction with other systems, user interfaces etc. and data. Most of the data that our applications manipulate have to be stored in datastores, retrieved, processed and analysed. Datastores are important: they store business data, act as a central point between applications, and can even sometimes process data themselves (e.g. relational databases have triggers and stored procedures). They can take several forms, from relational databases to schemaless databases. But relational databases are what are important to us when talking about JPA.

When developing an application in an object-oriented programming language such as Java, this data can be represented using objects. Objects encapsulate state and behaviour in a nice way, but this state is only accessible when the *Java Virtual Machine* (JVM) is running: if the JVM stops or the garbage collector cleans its memory content, objects disappear, as well as their state. Some objects need to be persistent and, by persistent, I mean objects that are deliberately stored in a permanent datastore so that their state can be reused later.

Object-Relational Mapping tools have been created in several languages in order to bridge objects and relational databases. *Java Persistence API* (JPA) is the standard Java framework to achieve this object-relational mapping. It relies on JDBC and transaction management.

Understanding JDBC

JDBC stands for *Java DataBase Connectivity*.[1] It is a Java-based technology providing methods for querying and updating data for relational databases. It arrived in the JDK 1.1 in 1997 and today, JDBC 4.2 is part of the JDK 1.8. This old API is very robust and used in many projects and frameworks. However, JDBC is a low-level API and it ends up being quite verbose: it takes several lines of code just to bind an object to an SQL (*Structured Query Language*) query.[2]

For example, take the code in Listing 1 that inserts a new book into the `BOOK` table. With JDBC, we need to create an SQL query to insert data. This query follows the SQL syntax and inserts data into the columns of `BOOK`. We do the mapping programmatically between each attribute of the `Book` class, and the `BOOK` table. The way the mapping works is that we bind the book identifier, as a `Long`, to the first argument of the query. The book title is mapped as a String, to the second argument of the query, the description to the third argument, and so on and so forth. Once the mapping is finished, we execute the query and the data is inserted into the database.

Listing 1. Using JDBC to Insert a New Row in the BOOK Table

```
String query = "INSERT INTO BOOK (ID, TITLE, DESCRIPTION, PRICE,
ISBN) VALUES (?, ?, ?, ?, ?)";

try (PreparedStatement stmt =
getConnection().prepareStatement(query)) {

  stmt.setLong(1, book.getId());
  stmt.setString(2, book.getTitle());
  stmt.setString(3, book.getDescription());
```

```
   stmt.setFloat(4, book.getPrice());
   stmt.setString(5, book.getIsbn());

   stmt.executeUpdate();
 }
```

There are a few things that are wrong with the code shown in Listing 1. First of all, SQL is a different language and Java developers might not be confident with it. Once SQL is mastered, the statements are not easy to refactor. If the database structure changes, or the object evolves, we need to manually update each SQL statement. That's because the structure of the DB is closely related to the object, there is no abstraction between both. This programmatic mapping makes the code really verbose, making it harder to read and harder to maintain. Object-Relational Mapping tools solve these problems by bringing some abstraction.

Understanding Transactions

Transaction management is an important matter for enterprises. It allows applications to have consistent data and to process that data in a reliable manner. Transaction management is a low-level concern that a business developer shouldn't have to code. That's why JTA (*Java Transaction API*) provides these services in a very simple way: either programmatically with a high level of abstraction or declaratively using metadata.

A *transaction* is used to ensure that the data is kept in a consistent state. It represents a logical group of operations that must be performed as a single unit, also known as a *unit of work*. These operations can involve persisting data in one or several databases, sending messages to a MOM (*Message-Oriented Middleware*), or invoking web services. Companies rely on transactions every day for their banking and e-commerce applications or business-to-business interactions with partners.

These indivisible business operations are performed either sequentially or in parallel over a relatively short period of time. Every operation must succeed in order for the transaction to succeed (we say that the transaction is *committed*). If one of the operations fails, the transaction fails as well (the transaction is rolled back). In terms of code, this is what it looks like:

```
entityManager.getTransaction().begin();
entityManager.getTransaction().commit();
entityManager.getTransaction().rollback();
```

Transactions must guarantee a degree of reliability and robustness and follow the ACID properties. ACID refers to the four properties that define a reliable transaction: *Atomicity*, *Consistency*, *Isolation*, and *Durability* (described in Table 1).

Table 1. ACID Properties

Property	Description
Atomicity	A transaction is composed of one or more operations grouped in a unit of work. At the conclusion of the transaction, either these operations are all performed successfully (a commit) or none of them is performed at all (a rollback) if something unexpected or irrecoverable happens.
Consistency	At the conclusion of the transaction, the data are left in a consistent state.
Isolation	The intermediate state of a transaction is not visible to external applications.
Durability	Once the transaction is committed, the changes made to the data are visible to other applications.

To explain these properties, I'll take the classic example of a bank transfer: you need to debit your savings account to credit your current account.

When you transfer money from one account to the other, you can imagine a sequence of database accesses: the savings account is debited using an SQL update statement, the current account is credited using a different update statement, and a log is created in a different table to keep track of the transfer. These operations have to be done in the same unit of work (*Atomicity*) because you don't want the debit to occur but not the credit. From the perspective of an external application querying the accounts, only when both operations have been successfully performed are they visible (*Isolation*). With isolation, the external application cannot see the interim state when one account has been debited and the other is still not credited (if it could, it would think the customer has less money than they really do). *Consistency* is when transaction operations (either with a commit or a rollback) are performed within the constraints of the database (such as primary keys, relationships, or fields). Once the transfer is completed, the data can be accessed from other applications (*Durability*).

Understanding Object-Relational Mapping

As we’ve just seen, relational databases store data in tables made of rows and columns. Data is identified by primary keys, which are special columns (or a combination of columns) designated to uniquely identify each table record. The relationships between tables use foreign keys and join tables with integrity constraints.

All this vocabulary is completely unknown in an object-oriented language such as Java. In Java, we manipulate objects that are instances of classes. Objects inherit from others, have references to collections of other objects, and sometimes point to themselves in a recursive manner. We have concrete classes, abstract classes, interfaces, enumerations, annotations, methods, attributes, and so on.

As seen in Figure 1, the principle of *Object-Relational Mapping* (ORM) is to bring the world of relational databases and objects together. ORMs are external tools that give an object-oriented view of relational data, and vice versa.

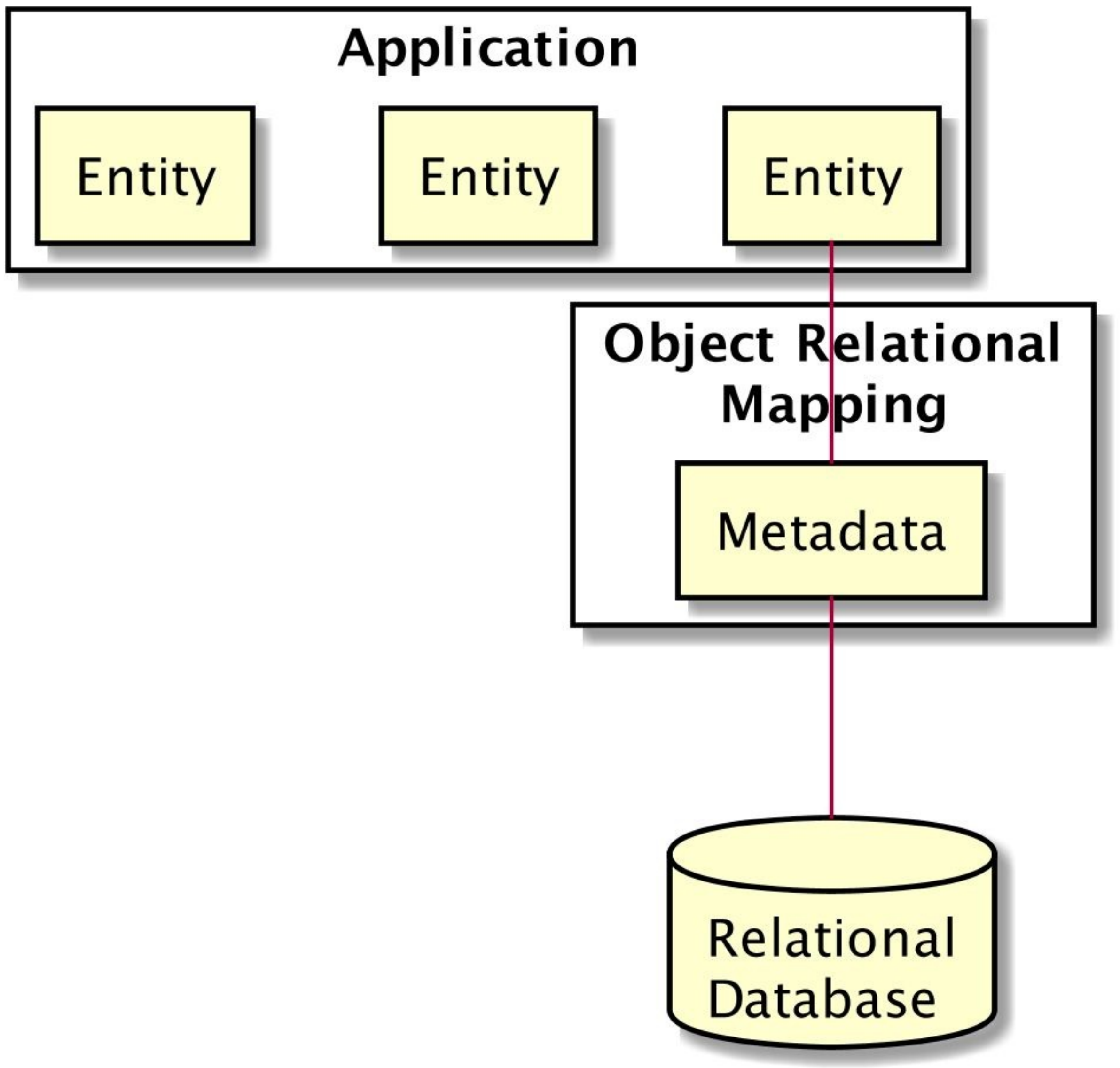

Figure 1. Mapping objects to a relational database

Relational Databases

The relational model organises data into one or more tables made of columns and rows, with a unique key identifying each row. Rows are also called records or tuples. Generally, each table represents one entity type (such as a book, author or purchase order). But in some cases, it can represent several entities (such as inheritance, as you'll see in Chapter 4). The rows represent instances of that type of entity (such as the book "H2G2" or "Design Patterns") with the columns representing values attributed to that instance (such as the title of the book or the price).

A new era of NoSQL (Not Only SQL) databases (or schemaless) has emerged with different storage structures: key-values, column, document, or graph. At the moment, JPA is not able to map entities to these structures. Hibernate OGM is an open source framework that attempts to address mapping objects to non-relational databases.[3] EclipseLink also has some extensions to map NoSQL structures.[4] Hibernate OGM or EclipseLink extensions are beyond the scope of this fascicle, but you should have a look at them if you plan to use NoSQL databases.

How would you store data representing a book into a relational database? Listing 2 shows an SQL script that creates such a table.

Listing 2. SQL Script Creating a BOOK Table Structure

```
CREATE TABLE BOOK
(
  ID            BIGINT NOT NULL,
  DESCRIPTION   VARCHAR,
  ILLUSTRATIONS BOOLEAN,
  ISBN          VARCHAR,
  NBOFPAGES     INTEGER,
  PRICE         DOUBLE,
  TITLE         VARCHAR,
  PRIMARY KEY (ID)
)
```

A *Data Definition Language* (DDL, or data description language) uses a syntax for defining database structures. The table `BOOK` is where we will find all the books of our application. Each book is identified by a unique primary key column (`PRIMARY KEY (ID)`) and each attribute is stored in a column (e.g. `TITLE`, `PRICE`, `ISBN` etc.). A column in a table has a type (e.g. `VARCHAR`, `INTEGER`, `BOOLEAN` etc.) and can accept null values or not (`NOT NULL`).

Entities

When talking about mapping objects to a relational database, persisting objects, or querying objects, the term *entity* should be used rather than *object*. Objects are instances that just live in memory. Entities are objects that live shortly in memory and persistently in a relational database. They have the ability to be mapped to a database; they can be concrete or abstract; and they support inheritance, relationships, and so on.

In the JPA persistence model, an entity is a *Plain Old Java Object* (POJO). This means an entity is declared, instantiated, and used just like any other Java class. An entity usually has attributes (its state), can have business methods (its behaviour), constructors, getters and setters. Listing 3 shows a simple entity.

Listing 3. Simple Example of a Book Entity

```
@Entity
public class Book {

  @Id
  @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
```

```
    private String isbn;
    private Integer nbOfPages;
    private Boolean illustrations;

    // Constructors, getters, setters
  }
```

The example in Listing 3 represents a `Book` entity from which I've omitted the getters and the setters for clarity. As you can see, except for some annotations, this entity looks exactly like any Java class: it has several attributes (`id`, `title`, `price` etc.) of different types (`Long`, `String`, `Float`, `Integer`, and `Boolean`), a default constructor, and getters and setters for each attribute. So how does this map to a table? The answer is: thanks to mapping.

Mapping Entities

The principle of Object-Relational Mapping (ORM) is to delegate the task of creating a correspondence between objects and tables, to external tools or frameworks (in our case, JPA). The world of classes, objects, and attributes can then be mapped to relational databases which are made up of tables containing rows and columns. Mapping gives an object-oriented view to developers who can transparently use entities instead of tables. And how does JPA map objects to a database? As seen in Figure 1, this is done through *metadata*.

Associated with every entity are metadata that describe the mapping. The metadata enable the persistence provider to recognise an entity and to interpret the mapping. The metadata can be written in two different formats:

- *Annotations*: The code of the entity is directly annotated with all sorts of annotations (see Listing 3).
- *XML descriptors*: Instead of (or in addition to) annotations, we can use XML descriptors. The mapping is defined in an external XML file that will be deployed with the entities (see Chapter 3).

These entities, once mapped, can be managed by JPA. You can persist an entity in the database, remove it, and query it. ORM lets you manipulate entities while, under the covers, the database is being accessed.

The `Book` entity (shown in Listing 3) uses JPA annotations so the persistence provider can synchronise the data between the attributes of the `Book` entity and the columns of the `BOOK` table. Therefore, if the attribute `isbn` is updated by the application, the `ISBN` column will be synchronised (see Figure 2). As you will see in Chapter 5, this synchronisation can be automatic or programmatic depending on the life cycle of the entity.

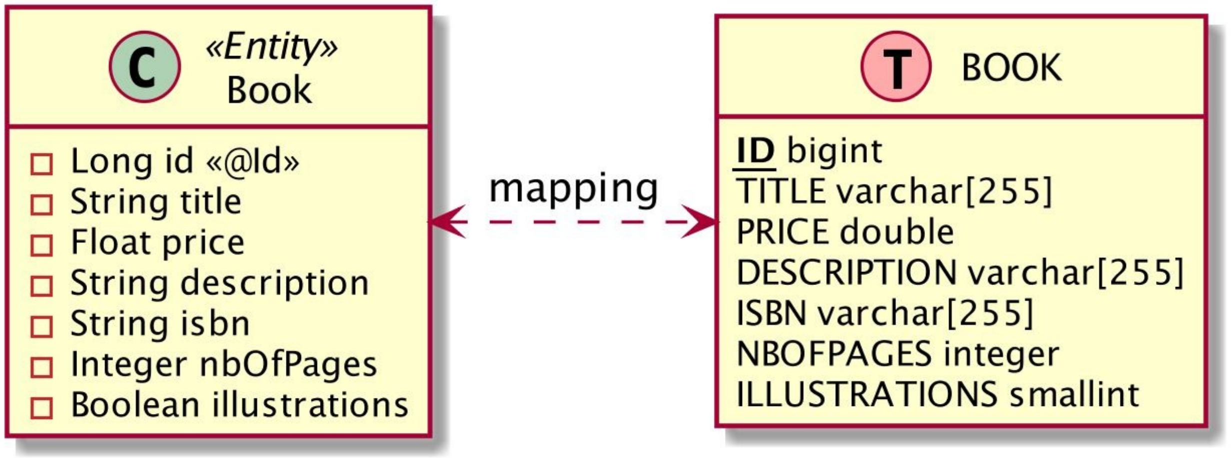

Figure 2. Data synchronisation between the entity and the table

As Listing 2 shows, the `Book` entity is mapped in a `BOOK` table, and each column is named after the attribute of the class (e.g. the `isbn` attribute of type `String` is mapped to a column named `ISBN` of type `VARCHAR`).

In this fascicle, I use CamelCase for Java code (e.g. `Book` entity, `isbn` attribute) and UpperCase for SQL script (e.g. `BOOK` table, `ISBN` column).

Without JPA and metadata, the `Book` entity in Listing 3 would be treated just like a POJO and not be persisted. That is the rule: if no special configuration is given, the default should be applied, and the default for the persistence provider is that the `Book` class has no database representation. But because you need to change this default behaviour, you annotate the class with `@Entity`. It is the same for the identifier. You need a way to tell the persistence provider that the `id` attribute has to be mapped to a primary key, so you annotate it with `@Id`. The value of this identifier is automatically generated by the persistence provider, using the optional `@GeneratedValue` annotation. This type of decision characterizes the configuration-by-exception approach, in which annotations are not required for the more common cases and are only used as metadata to be understood by an external provider.

Managing Entities

JPA allows us to map entities to a table and also to query them using different criteria. JPA’s power is that it offers the ability to query entities and their relationships in an object-oriented way without the developer having to use the foreign keys or columns of the underlying database. The central piece of the API responsible for orchestrating entities is the `javax.persistence.EntityManager`. Its role is to manage entities, read from and write to a given database, and allow simple CRUD (create, read, update, and delete) operations on entities as well as complex queries using JPQL (more on *Java Persistence Query Language* in Chapter 6). In a technical sense, the entity manager is just an interface whose implementation is done by the persistence provider (e.g. EclipseLink or Hibernate). At its core, the entity manager delegates all the low-level calls to JDBC bringing the developer a higher level of abstraction.

The JPA implementation (i.e. the persistence provider) used for this fascicle is EclipseLink 2.7.x.[5]

In Figure 3, you can see how the `EntityManager` interface can be used by a class (here `Main`) to manipulate entities (in this case, `Book`). With methods such as `persist()` and `find()`, the entity manager hides the JDBC calls to the database as well as the `INSERT` or `SELECT` SQL (*Structured Query Language*) statements.

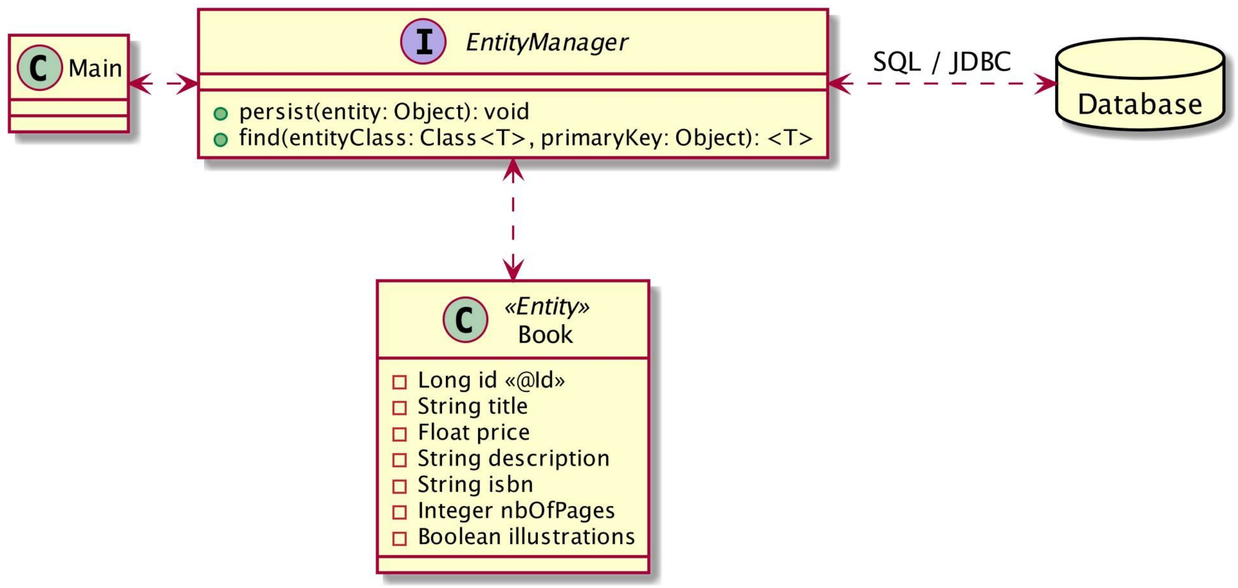

Figure 3. The entity manager interacts with the entity and the underlying database

JPA Overview

In the Java world, there are several frameworks and specifications to achieve object-relational mapping, such as Hibernate, TopLink, or *Java Data Objects* (JDO), but *Java Persistence API* (JPA) is the preferred technology.

Java Persistence API (JPA) is a Java specification that manages objects stored in a relational database.[6] JPA gives the developer an object-oriented view in order to transparently use entities instead of tables. It also comes with a query language (*Java Persistence Query Language*, or JPQL), allowing complex queries over objects.

JPA is just a specification that is part of Java EE and is governed by the JCP (*Java Community Process*). It is then implemented by frameworks such as EclipseLink, Hibernate or OpenJPA.

A Brief History of JPA

ORM solutions have been around for a long time, even before Java. Products such as TopLink originally started with Smalltalk in 1994 before switching to Java.[7] Commercial ORM products like TopLink have been available since the earliest days of the Java language. They were successful but were never standardised for the Java platform. A similar approach to ORM was standardised in the form of JDO (*Java Data Objects*), which failed to gain any significant market penetration.[8]

In 1998, EJB (*Enterprise JavaBeans*) 1.0 was created and later shipped with J2EE (*Java Enterprise Edition*) 1.2. It was a heavyweight, distributed component used for transactional business logic. Entity Bean CMP (*Container-Managed Persistence*) was introduced as optional in EJB 1.0, became mandatory in EJB 1.1, and was enhanced through versions up to EJB 2.1 (J2EE 1.4).[9] Persistence could only be done inside the container through a complex mechanism of instantiation using home, local, or remote interfaces. The capabilities of ORM were also very limited, as inheritance was difficult to map.

Parallel to the J2EE world was a popular open source solution that led to some surprising changes in the direction of persistence: Hibernate, which brought back a lightweight, object-oriented persistent model.[10]

After years of complaints about Entity CMP 2. x components and, in acknowledgment of the success and simplicity of open source frameworks such as Hibernate, the persistence model of the Enterprise Edition was completely rearchitected in Java EE 5. JPA 1.0 was created with a very lightweight approach that adopted many Hibernate design principles. The JPA 1.0 specification was bundled with EJB 3.0 (JSR 220).[11] In 2009, JPA 2.0 (JSR 317) was released with Java EE 6 and brought new APIs, extended JPQL, and added new functionalities such as a second-level cache, pessimistic locking, or the criteria API.[12] In 2013, with Java EE 7, JPA 2.1 followed the path of ease of development and brought new features such as schema generation, converters or CDI supports in JSR 338.

[13] Today JPA is on its version 2.2 that was shipped at the same time as Java EE 8.[14]

JCP and Eclipse Foundation

The JCP (*Java Community Process*) is an open organisation, created in 1998 by Sun Microsystems, that is involved in the definition of future versions and features of the Java platform.[15] When the need for standardising an existing component or API is identified, the initiator (a.k.a. specification lead) creates a JSR (*Java Specification Request*) and forms a group of experts. This group, made up of company representatives, organisations, universities, and private individuals, is responsible for the development of the JSR and has to deliver:

- One or more specifications that explain the details and define the fundamentals of the JSR,
- A *Reference Implementation* (RI), which is an actual implementation of the specification, and
- A *Technology Compatibility Kit* (TCK), which is a set of tests every implementation needs to pass before claiming to conform to the specification.

Once approved by the *Executive Committee* (EC), the specification is released to the community for other projects to implement.

This is exactly the standardisation process that followed JPA 2.2. It has been standardised under JSR 338, has a reference implementation (EclipseLink) and has a TCK.[16]

In 2017, most Java EE specifications were moved from the JCP to the Eclipse Foundation (including the JPA specification).[17] The Eclipse Foundation is an independent, not-for-profit corporation that acts as a steward of the Eclipse open source software development community. The Foundation focuses on key services such as: intellectual property management, ecosystem development, development process, and IT infrastructure. It was originally created by IBM in 2001 and is now supported by a consortium of several software vendors.

Java EE and Jakarta EE

Even if Java Persistence API does not rely on Java EE, it is strongly correlated with this platform. So, it is worth mentioning Java EE in a JPA fascicle.

Java EE, or *Java Enterprise Edition*, can be seen as an extension of the *Java Standard Edition* (Java SE). Usually, when you want to handle collections of objects, you don’t start by developing your own list. Instead, you use the collection API. Similarly, if you need a simple web application or a transactional, secure, interoperable and distributed application, there is no need to develop all the low-level APIs. Rather, you use the enterprise edition of Java. Just as Java Standard Edition provides an API to handle collections, Java EE provides a standard way to handle transactions with Java Transaction API (JTA), messaging with Java Message Service (JMS), or persistence with Java Persistence API (JPA). Java EE is a set of specifications intended for enterprise applications. It can be seen as an extension of Java SE in order to facilitate the development of distributed, robust, powerful, and highly available applications.

Java EE benefits from the dynamism of open source communities as well as the rigour of the JCP standardisation process. Today Java EE is a well-documented platform with experienced developers, a large community, and many deployed applications running on company servers. Java EE is a suite of APIs that can be used to build standard component-based multi-tier applications.

In 2017, with version 8 of the platform, Java EE was donated to the Eclipse Foundation and renamed Jakarta EE.[18] Jakarta EE is the name of the platform governed by the Jakarta EE Working Group.[19] The first version is Jakarta EE 8, which is based on the Java EE 8 technologies. Future versions will not be driven by the JCP but through the open Eclipse Foundation.

What's New in JPA 2.2?

JPA 2.2 is described under the JSR 338 and was released in 2017.[20] It was shipped in 2017 with Java EE 8 but it was just a *maintenance release*, meaning that it uses the same JSR as JPA 2.1 (still the JSR 338). A maintenance release means that a specification doesn't evolve much, and therefore doesn't need a new JSR. The changes made to JPA 2.2 are:

- Adds support for Java SE 8 Lambdas and Streams.
- Adds `@Repeatable` meta-annotation to JPA annotations.
- Support for CDI injection into `AttributeConverter` classes.

- Support for the mapping of the new `java.time` types (e.g. `LocalDate`, `LocalTime` etc.).[21]
- Adds default `Stream getResultStream()` method to `Query` and `TypedQuery` interfaces.

Most of these novelties will be discussed in the chapters that follow.

Appendix B lists all the revisions and major changes of the Java Persistence API specification.

Implementations

EclipseLink is the open source reference implementation of JPA.[22] The project was based on the TopLink product from which Oracle contributed the source code to create the EclipseLink project. Today, EclipseLink 2.7.x implements JPA 2.2. It also adds specific features as it also supports XML persistence through Java XML Binding (JAXB) and through other means such as Service Data Objects (SDOs). It provides support not only for ORM but also for object XML mapping (OXM), object persistence to enterprise information systems (EIS) using Java EE Connector Architecture (JCA), and database web services.

At the time of writing this fascicle, EclipseLink is not the only JPA 2.2 compliant implementation, there are others. Hibernate ORM is the famous JPA implementation from RedHat.[23] It is used in all the RedHat products including the JBoss Application Server or Wildfly.

OpenJPA is the Apache implementation and will soon implement JPA 2.2.[24]

As you can see, despite the reference implementation (EclipseLink), you have several implementations to choose from.

1 JDBC https://www.oracle.com/technetwork/java/overview-141217.html

2 SQL https://fr.wikipedia.org/wiki/Structured_Query_Language

3 Hibernate OGM http://hibernate.org/ogm

4 EclipseLink NoSQL extension https://wiki.eclipse.org/EclipseLink/Examples/JPA/NoSQL

5 EclipseLink http://www.eclipse.org/eclipselink/

6 JPA https://jcp.org/en/jsr/detail?id=338

7 Toplink http://www.oracle.com/technetwork/middleware/toplink

8 JDO https://jcp.org/en/jsr/detail?id=243

9 EJB 2.1 https://jcp.org/en/jsr/detail?id=153

10 Hibernate http://hibernate.org

11 JSR 220 https://jcp.org/en/jsr/detail?id=220

12 JSR 317 https://jcp.org/en/jsr/detail?id=317

13 JSR 338 https://jcp.org/en/jsr/detail?id=338

14 Java EE 8 https://jcp.org/en/jsr/detail?id=366

15 JCP https://jcp.org

16 JPA https://jcp.org/en/jsr/detail?id=338

17 Eclipse Foundation https://www.eclipse.org/org/foundation

18 Eclipse Foundation https://www.eclipse.org/org/foundation/

19 Jakarta EE https://jakarta.ee

20 JSR 338 https://jcp.org/en/jsr/detail?id=338

21 JSR 310 https://jcp.org/en/jsr/detail?id=310

22 EclipseLink http://www.eclipse.org/eclipselink

23 Hibernate ORM http://hibernate.org/orm

24 OpenJPA http://openjpa.apache.org