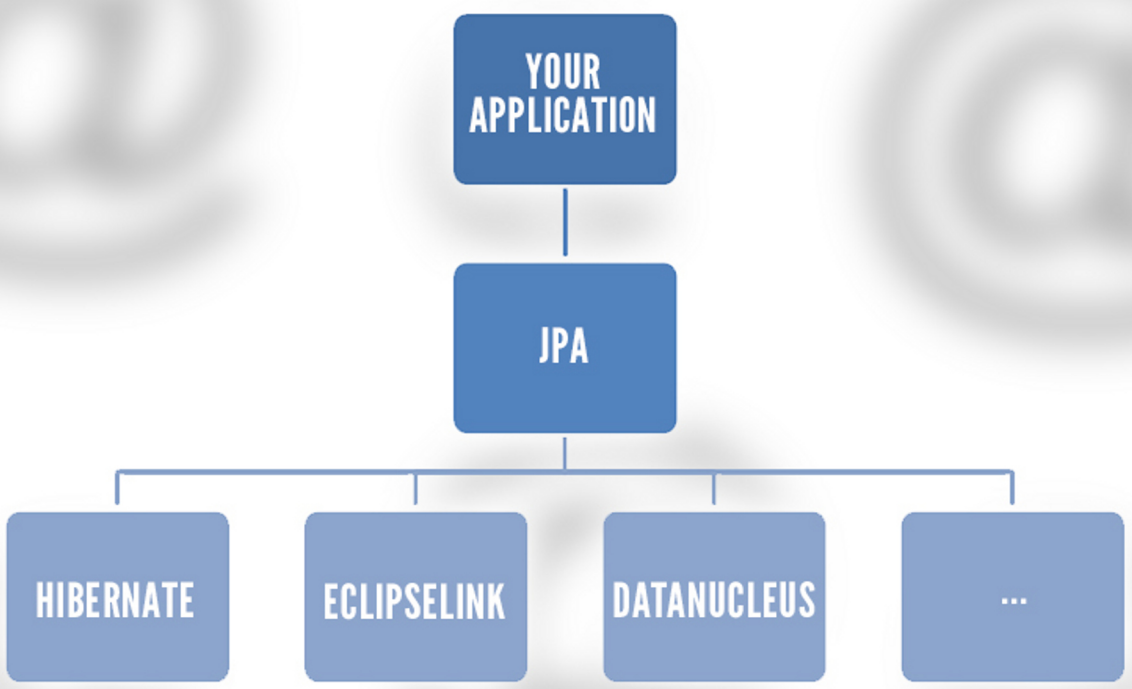
JPA TUTORIAL

THE ULTIMATE GUIDE



MARTIN MOIS

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

JPA Tutorial

Contents

Preface

The Java Persistence API (JPA) is a Java programming language application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

JPA has become the de-facto standard to write application code that interacts with Databases. For this reason we have provided an abundance of tutorials here at Java Code Geeks, most of which can be found here: http://www.javacodegeeks.com/tutorials/-java-tutorials/enterprise-java-tutorials/#JPA.

Now, we wanted to create a standalone, reference guide to provide a framework on how to work with JPA and help you quickly kick-start your JPA applications. Enjoy!

About the Author

Martin is a software engineer with more than 10 years of experience in software development. He has been involved in different positions in application development in a variety of software projects ranging from reusable software components, mobile applications over fat-client GUI projects up to larg-scale, clustered enterprise applications with real-time requirements.

After finishing his studies of computer science with a diploma, Martin worked as a Java developer and consultant for international operating insurance companies. Later on he designed and implemented web applications and fat-client applications for companies on the energy market. Currently Martin works for an international operating company in the Java EE domain and is concerned in his day-to-day work with larg-scale big data systems.

His current interests include Java EE, web applications with focus on HTML5 and performance optimizations. When time permits, he works on open source projects, some of them can be found at this github account. Martin is blogging at Martin's Developer World.

Chapter 1

Introduction

The Java Persistence API (JPA) is a vendor independent specification for mapping Java objects to the tables of relational databases. Implementations of this specification allow application developers to abstract from the specific database prodcut they are working with and allow them to implement CRUD (create, read, update and delete) operations such that the same code works on different database products. These frameworks do not only handle the code that interacts with the database (the JDBC code) but also map the data to object structures used by the application.

The JPA consists of three different components:

- `Entities`: In current versions of the JPA entities are plain old Java objects (POJOs). Older versions of the JPA required to subclass entities from classes provided by the JPA, but as these approaches are more difficult to test because of their hard dependecies to the framework, newer versions of the JPA do not require entities to subclass any framework class.
- `Object-relational metadata`: The application developer has to provide the mapping between the Java classes and their attributes to the tables and columns of the database. This can be done either trough dedicated configuration files or in newer version of the framework also by annotations.
- `Java Persistence Query Language (JPQL)`: As the JPA aims to abstract from the specific database product, the framework also provides a dedicated query langauge that can be used instead of SQL. This additional translation from JPQL to SQL allows the implementations of the framework to support different database dialects and allows the application developer to implement queries in a database independent way

In this tutorial we are going through different aspects of the framework and will develop a simple Java SE application that stores and retrieves data in/from a relational database. We will use the following libraries/environments:

- maven >= 3.0 as build environment
- JPA 2.1 as contained in the Java Enterprise Edition (JEE) 7.0
- Hibernate as an implementation of JPA (4.3.8.Final)
- H2 as relational database in version 1.3.176

Chapter 2

Project setup

As a first step we will create a simple maven project on the command line:

```
mvn archetype:create -DgroupId=com.javacodegeeks.ultimate -DartifactId=jpa
```

This command will create the following structure in the file system:

```
|-- src
|   |-- main
|   |   `-- java
|   |       `-- com
|   |           `-- javacodegeeks
|   |                       `-- ultimate
|   `-- test
|   |   `-- java
|   |       `-- com
|   |           `-- javacodegeeks
|   |                       `-- ultimate
`-- pom.xml
```

The libraries our implementation depends on are added to the dependencies section of the pom.xml file in the following way:

```
<properties>
        <jee.version>7.0</jee.version>
        <h2.version>1.3.176</h2.version>
        <hibernate.version>4.3.8.Final</hibernate.version>
</properties>

<dependencies>
        <dependency>
                <groupId>javax</groupId>
                <artifactId>javaee-api</artifactId>
                <version>${jee.version}</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>com.h2database</groupId>
                <artifactId>h2</artifactId>
                <version>${h2.version}</version>
        </dependency>
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-entitymanager</artifactId>
                <version>${hibernate.version}</version>
        </dependency>
</dependencies>
```

To get a better overview of the separate versions, we define each version as a maven property and reference it later on in the dependencies section.

Chapter 3

Basics

3.1 EntityManager and Persistence Unit

Now we cat start to implement our first JPA functionality. Let's start with a simple class that provides a `run()` method that is invoked in the application's `main` method:

```
public class Main {
        private static final Logger LOGGER = Logger.getLogger("JPA");

        public static void main(String[] args) {
                Main main = new Main();
                main.run();
        }

        public void run() {
                EntityManagerFactory factory = null;
                EntityManager entityManager = null;
                try {
                        factory = Persistence.createEntityManagerFactory("PersistenceUnit") ↩
                            ;
                        entityManager = factory.createEntityManager();
                        persistPerson(entityManager);
                } catch (Exception e) {
                        LOGGER.log(Level.SEVERE, e.getMessage(), e);
                        e.printStackTrace();
                } finally {
                        if (entityManager != null) {
                                entityManager.close();
                        }
                        if (factory != null) {
                                factory.close();
                        }
                }
        }
        ...
```

Nearly all interaction with the JPA is done through the `EntityManager`. To obtain an instance of an `EntityManager`, we have to create an instance of the `EntityManagerFactory`. Normally we only need one `EntityManagerFactory` for one "persistence unit" per application. A persistence unit is a set of JPA classes that is managed together with the database configuration in a file called `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
            xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <persistence-unit name="PersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="connection.driver_class" value="org.h2.Driver"/>
            <property name="hibernate.connection.url" value="jdbc:h2:~/jpa"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
            <property name="hibernate.show_sql" value="true"/>
                        <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

This file is created in the `src/main/resource/META-INF` folder of the maven project. As you can see, we define one `persistence-unit` with the name `PersistenceUnit` that has the `transaction-type` `RESOURCE_LOCAL`. The transaction-type determines how transactions are handled in the application.

In our sample application we want to handle them on our own, hence we specify here `RESOURCE_LOCAL`. When you use a JEE container then the container is responsible for setting up the `EntityManagerFactory` and only provides you he `EntityManager`. The container then also handles the begin and end of each transaction. In this case you would provide the value `JTA`.

By setting the `provider` to `org.hibernate.ejb.HibernatePersistence` we select the JPA implementation we want to use. Because we have included Hibernate as JPA provider on the classpath by defining a dependency to it, we can reference the provider implementation here by class name.

The next step in `persistence.xml` is to inform the JPA provider about the database we want to use. This is done by specifying the JDBC driver that Hibernate should use. As we want to use the H2 database (www.h2database.com), the property `connection.driver_class` is set to the value `org.h2.Driver`.

In order to enable Hibernate to create connections to the database we also have to provide the connection URL. H2 provides the option to create the database within a single file which path is given in the JDBC URL. Hence the JDBC URL `jdbc:h2:~/jpa` tells H2 to create in the user's home directory a file called `jpa.h2.db`.

H2 is a fast open source database that can be easily embedded in Java applications as it comes as a single jar file with a size of about 1.5 MB. This makes it suitable for our purpose of creating a simple sample application to demonstrate the usage of JPA. In projects that require solutions that scale better for huge amounts of data you would probably choose a different database product, but for small amount of data that have strong relations H2 is good choice.

The next thing we have to tell Hibernate is the JDBC dialect that it should use. As Hibernate provides a dedicated dialect implementation for H2, we choose this one with the property `hibernate.dialect`. With this dialect Hibernate is capable of creating the appropriate SQL statements for the H2 database.

Last but not least we provide three options that come handy when developing a new application, but that would not be used in production environments. The first of them is the property `hibernate.hbm2ddl.auto` that tells Hibernate to create all tables from scratch at startup. If the table already exists, it will be deleted. In our sample application this is a good feature as we can rely on the fact that the database is empty at the beginnig and that all changes we have made to the schema since our last start of the application are reflected in the schema.

The second option is `hibernate.show_sql` that tells Hibernate to print every SQL statement that it issues to the database on the command line. With this option enabled we can easily trace all statements and have a look if everything works as expected. And finally we tell Hibernate to pretty print the SQL for better readability by setting the property `hibernate.format_sql` to `true`.

Now that we have setup the `persistence.xml` file, we get back to our Java code above:

```
EntityManagerFactory factory = null;
EntityManager entityManager = null;
try {
        factory = Persistence.createEntityManagerFactory("PersistenceUnit");
```

```
        entityManager = factory.createEntityManager();
        persistPerson(entityManager);
} catch (Exception e) {
        LOGGER.log(Level.SEVERE, e.getMessage(), e);
        e.printStackTrace();
} finally {
        if (entityManager != null) {
                entityManager.close();
        }
        if (factory != null) {
                factory.close();
        }
}
```

After having obtained an instance of the `EntityManagerFactory` and from it an instance of `EntityManager` we can use them in the method `persistPerson` to save some data in the database. Please note that after we have done our work we have to close both the `EntityManager` as well as the `EntityManagerFactory`.

3.2 Transactions

The `EntityManager` represents a persistence unit and therefore we will need in `RESOURCE_LOCAL` applications only one instance of the `EntityManager`. A persistence unit is a cache for the entities that represent parts of the state stored in the database as well as a connection to the database. In order to store data in the database we therefore have to pass it to the `EntityManager` and therewith to the underlying cache. In case you want to create a new row in the database, this is done by invoking the method `persist()` on the `EntityManager` as demonstrated in the following code:

```
private void persistPerson(EntityManager entityManager) {
        EntityTransaction transaction = entityManager.getTransaction();
        try {
                transaction.begin();
                Person person = new Person();
                person.setFirstName("Homer");
                person.setLastName("Simpson");
                entityManager.persist(person);
                transaction.commit();
        } catch (Exception e) {
                if (transaction.isActive()) {
                        transaction.rollback();
                }
        }
}
```

But before we can call `persist()` we have to open a new transaction by calling `transaction.begin()` on a new transaction object we have retrieved from the `EntityManager`. If we would omit this call, Hibernate would throw a `IllegalStateException` that tells us that we have forgotten to run the `persist()` within a transaction:

```
java.lang.IllegalStateException: Transaction not active
        at org.hibernate.jpa.internal.TransactionImpl.commit(TransactionImpl.java:70)
        at jpa.Main.persistPerson(Main.java:87)
```

After calling `persist()` we have to commit the transaction, i.e. send the data to the database and store it there. In case an exception is thrown within the try block, we have to rollback the transaction we have begun before. But as we can only rollback active transactions, we have to check before if the current transaction is already running, as it may happen that the exception is thrown within the `transaction.begin()` call.

3.3 Tables

The class Person is mapped to the database table `T_PERSON` by adding the annotation `@Entity`:

```
@Entity
@Table(name = "T_PERSON")
public class Person {
	private Long id;
	private String firstName;
	private String lastName;

	@Id
	@GeneratedValue
	public Long getId() {
		return id;
	}

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

	@Column(name = "FIRST_NAME")
	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	@Column(name = "LAST_NAME")
	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
}
```

The additional annotation `@Table` is optional, but you can use it to specify a specific table name. In our example we want that all tables have the prefix T_, hence we tell Hibernate to create the table `T_PERSON`. The table `T_PERSON` has three columns: `ID`, `FIRST_NAME`, `LAST_NAME`.

This information is provided to the JPA provider with the annotations `@Column` and their attribute `name`. Even this annotation is optional. JPA will use all properties of the Java class that have a setter and getter method and create columns for them, as long as you do not exclude them by annotating them with `@Transient`. On the other hand you can specify more information for each column by using the other attributes that the `@Column` annotation provides:

```
@Column(name = "FIRST_NAME", length = 100, nullable = false, unique = false)
```

The example above restricts the length of the string colum to 100 characters, states that the column must not contain null values and is not unique. Trying to insert null as first name into this table would provoke a constraint violation on the database and cause the current transaction to roll back.

The two annotations `@Id` and `@GeneratedValue` tell JPA that this value is the primary key for this table and that it should be generated automatically.

In the example code above we have added the JPA annotations to the getter methods for each field that should be mapped to a database column. Another way would be to annotate the field directly instead of the getter method:

```
@Entity
@Table(name = "T_PERSON")
public class Person {
	@Id
	@GeneratedValue
```

```
        private Long id;
        @Column(name = "FIRST_NAME")
        private String firstName;
        @Column(name = "LAST_NAME")
        private String lastName;
        ...
```

The two ways are more or less equal, the only difference they have plays a role when you want to override annotations for fields in subclasses. As we will see in the further course of this tutorial, it is possible to extend an existing entity in order to inherit its fields. When we place the JPA annotations at the field level, we cannot override them as we can by overriding the corresponding getter method.

One also has to pay attention to keep the way to annotate entities the same for one entity hierarchy. You can mix annotation of fields and methods within one JPA project, but within one entity and all its subclasses is must be consistent. If you need to change the way of annotatio within a subclass hierarchy, you can use the JPA annotation `Access` to specify that a certain subclass uses a different way to annotate fields and methods:

```
@Entity
@Table(name = "T_GEEK")
@Access(AccessType.PROPERTY)
public class Geek extends Person {
...
```

The code snippet above tells JPA that this class is going to use annotations on the method level, whereas the superclass may have used annotations on field level.

When we run the code above, Hibernate will issue the following queries to our local H2 database:

```
Hibernate: drop table T_PERSON if exists
Hibernate: create table T_PERSON (id bigint generated by default as identity, FIRST_NAME  ↩
    varchar(255), LAST_NAME varchar(255), primary key (id))
Hibernate: insert into T_PERSON (id, FIRST_NAME, LAST_NAME) values (null, ?, ?)
```

As we can see, Hibernate first drops the table `T_PERSON` if it exists and re-creates it afterwards. If creates the table with two columns of the type `varchar(255)` (`FIRST_NAME`, `LAST_NAME`) and one column named `id` of type `bigint`. The latter column is defined as primary key and its values are automatically generated by the database when we insert a new value.

We can check that everything is correct by using the Shell that ships with H2. In order to use this Shell we just need the jar archive `h2-1.3.176.jar`:

```
>java -cp h2-1.3.176.jar org.h2.tools.Shell -url jdbc:h2:~/jpa

...

sql> select * from T_PERSON;
ID | FIRST_NAME | LAST_NAME
1  | Homer      | Simpson
(4 rows, 4 ms)
```

The query result above shows us that the table `T_PERSON` actually contains one row with id 1 and values for first name and last name.

Chapter 4

Inheritance

After having accomplished the setup and this easy use case, we turn towards some more complex use cases. Let's assume we want to store next to persons also information about geeks and about their favourite programming language. As geeks are also persons, we would model this in the Java world as subclass relation to Person:

```
@Entity
@Table(name = "T_GEEK")
public class Geek extends Person {
        private String favouriteProgrammingLanguage;
        private List<Project> projects = new ArrayList<Project>();

        @Column(name = "FAV_PROG_LANG")
        public String getFavouriteProgrammingLanguage() {
                        return favouriteProgrammingLanguage;
        }

        public void setFavouriteProgrammingLanguage(String favouriteProgrammingLanguage) {
                this.favouriteProgrammingLanguage = favouriteProgrammingLanguage;
        }
        ...
}
```

Adding the annotations `@Entity` and `@Table` to the class lets Hibernate create the new table `T_GEEK`:

```
Hibernate: create table T_PERSON (DTYPE varchar(31) not null, id bigint generated by  ↩
    default as identity, FIRST_NAME varchar(255), LAST_NAME varchar(255), FAV_PROG_LANG  ↩
    varchar(255), primary key (id))
```

We can see that Hibernate creates one table for both entities and puts the information whether we have stored a `Person` or a `Geek` within a new column named `DTYPE`. Let's persist some geeks in our database (for better readability I have omitted the block that catches any exception and rolls back the transaction):

```
private void persistGeek(EntityManager entityManager) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        Geek geek = new Geek();
        geek.setFirstName("Gavin");
        geek.setLastName("Coffee");
        geek.setFavouriteProgrammingLanguage("Java");
        entityManager.persist(geek);
        geek = new Geek();
        geek.setFirstName("Thomas");
        geek.setLastName("Micro");
        geek.setFavouriteProgrammingLanguage("C#");
        entityManager.persist(geek);
```

```
        geek = new Geek();
        geek.setFirstName("Christian");
        geek.setLastName("Cup");
        geek.setFavouriteProgrammingLanguage("Java");
        entityManager.persist(geek);
        transaction.commit();
}
```

After having executed this method, the database table `T_PERSON` contains the following rows (together with the person we have already inserted):

```
sql> select * from t_person;
DTYPE  | ID | FIRST_NAME | LAST_NAME | FAV_PROG_LANG
Person | 1  | Homer      | Simpson   | null
Geek   | 2  | Gavin      | Coffee    | Java
Geek   | 3  | Thomas     | Micro     | C#
Geek   | 4  | Christian  | Cup       | Java
```

As expected the new column `DTYPE` determines which type of person we have. The column `FAV_PROG_LANG` has the value `null` for persons that are no geeks.

If you do not like the name or type of the discriminator column, you can change it with the corresponding annotation. In the following we want that the column has the name `PERSON_TYPE` and is an integer column instead of a string column:

```
@DiscriminatorColumn(name="PERSON_TYPE", discriminatorType = DiscriminatorType.INTEGER)
```

This yields to the following result:

```
sql> select * from t_person;
PERSON_TYPE | ID | FIRST_NAME | LAST_NAME | FAV_PROG_LANG
-1907849355 | 1  | Homer      | Simpson   | null
2215460     | 2  | Gavin      | Coffee    | Java
2215460     | 3  | Thomas     | Micro     | C#
2215460     | 4  | Christian  | Cup       | Java
```

Not in every situation you want to have one table for all different types you want to store in your database. This is especially the case when the different types do not have nearly all columns in common. Therefore JPA lets you specify how to lay out the different columns. These three options are available:

- `SINGLE_TABLE`: This strategy maps all classes to one single table. As a consequence that each row has all columns for all types the database needs additional storage for the empty columns. On the other hand this strategy brings the advantage that a query never has to use a join and therefore can be much faster.
- `JOINED`: This strategy creates for each type a separate table. Each table therefore only contains the state of the mapped entity. To load one entity, the JPA provider has to load the data for one entity from all tables the entity is mapped to. This approach reduces storage space but on the other hand introduces join queries which can decrease query speed significantly.
- `TABLE_PER_CLASS`: Like the `JOINED` strategy, this strategy creates a separate table for each entity type. But in contrast to the `JOINED` strategy these tables contain all information necessary to load this entity. Hence no join queries are necessary for loading the entities but it introduces in situations where the concrete subclass is not known additional SQL quries in order to determine it.

To change our implementation to use the `JOINED` strategy, all we have to do is to add the following annotation to the base class:

```
@Inheritance(strategy = InheritanceType.JOINED)
```

Now Hibernate creates two tables for persons and geeks:

```
Hibernate: create table T_GEEK (FAV_PROG_LANG varchar(255), id bigint not null, primary key ↩
     (id))
Hibernate: create table T_PERSON (id bigint generated by default as identity, FIRST_NAME  ↩
    varchar(255), LAST_NAME varchar(255), primary key (id))
```

Having added the person and the geeks we get the following result:

```
sql> select * from t_person;
ID | FIRST_NAME | LAST_NAME
1  | Homer      | Simpson
2  | Gavin      | Coffee
3  | Thomas     | Micro
4  | Christian  | Cup
(4 rows, 12 ms)
sql> select * from t_geek;
FAV_PROG_LANG | ID
Java          | 2
C#            | 3
Java          | 4
(3 rows, 7 ms)
```

As expected the data is distributed over the two tables. The base table `T_PERSON` contains all the common attributes whereas the table `T_GEEK` only contains rows for each geek. Each row references a person by the value of the column `ID`.

When we issue a query for persons, the following SQL is send to the database:

```
select
        person0_.id as id1_2_,
        person0_.FIRST_NAME as FIRST_NA2_2_,
        person0_.LAST_NAME as LAST_NAM3_2_,
        person0_1_.FAV_PROG_LANG as FAV_PROG1_1_,
        case
                when person0_1_.id is not null then 1
                when person0_.id is not null then 0
        end as clazz_
from
        T_PERSON person0_
left outer join
        T_GEEK person0_1_
                on person0_.id=person0_1_.id
```

We see that a join query is necessary to include the data from the table `T_GEEK` and that Hibernate encodes the information if one row is a geek or by returning an integer (see case statement).

The Java code to issue such a query looks like the following:

```
TypedQuery<Person> query = entityManager.createQuery("from Person", Person.class);
List<Person> resultList = query.getResultList();
for (Person person : resultList) {
        StringBuilder sb = new StringBuilder();
        sb.append(person.getFirstName()).append(" ").append(person.getLastName());
        if (person instanceof Geek) {
                Geek geek = (Geek)person;
                sb.append(" ").append(geek.getFavouriteProgrammingLanguage());
        }
        LOGGER.info(sb.toString());
}
```

First of all we create a `Query` object by calling EntityManager's `createQuery()` method. The query clause can omit the select keyword. The second parameter helps to parameterize the method such that the `Query` is of type `Person`. Issuing the query is simply done by calling `query.getResultList()`. The returned list is iterable, hence we can just iterate over the `Person` objects. If we want to know whether we have a `Person` or a `Geek`, we can just use Java's `instanceof` operator.

Running the above code leads to the following output:

```
Homer Simpson
Gavin Coffee Java
Thomas Micro C#
Christian Cup Java
```

Chapter 5

Relationships

Until now we have not modelled any relations between different entities except the `extends` relation between a subclass and its superclass. JPA offers different relations between entities/tables that can be modelled:

- `OneToOne`: In this relationship each entity has exactly one reference to the other entity and vice versa.
- `OneToMany /ManyToOne`: In this relationship one entity can have multiple child entities and each child entity belongs to one parent entity.
- `ManyToMany`: In this relationship multiple entites of one type can have multiple references to entities from the other type.
- `Embedded`: In this relationship the other entity is stored in the same table as the parent entity (i.e. we have two entites for one table).
- `ElementCollection`: This relationship is similar to the `OneToMany` relation but in contrast to it the referenced entity is an `Embedded` entity. This allows to define `OneToMany` relationships to simple objects that are stored in contrast to the "normal" `Embedded` relationship in another table.

5.1 OneToOne

Let's start with an `OneToOne` relationship by adding a new entity `IdCard`:

```
@Entity
@Table(name = "T_ID_CARD")
public class IdCard {
        private Long id;
        private String idNumber;
        private Date issueDate;

        @Id
        @GeneratedValue
        public Long getId() {
                return id;
        }

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

        @Column(name = "ID_NUMBER")
        public String getIdNumber() {
                return idNumber;
        }
```

```
        public void setIdNumber(String idNumber) {
                this.idNumber = idNumber;
        }

        @Column(name = "ISSUE_DATE")
        @Temporal(TemporalType.TIMESTAMP)
        public Date getIssueDate() {
                return issueDate;
        }

        public void setIssueDate(Date issueDate) {
                this.issueDate = issueDate;
        }
}
```

Note that we have used a common `java.util.Date` to model the issue date of the ID card. We can use the annotation `@Temporal` to tell JPA how we want the `Date` to be serialized to the database. Depending on the underlying database product this column is mapped to an appropriate date/timestamp type. Possible values for this annotation are next to `TIMESTAMP`: `TIME` and `DATE`.

We tell JPA that each person has exactly one ID card:

```
@Entity
@Table(name = "T_PERSON")
public class Person {
        ...
        private IdCard idCard;
        ...

        @OneToOne
        @JoinColumn(name = "ID_CARD_ID")
        public IdCard getIdCard() {
                return idCard;
        }
```

The column in the table `T_PERSON` that contains the foreign key to the table `T_ID_CARD` is stored in the additional column `ID_CARD_ID`. Now Hibernate generates these two tables in the following way:

```
create table T_ID_CARD (
        id bigint generated by default as identity,
        ID_NUMBER varchar(255),
        ISSUE_DATE timestamp,
        primary key (id)
)

create table T_PERSON (
        id bigint generated by default as identity,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        ID_CARD_ID bigint,
        primary key (id)
)
```

An important fact is that we can configure when the ID card entity should be loaded. Therefore we can add the attribute `fetch` to the `@OneToOne` annotation:

```
@OneToOne(fetch = FetchType.EAGER)
```

The value `FetchType.EAGER` is the default value and specifies that each time we load a person we also want to load the ID card. On the other hand we can specify that we only want to load the ID when we actually access it by calling `person.getIdCard()`:

```
@OneToOne(fetch = FetchType.LAZY)
```

This result in the following SQL statements when loading all persons:

```
Hibernate:
    select
        person0_.id as id1_3_,
        person0_.FIRST_NAME as FIRST_NA2_3_,
        person0_.ID_CARD_ID as ID_CARD_4_3_,
        person0_.LAST_NAME as LAST_NAM3_3_,
        person0_1_.FAV_PROG_LANG as FAV_PROG1_1_,
        case
            when person0_1_.id is not null then 1
            when person0_.id is not null then 0
        end as clazz_
    from
        T_PERSON person0_
    left outer join
        T_GEEK person0_1_
            on person0_.id=person0_1_.id
Hibernate:
    select
        idcard0_.id as id1_2_0_,
        idcard0_.ID_NUMBER as ID_NUMBE2_2_0_,
        idcard0_.ISSUE_DATE as ISSUE_DA3_2_0_
    from
        T_ID_CARD idcard0_
    where
        idcard0_.id=?
```

We can see that we now have to load each ID card separately. Therefore this feature has to be used wisely, as it can cause hundreds of additional select queries in case you are loading a huge number of persons and you know that you are loading each time also the ID card.

5.2 OneToMany

Another important relationship is the `@OneToMany` relationship. In our example every `Person` should have one ore more phones:

```
@Entity
@Table(name = "T_PHONE")
public class Phone {
        private Long id;
        private String number;
        private Person person;

        @Id
        @GeneratedValue
        public Long getId() {
                return id;
        }

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

        @Column(name = "NUMBER")
        public String getNumber() {
                return number;
```

```
        }

        public void setNumber(String number) {
                this.number = number;
        }

        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "PERSON_ID")
        public Person getPerson() {
                return person;
        }

        public void setPerson(Person person) {
                this.person = person;
        }
}
```

Each phone has an internal id as well as the number. Next to this we also have to specify the relation to the `Person` with the `@ManyToOne`, as we have "many" phones for "one" person. The annotation `@JoinColumn` specifies the column in the `T_PHONE` table that stores the foreign key to the person.

On the other side of the relation we have to add a `List` of `Phone` objects to the person and annotate the corresponding getter method with `@OneToMany` as we have "one" person with "many" phones:

```
private List<Phone> phones = new ArrayList<>();
...
@OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
public List<Phone> getPhones() {
        return phones;
}
```

The value of the attribute `mappedBy` tells JPA which list on the other side of the relation (here `Phone.person`) this annotation references.

As we do not want to load all phones every time we load the person, we set the relationship to be fetched lazy (although this is the default value and we would have to set it explicitly). Now we get an additional selecte statment each time we load the phones for one person:

```
select
        phones0_.PERSON_ID as PERSON_I3_3_0_,
        phones0_.id as id1_4_0_,
        phones0_.id as id1_4_1_,
        phones0_.NUMBER as NUMBER2_4_1_,
        phones0_.PERSON_ID as PERSON_I3_4_1_
from
        T_PHONE phones0_
where
        phones0_.PERSON_ID=?
```

As the value for the attribute `fetch` is set at compile time, we unfortunately cannot change it at runtime. But if we know that we want to load all phone numbers in this use case and in other use cases not, we can leave the relation to be loaded lazy and add the clause `left join fetch` to our JPQL query in order to tell the JPA provider to also load all phones in this specific query even if the relation is set to `FetchType.LAZY`. Such a query can look like the following one:

```
TypedQuery<Person> query = entityManager.createQuery("from Person p left join fetch p. ↩
    phones", Person.class);
```

We give the `Person` the alias `p` and tell JPA to also fetch all instances of `phones` that belong to each person. This results with Hibernate in the following select query:

```
    select
        person0_.id as id1_3_0_,
```

```
        phones1_.id as id1_4_1_,
        person0_.FIRST_NAME as FIRST_NA2_3_0_,
        person0_.ID_CARD_ID as ID_CARD_4_3_0_,
        person0_.LAST_NAME as LAST_NAM3_3_0_,
        person0_1_.FAV_PROG_LANG as FAV_PROG1_1_0_,
        case
            when person0_1_.id is not null then 1
            when person0_.id is not null then 0
        end as clazz_0_,
        phones1_.NUMBER as NUMBER2_4_1_,
        phones1_.PERSON_ID as PERSON_I3_4_1_,
        phones1_.PERSON_ID as PERSON_I3_3_0__,
        phones1_.id as id1_4_0__
    from
        T_PERSON person0_
    left outer join
        T_GEEK person0_1_
            on person0_.id=person0_1_.id
    left outer join
        T_PHONE phones1_
            on person0_.id=phones1_.PERSON_ID
```

Please note that without the keyword `left` (i.e. only `join fetch`) Hibernate will create an inner join and only load persons that actually have at least one phone number.

5.3 ManyToMany

Another interesting relationship is the `@ManyToMany` one. As one geek can join many projects and one project consists of many geeks, we model the relationship between `Project` and `Geek` as `@ManyToMany` relationship:

```
@Entity
@Table(name = "T_PROJECT")
public class Project {
        private Long id;
        private String title;
        private List<Geek> geeks = new ArrayList<Geek>();

        @Id
        @GeneratedValue
        public Long getId() {
                return id;
        }

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

        @Column(name = "TITLE")
        public String getTitle() {
                return title;
        }

        public void setTitle(String title) {
                this.title = title;
        }

        @ManyToMany(mappedBy="projects")
        public List<Geek> getGeeks() {
                return geeks;
        }
```

```
        public void setGeeks(List<Geek> geeks) {
                this.geeks = geeks;
        }
}
```

Our project has an internal id, a title of type `String` and a list of `Geeks`. The getter method for the geeks attribute is annotated with `@ManyToMany(mappedBy="projects")`. The value of the attribute `mappedBy` tells JPA the class member on the other side of the relation this relation belongs to as there can be more than one list of projects for a geek. The class `Geek` gets a list of projects and :

```
private List<Project> projects = new ArrayList<>();
...
@ManyToMany
@JoinTable(
                name="T_GEEK_PROJECT",
                joinColumns={@JoinColumn(name="GEEK_ID", referencedColumnName="ID")},
                inverseJoinColumns={@JoinColumn(name="PROJECT_ID", referencedColumnName="ID ↩
                    ")})
public List<Project> getProjects() {
        return projects;
}
```

For a `@ManyToMany` we need an additional table. This table is configured by the `@JoinTable` annotation that describes the table used to store the geek's assignments to the different projects. It has the name `GEEK_PROJECT` and stores the geek's id in the column `GEEK_ID` and the project's id in the column `PROJECT_ID`. The referenced column is on both sides just `ID` as we have named the internal id in both classes `ID`.

A `@ManyToMany` is also per default fetched lazy, as in most cases we do not want to load all projects assignments when we load a single geek.

As the relation `@ManyToMany` is equal on both sides, we could have also annotated the two lists in both classes the other way round:

```
@ManyToMany
@JoinTable(
                name="T_GEEK_PROJECT",
                joinColumns={@JoinColumn(name="PROJECT_ID", referencedColumnName="ID")},
                inverseJoinColumns={@JoinColumn(name="GEEK_ID", referencedColumnName="ID") ↩
                    })
public List<Geek> getGeeks() {
        return geeks;
}
```

And on the other `Geek` side:

```
@ManyToMany(mappedBy="geeks")
public List<Project> getProjects() {
        return projects;
}
```

In both cases Hibernate creates a new table `T_GEEK_PROJECT` with the two colums `PROJECT_ID` and `GEEK_ID`:

```
sql> select * from t_geek_project;
PROJECT_ID | GEEK_ID
1          | 2
1          | 4
(2 rows, 2 ms)
```

The Java code to persist these relations is the following one:

```
List<Geek> resultList = entityManager.createQuery("from Geek g where g. ↩
    favouriteProgrammingLanguage = :fpl", Geek.class).setParameter("fpl", "Java"). ↩
    getResultList();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();
Project project = new Project();
project.setTitle("Java Project");
for (Geek geek : resultList) {
        project.getGeeks().add(geek);
        geek.getProjects().add(project);
}
entityManager.persist(project);
transaction.commit();
```

In this example we only want to add geeks to our "Java Project" whose favourite programming language is of course Java. Hence we add a where clause to our select query that restricts the result set to geeks with a specific value for the column `FAV_PROG_LANG`. As this column is mapped to the field `favouriteProgrammingLanguage`, we can reference it directly by its Java field name in the JPQL statement. The dynamic value for the query is passed into the statement by calling `setParameter()` for the corresponding variable in the JPQL query (here: `fpl`).

5.4 Embedded / ElementCollection

It can happen that you want to structure your Java model more fine-grained than your database model. An example for such a use case is the Java class `Period` that models the time between a start and an end date. This construct can be reused in different entities as you do not want to copy the two class fields `startDate` and `endDate` to each entity that has a period of time.

For such cases JPA offers the ability to model embedded entities. These entities are modeled as separate Java classes with the annotation `@Embeddable`:

```
@Embeddable
public class Period {
        private Date startDate;
        private Date endDate;

        @Column(name ="START_DATE")
        public Date getStartDate() {
                return startDate;
        }

        public void setStartDate(Date startDate) {
                this.startDate = startDate;
        }

        @Column(name ="END_DATE")
        public Date getEndDate() {
                return endDate;
        }

        public void setEndDate(Date endDate) {
                this.endDate = endDate;
        }
}
```

This entity can then be included in our `Project` entity:

```
private Period projectPeriod;

@Embedded
public Period getProjectPeriod() {
```

```
        return projectPeriod;
}

public void setProjectPeriod(Period projectPeriod) {
        this.projectPeriod = projectPeriod;
}
```

As this entity is embedded, Hibernate creates the two columns `START_DATE` and `END_DATE` for the table `T_PROJECT`:

```
create table T_PROJECT (
        id bigint generated by default as identity,
        END_DATE timestamp,
        START_DATE timestamp,
        projectType varchar(255),
        TITLE varchar(255),
        primary key (id)
)
```

Altough these two values are modelled within a separate Java class, we can query them as part of the project:

```
sql> select * from t_project;
ID | END_DATE                | START_DATE              | PROJECTTYPE       | TITLE
1  | 2015-02-01 00:00:00.000 | 2016-01-31 23:59:59.999 | TIME_AND_MATERIAL | Java Project
(1 row, 2 ms)
```

A JPQL query has to reference the embedded period in order to formulate a condition that restricts the result set to projects that started on a certain day:

```
entityManager.createQuery("from Project p where p.projectPeriod.startDate = :startDate",  ↩
    Project.class).setParameter("startDate", createDate(1, 1, 2015));
```

This yields to the following SQL query:

```
select
        project0_.id as id1_5_,
        project0_.END_DATE as END_DATE2_5_,
        project0_.START_DATE as START_DA3_5_,
        project0_.projectType as projectT4_5_,
        project0_.TITLE as TITLE5_5_
from
        T_PROJECT project0_
where
        project0_.START_DATE=?
```

Since version 2.0 of JPA you can even use `@Embeddable` entities in one-to-many relations. This is accomplished by using the new annotations `@ElementCollection` and `@CollectionTable` as shown in the following example for the class `Project`:

```
private List<Period> billingPeriods = new ArrayList<Period>();

@ElementCollection
@CollectionTable(
                name="T_BILLING_PERIOD",
                joinColumns=@JoinColumn(name="PROJECT_ID")
)
public List<Period> getBillingPeriods() {
        return billingPeriods;
}

public void setBillingPeriods(List<Period> billingPeriods) {
        this.billingPeriods = billingPeriods;
}
```

As `Period` is an `@Embeddable` entity we cannot just use a normal `@OneToMany` relation.

Chapter 6

Data Types and Converters

When dealing with legacy databases it can happen that you the standard mapping provided by JPA may not be enough. The following table lists how the Java types are mapped to the different database types:

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

An interesting point in this table is the mapping for enum types. To demonstrate the usage of enums in JPA let's add the enum `ProjectType` to the entity:

```
@Entity
@Table(name = "T_PROJECT")
public class Project {
...
        private ProjectType projectType;

        public enum ProjectType {
                FIXED, TIME_AND_MATERIAL
        }
        ...
        @Enumerated(EnumType.ORDINAL)
        public ProjectType getProjectType() {
                return projectType;
        }

        public void setProjectType(ProjectType projectType) {
                this.projectType = projectType;
        }
}
```

As we can see from the snippet above, the annotation `@Enumerated` allows us to map enums to database colums by specifying how to map the different values to the column. Choosing `EnumType.ORDINAL` means that each enum constant is mapped to a specific number in the database. When we set our "Java Project" to `TIME_AND_MATERIAL` we get the following output:

```
sql> select * from t_project;
ID | PROJECTTYPE | TITLE
1  | 1           | Java Project
(1 row, 2 ms)
```

As an alternative we could also use the value `EnumType.STRING`. In this case the column is of type `String` and encodes the enum by calling its `name()` method:

```
sql> select * from t_project;
ID | PROJECTTYPE       | TITLE
1  | TIME_AND_MATERIAL | Java Project
(1 row, 2 ms)
```

If both solutions do not satisfy your requirements, you can write your own converter. This is done by implementing the Java interface `AttributeConverter` and annotating the class with `@Converter`. The following class for example converts a boolean value into the two numeric values `1` and `-1`:

```
@Converter
public class BooleanConverter implements AttributeConverter<Boolean, Integer> {

        @Override
        public Integer convertToDatabaseColumn(Boolean aBoolean) {
                if (Boolean.TRUE.equals(aBoolean)) {
                        return 1;
                } else {
                        return -1;
                }
        }

        @Override
        public Boolean convertToEntityAttribute(Integer value) {
                if (value == null) {
                        return Boolean.FALSE;
                } else {
                        if (value == 1) {
                                return Boolean.TRUE;
                        } else {
                                return Boolean.FALSE;
                        }
                }
        }
}
```

This converter can be applied to our `IdCard` when we want provide the information if an ID card is valid or not as a boolean value:

```
private boolean valid;
...
@Column(name = "VALID")
@Convert(converter = BooleanConverter.class)
public boolean isValid() {
        return valid;
}

public void setValid(boolean valid) {
        this.valid = valid;
}
```

Inserting an ID card with `false` for the attribute `valid`, results in the following output:

```
sql> select * from t_id_card;
ID | ID_NUMBER | ISSUE_DATE              | VALID
1  | 4711      | 2015-02-04 16:43:30.233 | -1
```

Chapter 7

Criteria API

Until now we have used the Java Persistence Query Language (JPQL) to issue queries to the database. An alternative to the JPQL is the "Criteria API". This API provides a pure Java method based API to construct a query.

The following example queries the database for persons with `firstName ="Homer"`:

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Person> query = builder.createQuery(Person.class);
Root<Person> personRoot = query.from(Person.class);
query.where(builder.equal(personRoot.get("firstName"), "Homer"));
List<Person> resultList = entityManager.createQuery(query).getResultList();
```

When starting to build a criteria query, we have to ask the `EntityManager` for a `CriteriaBuilder` object. This builder can then be used to create the actual `Query` object. The way to tell this query which table(s) to query is accomplished by invoking the method `from()` and passing the entity that is mapped to the corresponding table. The `Query` object also offers a method to add the where clause:

```
query.where(builder.equal(personRoot.get("firstName"), "Homer"));
```

The condition itself is then created using the `CriteriaBuilder` and its `equal()` method. More complex queries can then be assembled by using the appropriate logical conjunction:

```
query.where(builder.and(
        builder.equal(personRoot.get("firstName"), "Homer"),
        builder.equal(personRoot.get("lastName"), "Simpson")));
```

In general `CriteriaQuery` defines the following clauses and options:

- `distinct()`: Specifies whether the database should filter out duplicate values.
- `from()`: Specifies the table/entity the query is submitted for.
- `select()`: Specifies a `select` query.
- `multiselect()`: Specifies a list of selections.
- `where()`: Specifies the `where` clause of the query.
- `orderBy()`: Specifies the ordering for the query.
- `groupBy()`: Specifies groups that are formed over the result.
- `having()`: Specifies restrictions for groups that are defined over the result.
- `subquery()`: Specifies a subquery that can be used in other queries.

The above methods allow you to assemble queries completely dynamically based on any filter restrictions the user has provided.

Chapter 8

Sequences

Until now we have used in this tutorial the annotation `@GeneratedValue` without any specific information on how this unique value should be assigned to each entity. Without any further information the JPA provider chooses on its on how to generate this unique value. But we can also decide the way how to generate unique ids for entities on our own. JPA provides therefore these three different approaches:

- `TABLE`: This strategy lets the JPA provider create a separate table that contains one row for each entity. This row contains next to the name of the entity also the current value for the id. Each time a new value is requsted, the row in the table is updated accordingly.
- `SEQUENCE`: If the database provides sequences, this strategy requests the unique values from the provided database sequence. Not all database products do support sequences.
- `IDENTITY`: If the database provides identity columns, this strategy uses this kind of column provided by the underlying database implementation. Not all database products support identity columns.

In order to use the `TABLE` strategy, we also have to tell the JPA provider some details about the table it should use for the sequence management:

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "TABLE_GENERATOR")
@TableGenerator(name = "TABLE_GENERATOR", table="T_SEQUENCES", pkColumnName = "SEQ_NAME",  ↩
    valueColumnName = "SEQ_VALUE", pkColumnValue = "PHONE")
public Long getId() {
        return id;
}
```

The `@TableGenerator` annotation tell our JPA provider that the table should have the name `T_SEQUENCES` and the two columns `SEQ_NAME` and `SEQ_VALUE`. The name for this sequence in the table should be `PHONE`:

```
sql> select * from t_sequences;
SEQ_NAME | SEQ_VALUE
PHONE    | 1
```

The `SEQUENCE` strategy can be used in a similar way:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "S_PROJECT")
@SequenceGenerator(name = "S_PROJECT", sequenceName = "S_PROJECT", allocationSize = 100)
public Long getId() {
        return id;
}
```

By using the annotation `@SequenceGenerator` we tell the JPA provider how the sequence should be named (`S_PROJECT`) and tell it an allocation size (here 100), i.e. how many values should be pre-allocated. The attributes `generator` and `name` connect the two annotations with each other.

As this strategy uses a separate table, it can easily become a performance bottleneck when requesting a lot of sequence values. This is especially true if you use the same table for a huge number of tables and the underlying database only supports table locks or locks on table pages. In this case the database has to lock the complete table/page until the current transaction has been committed. Therefore JPA allows to define a pre-allocation size such that the database is not hit too often.

In order to use the `IDENTITY` strategy, we just have to set the `strategy` attribute accordingly:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Long getId() {
        return id;
}
```

If the database supports identity columns, the table is created appropriately:

```
create table T_ID_CARD (
        id bigint generated by default as identity,
        ID_NUMBER varchar(255),
        ISSUE_DATE timestamp,
        VALID integer,
        primary key (id)
)
```

Chapter 9

Download

You can download the full source code of this tutorial here: **jpa_tutorial**