Object-relational Mapping Using JPA, Hibernate and Spring Data JPA

Cătălin Tudose
Learning Management Department
Luxoft Romania
Bucharest, Romania
catalin.tudose@gmail.com

Carmen Odubăşteanu
Department of Computer Science
"POLITEHNICA" University of Bucharest
Bucharest, Romania
carmen_od@yahoo.com

***Abstract*— Object persistence means that information from the objects in an application can survive after that application ended its execution; this information can be saved and retrieved. Persistence in Java will require object-relational mapping - mapping the object instances to the tables of a relational database. There are a few alternatives to make this object-relational mapping, from which we analyze, compare, and contrast Jakarta Persistence API (JPA), Hibernate, and Spring Data JPA.**

Keywords*— *Object-relational mapping, entity, persistent class, Jakarta Persistence API, Hibernate, Spring Data JPA, MySQL

I. Introduction

Many applications require persistent data. If an information system does not save data when its execution stops, the practical use of the system will strongly diminish.

Managing persistent data may be a fundamental design decision in software projects. Can we use only SQL and stored procedures and triggers, or this problem must be addressed by Java frameworks? Should we write by ourselves even basic CRUD (create, read, update, delete) operations in SQL and JDBC, or should this work be handed to an intermediary layer? Relational Database Management Systems have proprietary SQL dialects, how can we address portability? [1] ORM *(object/relational mapping)* has now widespread. This is mainly due to Hibernate, an open-source ORM, and Spring Data, an umbrella project from the Spring family whose purpose is to unify and facilitate access to different kinds of persistence stores, including relational database systems and NoSQL databases.

Relational databases manage data integrity. This is mostly due to long-time established fundamentals of the relational data model [2][3].

Usually, data is needed and accessed much longer than the application managing them does [4]. This data independence principle will mean that the applications accessing data may change or disappear, while the logical structure of data should survive.

The JDBC API is the classical way to access databases from Java applications. One needs to create and open the connection to the database, write the query in a particular SQL dialect, bind the query parameters, execute the query, browse the results, then close the connection. So, data access is tedious and implies many boilerplate operations.

Hibernate is a project focused on managing persistent data in Java. The Hibernate suite includes projects as Hibernate ORM, Hibernate EntityManager, Hibernate Search, Hibernate Validator, Hibernate OGM, Hibernate Envers, or Hibernate Reactive [1].

Hibernate ORM belongs to the larger Hibernate framework suite. It provides a native proprietary API and a core for persisting information in databases. Hibernate ORM is at the ground of other projects belonging to the same suite. It is an independent framework and may be used on any modern Java version and eventually in combination with other frameworks (including here Spring and Jakarta Enterprise Edition) [1].

Spring Data is a family of projects belonging to the Spring framework whose purpose is to simplify access to both relational and NoSQL databases.

The Spring Data suite includes many projects, like Spring Data Commons, Spring Data JPA, Spring Data JDBC, Spring Data MongoDB, Spring Data Redis.

Spring Data JPA deals with the implementation of JPA-based repositories. It provides improved support for JPA-based data access layers by reducing the boilerplate code and creating implementations for the repository interfaces.

Hibernate and Spring Data are both frameworks. In a framework, the working flow is already included, there are hook points that we should fill out with the code to implement domain-specific functions.

A framework is more complex than a library. It defines a skeleton where the application may include its own features to fill out that skeleton. This way, the code will be called by the framework when appropriate. Developers need to focus mainly to implement the business logic.

Frameworks rely on the "Inversion of Control" idea. Libraries expose APIs that may be called from the developer's applications. When the programmer writes his own code, he will call a method belonging to a class from the library, while maintaining control of the working flow. On the contrary, the framework calls your code, thus resulting in the "Inversion of Control".

Internally, Java frameworks are heavily based on reflection capabilities.

Reflection or introspection is the capability of a program to inspect and change its structure and behavior at runtime. The Java Reflection API can view information about classes, interfaces, methods, fields, constructors, and annotations during Java program runtime. Frameworks do not need to know the names of inspected elements to do that, but may find them at runtime and alter the behavior of the program.

Due to their heavy usage inside frameworks and to the increase of the popularity of the frameworks during the last years, Java has also improved the performances of its reflection capabilities for the new versions [5].

This paper will set JPA, Hibernate and Spring Data JPA face to face, compare and contrast them and provide objective criteria about when to work with one or with another one.

2021 23rd International Conference on Control Systems and Computer Science (CSCS) | 978-1-6654-3939-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSCS52396.2021.00076

2379-0482/21/$31.00 ©2021 IEEE
DOI 10.1109/CSCS52396.2021.00076

II. PERSISTENCE WITH JPA

A. JPA application

We'll write a JPA application to save an item in the database and retrieve it. The machine running the code has MySQL Release 8.0 installed and a database named CSCS.

Let's start by installing and configuring JPA, Hibernate, and the other needed dependencies. The source code is under the control of Apache Maven. We declare these dependencies:

Listing 1 The Maven dependencies on Hibernate, JUnit Jupiter, and MySQL:

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>5.4.17.Final</version>
</dependency>
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
   <version>5.6.2</version>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.20</version>
</dependency>
```

The `hibernate-entitymanager` module includes transitive dependencies on other modules we'll need, such as hibernate-core and the JPA interface stubs.

We need the `junit-jupiter-engine` dependency to run the JUnit 5 tests, and the `mysql-connector-java` dependency, the official JDBC driver for MySQL.

We insert the *persistence unit* in our code. This one contains configuration settings, including here the persistence provider, information about the database to connect to, entity classes, SQL dialect to use, credentials, SQL formatting, and others. JPA applications need one or more persistence units.

1) The persistence unit

- The `persistence.xml` file is located in `resources/META-INF/`. Its content looks like this:

Listing 2 The `persistence.xml` configuration file

```
<provider>org.hibernate.jpa.
  HibernatePersistenceProvider</provider>          #1
<properties>
  <property name="javax.persistence.jdbc.driver"
    value="com.mysql.cj.jdbc.Driver"/>             #2
  <property name="javax.persistence.jdbc.url"
    value="jdbc:mysql://localhost:3306/CSCS"/>     #3
  <property name="javax.persistence.jdbc.user"
    value="root"/>                                 #4
  <property name="javax.persistence.jdbc.password"
    value=""/>                                     #5
  <property name="hibernate.dialect"               #6
     value="org.hibernate.dialect.MySQL8Dialect"/>
  <property name="hibernate.hbm2ddl.auto"
     value="create"/>                              #7
</properties>
```

- As JPA is only a specification, we need to indicate the vendor-specific `PersistenceProvider` implementation of the API. The persistence we define will be backed by a Hibernate provider. #1
- We indicate the JDBC properties - driver #2, URL of the database #3, username #4, and password #5 to access it. The machine we are running the programs on has MySQL 8 installed and the access credentials are the ones from persistence.xml.
- The Hibernate dialect is MySQL8 #6, as the database to interact with is MySQL Release 8.0.
- Every time the program is executed, the database will be created from scratch #7. This is ideal for automated testing when we want to work with a clean database for every test run.

2) The persistent class

- The application we implement has a persistent class, `Item`:

Listing 3 The Item class

```
@Entity                                            #1
public class Item {

   @Id                                             #2
   @GeneratedValue                                 #3
   private Long id;

   private String info;                            #4
}
```

- Persistent classes need the `@Entity` annotation #1, to indicate they correspond to a database table. If no table name is specified, this class will be mapped by default to a table called `ITEM`.
- Persistent classes need an attribute annotated with `@Id` #2, to correspond to the primary key of the database table. This attribute will be mapped to a column named `ID`.
- The `@GeneratedValue` annotation indicates the automatic generation of primary keys #3. Many strategies may be used here, depending on the database: identity fields, values obtained from a sequence, or a table.
- Attributes will be mapped to columns in the table. As no column name is specified, `info` attribute #4 will be mapped by default to a column called `INFO`.

3) Saving and retrieving items

- Let's save a new `Item` to the database.

Listing 4 The ItemJPATest class

```
public class ItemJPATest {

    @Test
    void saveRetrieveItem() {
        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory(
                    "cscs");                       #1

        try {
           EntityManager em =
               emf.createEntityManager();          #2
           em.getTransaction().begin();            #3
           Item item = new Item();                 #4
           item.setInfo("Item");                   #4
           em.persist(item);                       #5
           em.getTransaction().commit();           #6
```

```
            em.close();                                  #7

        } finally {
            emf.close();                                 #8
        }
    }

}
```

- We need an `EntityManagerFactory` to access the database #1. This factory corresponds to the persistence unit that we have previously defined.
- Create an `EntityManager` and start a new database session #2.
- Get access to the standard transaction API and begin a transaction #3.
- Create a new instance of the `Item` class, and set its `info` property #4.
- The transient instance becomes persistent on the side of the persistence context #5.
- Commit the transaction #6. A new row is inserted in the ITEM table. Access to the database must be transactional, even if it is read-only.
- As we created an `EntityManager`, we must close it #7.
- As we created an `EntityManagerFactory`, we must close it #8.
- There is no SQL code and no JDBC usage in the JPA application. There are no CRUD (create, read, update, delete) operations inside the Java code, but only working object-oriented way, with classes, objects, and methods. The translation to each SQL dialect is made by the ORM which also addresses portability.

III. Native Hibernate configuration

The `org.hibernate.SessionFactory` Hibernate interface is the native correspondent of the JPA `EntityManagerFactory`.

To configure the native Hibernate, we may use a `hibernate.properties` Java properties file, or a `hibernate.cfg.xml` XML file. We chose the second option, and the configuration contains database and session-related options. This XML file is generally placed under the folder `src/main/resource` or `src/test/resource`. As we need the information for the Hibernate configuration in the tests, we chose the second location alternative.

Listing 5 The `hibernate.cfg.xml` configuration file

```
<hibernate-configuration>                                #1
  <session-factory>                                      #2
    <property
       name="hibernate.connection.driver_class">
            com.mysql.cj.jdbc.Driver                     #3
    </property>
    <property name="hibernate.connection.url">
          jdbc:mysql://localhost:3306/CSCS               #4
    </property>
    <property name="hibernate.connection.username">
       root</property>                                   #5
    <property name="hibernate.connection.password">
       </property>                                       #6
    <property
       name="hibernate.connection.pool_size">
       50</property>                                     #7
    <property name="hibernate.hbm2ddl.auto">
       create</property>                                 #8
  </session-factory>
</hibernate-configuration>
```

- First, we use the tags to indicate the fact that we are configuring Hibernate #1, more exactly the `SessionFactory` object #2. `SessionFactory` is an interface and we need one `SessionFactory` to interact with one database.
- We indicate the JDBC properties - driver #3, URL of the database #4, username #5, and password #6 to access it.
- We limit the number of connections waiting in the Hibernate database connection pool to 50 #7.
- Every time the program is executed, the database will be created from scratch #8. This is ideal for automated testing when we want to work with a clean database for every test run.

Let's save an `Item` to the database using native Hibernate.

Listing 6 The `ItemHibernateTest` class

```
public class ItemHibernateTest {

   private static SessionFactory
     createSessionFactory() {
        Configuration configuration = new
            Configuration();                             #1
     configuration.configure().
        addAnnotatedClass(Item.class);                   #2
     ServiceRegistry serviceRegistry = new
          StandardServiceRegistryBuilder().              #3
     applySettings(configuration.
     getProperties()).build();                           #3
     return configuration.
        buildSessionFactory(serviceRegistry);            #4
  }

  @Test
  void saveRetrieveItem() {
   try (SessionFactory sessionFactory =
          createSessionFactory();                        #5
        Session session =
          sessionFactory.openSession()){                 #6

        session.beginTransaction();                      #7
        Item item = new Item();                          #8
        item.setInfo(
               "Item from Hibernate");                   #8
        session.persist(item);                           #9
        session.getTransaction().commit();               #10
   }
}
```

- To create a `SessionFactory`, we need to create a `Configuration` #1, to call the `configure` method on it, and to add `Item` to it as annotated class #2. The execution of the `configure` method will load the content of the default `hibernate.cfg.xml` file.
- We create and configure the service registry #3. A `ServiceRegistry` hosts and manages services that need access to the `SessionFactory`.
- We build a `SessionFactory` using the configuration and the service registry we have previously created #4.

- The `SessionFactory` created with the `createSessionFactory` method we have previously defined is passed as an argument to a `try` with resources, as `SessionFactory` implements the `AutoCloseable` interface #5. Similarly, we begin a new session with the database by creating a `Session` #6, which also implements the `AutoCloseable` interface. This is our context for all persistence operations.
- Get access to the standard transaction API and begin a transaction on this thread of execution #7.
- Create a new instance of `Item` class, and set its info property #8.
- The transient instance becomes persistent on the side of the persistence context #9.
- Synchronize the session with the database and close the current session on commit of the transaction automatically #10.
- As in the case of JPA, there is no SQL code and no JDBC usage. There are no CRUD (create, read, update, delete) operations inside the Java code, but only working object-oriented way, with classes, objects, and methods. The translation to each SQL dialect is made by the ORM which also addresses portability.

IV. Switching between JPA and Hibernate

It is possible that we are working with JPA and we need to access the Hibernate API. Or, vice-versa, we work with Hibernate native and we need to create an `EntityManagerFactory` from the Hibernate configuration.

To obtain a `SessionFactory` from an `EntityManagerFactory`, we'll have to unwrap the first one from the second one.

Listing 7 Obtaining a `SessionFactory` from an `EntityManagerFactory`

```
private static SessionFactory getSessionFactory
(EntityManagerFactory entityManagerFactory) {
   return entityManagerFactory.
          unwrap(SessionFactory.class);
}
```

Starting with JPA version 2.0, we may get access to the APIs of the underlying implementations. The `EntityManagerFactory` (and also the `EntityManager`) declare an `unwrap` method that will return objects belonging to the classes of the JPA implementation. When using the Hibernate implementation, we may get the corresponding `SessionFactory` or `Session` objects.

We may be interested in the reverse operation: create an `EntityManagerFactory` from an initial Hibernate configuration.

Listing 8 Obtaining an `EntityManagerFactory` from a Hibernate configuration

```
private static EntityManagerFactory
      createEntityManagerFactory() {
    Configuration configuration = new
       Configuration();                          #1
    configuration.configure().
       addAnnotatedClass(Item.class);            #2

    Map<String, String> properties =
          new HashMap<>();                       #3
    Enumeration<?> propertyNames =
          configuration.getProperties().
          propertyNames();                       #4
    while (propertyNames.hasMoreElements()) {
        String element = (String)
          propertyNames.nextElement();           #5
        properties.put(element,
          configuration.getProperties().
          getProperty(element));                 #5
    }
    return Persistence.
        createEntityManagerFactory("cscs",
             properties);                        #6
}
```

- We create a new Hibernate configuration #1, then call the `configure` method, which adds the content of the default `hibernate.cfg.xml` file to the configuration.
- We explicitly add `Item` as annotated class #2.
- We create a new hash map to be filled in with the existing properties #3.
- We get all property names from Hibernate configuration #4, then we add them one by one to the previously created map #5.
- We return a new `EntityManagerFactory`, providing to it the `cscs` persistence unit name and the previously created map of properties #6.

V. PERSISTENCE with Spring Data JPA

In this section, we'll write a Spring Data JPA application, which saves an item in the database and then retrieves it.

We add the Spring dependencies on the side of the Apache Maven configuration.

Listing 9 The Maven dependencies on Spring

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-jpa</artifactId>
  <version>2.3.3.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
```

The `spring-data-jpa` module provides the repository support for JPA and includes transitive dependencies on other modules we'll need, such as `spring-core` and `spring-context`.

We also need the `spring-test` dependency, to run the tests with the help of the Spring extension.

The standard configuration file for Spring Data JPA is a Java class to create and setup the beans needed by Spring Data. The configuration can be done using either an XML file or Java code and we chose this second alternative. We create the following configuration file for the persistence application:

Listing 10 The `SpringDataConfiguration` class

```
@EnableJpaRepositories
                ("cscs23.orm.repositories")       #1
public class SpringDataConfiguration {
    @Bean
    public DataSource dataSource() {              #2
      DriverManagerDataSource dataSource =
         new DriverManagerDataSource();           #2
      dataSource.setDriverClassName
            ("com.mysql.cj.jdbc.Driver");         #3
      dataSource.setUrl
          ("jdbc:mysql://localhost:3306/CSCS");   #4
      dataSource.setUsername("root");             #5
      dataSource.setPassword("");                 #6
      return dataSource;                          #2
    }

    @Bean
    public JpaTransactionManager transactionManager
           (EntityManagerFactory emf) {
      return new JpaTransactionManager(emf);      #7
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
      HibernateJpaVendorAdapter jpaVendorAdapter =
         new HibernateJpaVendorAdapter();         #8
      jpaVendorAdapter.
        setDatabase(Database.MYSQL);              #9
      return jpaVendorAdapter;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean
     entityManagerFactory() {
      LocalContainerEntityManagerFactoryBean
       localContainerEntityManagerFactoryBean =
       new
       LocalContainerEntityManagerFactoryBean();#10
       localContainerEntityManagerFactoryBean.
          setDataSource(dataSource());          #11
       localContainerEntityManagerFactoryBean.
          setJpaVendorAdapter(
           jpaVendorAdapter());                 #12
        localContainerEntityManagerFactoryBean.
          setPackagesToScan(
            "cscs23.orm");                      #13
       return
         localContainerEntityManagerFactoryBean;
    }
}
```

- The `@EnableJpaRepositories` annotation will scan the package of the annotated configuration class for Spring Data repositories #1.
- We create a data source bean #2 and we indicate the JDBC properties - driver #3, URL of the database #4, the username #5, and password #6 to access it.
- We create a transaction manager bean based on an entity manager factory #7. Every interaction with the database should occur within transaction boundaries and Spring Data needs a transaction manager bean.
- We create a JPA vendor adapter bean that is needed by JPA to interact with Hibernate #8. We configure this vendor adapter to access a MySQL database #9.
- We create an object belonging to the class `LocalContainerEntityManagerFactoryBean` – this is a factory bean that produces an `EntityManagerFactory` following the JPA standard container bootstrap contract #10.
- We set the data source #11, the vendor adapter #12, and the packages to scan for entity classes #13. As the `Item` entity is located in `cscs23.orm`, we set this package to be scanned.

Spring Data JPA provides support for JPA-based data access layers reducing the boilerplate code and creating implementations for the repository interfaces. We need only to define our own repository interface, to extend one of the Spring Data interfaces.

Listing 11 The `ItemRepository` interface

```
public interface ItemRepository extends
CrudRepository<Item, Long> {

}
```

The `ItemRepository` interface extends `CrudRepository<Item, Long>`. This means that it is a repository of `Item` entities, having a `Long` identifier. Remember, the `Item` class has an `id` field annotated as `@Id` of type `Long`. We can directly call methods as `save`, `findAll` or `findById`, inherited from `CrudRepository` and we can use them without additional information, to execute operations against a database.

For additional operations, one may use the `JpaRepository` interface that in turn extends the `CrudRepository` interface. This one comes with more additional methods, including here methods that allow pagination and sorting or batch operations.

Spring Data JPA uses the proxy pattern to provide a mechanism to interact with the database. A proxy is a substitute that will have the purpose to replace another object. Spring Data JPA will create a proxy class implementing the `ItemRepository` interface and will create its methods (figure 1). This proxy class will define all the methods inherited from `CrudRepository`, including here `save`, `findAll` or `findById`. The programmer will no longer waste time writing the logic of these methods – their implementation will come from the side of Spring Data JPA. This acts in the spirit of frameworks and of Spring in particular – invert the control, the working flow is already included, provide hook points that the programmer should fill out by focusing on the business logic.

Let's save an `Item` to the database using Spring Data JPA.

Listing 12 The `ItemSpringDataJPATest` class

```
@ExtendWith(SpringExtension.class)                #1
@ContextConfiguration(classes = {
  SpringDataConfiguration.class})                 #2
public class ItemSpringDataJPATest {

  @Autowired
  private ItemRepository itemRepository;          #3

  @Test
  void saveRetrieveItem() {
    Item item = new Item();                       #4
    item.setInfo(
      "Item from Spring Data JPA");               #4
    itemRepository.save(item);                    #5
  }
}
```

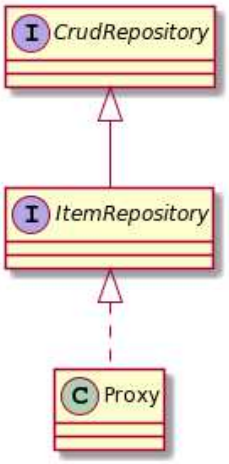

Fig. 1 The Spring Data JPA Proxy class implements the `ItemRepository` interface

- We extend the test using `SpringExtension` #1. This extension is used to integrate the Spring test context with the JUnit 5 Jupiter test.
- The Spring test context is configured using the beans defined in the previously presented `SpringDataConfiguration` class #2.
- An `ItemRepository` bean is injected by Spring through auto-wiring #3. This is possible as the `cscs23.orm.repositories` package where `ItemRepository` is located was used as the argument of the `@EnableJpaRepositories` annotation in listing 10.
- If we would like to check and call `itemRepository.getClass()`, we'll see that it is something like `com.sun.proxy.$Proxy41` – a proxy generated by Spring Data JPA, as explained in figure 1.
- Create a new instance of the `Item` class, and set its info property #4.
- Persist the item object #5. The `save` method is inherited from the `CrudRepository` interface and its body will be generated by Spring Data JPA when the proxy class is created. It will simply save an `Item` entity to the database.
- The advantages of working with JPA or Hibernate native are still here: no SQL code or JDBC needed; no CRUD operations inside the Java code, but only classes, objects, and methods; the portability to each SQL dialect is addressed by the ORM.
- Additionally, the Spring Data JPA test is considerably shorter than the ones using JPA or Hibernate native. This is because the boilerplate code has been removed – no more explicit object creation or explicit control of the transactions. The repository object is injected and it provides the generated methods of the proxy class. The burden is heavier now on the configuration side – but this should be done only once per application.

VI. Comparing the Approaches of Persisting Entities

We have followed the implementation of an application interacting with a database and that uses, alternatively, JPA, Hibernate native, and Spring Data JPA. Our purpose was to analyze each approach and how the needed configuration and the code to be written varies.

With JPA, we used its general API and needed a persistence provider. We may switch between persistence providers from the configuration.

We needed explicit management of the `EntityManagerFactory`, `EntityManager`, and transactions. The way the configuration is done and the amount of code to be written is similar to the Hibernate native approach. We may switch to the JPA approach by constructing an `EntityManagerFactory` from a Hibernate native configuration.

With Native Hibernate, we used the specific Hibernate native API. We built its configuration starting from the default Hibernate configuration files (`hibernate.cfg.xml` or `hibernate.properties`).

We needed explicit management of the `SessionFactory`, `Session`, and transactions. The way the configuration is done and the amount of code to be written is similar to the JPA approach. We may switch to the Hibernate native approach by unwrapping a `SessionFactory` from an `EntityManagerFactory` or a `Session` from an `EntityManager`.

With Spring Data JPA, we needed the additional Spring Data dependencies in the project. The configuration part also takes care of the creation of the beans needed for the project, including the transaction manager.

The repository interface needs only to be declared, and Spring Data will create an implementation for it as a proxy class with generated methods that interact with the database. We may use these methods directly, as they are generated by the framework, without taking care of defining them by ourselves.

The needed repository is injected and not explicitly created by the programmer. The amount of code to be written is the shortest from all these approaches, as the configuration part has taken most of the burden.

Looking back to the questions from the beginning of the article, we notice that Java frameworks address the problem of persisting information to the database in a more transparent way. Using Java frameworks, there is no more need to hand-code the CRUD (create, read, update, delete) operations in SQL and JDBC, this work is now managed by the ORM intermediary layer. ORM addresses portability, in the world of Relational Database Management Systems having proprietary dialects.

VII. Comparing the Performances of Persisting Entities

We'll analyze the performances of persisting entities through the three approaches, considering the number of needed dependencies and JAR files to run the project, the number of lines of code to write, and the execution times.

Working with a particular framework will mean that its classes have to be imported into our code and also have to be

accessible on the classpath. Fewer classes on the classpath will also mean a lower memory footprint.

Table 1 shows the number of needed Maven dependencies, needed JAR files on the classpath (including transitive dependencies and excluding the MySQL and JUnit ones) and the number of lines of code to be written to insert, update, retrieve and delete a batch of records.

	Hibernate	JPA	Spring Data JPA
Maven dependencies	1	1	2
Needed JAR files	22	29 (22 from Hibernate)	62
Lines of code	47	57	29

Table 1 Maven dependencies, JAR files and lines of code to be written

To analyze the running times, we executed a batch of insert, update, retrieve and delete operations using the three approaches, progressively increasing the number of records from 1000 to 50000. Tests were made on Windows 10 Enterprise, running on a 4 cores Intel i7-5500U processor at 2.40GHz and 8 GB RAM. The results may be examined in tables 2-5 and figures 2-5.

Number of records	Hibernate	JPA	Spring Data JPA
1000	1138	1127	2288
5000	3187	3307	8410
10000	5145	5341	14565
20000	8591	8488	26313
30000	11146	11859	37579
40000	13011	13300	48913
50000	16512	16463	59629

Table 2 Insert execution times by framework (times in ms)

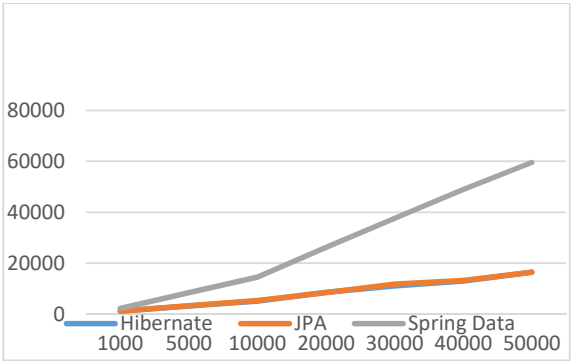

Fig. 2 Insert execution times by framework (times in ms)

Number of records	Hibernate	JPA	Spring Data JPA
1000	706	759	2683
5000	2081	2256	10211
10000	3596	3958	17594
20000	6669	6776	33090
30000	9352	9696	46341
40000	12720	13614	61599
50000	16276	16355	75071

Table 3 Update execution times by framework (times in ms)

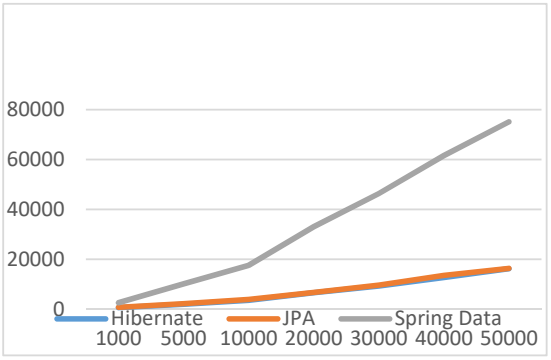

Fig. 3 Update execution times by framework (times in ms)

Number of records	Hibernate	JPA	Spring Data JPA
1000	251	232	338
5000	214	192	401
10000	223	210	718
20000	246	248	1099
30000	276	308	1475
40000	311	297	1964
50000	362	344	2252

Table 4 Retrieve execution times by framework (times in ms)

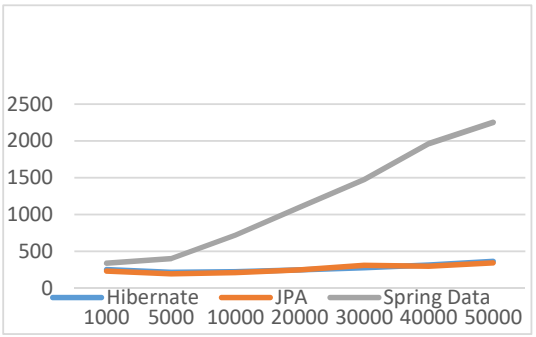

Table 4 Retrieve execution times by framework (times in ms)

Number of records	Hibernate	JPA	Spring Data JPA
1000	584	551	2430
5000	1537	1628	9685
10000	2992	2763	17930
20000	5344	5129	32906
30000	7478	7852	47400
40000	10061	10493	62422
50000	12857	12768	79799

Table 5 Delete execution times by framework (times in ms)

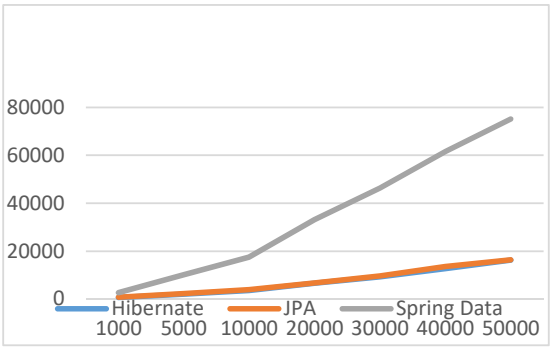

Fig. 5 Delete execution times by framework (times in ms)

VIII. CONCLUSIONS

The three approaches provide different performances from the points of view that were under analysis. The ease of development and the code reduction of Spring Data JPA is impressive (half of the number of lines of code of JPA), but it comes at a price. It requires more dependencies and more JAR files on the classpath (almost 3 times more than for Hibernate). The JPA solution needs all the Hibernate dependencies, plus its own.

There are remarkable notes regarding the execution times. Hibernate and JPA go head to head, the graphics of their times almost overlap for all four operations (insert, update, retrieve and delete). Even if JPA comes with its own API on top of Hibernate, this additional layer introduces no overhead.

The execution times of Spring Data JPA insertions start from about 2 times more than Hibernate and JPA for 1000 records to about 3.5 times more for 50000 records. The overhead of the Spring Data JPA framework is considerable.

For Hibernate and JPA, the update and delete execution times decrease in comparison with the insert execution times. On the contrary, the Spring Data JPA update and delete execution times increase in comparison with the insert execution times.

For Hibernate and JPA, the retrieve times grow very slowly with the number of rows. The Spring Data JPA retrieve execution times strongly grow with the number of rows.

Using Spring Data JPA is justified in particular situations: the project already uses the Spring framework and needs to rely on its existing paradigm (e.g. inversion of control, automatically managed transactions); there is a strong need to decrease the amount of code and thus shorten the development time; the number of manipulated rows at a time is small.

IX. SUMMARY

This article has focused on alternatives for working with a database from a Java application: JPA, Hibernate native, and Spring Data JPA and provided examples for each of them.

We implemented Java persistence code using three alternatives: JPA, Hibernate native, and Spring Data JPA.

We created a persistent class and its mapping with annotations.

Using JPA, we implemented the configuration and bootstrap of a persistence unit, and how to create the `EntityManagerFactory` entry point. Then we called the `EntityManager` to interact with the database, storing and loading an instance of the persistent domain model class.

We demonstrated some of the native Hibernate bootstrap and configuration options, as well as the equivalent basic Hibernate APIs, `SessionFactory`, and `Session`.

We examined how we can switch between the JPA approach and the Hibernate approach.

We implemented the configuration of a Spring Data JPA application, created the repository interface, then used it to save an instance of the persistent class.

We finally compared and contrasted these three approaches: JPA, Hibernate native, and Spring Data JP using a few criteria: the number of needed dependencies, the number of JAR files on the classpath, the number of lines of code to be written, and the execution times for batch processing (insert, update, retrieve and delete operations).

Tables 6 and 7 summarize the pros and cons of each approach.

	Pros
Hibernate	• Fewest needed JAR files • Less memory footprint
JPA	• Easy to switch to another persistence provider • No overhead compared with native Hibernate
Spring Data JPA	• The fewest lines of code • Highest development speed • Automatic management of transactions

Table 6 Pros of each of the three approaches

	Cons
Hibernate	• Hard to replace with another persistence provider
JPA	• The largest amount of code to be written
Spring Data JPA	• Big overhead, slowest of all frameworks for batch operations

Table 7 Cons of each of the three approaches

REFERENCES

[1] Christian Bauer, Gavin King, and Gary Gregory, Java Persistence with Hibernate, Second Edition. Manning, 2015

[2] Codd, E.F., "A relational model of data for large shared data banks." [Communications of the ACM 13 (6): 377-87, 1970].

[3] Date, C.J., SQL and Relational Theory: How to Write Accurate SQL Code, 3rd edition. O'Reilly Media, 2015.

[4] Ramez Elmasri and Shamkant Navathe, Fundamentals of Database Systems. Pearson, 2016.

[5] Radu Şerban, Cătălin Tudose, Carmen Odubăşteanu, Java Reflection Performance Analysis Using Different JDKs, CSCS 18, 2011