Comparative study of ORM tools

Benedetta Pacilli

Academic Year: 2022/2023

Contents

1 Introduction to ORM tools

ORM stands for Object-Relational Mapping and refers to the computer science technique of creating a layer between an Object-Oriented language and a Database. With ORM tools, programmers can easily connect classes with their respective tables in the database without writing SQL queries.
There are many benefits to this, and the following are only a few of them:

- the overcoming of inconsistency between the object-oriented programming and the relational model;
- a significant reduction of the developing time, leading also to a cost lowering;
- a one layer-isolation of the logic regarding data persistence.

There have been many of these tools created over the years, and today there is a wide selection available. The following section aims to cite and briefly describe the currently most popular and used ORM tools for Java, .NET, Python, and PHP.

2 Overview of some ORM tools

2.1 Java

In 2018 snyk put out research on Java Ecosystems, taking also into consideration ORM frameworks. From this study emerged that Hibernate is used by 54% of Java developers who need to interact with relational databases.
Another analysis can be done using Google Trends by comparing the trends of different Java ORM tools over time. The trends here reported are the ones of Hibernate, Apache OpenJPA, EclipseLink and TopLink.

2.1.1 Hibernate

It is an open-source tool by Red Hat Software that provides an ORM framework for the Java language. It was created in 2001 and first released in 2002 by Gavin King as an alternative to JDBC (Java DataBase Connectivity API) which was too slow and complex to use.
As Hibernate was rapidly gaining popularity, its ideas were adopted and standardized into the first JPA specification in 2006. Thus, Hibernate is the standard implementation of JPA and it guarantees many benefits to those who decide to make use of it. It is lightweight, flexible, powerful, and has a large community.

2.1.2 Non-JPA tools

JPA is not the only solution when it comes to connecting the code to the database in Java. There have been numerous frameworks developed over the years that do not implement JPA.
Here are reported only some of the main competitors of JPA:

- **MyBatis:**
  It is actually not an ORM framework since that, instead of Java Objects to database tables, it maps Java methods to SQL statements and vice versa. It has many interesting features such as support for dynamic SQL and inline SQL; it also provides encapsulation of SQL in the form of stored procedures so to separate the business logic from the database. Moreover, there are two particularly interesting features of MyBatis: its support for many of the same features as an ORM tool and its portability, in fact, MyBatis is a fork of **iBATIS**, a persistence

framework available not only for Java but also for .NET and Ruby on Rails.

- **jOOQ:**
  jOOQ is a database-mapping software library that uses JDBC and that, contrary to other technologies like Hibernate, takes a database-first, or SQL-centric, approach. This means that before using jOOQ code generator for the classes, the database needs to exist already.
  The main feature of jOOQ is that one can have full control over the queries, as they can be visualized and written directly into the code, while at the same time providing type safety. As for functionality, it does not provide as much as standard object-relational mapping technologies.

2.2 .NET

Among all the ORM technologies developed for the .NET platform the most popular ones are probably Entity Framework (along with the Entity Framework Core version) and NHibernate; as the analysis here reports.

2.2.1 Entity Framework and Entity Framework Core

Entity Framework or EF is an open-source ORM framework for ADO.NET, the data access technology from Microsoft .NET.
EF is the younger sibling of a previous ORM technology, also created by Microsoft, called LINQ to SQL which is no longer under development, according to Microsoft Documentation.
EF Core is the modern, lightweight, and extensible version of Entity Framework and it is also the only one of the two under active development. Although it is still currently lacking some of the most advanced mapping functionalities of EF, EF Core brings to the table some brand new features, never been seen in the past EF versions. EF Core was introduced during the EF 6.x versions period and the latest released version is EF Core 7, EF7 shortly. Thus, it appears that Entity Framework Core is the future of Entity Framework and it is going to fully replace EF.

2.2.2 NHibernate

NHibernate, by Red Hat Software, is the main competitor of Microsoft concerning ORM tools for the .NET platform and is the sibling of Hibernate, for Java.
Many differences between NHibernate and EF will be addressed and deepened in the next chapter.

2.3 Python

For some years now, jetbrains has submitted a survey to python developers regarding the python technologies that they use. In the 2021 report of this analysis, under the heading "ORMs", it is stated that the two most used ORM technologies for Python are: SQLAlchemy and Django ORM

2.3.1 SQLAlchemy

SQLAlchemy is an open-source SQL toolkit and ORM for Python.
The fact that it has been actively developed since 2006, its large community, and its simplicity to implement make it the ORM of choice for working with relational databases in Python.

2.3.2 Django ORM

Django is an open-source web framework that comes with its own built-in ORM module, the Django ORM.
Django ORM comes with excellent documentation and a large community and it leaves little for the developer to worry about, to the extent that it is not required for the developer to know SQL. Despite its numerous advantages, Django ORM is monolithic, it lacks some of the features that other ORMs have and its use is not advised for small projects.

2.3.3 Peewee

Peewee is a simple and small open-source ORM. Peewee's philosophy is to be as easy to learn and intuitive to use as possible and at the same time provide few but expressive concepts.

2.4 PHP

PHP is a general-purpose programming language that reached the peak of success in web development, being used in more than 70% of the existing websites.
Its reputation led to the creation of several frameworks and ORM tools dedicated to PHP. The following cited are the most known and used ones.

2.4.1 Laravel Eloquent

Laravel, by Taylor Otwell, is a PHP framework that in the latest years has had massive growth and, as reported in the jetbrains study of 2021, it is extremely popular.
Laravel comes by default with an ORM, the Eloquent ORM.
Eloquent makes it enjoyable to interface with the database thanks to its simplicity in syntax and usage and the wide range of features it provides. However, implementation of the Active Record pattern leads to some performance issues which will be later discussed in the next chapters.

2.4.2 Doctrine

Doctrine provides various sets of libraries for different purposes but is well-known for its ORM services.
Although Doctrine can be used on its own, it is often used together with a popular PHP framework: Symfony.

2.4.3 CakePHP ORM

CakePHP is a PHP framework that comes with its ORM.
CakePHP ORM implements the DataMapper pattern which makes this ORM a powerful and flexible tool.
It is also the longest-lived among the other PHP frameworks listed, being under active development since 2005.

3 Installation and Configuration

The purpose of this section is to discuss and compare one by one, all the ORM tools introduced in the previous section, in terms of installation and configuration.

Premise:
In order to serve as a reference point, here is a list of the ORM versions used in this study.

- Hibernate: 6.1.5.Final
- MyBatis: 3.5.11
- Entity Framework Core: 7.0.0
- NHibernate: 5.4.0
- SQLAlchemy: 1.4.44
- peewee: 3.15.4
- Django: 4.1.4
- CakePHP: 4.4.7
- Doctrine: 2.14.1
- Laravel: 9.38.0

3.1 Hibernate

Depending on individual needs and preferences, Hibernate offers multiple options for installation and configuration. When installing, there are essentially two main paths that can be followed:

1. Downloading one of the releases from the official Hibernate website. In this case, it is important to ensure that all the libraries and packages required by Hibernate are installed as well. Hibernate and its dependencies have to be installed into the "CLASSPATH" variable, which needs to be set properly. Since Hibernate requires knowing in advance

where to find the mapping information along with other data regarding the database, there is a configuration file named hibernate.cfg.xml, which must be manually configured in the present case.

2. Downloading Hibernate through an IDE, such as Eclipse IDE or IntelliJ IDEA. By choosing this option you can benefit from many simplifications made available by the IDE; for example, the table-class mapping is automatically reported in the configuration file, in case you let the IDE create the classes for you.

In general, the configuration file reports the location of the classes that were mapped from the database, and the data necessary to establish the connection to the database.
For this study, I used Hibernate with IntelliJ IDEA while I used a persistence.xml file for the configuration file. The apparent inconsistency of files stems from which API is used. With Hibernate's proprietary API, the hibernate.cfg.xml file is required, while for Hibernate EntityManager (a JPA standard), the file requested is the persistence.xml. The two files are equivalent but opting for the latter solution allows the programmer to change the JPA implementation (Hibernate, EclipseLink, TopLink, etc...) freely.

3.2 MyBatis

Regarding MyBatis, its installation is a very straightforward process; all it takes is adding the mybatis-x.x.x.jar file to the classpath. If a build automation tool like Gradle or Maven is being used, it is sufficient to add MyBatis as a dependency to the tool's configuration file. For further information on this matter, this page of the documentation of MyBatis reports which dependency to use for some of the most popular Java build tools.
About the configuration, MyBatis has its XML file (in the documentation the file is named "mybatis-config.xml" but other names are permitted as well) where the following information is given: the location of the mapper files and the environments. MyBatis can be configured with multiple environments. Each environment has its configuration and can be used with a different database, in this way the same mappers can be applied to different databases. The information contained in one environment is its name and ID, the parameters needed to connect to the database, what kind of connection is requested(POOLED, UNPOOLED, or JNDI), and the transaction manager (MyBatis supports two transaction managers, JDBC and MANAGED).

The following example shows how a MySQL database's environment can be configured:

```
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver"
            value="com.mysql.cj.jdbc.Driver"/>
            <property name="url"
            value="jdbc:mysql://host:port/dbname"/>
            <property name="username" value="username"/>
            <property name="password" value="password"/>
        </dataSource>
    </environment>
</environments>
```

The mappers can be created in two ways:

1. Each mapper is an XML file. Each of these files contains various tags, for instance: <insert>, <select>, <update>, and <delete> which specify a mapped SQL statement. Here is an example of a select SQL query made with the <select> tag:

   ```
   <select id = "selectAllCategories" resultType = "Category">
       SELECT * FROM category
   </select>
   ```

   Other additional tags are well described in here.

2. Each mapper is a Java interface. These interfaces are never implemented, they just exist as they are. Each of them contains the signature of particular methods, usually named "selectEntity", "selectEntities" or "selectAll", "insertEntity", "updateEntity" and "deleteEntity". Above each of the signatures, for the mapping to work, there is the corresponding query, passed to the proper annotation (retrieved via org.apache.ibatis.annotations).
   The following example is the "interface method" equivalent to the "XML file" one.

```
@Select("select * from categories")
List<Category> selectCategories();
```

In both cases, it is necessary to create by hand the classes which are mirrored by the database tables.
For this study, I used the second method.

3.3 Entity Framework Core

Moving on to C#, with EF Core the first thing to do is to create the Visual Studio project and install the appropriate NuGet packages. Several Entity Framework Core packages are available, each providing different features and tools, suitable for various needs. For example, the *Microsoft.EntityFrameworkCore.SqlServer* package provides a Microsoft SQL server. After installing the desired packages one can connect to the database and create the model classes mapping the database tables.

3.4 NHibernate

As opposed to EF Core, NHibernate requires some manual configuration. In fact, after downloading the NHibernate package with NuGet, it is necessary to create an XML file named "hibernate.cfg.xml" which results very similar to the Hibernate configuration file. The file must contain all the information required to establish a connection to the database as well as the locations of the mapper files.

3.5 SQLAlchemy, peewee, and Django

Regarding Python, the ORM tools considered for this study do not require special attention to their installation and configuration. To install any of them it is sufficient to run a command on the terminal:

```
pip install sqlalchemy

pip install peewee

pip install django
```

respectively for SQLAlchemy, peewee and Django. For the first two, no configuration is required other than to establish the database connection and create the classes that mirror the database tables:

- With SQLAlchemy, a function named create_engine can be used to establish the connection to the database. This function accepts the URL of the connection, formed as follows:

  ```
  dialect[+driver]://user:password@host/dbname
  ```

  A few rules must be followed when creating mapping classes in SQLAlchemy, which are well described on this documentation page.

- With peewee, the connection information is passed to the constructor of a class. The class differs according to which database type one is using. For example, a connection to a MySQL database would look like this:

  ```
  MySQLDatabase('dbname', host='host', port=portnumber,
  user='user', password='password')
  ```

  There is a lot of similarity between SQLAlchemy's mapping class creation rules and those of Peewee. On this Peewee documentation page, there is an in-depth description of model class creation.

In contrast, Django and its ORM require precise configuration, which is well described on this documentation page.

3.6 CakePHP, Doctrine and Laravel Eloquent

As for PHP, every ORM considered for this study requires Composer. Composer is an open-source dependency manager for PHP and it provides a standard format for managing dependencies and libraries.
Once Composer is installed, the "composer" command can be used to create the preferred PHP project.
For CakePHP, the following command:

```
composer create-project --prefer-dist cakephp/app:~4.0
projectname
```

will create a complete CakePHP project in a new directory called "projectname".
Moving on to Doctrine, the documentation suggests creating the "composer.json" file first and then creating the rest of the project by calling this command:

```
composer install
```

Lastly, for Laravel Eloquent, by running:

```
composer create-project --prefer-dist laravel/laravel
projectname
```

a complete Laravel project will be created in a directory named "projectname".
In terms of database connection data, both CakePHP and Laravel require this information to be stored in a specific file, automatically created by Composer.

- For CakePHP: From the 4.x.x version, this data is stored in the app_local.php file, instead of the app.php file, used in the previous versions. Here is an example:

```
'Datasources' => [
'default' => [
    'host' => 'host',
    'username' => 'username',
    'password' => 'password',
    'database' => 'dbname',
],
```

- For Laravel Eloquent: the configuration data for a database connection is found in the config/database.php configuration file. Inside this file, the connections defined can be as many as desired and it is possible to label which is the default connection. For example:

```
'connections' => [

    'mysql' => [
        'driver' => 'mysql',
        'host' => env('DB_HOST', 'host'),
        'port' => env('DB_PORT', 'port'),
        'database' => env('DB_DATABASE', 'dbname'),
        'username' => env('DB_USERNAME', 'username'),
        'password' => env('DB_PASSWORD', 'password'),
        ]) : [],
    ],
]
```

After establishing the connection it is important to create the class models, as described in the documentation.

Unlike CakePHP and Laravel, Doctrine allows the programmer to store connection data in any file. This ORM comes with the Doctrine Database Abstraction Layer (DBAL), an abstraction layer above PHP Data Objects (PDO), which makes it possible to communicate with relational databases. It is sufficient to use the Doctrine\DBAL\DriverManager class to create the connection.
For example, a connection to a MySQL database would look like the following code:

```
use Doctrine\DBAL\DriverManager;

$connectionParams = [
    'dbname' => 'dbname',
    'user' => 'user',
    'password' => 'password',
    'host' => 'host',
    'driver' => 'pdo_mysql',
];

DriverManager::getConnection($connectionParams);
```

After establishing the connection, model classes should be created, as described on this Doctrine documentation page.

4 Mapping automation

This chapter will discuss whether and how the ORM tools offer an automatic mapping from database tables to model classes and vice versa.
The examples presented in this chapter are based on Northwind database tables.

4.1 Database-first approach

A database-first approach is a methodology in which the database schema is designed and created before the application is built. Some ORMs considered offer automatic mapping from the database tables to the model classes, which proves useful when using this approach.

4.1.1 Hibernate

There are several ways to generate Hibernate model classes from an existing database:

- Hibernate Tools: a suite of tools that includes a code generator for creating Hibernate model classes from existing database tables.
- JPA Annotations: Hibernate supports JPA annotations to generate the model classes. Annotations such as @Entity, @Table, and @Column can be used to define the mapping between the database tables and Java classes. Hibernate will then generate the necessary code based on the annotations specified.
  Here is an example:

```
import javax.persistence.*;

@Entity
@Table(name="categories")
public class Category {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="CategoryId")
    private int id;
```

```
    @Column(name="CategoryName")
    private String name;

    @Column(name="Description")
    private String description;

    @Column(name="Picture")
    private String picture;

    // Getters and setters
}
```

@Entity indicates that it is a JPA entity.
@Table specifies the name of the database table that the class maps to.
@Id denotes that field as the primary key for the entity.
@GeneratedValue is used to specify that the primary key is automatically generated by the database.
@Column indicates which database table column that field refers to.

Hibernate will then use this information to generate the necessary SQL statements to create and manage the database table and to map the class to the table.

- JPA metamodel generation: the Java Persistence API (JPA) metamodel generator can create Hibernate model classes from existing database tables. The metamodel generator will create a Java representation of the database schema, including the entities, attributes, and relationships. A usage example follows.
  As a first step, it is necessary to add the JPA API and the Hibernate JPA Metamodel Generator dependencies to the project:
  Using Maven:

```
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>x.x.x</version>
</dependency>
```

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
    <version>x.x.x</version>
</dependency>
```

Using Gradle:

```
dependencies {
    compile group: 'javax.persistence',
    name: 'javax.persistence-api', version: 'x.x.x'

    compile group: 'org.hibernate',
    name: 'hibernate-jpamodelgen', version: 'x.x.x'
}
```

Afterward, in the *persistence.xml* file, one has to specify the hibernate.ejb.metamodel.generation property to enable JPA metamodel generation:

```
<persistence-unit name="persistence-unit-name"
                  transaction-type="RESOURCE_LOCAL">
<properties>
    <property name="hibernate.ejb.metamodel.generation"
    value="enabled"/>
</properties>
</persistence-unit>
```

Lastly, the JPA metamodel generation process has to be configured. When using Maven, the following should be added to the *pom.xml* file:

```
<build>
  <plugins>
    <plugin>
```

```
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>x.x.x</version>
      <configuration>
        <compilerArgs>
          <arg>-proc:only</arg>
        </compilerArgs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Using Gradle, the following should be added to the build file:

```
task generateJpaMetamodel(type: JavaCompile) {
  options.compilerArgs << "-proc:only"
  classpath = configurations.compile + sourceSets.main.output
  source = sourceSets.main.java
}
compileJava.dependsOn generateJpaMetamodel
```

- Reverse Engineering: some integrated development environments (IDEs), such as IntelliJ IDEA and Eclipse, have built-in tools for reverse engineering a database schema into Hibernate model classes.
  When using Hibernate to map a database schema through reverse engineering, the associations between tables are typically managed through the use of entity classes and annotations. Hibernate generates a set of entity classes, one for each table in the schema. These entity classes are typically annotated with Hibernate annotations that specify the relationships between the tables.
  @OneToMany annotation defines a one-to-many association between two tables and it is specified in the entity class that corresponds to the "one" side of the relationship.
  @ManyToOne annotation defines a one-to-many association between two tables and it is specified in the entity class that corresponds to the "many" side of the relationship.

@OneToOne annotation where each record in one table is associated with exactly one record in another table, and vice versa.
@ManyToMany annotation used when many records in one table are associated with many records in another table.
Here are some examples of annotations automatically created by Hibernate by reverse engineering the Northwind database:
The Orders class has several many-to-one relations. Many orders can be made by the same customer, many orders can be taken care of by the same employee and many orders can be shipped by the same shipper.

```
@ManyToOne
@JoinColumn(name = "CustomerID",
referencedColumnName = "CustomerID")
private Customers customersByCustomerId;

@ManyToOne
@JoinColumn(name = "EmployeeID",
referencedColumnName = "EmployeeID")
private Employees employeesByEmployeeId;

@ManyToOne
@JoinColumn(name = "ShipVia",
referencedColumnName = "ShipperID")
private Shippers shippersByShipVia;
```

Here are the corresponding one-to-many relations in the Customers, Employees and Shippers classes:

```
@OneToMany(mappedBy = "customersByCustomerId")
private Collection<Orders> ordersByCustomerId;

@OneToMany(mappedBy = "employeesByEmployeeId")
private Collection<Orders> ordersByEmployeeId;

@OneToMany(mappedBy = "shippersByShipVia")
private Collection<Orders> ordersByShipperId;
```

4.1.2 MyBatis

With MyBatis it is possible to use the MyBatis Generator to inspect database tables and generate the corresponding classes. MBG can be used in different ways:

- by downloading the latest jar and then running the following command inside the project:

```
java -jar mybatis-generator-core-x.x.x.jar
-configfile./path-to/generatorConfig.xml
-overwrite
```

- by using Maven and adding the dependency to the *pom.xml* file:

```
<dependency>
  <groupId>org.mybatis.generator</groupId>
  <artifactId>mybatis-generator-core</artifactId>
  <version>1.4.0</version>
</dependency>
```

  and then telling Maven to execute MBG:

```
mvn mybatis-generator:generate
```

- by creating a Java program like the following:

```
import org.mybatis.generator.*;

import java.io.File;
import java.util.ArrayList;
import java.util.List;

public class MybatisGenerator {
    public static void main(String[] args) throws Exception {
        Configuration config = cp.parseConfiguration(
```

```
            new File("./path-to/generatorConfig.xml"));
            MyBatisGenerator myBatisGenerator =
            new MyBatisGenerator(config);
            myBatisGenerator.generate(null);
        }
    }
```

In both cases, it is necessary to have an XML file configuration file for MBG. This file is usually named *generatorConfig.xml* and tells the generator how to connect to the database, what tables to generate models for, and how to create the models.

4.1.3 Entity Framework Core

EF Core allows a database-first approach. Reverse engineering can derive classes from the database, using the *Scaffold-DbContext* command in the packet manager console. This command creates entity and context classes from an existing database and has different parameters that can be specified:

```
Scaffold-DbContext [-Connection] [-Provider] [-OutputDir]
[-Context] [-Schemas>] [-Tables>] [-DataAnnotations]
[-Force] [-Project] [-StartupProject] [<CommonParameters>]
```

"Connection" is for a connection string, for example:

```
"Server=.\servername;Database=dbname
                    ;Trusted_Connection=True;"
```

"Provider" stands for the NuGet server package installed.
For example, when working with a MySQL database, the provider would be *Microsoft.EntityFrameworkCore.SqlServer*.
"OutputDir" specifies the directory where the generated classes will be placed. Along with the mapping classes, the DbContext class is also created. An instance of DbContext represents a session that can query the database and save data in it.

Entity Framework Core also provides a way to automatically generate entity classes based on an existing SQL Server database using a GUI in Visual Studio through ADO.NET Entity Data Model Tools. Once the entity classes

and DbContext have been generated, they can be used to perform CRUD operations on the database.

4.1.4 NHibernate

There are different ways to generate model classes from an existing database with NHibernate:

- Fluent NHibernate Auto Mapping: Fluent Hibernate provides an API for completely configuring NHibernate, one of its concepts is auto-mapping. Its purpose is to allow the developer to map the entire domain with little code and no XML.
  The documentation illustrates all the steps to configure and use this API.

- NHibernate Reverse Engineering: *Db2hbm* stands for DataBase To Hibernate Mapping and is a reverse engineering tool able to generate hibernate mapping files from a database schema. Presently, this tool is neither popular nor widely used. Its documentation can be found here.

4.1.5 SQLAlchemy

SQLAlchemy allows database-code automatic mapping and has, within its documentation, a page about the API that enables the auto-mapping.
To create a new model from an existing database, it is necessary to import the *automap_base* function from the *sqlalchemy.ext.automap* module. The *automap_base()* function creates and returns an AutomapBase class. The *prepare()* function is then called on this class, passing the previously set engine as the parameter.
An example follows:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

Base = automap_base()

engine = create_engine(
    'mysql://root:@localhost:3306/northwind',
```

```
        encoding="utf8",
        echo=False)

    Base.prepare(autoload_with=engine)
```

The mapped classes are sourced from the AutomapBase class and stored in apposite variables:

```
    Category = Base.classes.categories
    Product = Base.classes.products
    Customer = Base.classes.customers
    Order = Base.classes.orders
    Employee = Base.classes.employees
    ...
```

Accordingly, SQLAlchemy's automatic mapping does not create class files physically, unlike many other ORM tools that offer this type of mapping.

4.1.6 Peewee

Peewee does not support the automatic creation of classes.

4.1.7 Django ORM

As stated in the Django documentation, by reverse engineering existing database tables, it is possible to automatically generate classes.
After setting the parameters for the default database connection, it is possible to run the inspectdb command:

```
    python manage.py inspectdb > models.py
```

This will inspect the database tables and save the model classes in the *models.py* file.
Here is an example of an automatically created class by Django:

```
class Orders(models.Model):
    orderid = models.AutoField(db_column='OrderID',
    primary_key=True)  # Field name made lowercase.
    customerid = models.ForeignKey(Customers,
    models.DO_NOTHING, db_column='CustomerID',
```

```
    blank=True, null=True)  # Field name made lowercase.
    employeeid = models.ForeignKey(Employees,
    models.DO_NOTHING, db_column='EmployeeID', blank=True,
    null=True)  # Field name made lowercase.
    orderdate = models.DateTimeField(db_column='OrderDate',
    blank=True, null=True)  # Field name made lowercase.
    requireddate = models.DateTimeField(db_column='RequiredDate',
    blank=True, null=True)  # Field name made lowercase.
    shippeddate = models.DateTimeField(db_column='ShippedDate',
    blank=True, null=True)  # Field name made lowercase.
    shipvia = models.ForeignKey('Shippers'
    , models.DO_NOTHING, db_column='ShipVia',
    blank=True, null=True)  # Field name made lowercase.
    freight = models.DecimalField(db_column='Freight',
    max_digits=10, decimal_places=4, blank=True,
    null=True)  # Field name made lowercase.
    shipname = models.CharField(db_column='ShipName',
    max_length=40, blank=True, null=True)  # Field name made lowercase.
    shipaddress = models.CharField(db_column='ShipAddress',
    max_length=60, blank=True, null=True)  # Field name made lowercase.
    shipcity = models.CharField(db_column='ShipCity',
    max_length=15, blank=True, null=True)  # Field name made lowercase.
    shipregion = models.CharField(db_column='ShipRegion',
    max_length=15, blank=True, null=True)  # Field name made lowercase.
    shippostalcode = models.CharField(db_column='ShipPostalCode',
    max_length=10, blank=True, null=True)  # Field name made lowercase.
    shipcountry = models.CharField(db_column='ShipCountry',
    max_length=15, blank=True, null=True)  # Field name made lowercase.

    class Meta:
        managed = False
        db_table = 'orders'
```

Eventually, running the migrate command will synchronize the database tables and the respective model classes:

```
python manage.py migrate
```

4.1.8 CakePHP

Several commands are available in the CakePHP bake console for code generation. The list of all the bake commands can be retrieved by calling:

```
bin/cake bake --help
```

By running:

```
bin/cake bake model [db_table_name]
```

two files will automatically be generated in the *src/Model directory* unless a different location is specified with the following command option:

```
--dir
```

The files generated are:

- A model class file which contains the code for the model, including the database table it maps to, the fields, and the associations with other models.
- A table class file containing the code for the corresponding database table, including the schema definition and any custom methods for accessing and manipulating the table data.

Here is an example of the two files for the *customers* table of the Northwind database:

- Model class file:

```
declare(strict_types=1);

namespace App\Model\Entity;

use Cake\ORM\Entity;

/**
 * Category Entity
 *
 * @property int $CategoryID
 * @property string $CategoryName
```

```
 * @property string|null $Description
 * @property string|resource|null $Picture
 */
class Category extends Entity
{
    /**
     * Fields that can be mass assigned using newEntity()
     * or patchEntity().
     *
     * Note that when '*' is set to true, this allows
     * all unspecified fields to be mass assigned.
     * For security purposes, it is advised to set '*' to false
     * (or remove it), and explicitly make individual
     * fields accessible as needed.
     *
     * @var array<string, bool>
     */
    protected $_accessible = [
        'CategoryName' => true,
        'Description' => true,
        'Picture' => true,
    ];
}
```

The model class file is responsible for defining the business logic of the model, such as data validation, data manipulation, and relationships with other models. In this example, the Category model class extends the Entity class, which provides several methods and properties that can be used in entity classes, such as:
$_accessible: it controls which fields can be mass-assigned during data manipulation operations, such as save().
$_hidden: it controls which fields should be hidden when the entity is converted to an array or JSON format.
$_virtual: it defines virtual properties that can be accessed on the entity. Virtual properties are not stored in the database, but can be computed based on other properties or data.
isNew(): it returns true if the entity has not been saved to the database yet, or false if it has been saved before.

isDirty(): it returns true if any of the entity's properties have been modified since it was last saved to the database, or false if it has not been modified.
get(): it returns the value of a specific property on the entity.
set(): it sets the value of a specific property on the entity.

- Table class file:

```
declare(strict_types=1);

namespace App\Model\Table;

use Cake\ORM\Query;
use Cake\ORM\RulesChecker;
use Cake\ORM\Table;
use Cake\Validation\Validator;

/**
 * Categories Model
 *
 * @method \App\Model\Entity\Category newEmptyEntity()
 * @method \App\Model\Entity\Category newEntity(array $data,
 * array $options = [])
 * @method \App\Model\Entity\Category[] newEntities(
 * array $data, array $options = [])
 * @method \App\Model\Entity\Category get(
 * $primaryKey, $options = [])
 * @method \App\Model\Entity\Category findOrCreate(
 * $search, ?callable $callback = null, $options = [])
 * @method \App\Model\Entity\Category patchEntity(
 * \Cake\Datasource\EntityInterface $entity,
 * array $data, array $options = [])
 * @method \App\Model\Entity\Category[] patchEntities(
 * iterable $entities, array $data, array $options = [])
 * @method \App\Model\Entity\Category|false save(
 * \Cake\Datasource\EntityInterface $entity, $options = [])
 * @method \App\Model\Entity\Category saveOrFail(
 * \Cake\Datasource\EntityInterface $entity, $options = [])
 * @method \App\Model\Entity\Category[]|
```

```
 * \Cake\Datasource\ResultSetInterface|false saveMany(
 * iterable $entities, $options = [])
 * @method \App\Model\Entity\Category[]|\Cake\Datasource\
 * ResultSetInterface saveManyOrFail(iterable $entities, $options = [])
 * @method \App\Model\Entity\Category[]|\Cake\Datasource\
 * ResultSetInterface|false deleteMany(iterable $entities, $options = [])
 * @method \App\Model\Entity\Category[]|\Cake\Datasource\
 * ResultSetInterface deleteManyOrFail(
 * iterable $entities, $options = [])
 */
class CategoriesTable extends Table
{
    /**
     * Initialize method
     *
     * @param array $config The configuration for the Table.
     * @return void
     */
    public function initialize(array $config): void
    {
        parent::initialize($config);

        $this->setTable('categories');
        $this->setDisplayField('CategoryID');
        $this->setPrimaryKey('CategoryID');
    }

    /**
     * Default validation rules.
     *
     * @param \Cake\Validation\Validator $validator
     * Validator instance.
     * @return \Cake\Validation\Validator
     */
    public function validationDefault(Validator $validator): Validator
    {
        $validator
            ->scalar('CategoryName')
```

```
            ->maxLength('CategoryName', 15)
            ->requirePresence('CategoryName', 'create')
            ->notEmptyString('CategoryName');

        $validator
            ->scalar('Description')
            ->maxLength('Description', 16777215)
            ->allowEmptyString('Description');

        $validator
            ->allowEmptyString('Picture');

        return $validator;
    }
}
```

The table class file is responsible for defining the schema of the database table, like its columns, data types and the relationships that the table has with other tables in the database. In this example, the CategoriesTable class extends the Table class, which provides several methods and properties that can be used in entity classes, such as:
$alias: The alias of the table. By default, the alias is the underscored version of the table name.
$primaryKey: The name of the primary key field for the table. By default, this is set to 'id'.
$table: The name of the database table for the model. By default, this is set to the underscored version of the class name.
$displayField: The name of the field that should be used as the default display field for the table's entities. By default, this is set to the first field that is not a primary key or a foreign key.
$entityClass: The name of the entity class for the table. By default, this is set to Entity.
$connection: The name of the database connection that the table should use. By default, this is set to 'default'.
query(): it returns a Query object that can be used to build complex queries using a fluent interface.
getConnection(): it returns the database connection object that the table is using.

newEmptyEntity(): it returns a new, empty entity object for the table.
newEntity(): it creates a new entity object and populates it with data from an array or an object.
patchEntity(): it updates an existing entity object with data from an array or an object.
getValidator(): it returns the validator object for the table. This object is used to define validation rules for the table's entities.

4.1.9 Doctrine

The *doctrine:mapping:import* command was used to generate Doctrine entities from an existing database, but it was deprecated in 2019.
At the present time, the *make:entity* command can be used for the same purpose. This command is part of the Symfony CLI and can be used in Symfony projects having Doctrine as their database layer.

4.1.10 Laravel Eloquent

Moving on to Laravel Eloquent, the *make:model* command can be used and executed as follows:

```
php artisan make:model ModelClassName
```

Artisan is the command line interface included with Laravel. One can view a list of all available Artisan commands, by calling :

```
php artisan list
```

This command only creates a basic structure for a model class without including any specific properties or fields:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Customer extends Model
{
    //
}
```

If the *ModelClassName* given to the command is for example, "Customer", Eloquent assumes that the referred table is "customers". In case of the database tables' names not fitting this convention, the corresponding table name can be specified inside the model class by defining a table property:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Customer extends Model
{
    use HasFactory;
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_customers';
}
```

4.2 Code-first approach

A code-first approach is a technique in which the application code is written first, and the database schema is generated from it. This approach suits situations where the database is relatively simple.

4.2.1 Hibernate

Hibernate supports the creation of database tables from Java classes through hbm2ddl, which stands for Hibernate Mapping to Database Definition Language.
An example model class is provided:

```
@Entity
@Table(name = "categories")
public class Category {

  @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "CategoryId")
    private Long id;

    @Column(name = "CategoryName")
    private String name;

    @Column(name = "Description")
    private String description;

    @Column(name = "Picture")
    private String picture;

    // getters and setters
}
```

To create the table, it is crucial to specify the hbm2ddl.auto property in the Hibernate configuration file:

```
<property name="hbm2ddl.auto">create</property>
```

"Hbm2ddl.auto" can have different values:

- none: no actions will be performed
- create-only: database creation will be generated
- drop: database dropping will be generated
- create: will create the schema, dropping previous data
- create-drop: will drop the schema and recreate it on SessionFactory startup, will also drop the schema on SessionFactory shutdown
- validate the database schema
- update: update the schema

4.2.2 MyBatis

MyBatis does not provide a feature to create database tables from model classes.

4.2.3 Entity Framework Core

Database tables can be automatically created in EF Core using the Microsoft.EntityFrameworkCore.Design NuGet package that includes the commands for running migrations and generating database tables from model classes.
Here is a model class example:

```
using Microsoft.EntityFrameworkCore;

public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public string Picture { get; set; }
}

public class ApplicationDbContext: DbContext
{
    public DbSet<Category> categories { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder)
    {
        optionsBuilder.UseSqlServer("connectionString");
    }
}

class Program
{
    static void Main(string[] args)
    {
        using (var context = new ApplicationDbContext())
        {
            context.Database.Migrate();
        }
    }
}
```

DbSet is a generic class provided by Entity Framework Core that represents a collection of entities in the database. The name of the DbSet "categories" is the name that will be given to the corresponding database table. Each time the "categories" property is queried, EF Core executes a database query to retrieve the data. Similarly, when entities are added, updated, or deleted in the "categories" collection, EF Core automatically generates and executes the corresponding database statements to apply the changes in the database. All the ORMs in this study provide this functionality, as do most modern ORMs.
The following example illustrates how to create composite foreign keys:

```
public class EmployeeTerritory
{
    [Key]
    [Column(Order = 0)]
    [ForeignKey("Employee")]
    public int EmployeeId { get; set; }

    [Key]
    [Column(Order = 1)]
    [ForeignKey("Territory")]
    public int TerritoryId { get; set; }

    public Employee Employee { get; set; }
    public Territory Territory { get; set; }
}
```

4.2.4 NHibernate

Database table automatic generation is possible in NHibernate.
It can be accomplished using SchemaExport class:

```
using System;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Tool.hbm2ddl;

class Program
{
```

```
    static void Main(string[] args)
    {
        var configuration = new Configuration();
        configuration.Configure();

        configuration.AddAssembly(typeof(Category).Assembly);
        var schemaExport = new SchemaExport(configuration);
        schemaExport.Execute(false, true, false);
    }
}
```

The Configuration class is used to create an NHibernate configuration, and the AddAssembly method is used to add the assembly that contains the model classes to the configuration. The SchemaExport class generates the database tables based on the mapping information in the model classes.
For the SchemaExport class to work is necessary to define both a model class and an XML mapping file. The model class defines the structure of the data. The mapping file tells NHibernate how to map the class to the database table.

```
public class Category
{
    public virtual int Id { get; set; }
    public virtual string Name { get; set; }
    public virtual string Description { get; set; }
    public virtual string Picture { get; set; }
}
```

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-x.x.x"
                   namespace="ModelNamespace"
                   assembly="ModelAssembly">

  <class name="Category" table="categories">
    <id name="Id" column="CategoryId">
      <generator class="native"/>
    </id>
    <property name="Name" column="CategoryName"/>
    <property name="Description"/>
```

```
    <property name="Picture"/>
  </class>

</hibernate-mapping>
```

4.2.5 SQLAlchemy

With SQLAlchemy is possible to generate a database table starting from an existing class, declared as a sub-class of declarative_base. Inheriting from *declarative_base*, the class is recognized as a mapped class, which means SQLAlchemy will use it to represent a database table.
The following shows how to declare a mapped class:

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Category(Base):
    __tablename__ = 'categories'

    id = Column(Integer, primary_key=True, name='CategoryId')
    name = Column(String, name='CategoryName')
    description = Column(String, name='Description')
    picture = Column(String, name='Picture')
```

__tablename__ tells SQLAlchemy how to name the table in the database.
name states how to name the column corresponding to that field.
primary_key=True specifies the primary key of the table. Composite primary keys can be defined as follows:

```
class EmployeeTerritory(Base):
__tablename__ = 'employeeterritories'
employeeId = Column(Integer, primary_key=True,
name='EmployeeId', ForeignKey('employees.EmployeeId'))
territoryId = Column(String, primary_key=True,
name='TerritoryId', ForeignKey('territories.TerritoryId'))

__table_args__ = (
```

```
        PrimaryKeyConstraint('employeeId', 'territoryId'),
    )
```

The next and final step is to call the create_all() method to After creating the class, the "engine" retrieved from the database connection is used to specify in which database the tables should be created. A table is generated for each class that inherits from the declarative base:

```
engine = create_engine('db_connection_string')
Base.metadata.create_all(engine)
```

The *create_all()* method does not overwrite tables that already exist in the database.

4.2.6 Peewee

Peewee allows for the automatic creation of database tables and the process resembles that of SQLAlchemy.
For example:

```
from peewee import *

db = MySQLDatabase('db_connection_string')

class Category(Model):
    category_id = IntegerField(column_name='CategoryID',
                               primary_key=True)
    category_name = CharField(column_name='CategoryName')
    description = TextField(column_name='Description')
    picture = BlobField(column_name='Picture')

    class Meta:
        database = db
        db_table = 'categories'

db.connect()
db.create_tables([Category], safe=True)
```

Model is the base class for creating models in Peewee.
The inner class Meta is used to define different model options, in this case,

the database where the table will be created and the table name.
create_tables() method is used to create tables, indexes, and associated metadata for the given models' list.
The following shows how to create a composite index:

```
class EmployeeTerritory(Model):
employeeId = ForeignKeyField(Employee, column_name='EmployeeId')
territoryId = ForeignKeyField(Territory, column_name='TerritoryId')

class Meta:
        database = db
        db_table = 'employeeterritories'
        indexes = (
        (('employeeId', 'territoryId'), True),
        )
```

True indicates that the couple ('employeeId', 'territoryId') is unique in the table.

4.2.7 Django ORM

Moving on to Django, the tables' automatic generation is done through the django.db.models.Model class.

```
from django.db import models

class Category(models.Models):
    id = models.AutoField(primary_key=True, db_column="CategoryId")
    name = models.CharField(db_column="CategoryName")
    description = models.CharField(db_column="Description")
    picture = models.CharField(db_column="Picture")

    class Meta:
        db_table = "categories"
```

Afterward, the corresponding database tables are created by calling two commands:

```
python manage.py makemigrations
python manage.py migrate
```

The first one generates migration files that describe the changes you made to your models. The second command applies those changes to the database and creates the tables. Whenever changes to the models are made, the database tables can be updated by running these two commands.

It is possible to create composite keys in Django by downloading the django-composite-field library.
Here is an example:

```
from composite_field import CompositeKey
from django.db import models

class EmployeeTerritory(models.Model):
    employeeId = models.ForeignKey(Employee,
    on_delete=models.CASCADE, db_column="EmployeeId")
    territoryId = models.ForeignKey(Territory,
    on_delete=models.CASCADE, db_column="TerritoryId")

    class Meta:
        unique_together = (("employeeId", "territoryId"),)
        index_together = [("employeeId", "territoryId"),]
        primary_key = CompositeKey("employeeId", "territoryId")
```

The *unique_together* option states that the combination of "employeeId" and "territoryId" should be unique.
The *index_together* option is used to create an index on the "employeeId" and "territoryId" fields.
The *CompositeKey* class is used to specify a composite primary key made up of the "employeeId" and "territoryId" fields.

4.2.8 CakePHP

It is possible to automatically generate database tables from existing model classes in CakePHP using the Bake console tool.
After creating a class, it can be translated into a table by running the following command in the bake console:

```
bin/cake bake all [ModelClassName]
```

This command will only create a table if it doesn't already exist and will not overwrite or update an existing table.

The following is an example of how a class would appear once it has been created:

```
    class Category extends Model {
    public $table = 'my_categories';

    public $validate = [
        'name' => [
            'rule' => 'notBlank'
        ],
        'description' => [],
        'picture' => []
    ];
}
```

Calling

```
    bin/cake bake all Category
```

will automatically generate an "id" column for the corresponding table and name it "categories". The $table variable must be specified if a different table name is desired.

Follows an example with composite and foreign keys:

```
class Region extends Model {
    protected $_schema = [
        'RegionId' => ['type' => 'integer'],
        'RegionDescription' => ['type' => 'string', 'length' => 50]
    ];

    public $hasMany = ['Territory'];
}

class Territory extends Model {
    protected $_primaryKey = ['TerritoryId', 'RegionId'];
```

```
    protected $_schema = [
        'TerritoryId' => ['type' => 'string', 'length' => 20],
        'RegionId' => ['type' => 'integer'],
    ];

    public $belongsTo = [
        'Region' => [
            'foreignKey' => 'RegionId'
        ]
    ];

    public $validate = [
        'name' => [
            'TerritoryDescription' => 'notBlank'
        ]
    ];
}
```

4.2.9 Doctrine

Doctrine provides the

```
doctrine:schema:update
```

command to generate database tables from model classes. This command analyzes the model classes and compares them with the current schema in the database. If there are any changes, it generates the necessary SQL statements to update the schema accordingly.
Model classes used to be created with the use of annotations:

```
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="categories")
 */
class Category
{
```

```
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer", name="CategoryId")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=15, name="CategoryName")
     */
    private $name;

    /**
     * @ORM\Column(type="string", name="Description")
     */
    private $description();

    /**
     * @ORM\Column(type="string", name="Picture")
     */
    private $picture();

    // Getters and setters ...
}
```

However, starting from Doctrine 3.0, annotations will be deprecated. As a result, XML mapping, and PHP mapping are replacing this functionality. Following is the "Category" model class first in the XML mapping version and then in the PHP mapping one:

- XML mapping

```
<!-- config/doctrine/User.orm.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<doctrine-mapping xmlns="http://doctrine-
project.org/schemas/orm/doctrine-mapping"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://doctrine-project.org/
schemas/orm/doctrine-mapping
```

```
http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">
    <entity name="App\Entity\Category" table="categories">
        <id name="id" type="integer" column="CategoryId">
            <generator strategy="AUTO"/>
        </id>
        <field name="name" type="string"
               column="CategoryName" length="15"/>
        <field name="description" type="string"
               column="Description"/>
       <field name="picture" type="string"
       column="Picture"/>
    </entity>
</doctrine-mapping>
```

- PHP mapping

```
namespace App\Doctrine;

use Doctrine\ORM\Mapping\ClassMetadata;

class CategoryMapping
{
    public static function loadMetadata(ClassMetadata $metadata)
    {
        $metadata->setTableName('categories');

        $metadata->mapField([
            'id' => true,
            'fieldName' => 'id',
            'columnName' => 'CategoryId',
            'type' => 'integer',
            'generator' => [
                'strategy' => 'AUTO',
            ],
        ]);

        $metadata->mapField([
```

```
            'fieldName' => 'name',
            'type' => 'string',
            'length' => 15,
            'columnName' => 'CategoryName',
        ]);

        $metadata->mapField([
            'fieldName' => 'description',
            'type' => 'string',
            'columnName' => 'Description',
        ]);

        $metadata->mapField([
            'fieldName' => 'picture',
            'type' => 'string',
            'columnName' => 'Picture',
        ]);
    }
}
```

4.2.10 Laravel Eloquent

Laravel provides a command line tool, *php artisan migrate*, that can be used to generate database tables based on the structure of the model classes. First, one has to define the model classes and then, create the corresponding migration files by running the *php artisan make:migration [create_tableName_table]* command. A migration file determines the structure of the database tables that map to a model class. Once the migration is created, running the *php artisan migrate* command will generate the database tables.
Here is an example of a model class:

```
namespace App\Model;

use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
    protected $table = 'categories';
```

```
    protected $fillable = [
        'CategoryName',
        'Description',
        'Picture'
    ];
}
```

The model migration file can either be done automatically by calling:

```
php artisan make:migration create_caegories_table
```

or manually.
The migration file would look like the following:

```
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateCategoriesTable extends Migration
{
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('CategoriesName', 15);
            $table->string('Description');
            $table->string('Picture');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('categories');
    }
}
```

The following is an example of a composite key made of two foreign keys:

```
use Illuminate\Support\Facades\Schema;
```

```
use Illuminate\Database\Migrations\Migration;

class CreateEmployeeTerritoriesTable extends Migration
{
    public function up()
    {
        Schema::create('employeeterritories',
                        function (Blueprint $table) {
            $table->unsignedInteger('EmployeeId');
            $table->unsignedInteger('TerritoryId');

            $table->primary(['EmployeeId', 'TerritoryId']);

            $table->foreign('EmployeeId')
                ->references('id')
                ->on('employees')
                ->onDelete('cascade');

            $table->foreign('TerritoryId')
                ->references('id')
                ->on('territories')
                ->onDelete('cascade');

            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::table('employeeterritories',
                       function (Blueprint $table) {
            $table->dropForeign(['EmployeeId', 'TerritoryId']);
        });

        Schema::dropIfExists('employeeterritories');
    }
}
```

4.3 Overview of mapping differences

4.3.1 General differences

- **Hibernate**: Hibernate uses annotations or XML files to define the mapping between Java objects and database tables. It supports various types of associations, such as one-to-one, one-to-many, many-to-one, and many-to-many. It also supports inheritance mapping and composite primary keys.
- **MyBatis**: MyBatis uses XML files or annotations to define SQL statements and maps the results of those statements to Java objects. MyBatis does not have built-in support for associations, inheritance mapping, or composite primary keys.
- **EF Core**: EF Core uses attributes or Fluent API to define the mapping between .NET objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. EF Core also provides support for lazy loading, caching, and advanced querying.
- **NHibernate**: NHibernate uses attributes or XML files to define the mapping between .NET objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. NHibernate also provides support for lazy loading and caching.
- **SQLAlchemy**: SQLAlchemy uses Python classes to define the mapping between Python objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. SQLAlchemy also provides support for lazy loading, caching, and dynamic querying.
- **Peewee**: Peewee uses Python classes to define the mapping between Python objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys.
- **Django ORM**: Django ORM uses Python classes to define the mapping between Python objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. Django ORM also provides support for lazy loading, caching, and advanced querying.

- **CakePHP ORM**: CakePHP ORM uses PHP classes to define the mapping between PHP objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. CakePHP ORM also provides support for lazy loading, caching, and simple querying.

- **Doctrine**: Doctrine uses annotations or XML files to define the mapping between PHP objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. Doctrine also provides support for lazy loading, caching, and advanced querying.

- **Laravel Eloquent**: Laravel Eloquent uses PHP classes to define the mapping between PHP objects and database tables. It supports various types of associations, inheritance mapping, and composite primary keys. Laravel Eloquent also provides support for lazy loading, caching, and advanced querying.

4.3.2 Bidirectional references

- **Hibernate**: Hibernate supports bidirectional references with the *@ManyToOne* and *@OneToMany* annotations as discussed in Section 4.1.1

- **MyBatis**: MyBatis does not have built-in support for bidirectional references. However, it is possible to manually implement bidirectional relationships by defining the necessary SQL statements and mapping the results to Java objects.

- **EF Core**: EF Core supports bidirectional references with the *HasMany()* and *WithOne()* methods. A bidirectional one-to-many relationship is defined by using the *WithOne()* method to configure the reference navigation property on the "many" side of the relationship.

- **NHibernate**: NHibernate supports bidirectional references with the *many-to-one* and *one-to-many* associations. A bidirectional one-to-many relationship is defined by using the inverse attribute on the one-to-many element.

- **SQLAlchemy**: SQLAlchemy supports bidirectional references with the *relationship()* function. To define a bidirectional one-to-many rela-

tionship the *back_populates* parameter has to be used on the *relationship()* function.

- **Peewee**: Peewee supports bidirectional references with the *ForeignKeyField* and *ReverseForeignKeyField* fields. A bidirectional one-to-many relationship can be defined by using the ReverseForeignKeyField() field to configure the reference navigation property on the "many" side of the relationship.

- **Django ORM**: Django ORM supports bidirectional references with the *ForeignKey* and *related_name* attributes. It is possible to define a bidirectional one-to-many relationship by using the *ForeignKey* attribute to configure the reference navigation property on the "many" side of the relationship.

- **CakePHP ORM**: CakePHP ORM supports bidirectional references with the *belongsTo()* and *hasMany()* methods. A bidirectional one-to-many relationship can be defined by using the *hasMany()* method to configure the reference navigation property on the "many" side of the relationship.

- **Doctrine**: Doctrine supports bidirectional references with the *many-to-one* and *one-to-many* associations. A bidirectional one-to-many relationship can be defined by using the *mappedBy* attribute on the *one-to-many* annotation.

- **Laravel Eloquent**: Laravel Eloquent supports bidirectional references with the *belongsTo()* and *hasMany()* methods. It is possible to define a bidirectional one-to-many relationship by using the *hasMany()* method to configure the reference navigation property on the "many" side of the relationship.

4.3.3 Many-to-many relations

In Hibernate and NHibernate, there is the *@ManyToMany* annotation that defines a many-to-many relationship between two entities.
In SQLAlchemy, the *relationship()* function can be used with the *secondary* parameter to define a many-to-many relationship between two tables:

```
class Student(Base):
```

```
__tablename__ = 'students'

id = Column(Integer, primary_key=True)
name = Column(String)

courses = relationship('Course', secondary=students_courses_table)

class Course(Base):
__tablename__ = 'courses'

id = Column(Integer, primary_key=True)
```

In this example the *relationship()* methods specifies a many-to-many relation between the tables "courses" and "students" as a student can take several courses and a single course is taken by multiple students.
In EF Core, the *HasMany()* and *WithMany()* methods define a many-to-many relationship between two entities.

4.3.4 Cascading Deletes

In Django ORM, the *ForeignKey()* attribute with the *on_delete* parameter can be used to define the behavior when the referenced object is deleted. There are several options one can choose from, such as CASCADE (delete the referencing object as well), PROTECT (raise an exception), SET_NULL (set the referencing object's foreign key to NULL), and SET_DEFAULT (set the referencing object's foreign key to its default value).

4.3.5 Eager loading definition

In Laravel Eloquent, it is possible to define eager loading of models through the *with()* method. This can help to reduce the number of database queries required to retrieve related data. For example, with the Northwind database, the *with()* method can be used to load all of a customer's orders at once, instead of querying the database for each order individually.

4.3.6 Embedded objects

In Doctrine, one can use the *@Embedded* and *@Embeddable* annotations to define embedded objects. This allows the storage of a group of related values

as a single field in the database. For example, if there is a model for addresses, it is made possible to define an embedded object for the address fields.

5 CRUD operations

This next chapter intends to illustrate how CRUD operations can be performed with the ORMs previously mentioned.
Operations testing was conducted using*Northwind* database, in particular, many examples refer to *orders* table.

5.1 Hibernate

This ORM has its query language: HQL, which stands for "Hibernate Query Language". HQL has its syntax and was designed to be fully object-oriented. It understands the concepts of inheritance, polymorphism, and association. Before any HQL query execution, it is imperative to create a session using *org.hibernate* library and, as described in this section of the Hibernate User Guide, different strategies can be used to create one.

- Create

```
String hqlInsert = "insert into Order (...)";
Query query = session.createQuery( hqlInsert )
query.executeUpdate();
```

- Read

```
Query query = session.createQuery("from Order");
List<Order> orderList = query.list();

Query query = session.createQuery("from Order
where order_id= :orderId");
query.setLong("orderId", idNumber);
Order order = (Order) query.uniqueResult();
```

- Update

```
update Order o
set o.ship_address= :'shipAddress'
where o.order_id= :orderId

Query query = session.createQuery("update Order o
```

```
set o.ship_address= :'shipAddress'
where o.order_id= :orderId");
query.setParameter("shipAddresString");
query.setLong("orderId", idNumber);
query.executeUpdate();
```

- Delete

```
Query query = session.createQuery("delete from Order
where order_id= :orderId");
query.setLong("orderId", idNumber);
query.executeUpdate();
```

The HQL language provides a wide variety of statements to allow the generation of all types of queries that can be created with row SQL. For an in-depth reading of the HQL language, there is a dedicated page in the Hibernate User Guide.
Hibernate does not require the programmer to use HQL. It is possible to execute native SQL queries as well using the *createNativeQuery* method provided by Jakarta Persistence library.

5.2 MyBatis

When using MyBatis to perform CRUD operations, it is a must to take advantage of the mapper files, whether they are XML files or Java interfaces. A more detailed description of mapper files and MyBatis configuration can be found in section 3.2.
Performing any operation requires the reading of the MyBatis configuration file followed by the creation of a SQL session. An example is given here:

```
InputStream inputStream = Resources
.getResourceAsStream("mybatis-config.xml");
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
SqlSession session = factory.openSession();
```

Afterward, it is possible to call any of the operations defined in the mapper files by making use of the *session* element:

- Create:

```
session.insert("insertOrder", new Order(orderId, customerId, ...);
```

- Read:

```
List<Order> orderList = session.selectList("selectOrders");

Order order = session.selectOne("selectOrder", idOrder);
```

- Update:

```
Order newOrder = new Order(oldOrderId, newInformation, ...)
session.update("updateOrder", newOrder);
```

- Delete

```
session.delete("deleteOrder", orderId);
```

As can be observed, once the configuration process has been completed, performing operations is a very straightforward and time-saving process.
With MyBatis, any query can be executed once it is correctly declared and defined in a mapper file. Here is an example of a "JOIN" between shippers and orders:

```
@Select("SELECT s.*, count(distinct o.OrderID)
FROM shippers s
LEFT JOIN orders o
ON (o.ShipVia = s.ShipperID)
GROUP BY s.ShipperID")
List<Shipper> joinShippersOrders();
```

5.3 Entity Framework Core

EF Core uses Language-Integrated Query (LINQ) to query data from the database.
A LINQ query can be written in any .NET language and database objects are referenced using the entity classes that have been previously generated.

- Create:

```
using (var context = new DbContext())
{
    var blog = new Blog { CustomerID = customerId
                          EmployeeID = employeeId,
                          ... };
    context.Orders.Add(order);
    context.SaveChanges();
}
```

- Read:

```
using (var context = new DbContext())
{
    var orders = context.Orders.ToList();
}

using (var context = new DbContext())
{
    var order = context.Orders
    .Single(o => o.OrderId == orderId);
}

using (var context = new DbContext())
{
v   ar orders = context.Orders
    .Where(o => o.RequiredDate == date)
    .ToList();
}
```

- Update:

```
using (var context = new DbContext())
{
    var order = context.Orders
    .Single(o => o.OrderId == orderId);
    order.ShipPostalCode = newShipPostalCode;
    context.SaveChanges();
}
```

- Delete:

```
using (var context = new DbContext())
{
    var order = context.Orders
    .Single(o => o.OrderId == orderId);
    context.Remove(order);
    context.SaveChanges();
}
```

Additionally, EF Core allows the programmer to write raw SQL queries and to treat them similarly to LINQ queries:

```
using (var context = new DbContext())
{
    var orders = context.Orders.
                 SqlQuery("SELECT * FROM orders")
                 .ToList();
}
```

5.4 NHibernate

Like its Java sibling Hibernate, NHibernate uses HQL while at the same time offering support for LINQ as well.
The process leading to the creation and execution of any query with NHibernate, whether HQL or LINQ, is similar to the one for Hibernate. First of all, it is necessary to create a session, this time using *NHibernate.Cfg* namespace.

```
var cfg = ConfigureNHibernate();
var sessionFactory = cfg.BuildSessionFactory();
using(var session = sessionFactory.OpenSession())
```

After this passage, the session created can be used to execute any query.

5.4.1 HQL

The following example shows the syntax to write and execute any HQL query with NHibernate:

```
using(var transaction = session.BeginTransaction()) {
    var orders = session.CreateQuery(queryString);
    transaction.Commit();
}
```

The *queryString* parameter can be replaced with any query that has a valid HQL syntax, shown in section4.1. Any knowledge of HQL can be reused both for Hibernate and NHibernate.

5.4.2 LINQ

The *NHibernate.Linq* namespace allows LINQ to be used in NHibernate. As with HQL, any query that complies with LINQ syntax can be written and executed using NHibernate.
Here is an example of how to execute a LINQ query,

```
using(var transaction = session.BeginTransaction()) {
    var orders = session.Query<Order>()
    .Where(o => o.OrderId == orderId).First();
    transaction.Commit();
 }
```

In section 4.3, other examples of CRUD operations using LINQ have already been provided.

5.5 SQLAlchemy

When using SQLAlchemy, the first thing to do to execute CRUD operations is to create a session using the *sessionmaker* class from the *sqlalchemy.orm* module.

```
Session = sessionmaker(bind=engine)
session = Session()
```

Then it is possible to execute any query thanks to the mapping classes and the methods that Session offers:

- Create:

```
new_order = Order(customerID, employeeID, orderDate,
requiredDate, shippedDate, shipVia, freight, shipName,
shipAddress, shipCity, shipRegion, shipPostalCode, shipCountry)
session.add(new_order)
session.commit()
```

- Read:

```
orders = session.query(Order).all()

orders = session.query(Order).filter(Order.id == orderId).first()
```

- Update:

```
order = session.query(Order).filter(Order.id == orderId).first()
order.shipName = "new ship name"
order.requiredDate = "newRequiredDate"
session.commit()
```

- Delete

```
order = session.query(Order).filter(Order.id == orderId).first()
session.delete(order)
session.commit()
```

The Session class offers a variety of methods that make it easy to construct any query. For example, here is a query to retrieve, for each shipper, the number of deliveries done, including the shippers that haven't done a delivery yet:

```
query = (
session.query(Shipper.id, Shipper.company name,
func.count(func.distinct(Order.id))
.label('num_orders'))
.select_entity_from(Shipper)
.outerjoin(Order, Order.shipVia == Shipper.id)
.group_by(Shipper.id)
)
```

5.6 peewee

Once the model classes have been defined, Peewee enables the formulation of any query, usually in a variety of ways as well.

- Create

  ```
  new_order = Order.create(customerID, employeeID, orderDate,
  requiredDate, shippedDate, shipVia, freight, shipName,
  shipAddress, shipCity, shipRegion, shipPostalCode, shipCountry)

  Order.insert(customerID, employeeID, orderDate,
  requiredDate, shippedDate, shipVia, freight, shipName,
  shipAddress, shipCity, shipRegion, shipPostalCode, shipCountry)
  ```

  While the *insert* function will only create the database table entry, the *create* one will create the entry and return the class instance too.

- Read

  ```
  Order.select()

  Order.get(Order.id == orderId)

  Order.get_by_id(orderId)

  Order[orderId]
  ```

  The first example shown retrieves all entries of a table, in this case, the *orders* table, while the other two retrieve a single entry based on its primary key.

- Update

  ```
  order = Order.get(Order.order_id == orderId)
  order.ship_name = 'Name Surname'
  order.save()

  Order.update(ship_name = 'Name Surname')
  .where(Order.order_id == orderId).execute()
  ```

The former method shown is the classic one while the latter is called "Atomic Update" and, apart from being much faster, it is guarded against transaction errors in case multiple processes update the entry simultaneously.

- Delete

```
order = Order.get(Orders.order_id == orderId)
order.delete_instance()

Product.delete().where(Product.discontinued == true).execute()
```

While the first method can delete a single entry, the second can erase an arbitrary number of rows according to the given condition.

5.7 Django

Once the data models have been created, Django automatically offers a database-abstraction API to create, retrieve, update, and delete objects.

- Create

```
from db.models import Order
o = Order(customerID, employeeID, orderDate, requiredDate,
shippedDate, shipVia, freight, shipName, shipAddress,
shipCity, shipRegion, shipPostalCode, shipCountry)
o.save()

o = Order.objects.create(customerID, employeeID, orderDate,
requiredDate, shippedDate, shipVia, freight, shipName,
shipAddress, shipCity, shipRegion, shipPostalCode, shipCountry)
```

The first method creates the database entry only when the *save* function is called. The second method uses the *create* function to create both the object and the entry atomically.

- Read

```
Order.objects.all()

Order.objects.filter(ShipName=shipname)
            .exclude(OrderDate=datetime.date.today())

Order.objects.get(OrderId=orderId)
```

The first example retrieves all table entries while the second and third examples filter the data in different ways. *Filter* and *exclude* functions return a *QuerySet*, which is a collection of objects from the database, even if there is only a single entry matching the required parameters. In contrast, the *get* function always returns a single entry.

- Update

```
o = Order.objects.get(OrderId=orderId)
o.ShipAddress = 'New ship address'
o.save()
```

- Delete

```
o = Order.objects.get(OrderId=orderId)
o.delete()

Order.objects.filter(OrderDate__year<1997).delete()
```

The former example shown deletes one entry while the latter one delete can be used to delete multiple database rows. In both cases, when the *delete* function is called, Django emulates the behavior of the SQL constraint "ON DELETE CASCADE", so any objects that have foreign keys pointing at the deleted object are removed as well.

5.8 CakePHP

CakePHP differs quite much from all the other ORMs previously illustrated in terms of CRUD operations.
Using the orders table ad an example, if one wants to operate any transaction they have to create the related controller class, which extends *AppController*.

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\ORM\TableRegistry;
use Cake\Datasource\ConnectionManager;

class OrdersController extends AppController
{
    //
}
```

This class has to be placed under the *projectname\src\Controller* directory. The class methods will be used to implement the different CRUD operations. With a controller class, view files have to be created under the *projectname\src\Template\TableName* directory (TableName is Orders in this case), one for each operation.
Composer automatically generates the directory paths mentioned at the moment of the project creation (section 3.6)

- Create

Foremost, a code line like the following has to be put inside the *projectname\config\routes.php* file:

```
$builder->connect('/orders/add', ['controller' => 'Orders',
                    'action' => 'add']);
```

Routes are a way of connecting request URLs to objects in the application. The first parameter is the controller name while the second is the action name, respectively "Orders" and "add" in this case.
The function that will be written inside the controller class is the following:

```
public function add()
{
    if ($this->request->is('post')) {
        $customerId = $this->request->getData('customerId');
        $employeeId = $this->request->getData('employeeId');
        $orderDate = $this->request->getData('orderDate');
        $requiredDate = $this->request->getData('requiredDate');
        $shippedDate = $this->request->getData('shippedDate');
```

```
            ...

            $orders_table = TableRegistry::getTableLocator()
                            ->get('Orders');
            $order = $orders_table
                     ->newEntity($this->request->getData());

            $order->customerId = $customerId;
            $order->employeeId = $employeeId;
            $order->orderDate = $orderDate;
            $order->requiredDate = $requiredDate;
            $order->shippedDate = $shippedDate;
            ...
        }
    }
```

As the data is passed through a post request, this class method retrieves it and stores it in a new Order entity.
For the post request to be created and sent the view file "add.php" has to be created:

```
echo $this->Form->create(NULL, array('url' => '/orders/add'));
echo $this->Form->control('customerId');
echo $this->Form->control('employeeId');
echo $this->Form->control('orderDate');
echo $this->Form->control('requiredDate');
...
echo $this->Form->button('Submit');
echo $this->Form->end();
```

This file creates the forms to input the required data, then submits everything as a post request. The forms are accessible via the URL of the project, for example, http://server/projectname/orders/add

The steps described are repeated for each of the other CRUD operations.

- Read

The route:

```
$builder->connect('/orders', ['controller' =>'Orders',
'action' => 'index']);
```

The class method:

```
 public function index(){
     $orders = TableRegistry::getTableLocator()->get('Orders');
     $query = $orders->find();
     $this->set('results',$query);
}
```

The structure of the index.php view file depends on the content that will be displayed. For example, using the method shown above, all columns of the orders table will be retrieved. This data could be organized in an HTML table using the index.php file. This time the URL is http://server/projectname/orders

- Update

```
$builder->connect('/orders/edit', ['controller' => 'Orders',
'action' => 'edit']);
```

As shown in the following example, the controller class method updates the ShippedDate field, but it can be modified at will to update any column.

```
public function edit($OrderId){
     if($this->request->is('post')){
        $shippedDate = $this->request->getData('shippedDate');

        $orders_table = TableRegistry::getTableLocator()
                        ->get('orders');
        $order = $orders_table->get($OrderId);
        $order->ShippedDate = $shippedDate;
        if($orders_table->save($order))
        echo "Order is updated";
     }
  }
```

The edit.php file will differ depending on the update to be implemented.

- Delete

The route:

```
$builder->connect('/orders/delete', ['controller' => 'Orders',
                  'action' => 'delete']);
```

The controller class method:

```
public function delete($CustomerId, $EmployeeId)
{
    $orders_table = TableRegistry::getTableLocator()
                    ->get('orders');
    $order = $orders_table->get($CustomerId, $EmployeeId);
    $orders_table->delete($order);
    echo "Order deleted successfully.";
}
```

5.9 Doctrine

Doctrine allows the use of both raw SQL and DQL, Doctrine Query Language.
Examples of CRUD operations with DQL are shown below.

- Create

```
$entityManager = new EntityManager($connection, $config);

$order = new \Entities\Order();
$order->setEmployeeId(employeeId);
$order->setOrderDate(orderDate);
...
$entityManager->persist($order);
```

The entity manager is obtained through the *Doctrine\ORM\EntityManager* class and the established connection (section 3.6)

- Read

```
$query = $entityManager->createQuery('SELECT o
                        FROM projectname\Model\Order o
                        WHERE o.OrderId = orderId');
```

```
$orders = $query->getResult();
```

```
$query = $entityManager->createQuery('SELECT o.CustomerId
                        FROM projectname\Model\Order o);
$orders = $query->getResult();
```

- Update

```
$query = $entityManager->createQuery('UPDATE
                        projectname\Model\Order o
                        SET o.ShipAddress = newShipAddress
                        WHERE o.OrderId = orderId');
```

- Delete

```
$query = $entityManager->createQuery('DELETE
                            projectname\Model\Order o
                            WHERE o.OrderId = orderId');
```

5.10 Laravel Eloquent

When executing CRUD operations, Eloquent is very similar to CakePHP. To perform operations on a certain table, it is necessary to create the corresponding migration by running the following command:

```
php artisan make:migration create_tablename_table
```

For example:

```
php artisan make:migration create_order_table
```

This will create a class in the *database\migration* folder. This class contains two methods, up() and down(). In the first case, the migration is run, while in the second case, it is reversed. The *up* function contains the table fields, for example:

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
```

```
return new class extends Migration
{
    public function up(): void
    {
        Schema::create('order', function (Blueprint $table) {
            $table->id();
            $table->string('customer_id');
            $table->integer('employee_id');
            $table->string('order_date');
            $table->string('required_date');
            $table->string('shipped_date');
            $table->integer('ship_via');
            $table->decimal('freight');
            $table->string('ship_name');
            $table->string('ship_address');
            $table->string('ship_city');
            $table->string('ship_region');
            $table->string('ship_postal_code');
            $table->string('ship_country');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('order');
    }
};
```

After the migration creation, it is crucial to create the models as well through this command:

```
php artisan make:model modelname
```

For example:

```
php artisan make:model Order
```

This will create a class in the *app\Models* folder in which mandatory fields and one to many relations can be specified. An example follows:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Order extends Model
{
    protected $fillable = [
        'id', 'company_name', 'contact_name',
        'contact_title', 'address', 'city', 'region',
        'postal_code', 'country', 'phone', 'fax',
        'home_page',
    ];
    public function products()
    {
        return $this->hasMany('App\Models\Product');
    }
}
```

Last step is to create a controller through the following command:

```
php artisan make:controller modelnameController
```

For example:

```
php artisan make:controller OrderController
```

This class contains several methods, one for each operation to perform. The following are some examples of CRUD operations with Eloquent:

- Create

```
$order = Order::create([
'CustomerId' => customerId,
'EmployeeId' => employeeId,
...
]);
```

- Read

```
foreach (Order::all() as $order) {
    echo $order->OrderId;
    echo $order->OrderedDate;
    ...
}

$orders = Order::where('OrderId', orderID)
           ->orderBy(OrderedDate)
           ->get();
```

The first method retrieves all entries of the orders table. The second one retrieves a single row according to the conditions specified.

- Update

```
$order = Order::where('OrderId', orderID);
$order->ShipCountry = newShipCountry;
$order->save();

$order = Order::where('OrderId', oldOrderId);
$order->OrderId = newOrderId;
$order->refresh();
$order->OrderId; // oldOrderId
```

The first example shows how to update data of a certain database entry. The second example illustrates the operation of the *refresh* function. When some changes are applied to the model data but not saved to the database, *refresh* can be used to reload the database data into the model data.

- Delete

```
$order = Order::where('OrderId', orderId);
$order->delete();

Order::destroy(orderId);

Order::truncate();
```

The first two cases will result in the deletion of a single row, while the third case will result in the deletion of all database records associated with the model. The difference between the first and the second example is that the *destroy* function can only be called on the object's primary key to remove. Several primary keys can be passed to this function, one for each entry to delete:

```
Order::destroy(orderId1, orderId2, orderId3);
```

In conclusion, a controller class would result as the following:

```
namespace App\Http\Controllers;

use App\Models\Order;

class OrderController extends Controller
{
    // create a new order
    public function create()
    {
        Order::create([
            'id' => id,
            'customer_id' => customerId,
            'employee_id' => employeeId,
            'order_date' => orderDate,
            ....
        ]);
        echo 'Order created';
    }

    public function select()
    {

        $orders = Order::all();
        return $orders;
    }

    public function update()
    {
```

```
        $order = Order::find(orderId);
        // changes to $order
        $order->save();
        echo 'Order updated';
    }

    public function delete()
    {
        $order = Order::find(orderId);
        $order->delete();
        echo 'Order deleted';
    }
}
```

In order to actually execute the methods and consequently, the operations, specific routes have to be added to the *routes\web.php* file:

```
Route::get('newOrder', 'OrderController@create');

Route::get('selectOrders', 'OrderController@select');

Route::get('updateOrder', 'OrderController@update');

Route::get('deleteOrder', 'OrderController@delete');
```

With these routes, entering the *http://localhost:8000/selectOrders* URL would execute the *select()* method of the OrderController class.
In order to execute any Laravel CRUD operation, the Laravel server must be started by running the following command:

```
php artisan serve
```

5.11 Query languages

This section is devoted to the query languages previously mentioned (LINQ, HQL, DQL), their similarities, and their differences.
LINQ(Language Integrated Query) is a query language used in .NET applications, specifically in C# and other .NET languages. HQL(Hibernate Query Language) is used in both the Hibernate ORM framework for Java

and the NHibernate ORM for .NET. DQL(Doctrine Query Language) is a query language used in the Doctrine ORM for PHP.
There are some similarities between these three technologies.
All three languages:

- allow writing queries in a high-level, object-oriented syntax that's abstracted from the underlying database.
- provide a way to write type-safe queries.
- provide a way to perform complex operations like grouping, filtering, and aggregating data.

Here is an example query that returns, for each employee, the number of customers served in each country, ordered by employee and country:

- HQL (Hibernate):

```
SELECT e.employeeID, e.lastName, e.firstName,
c.country, count(distinct c.customerID) as numCustomers
FROM Employee e
JOIN e.orders o
JOIN o.customer c
GROUP BY e.employeeID, e.lastName, e.firstName, c.country
ORDER BY e.lastName, e.firstName, c.country
```

- HQL (Nhibernate):

```
var result = session.CreateQuery(
    "SELECT e.EmployeeID, e.LastName, e.FirstName, c.Country,
    count(distinct c.CustomerID) as numCustomers
     FROM Employee e
     JOIN e.Orders o
     JOIN o.Customer c
     GROUP BY e.EmployeeID, e.LastName, e.FirstName, c.Country
     ORDER BY e.LastName, e.FirstName, c.Country"
).List<object[]>();
```

- LINQ (C#):

```
result = (
     from e in dbContext.Employees
     from o in e.Orders
     from c in o.Customer
     group c by new { e.EmployeeID, e.LastName,
                    e.FirstName, c.Country }
     into g
     orderby g.Key.LastName, g.Key.FirstName, g.Key.Country
     select new
     {
         EmployeeID = g.Key.EmployeeID,
         LastName = g.Key.LastName,
         FirstName = g.Key.FirstName,
         Country = g.Key.Country,
         numCustomers = g.Select(x => x.CustomerID)
                                 .Distinct().Count()
     }
 ).ToList();
```

- DQL

```
$result = $em->createQuery(
    "SELECT e.employeeID, e.lastName, e.firstName,
     c.country, count(distinct c.customerID) as numCustomers
     FROM Employee e
     JOIN e.orders o
     JOIN o.customer c
     GROUP BY e.employeeID, e.lastName, e.firstName, c.country
     ORDER BY e.lastName, e.firstName, c.country"
)->getResult();
```

Each query shows a high-level and object-oriented syntax and provides mechanisms for grouping, ordering, and joining data.
Moving on to the differences between the three query languages, as shown by the examples above, all of them have different syntax and support different features.
Despite the fact that they all provide type safety, LINQ is the only strongly typed one, which means that the compiler can check that the programmer

is using the correct types and objects' properties. This reduces the chance of run-time errors and makes it easier to understand and maintain the code. On the other hand, HQL and DQL are dynamically typed, which means that type checking is performed at run-time. This can make them more flexible, but also less safe.
Regarding their syntax, HQL uses a syntax that resembles that of SQL but it's specifically designed to work with objects rather than tables. DQL uses a similar syntax that is however designed to work with PHP objects and the Doctrine ORM. LINQ syntax is based on C# and is the one that deviates the most from SQL, with a focus on readability and expressiveness.
Another difference can be found in their nature. LINQ is part of the .NET framework, while HQL and DQL are specific to their respective ORM frameworks and can be used in different environments. However, LINQ has a rich ecosystem of extensions, including LINQ to SQL, LINQ to XML, LINQ to Objects, and LINQ to Entities, which provide additional functionality and make it easier to work with different data sources.
In terms of performance, HQL and DQL can be slower than LINQ due to the overhead of mapping the data to and from objects. LINQ is known for its fast performance. The strongly-typed nature of LINQ can make it easier to write efficient queries, and the LINQ extensions, such as LINQ to SQL and LINQ to Entities, provide additional optimizations and features that can improve performance.
The three languages support a variety of security features, such as atomic updates. An atomic update is a type of database transaction in which a set of changes to the database is treated as a single, indivisible operation. The changes are either all committed or all rolled back. They prevent partial updates that can leave the database in an inconsistent state. They also ensure that multiple updates to the same data can be made safely and consistently, even when being executed simultaneously.
With HQL, DQL, and LINQ it is possible to choose the level of isolation to work with. Isolation levels determine how much one transaction can see the changes made by other concurrent transactions.
The most common isolation levels are:

- **Read Uncommitted**: transactions can see uncommitted changes made by other transactions.
- **Read Committed**: transactions can see only committed changes made by other transactions.

- **Repeatable Read**: transactions can see only the data that existed at the beginning of the transaction and will not see changes made by other transactions during the transaction's execution.
- **Serializable**: transactions are executed serially, one after another, to prevent conflicts and ensure that all transactions see the same data.

Here is an example of an atomic update using HQL, DQL, and LINQ:

- HQL (Hibernate)

```
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Query query = session.createQuery("update Employee
    set Salary = Salary + :increment
    where EmployeeId = :id");
    query.setParameter("increment", incrementValue);
    query.setParameter("id", idValue);
    int result = query.executeUpdate();
    tx.commit();
} catch (HibernateException e) {
    if (tx!=null) {
        tx.rollback();
    }
    e.printStackTrace();
} finally {
    session.close();
}
```

- HQL (NHibernate)

```
using (ISession session = sessionFactory.OpenSession())
{
    using (ITransaction transaction = session.BeginTransaction())
    {
        session.CreateQuery("update Employee
        set Salary = salary + :increment
```

```
        where EmployeeId = :id")
               .SetInt32("increment", incrementValue)
               .SetString("id", idValue)
               .ExecuteUpdate();
        transaction.Commit();
    }
}
```

- LINQ

```
using (var db = new SampleDataContext())
{
    using (var transaction = db.Database.BeginTransaction())
    {
        try
        {
            var employee = db.Employees
            .FirstOrDefault(e => e.EmployeeId == idValue);
            if (employee != null)
            {
                employee.Salary += increment;
                db.SaveChanges();
                transaction.Commit();
            }
        }
        catch (Exception)
        {
            transaction.Rollback();
        }
    }
}
```

- DQL

```
$entityManager = $this->getDoctrine()->getManager();
$entityManager->beginTransaction();
try {
    $query = $entityManager->createQuery('UPDATE Employee e
```

```
    SET e.Salary = e.Salary + increment WHERE e.EmployeeId = :id');
    $query->setParameter('id', idValue);
    $query->execute();
    $entityManager->commit();
} catch (Exception $e) {
    $entityManager->rollback();
    throw $e;
}
```

When compared to raw SQL, using a query language offers several benefits. Here are a few:

- Lambda Expressions: LINQ allows the use of lambda expressions. This feature can be used for filtering, transforming, or joining data.
  - Filtering:

    ```
    products.Where(p => p.UnitPrice > 20);
    ```

  - Transforming:

    ```
    products.Select(p => p.ProductName);
    ```

  - Grouping:

    ```
    customers.GroupBy(c => c.Country)
             .Select(g => new
             {
                Country = g.Key,
                Count = g.Count()
             });
    ```

  - Ordering:

    ```
    orders.OrderByDescending(o => o.OrderDate);

    orders.OrderBy(o => o.ShippedDate);
    ```

  - Joining:

```
customers.Join(orders,
     c => c.CustomerId,
     o => o.CustomerId,
     (c, o) => new
     {
         Customer = c.ContactName,
         Order = o.OrderId
});
```

- Object-Oriented Queries: query languages allow developers to write queries against objects rather than tables.
- Method Chaining: query languages allow to chain multiple query operations together, making it possible to perform complex operations in a single, readable expression. For example:
  - LINQ:

```
customers.Where(c => c.Country == "Germany")
.OrderBy(c => c.ContactName)
.Select(c => c.ContactName);
```

  - HQL (Hibernate):

```
String hql = "FROM Customer c
WHERE c.Country = 'Germany'
ORDER BY c.ContactName";
Query query = session.createQuery(hql);
List results = query.list();
```

  - HQL (NHibernate):

```
IQueryable<Customer> query = session
.Query<Customer>()
.Where(c => c.Country == "Germant")
.OrderBy(c => c.ContactName);
List<CUstomer> results = query.ToList();
```

- DQL:

```
$query = $em->createQuery("SELECT c.ContactName
FROM Customer c
WHERE c.Country = 'Germany'
ORDER BY c.ContactName");
$results = $query->getResult();
```

- LINQ Specifics: as previously mentioned, LINQ has additional features like LINQ to Objects, LINQ to XML, LINQ to SQL, and LINQ to Entities, which provide a way of querying data in a variety of formats. Here are some examples:

  - LINQ to Objects:

```
List<Customer> customers = GetCustomers();
query = from c in customers
where c.Country == "Germany"
orderby c.ContactName
select p.ContactName;
```

    The *GetPersons* method is used to retrieve a list of Customer objects.

  - LINQ to XML:

```
XDocument doc = XDocument.Load("customers.xml");
query = from c in doc.Descendants("Customer")
where (string)c.Attribute("Country") == "Germany"
orderby (string)c.Attribute("ContactName")
select (string)c.Attribute("ContactName");
```

    XDocument.Load is used to load an XML document from a file. *Descendants* selects the Customer elements from the document. *Attribute* method accesses the values of the "Country" and "ContactName" attributes.

  - LINQ to SQL:

```
DataContext db = new DataContext("ConnectionString");
query = from c in db.Customers
where c.Country == "Germany"
orderby c.ContactName
select c.ContactName;
```

The *DataContext* class is used to represent a connection to a database. The *Customers* property of the db object is used to access the customers' table in the database.

– LINQ to Entities:

```
query = from c in context.Customer
where c.Country == "Germany"
orderby c.ContactName
select c.ContactName;
```

The *context* object is an instance of a class that implements the *DbContext* class.
The query uses the Customer's property of the context object to access the Customer entities in the database.

6 Criteria query

Hibernate and NHibernate both offer support for criteria queries. Criteria queries provide a way to retrieve data from a database using an object-oriented approach, rather than writing raw SQL. This allows for a more flexible and maintainable way to query a database, as the data model can change freely without requiring changes to the related queries. The Criteria API is used only to query data, not for inserting, updating, or deleting data. Even if both ORMs provide Criteria Queries, their implementation differs. Here are two examples of criteria queries, one for Hibernate and one for NHibernate:

- Hibernate

```
CriteriaBuilder builder = entityManager
.getCriteriaBuilder();
CriteriaQuery<Customer> criteria = builder
.createQuery(Customer.class);
Root<Customer> root = criteria.from(Customer.class);
criteria.select(root);

criteria.where(builder
.equal(root.get(Customer_.Country), country));

List<Customer> customers =
entityManager.createQuery(criteria).getResultList();
```

- NHibernate

```
using (var session = sessionFactory.OpenSession())
{
    var criteria = session.CreateCriteria<Customer>();
    criteria.Add(Restrictions.Eq(CustomerId, customerId));
    var customers = criteria.List<Customer>();
}
```

7 QueryOver query

NHibernate offers support for QueryOver queries as well. They are similar to Criteria queries but use a more fluent and strongly typed syntax. They also may perform better than Criteria Queries as they are translated into more efficient SQL statements. Criteria Queries are more flexible and easy to maintain. In contrast, QueryOver queries are usually more suited for simple interrogations and in cases where the query structure is known at compile-time.
Here is an example of a QueryOver query:

```
using(var transaction = session.BeginTransaction()) {
    var query = session.QueryOver<Customer>(() => customer)
    .Where(() => customer.CustomerId == customerId)
    .Select(c => c.ContactName, c => c.CompanyName);
    var result = query.List<object[]>();
    transaction.Commit();
}
```

8 Performances

Among the ORMs discussed thus far, a few have been selected to be evaluated based on their efficiency in executing CRUD operations. All the transactions were performed on the "orders" table of the "Northwind" database.
This performance analysis was conducted on a PC with the following specifics:

- Processor: Intel(R) Core(TM) i5-1035G4 CPU @ 1.10GHz
- RAM: 8.00GB
- Storage: 256GB SSD
- Operating System: Windows 11 Home 22H2

8.1 Basic CRUD operations

Listed below are the operations whose performances were measured:

- Retrieve all orders:
  - Hibernate:

```
Query query = session.createQuery("FROM Orders");
List<Orders> orders = query.list();
```

  - MyBatis:

```
@Select("select * from orders")
List<Order> selectOrders();

List<Order> orderList = session.selectList("selectOrders");
```

  - EF Core:

```
var orders = context.Orders;
```

  - SQLAlchemy:

```
orders = session.query(Order).all()
```

- Peewee:

```
orders = Orders.select()
```

- CakePHP:

```
$orders = TableRegistry::getTableLocator()->get('Orders');
$orders->find();
```

- Retrieve an order based on the ID:
  - Hibernate:

```
Orders order = session.get(Orders.class, 10248);
```

  - MyBatis:

```
@Select("select * from orders where OrderID = #{orderID}")
Order selectOrder(int orderID);

Order order = session.selectOne("selectOrder", 10248);
```

  - EF Core:

```
var order = context.Orders.Find(10248);
```

  - SQLAlchemy:

```
order = session.query(Order)
.filter(Order.OrderID == 10248).first()
```

  - Peewee:

```
order = Orders.select().where(Orders.customer_id == 10248)
```

  - CakePHP:

```
$orders = TableRegistry::getTableLocator()->get('Orders');
$orders->find()->where(['orderId' => 10248]);
```

- Create an order:
  - Hibernate:

```
Orders order = new Orders();
order.setCustomerId("VINET");
order.setEmployeeId(5);
order.setOrderDate(new
Timestamp(System.currentTimeMillis()));
...
session.save(order);
```

  - MyBatis:

```
@Insert("insert into orders (OrderID, CustomerID, EmployeeID,
OrderDate, ..., ShipCountry) values (#{orderID}, #{customerID},
..., #{shipCountry})")
int insertOrder(Order Order);

Order order = new Order(10249, "VINET", 5, ...);
session.insert("insertOrder", order);
```

  - EF Core:

```
var newOrder = new Order
    {
        CustomerId = "VINET",
        EmployeeId = 5,
        OrderDate = DateTime.Now,
        RequiredDate = DateTime.Now.AddDays(7),
        ShippedDate = DateTime.Now.AddDays(7),
        ...
    };
context.Orders.Add(newOrder);
context.SaveChanges();
```

- SQLAlchemy:

```
new_order = Order(CustomerID='VINET', EmployeeID=5,
OrderDate=time.strftime('%Y-%m-%d'), ...)
session.add(new_order)
session.commit()
```

- Peewee:

```
order = Orders.create(customer_id='VINET', employee_id=5, ...)
```

- CakePHP:

```
$orders_table = TableRegistry::getTableLocator()->get('Orders');
$order = $orders_table->newEntity($this->request->getData());
$order->OrderID = 10249;
$order->CustomerID = "VINET";
$order->EmployeeID = 5;
...

$orders_table->save($order);
```

- Update an order:
  - Hibernate:

```
order = session.get(Orders.class, 10249);
order.setShipName("Vins et alcools Chevalier");
session.update(order);
```

  - MyBatis:

```
@Update("update orders set CustomerID = #{customerID},
EmployeeID = #{employeeID}, OrderDate = #{orderDate}, ...}
where OrderID = #{orderID}")
int updateOrder(Order Order);
```

```
Order order = new Order(10249, "VINET", 5,
..., "Vins et alcools Chevalier", ...);
session.update("updateOrder", order);
```

- EF Core:

```
order = context.Orders.Find(10249);
order.ShipName = "Vins et alcools Chevalier";
context.SaveChanges();
```

- SQLAlchemy:

```
order = session.query(Order)
.filter(Order.OrderID == 10249).first()
.update({'ShipName': "Vins et alcools Chevalier"})
session.commit()
```

- Peewee:

```
order = Orders.get(Orders.order_id == 10249)
order.ship_name = 'Vins et alcools Chevalier'
order.save()
```

- CakePHP:

```
$orders_table = TableRegistry::getTableLocator()->get('Orders');
$order = $orders_table->get(10249);
$order->ShipName = "Vins et alcools Chevalier";
$orders_table->save($order);
```

- Delete an order:
  - Hibernate:

```
order = session.get(Orders.class, 10249);
session.delete(order);
```

  - MyBatis:

```
@Delete("delete from orders where OrderID = #{orderID}")
int deleteOrder(int orderID);

session.delete("deleteOrder", 10249);
```

- EF Core:

```
order = context.Orders.Find(10249);
context.Orders.Remove(order);
context.SaveChanges();
```

- SQLAlchemy:

```
order = session.query(Order)
.filter(Order.OrderID == 10249).first()
session.delete(order)
session.commit()
```

- Peewee:

```
order = Orders.get(Orders.order_id == 10249)
order.delete_instance()
```

- CakePHP:

```
$orders_table = TableRegistry::getTableLocator()->get('orders');
$order = $orders_table->get(10249);
$orders_table->delete($order);
```

Here are the performance study results:

ORM	Create	Read	Update	Delete
Hibernate	$105 * 10^6 ns$	$346 * 10^6 ns$	$100 * 10^6 ns$	$615 * 10^5 ns$
MyBatis	$383 * 10^6 ns$	$348 * 10^6 ns$	$299 * 10^6 ns$	$300 * 10^6 ns$
EF Core	$280 * 10^7 ns$	$458 * 10^7 ns$	$412 * 10^7 ns$	$587 * 10^7 ns$
SQLAlchemy	$280 * 10^5 ns$	$238 * 10^5 ns$	$249 * 10^5 ns$	$401 * 10^5 ns$
Peewee	$185 * 10^5 ns$	$207 * 10^5 ns$	$968 * 10^4 ns$	$113 * 10^5 ns$
CakePHP	$615 * 10^5 ns$	$256 * 10^6 ns$	$312 * 10^6 ns$	$519 * 10^5 ns$

ORM	Read All
Hibernate	$202 * 10^6 ns$
MyBatis	$504 * 10^6 ns$
EF Core	$483 * 10^7 ns$
SQLAlchemy	$458 * 10^5 ns$
Peewee	$483 * 10^2 ns$
CakePHP	$247 * 10^5 ns$

8.2 Advanced CRUD operations

Aside from basic CRUD operations, the performance of the various ORMs was also evaluated with more complex operations. The operations considered are listed below:

- OP1 (select orders ordered by their order date):
  - Hibernate

```
session.createQuery("FROM Orders ORDER BY orderDate");
```

- MyBatis

```
@Select("select * from orders order by OrderDate")
List<Order> selectOrdersByOrderDate();

List<Order> orders = session
.selectList("selectOrdersByOrderDate");
```

- EF Core

```
var orders = context.Orders.OrderBy(o => o.OrderDate);
```

- SQLAlchemy

```
session.query(Order).order_by(Order.OrderDate).all()
```

- Peewee

```
orders = Orders.select().order_by(Orders.order_date)
```

- CakePHP

```
$orders = TableRegistry::getTableLocator()
    ->get('Orders');
$query = $orders->find()
    ->select(['OrderID', 'OrderDate'])
    ->order(['OrderDate' => 'ASC']);
```

- OP2 (select the number of orders for each year):
  - Hibernate

```
session.createQuery("select year(o.orderDate), count(*)
from Orders o group by year(o.orderDate)");
```

– MyBatis

```
@Select("select count(OrderID) as NumberOfOrders,
year(OrderDate) as Year
from orders
group by year(OrderDate)")
List<Order> selectOrdersPerYear();

List<Order> orders = session.selectList("selectOrdersPerYear");
```

– EF Core

```
var orders = context.Orders.GroupBy(o => o.OrderDate.Year)
.Select(g => new { Year = g.Key, Count = g.Count() });
```

– SQLAlchemy

```
session.query(Order.OrderDate,
func.count(Order.OrderID))
.group_by(func.year(Order.OrderDate)).all()
```

– Peewee

```
orders = Orders.select(fn.Count(Orders.order_id),
fn.Year(Orders.order_date)
.group_by(fn.Year(Orders.order_date))
```

– CakePHP

```
$orders = TableRegistry::getTableLocator()
        ->get('Orders');
$query = $orders->find();
$query->select([
    'NumberOfOrders' => $query
        ->func()->count('OrderID'),
    'Year' => $query
        ->func()->year(['OrderDate' => 'literal'])
])->group('Year');
```

- OP3 (select, for each shipper, the number of orders shipped even when the shipper has not shipped any order):
  - Hibernate

```
session.createQuery("select s.shipperId, s.companyName,
count(o.orderId)
from Shippers s left join Orders o
on s.shipperId = o.shipVia
group by s.shipperId, s.companyName");
```

  - MyBatis

```
@Select("select count(o.OrderID) as NumberOfOrders,
s.ShipperID, s.CompanyName
from shippers s
left join orders o on s.ShipperID = o.ShipVia
group by ShipperID, CompanyName")
List<Shipper> selectOrdersPerShipper();
```

  - EF Core

```
var orders = from s in context.Shippers
join o in context.Orders on s.ShipperID equals o.ShipVia
into orders from order in orders.DefaultIfEmpty()
group order by new { s.ShipperID, s.CompanyName } into g
select new
{
  ShipperID = g.Key.ShipperID,
  CompanyName = g.Key.CompanyName,
  OrderCount = g.Count(x => x != null)
};
```

  - SQLAlchemy

```
orders = session.query(Shipper.CompanyName,
func.count(Order.OrderID))
.outerjoin(Order, Shipper.ShipperID == Order.ShipVia)
.group_by(Shipper.CompanyName).all()
```

- Peewee

```
orders = Shippers.select(fn.Count(Orders.order_id),
Shippers.shipper_id, Shippers.company_name)
.join(Orders, JOIN.LEFT_OUTER,
on=(Shippers.shipper_id == Orders.ship_via))
.group_by(Shippers.shipper_id, Shippers.company_name)
```

- CakePHP

```
$shippers = TableRegistry::getTableLocator()
        ->get('Shippers');
$query = $shippers->find();
$query->select([
    'ShipperID',
    'CompanyName',
    'NumberOfOrders' => $query->func()
    ->count('OrderID')
])
    ->leftJoinWith('Orders')
    ->group('ShipperID');
```

Here are the results:

ORM	OP1	OP2	OP3
Hibernate	$518 * 10^6 ns$	$536 * 10^6 ns$	$408 * 10^6 ns$
MyBatis	$754 * 10^6 ns$	$482 * 10^6 ns$	$484 * 10^6 ns$
EF Core	$280 * 10^7 ns$	$277 * 10^7 ns$	$311 * 10^7 ns$
SQLAlchemy	$568 * 10^5 ns$	$260 * 10^5 ns$	$271 * 10^5 ns$
Peewee	$495 * 10^2 ns$	$561 * 10^2 ns$	$854 * 10^2 ns$
CakePHP	$330 * 10^5 ns$	$344 * 10^5 ns$	$327 * 10^5 ns$

9 Final ORMs' overview

This final chapter is intended to give an overview of the differences in the ORMs considered for this study. The criteria for comparison are: performance, ease of use, documentation, community support, database support, flexibility, query language, maturity, security, and scalability.
Here is the legend:

Color	Score
	5
	4
	3
	2
	1

Parameter	**Hibernate**	**MyBatis**	**EF Core**	**NHibernate**
Performance				
Ease of Use				
Documentation				
Community Support				
Database Support				
Flexibility				
Query Language				
Maturity				
Security				
Scalability				

Paramenter	SQLAlchemy	Peewee	Django ORM	CakePHP ORM
Performance				
Ease of Use				
Documentation				
Community Support				
Database Support				
Flexibility				
Query Language				
Maturity				
Security				
Scalability				

Parameter	Doctrine	Eloquent
Performance		
Ease of Use		
Documentation		
Community Support		
Database Support		
Flexibility		
Query Language		
Maturity		
Security		
Scalability		

10 Bibliography

- Vlad Mihalcea, Steve Ebersole, Andrea Boriero, Gunnar Morling, Gail Badner, Chris Cranford, Emmanuel Bernard, Sanne Grinovero, Brett Meyer, Hardy Ferentschik, Gavin King, Christian Bauer, Max Rydahl Andersen, Karel Maesen, Radim Vansa, Louis Jacomet; Hibernate ORM 6.1.7.Final User Guide 2023
- Clinton Begin, Brandon Goodin, Christian Poitras, Eduardo Macarron, Frank Martinez, Andrew Gustafson, Hunter Presnall, Iwao Ave, Jeff Butler, Jeremy Landis, Kai Grabfelder, Larry Meadors, Marco Speranza, Nathan Maves, Putthiphong Boonphong, Simone Tripodi, Tim Chen; MyBatis 3.5.11 Documentation 2022
- Arthur Vickers, Dennis Seders, Smit Patel, Shay Rojansky, Rick Anderson, Diego Vega, Gurmeet Singh, Tom Dykstra, Maira Wenzel, Brice Lambson, Rowan Miller, Martin Milan; Entity Framework Core 7.0 Documentation 2023
- Arthur Vickers; Entity Framework and Entity Framework Core 2022
- The NHibernate Community; NHibernate 5.4 Documentation 2022
- The SQLAlchemy authors and contributors; SQLAlchemy 1.4 Documentation 2022
- Charles Leifer; Peewee 3.16 Documentation
- Django Software Foundation and individual contributors; Django 4.1 Documentation 2022
- The CakePHP community; CakePHP 4.4 Documentation 2022
- The Doctrine community; Doctrine 2.14 Documentation 2023
- The Laravel community; Eloquent Documentation 2022