Innovative Solutions Application Report

Table of Contents

Introduction

The following paper illustrates the design steps and the choices made for the development of an application based on a relational database devised to meet the data management requirements of a fictional tech company named *Innovative Solution*.

The first part of the paper covers the application's functional and non-functional requirements from which the application's specifications are derived, consisting among other things of the actors involved and their use cases, the application dataflow, and an overall description of the software architecture.

The second part of the paper presents the object-oriented design of the software, where the relevant classes for the application and their relationships are defined, along with their POJO implementations.

The third part of the paper covers the details of how the *Hibernate* implementation of the *Java Persistence API* can be used in order to persist the data of the application in an underlying MySQL database, with the derivation of the POJO classes previously defined into *persistence entities*, how their relationships are implemented, and a copy the *persistence.xml* file used in the application.

The fourth and final part of the paper presents the study of the adoption of a Key-Value database in the application, with its feasibility study and considerations, its design, and how the data consistency between the Key-Value and the MySQL database is managed.

Requirements

Innovative Solutions is a company whose core business consists in the retailing of electronic IoT-oriented products assembled in-house by teams of employees from components purchased from various suppliers.

Functional Requirements

The company requires a software solution which should be used both internally by its system administrator and team leaders to oversee and control operations in the company and by the company's customers, who will be allowed to browse and purchase the products available in stock, as well as review their past orders.
More precisely, the software solution should:

- Provide a login system based on the username/password model, through which to distinguish the different actors of the application (namely the system administrator, the team leaders, and the customers).
- Allow the System Administrator to browse all the users of the application, add and delete a user, and update the salary of an employee.
- Allow the Team Leaders to browse information on the members of their teams and the products the teams assemble, as well as adding new finished products to the company's stock.
- Allow the company's Customers to view the products available in stock, purchase them, and review their past orders.

Non-Functional Requirements

Since the application should not require computer expertise to be used, especially by customers, to enhance the non-functional requirement of usability a graphical interface should be provided for the application.

In addition, due to the company's system administrator strong background in object-oriented programming, the application should be developed attuning to an object-oriented paradigm, concealing as much as possible the data persistence aspects.

Lastly, given its key role in the business's activities, the application should guarantee a professional-grade quality of service (QoS), both in terms of fast response times, especially for the customers, and in terms of tolerance to single points of failure.

Working Hypotheses

The design of the application will also rely on the following working hypotheses:

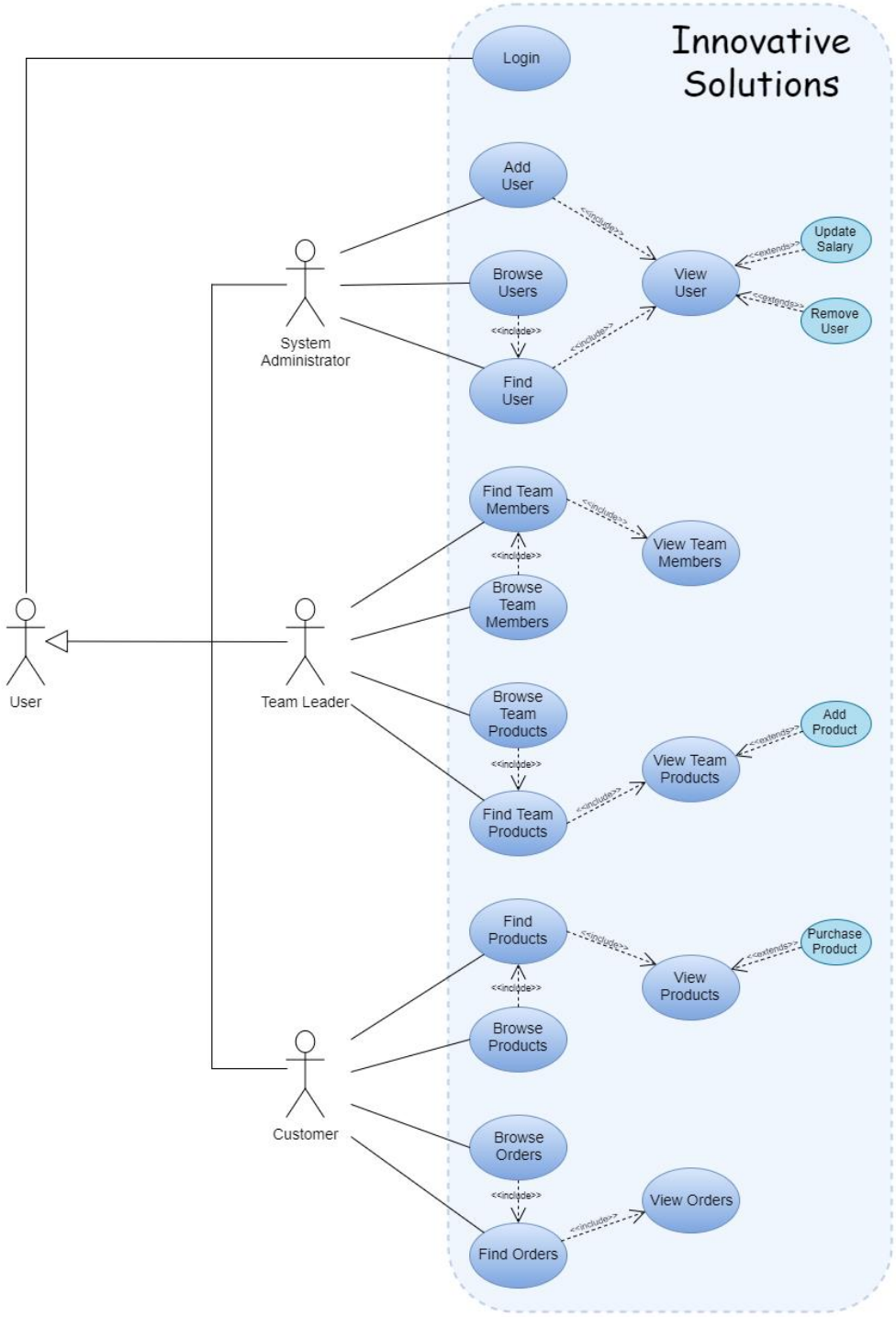
- *Customers* may purchase any *Products* up to their available quantities
- Each *Employee* in the company may belong to up to one *Team*
- Each *Team* is composed of at least one member, representing its leader
- Each category of *Product* offered by the company is assembled by a single team

Specification

Actors and Use Cases Diagram

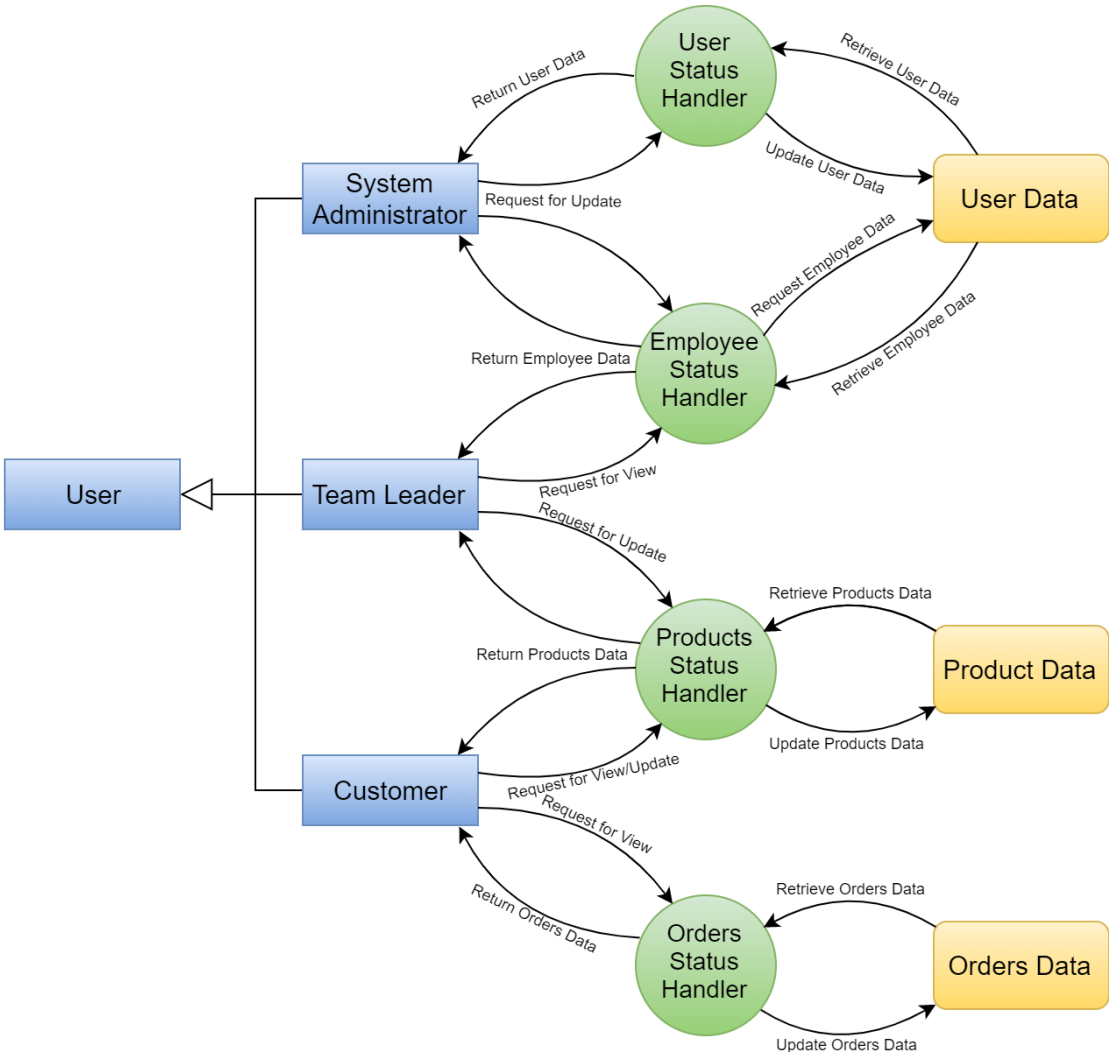
Given the software functional requirements, the application is meant to be used by three different actors, each being allowed to perform a different set of operations:

- The company's system administrator, who is permitted to perform a number of actions requiring a high level of privilege, such as inserting or removing a customer or an employee from the database or updating the salary of an employee.
- The team leaders, who are allowed to add finished products to the company stock and review information on the member assigned to their team.
- The customers, who may view and purchase the products on offer, as well as review their past orders.



Application Dataflow

The projected dataflow of the application is outlined below:



Software Architecture

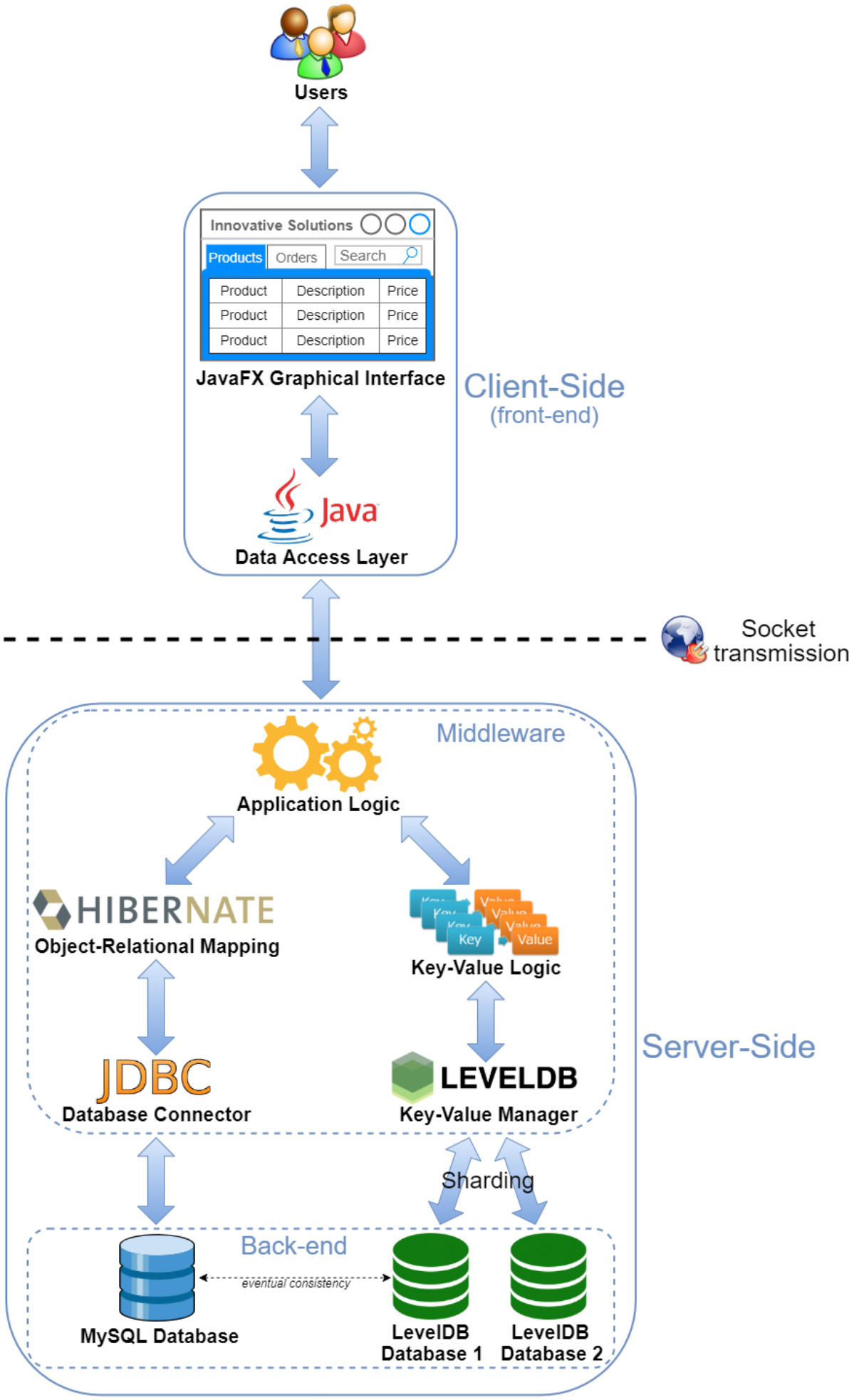
The application will be developed attuning to a *Client-Server* paradigm using the *Java 8* programming language, where:

- The client-side will consist in a graphical interface developed through the JavaFX API to allow users to interact with the application, along with the modules required to exchange data with the application's back-end.

- The server-side will comprise the main logic and the data persistence aspects of the application where:

    - The primary database will consist in a relational MySQL database, where the *Hibernate* implementation of the *Java Persistence API* will be used in order to allow us to transparently exchange the data between the application and the database.

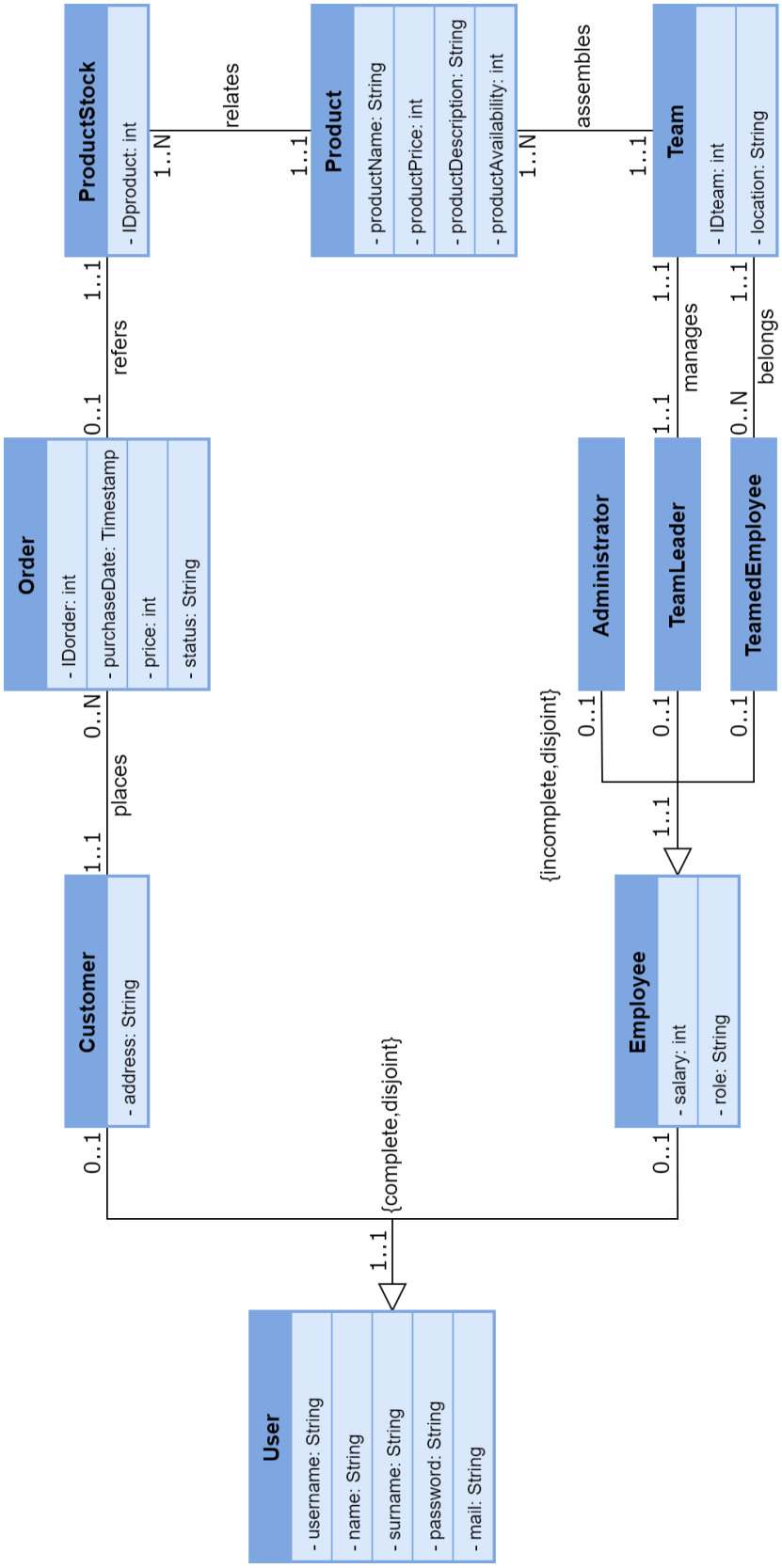
    - In order to improve the access times and the fault-tolerance of the application, a partition of the primary database will be replicated in a pair of key-value databases managed through the *Google LevelDB API*, where the data will be sharded so as to balance the read/write operations, with an *eventual consistency* mechanism with the primary relational database being also provided.


Users
Innovative Solutions
Products
Orders
Search
Product
Description
Price
JavaFX Graphical Interface
Client-Side
(front-end)
Java
Data Access Layer
Socket
transmission
Middleware
Application Logic
HIBERNATE
Object-Relational Mapping
Key
Value
Key-Value Logic
JDBC
Database Connector
LEVELDB
Key-Value Manager
Server-Side
Sharding
Back-end
eventual consistency
MySQL Database
LevelDB
Database 1
LevelDB
Database 2

Object-Oriented Application Design

Analysis Classes Diagram

From the application's requirements and the additional working hypotheses the following classes can be identified as the first step in the application's design:



Classes Definitions

CLASS	DESCRIPTION
User	An individual of interest in the context of the application, who may be a customer or an employee of the company
Customer	A customer of the company, who may purchase products and review their orders
Employee	An employee of the company who, other than a generic employee, can represent its system administrator, a team leader, or a team member
Administrator	The company's system administrator
TeamLeader	The leader of a team of employees
TeamedEmployee	A member of a team of employees
Team	A team in the company, which is composed of a team leader and other members, where each team is assigned the assembly of one or more products
Product	A specific category of product offered by the company, which can be purchased by customers
ProductStock	A physical instance of a product sold by the company, which is related to a specific product category and is sold to customers
Order	An order placed by a customer, which is relative to a physical instance of a product sold by the company

Classes Attributes

User		
Attribute	**Type**	**Description**
username	String	A unique string identifying a user, which is also used by them to access the application
surname	String	The user's name
surname	String	The user's surname
password	String	The password required for the user to access the application
mail	String	The user's e-mail address

Customer		
Attribute	**Type**	**Description**
address	String	The customer's address, which is used for product shipping purposes

Employee		
Attribute	**Type**	**Description**
salary	int	The employee's salary
role	String	The employee's role in the company

Administrator		
Attribute	**Type**	**Description**
none (same as the superclass)		

TeamLeader		
Attribute	**Type**	**Description**
none (same as the superclass)		

TeamedEmployee		
Attribute	**Type**	**Description**
none (same as the superclass)		

Team		
Attribute	**Type**	**Description**
IDteam	int	The team's unique identifier
location	String	The team's working location

Product		
Attribute	**Type**	**Description**
productName	String	The product's name
productPrice	int	The product's current asking price
productDescription	String	A brief description of the product
productAvailability	int	The product's current stock availability

ProductStock		
Attribute	**Type**	**Description**
IDstock	int	The unique identifier of an instance of a product category

Order		
Attribute	**Type**	**Description**
IDorder	int	The order's unique identifier
purchaseDate	Timestamp	The date and time the order was placed
price	int	The price of the product relative to the order
status	String	The status of the order's shipment (received, shipping, delivered)

Classes POJO Implementations

As the first step in the application's development, the following represent the POJO implementations of the classes of our application:

User.java

```
public class User
{
 private String username;
 private String name;
 private String surname;
 private String password;
 private String mail;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Customer.java

```
public class Customer extends User
{
 private String address;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Employee.java

```
public class Employee extends User
{
 private int salary;
 private String role;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Administrator.java

```
public class Administrator extends Employee
{
 //Same attributes as the superclass

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

TeamLeader.java

```
public class TeamLeader extends Employee
{
 //Same attributes as the superclass

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

TeamedEmployee.java

```
public class TeamedEmployee extends Employee
{
 //Same attributes as the superclass

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Team.java

```
public class Team
{
 private int IDteam;
 private String location;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Product.java

```
public class Product
{
 private String productName;
 private int    productPrice;
 private String productDescription;
 private int    productAvailability;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

ProductStock.java

```
public class ProductStock
{
 private int IDstock;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Order.java

```
public class Order
{
 private int IDorder;
 private Timestamp purchaseDate;
 private int price;
 private String status;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

JPA-based Database Implementation

POJO classes derivation into persistence entities

As the first step of the JPA-based relational database implementation, the POJO classes previously defined must be derived into *persistence entities,* which is obtained by complementing their definitions with the appropriate JPA annotations.
It should also be noted that regarding the inheritance we have opted, as a design choice, to map the User superclass and all its subclasses (Customer, Employee, Administrator, TeamLeader, TeamedEmployee) into a single table in the database, a solution which provides the best performance in terms of number of accesses required to retrieve the state of an entity, at the cost of a higher memory footprint, given that the attributes specific to each subclass are repeated as fields in all other subclasses.

User.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "usertype")
@Table(name="Users")
public class User
{
 @Id
 @Column( name = "username", length = 45, nullable = false )
 private String username;

 @Column( name = "name", length = 45, nullable = false )
 private String name;

 @Column( name = "surname", length = 45, nullable = false )
 private String surname;

 @Column( name = "password", length = 45, nullable = false )
 private String password;

 @Column( name = "mail", length = 45, nullable = false )
 private String mail;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Customer.java

```
@Entity
public class Customer extends User
{
 @Column( name = "mail", length = 45, nullable = false )
 private String address;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Employee.java

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Employee extends User
{
 @Column( name = "salary", nullable = true )
 private int salary;

 @Column( name = "role", length = 45, nullable = true )
 private String role;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Administrator.java

```java
@Entity
public class Administrator extends Employee
{
 //Same attributes as the superclass

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

TeamLeader.java

```java
@Entity
public class TeamLeader extends Employee
{
 //Same attributes as the superclass

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

TeamedEmployee.java

```java
@Entity
public class TeamedEmployee extends Employee
{
 //Same attributes as the superclass

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Team.java

```java
@Entity
@Table(name="Teams")
public class Team
{
 @Id
 @Column( name="IDteam", nullable = false )
 private int IDteam;

 @Column( name="location", length = 45, nullable = false )
 private String location;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Product.java

```java
@Entity
@Table(name = "Products")
public class Product
{
 @Id
 @Column( name="productName", length = 45, nullable = false )
 private String productName;

 @Column( name = "productPrice", nullable = false )
 private int productPrice;

 @Column( name = "productDescription", length = 200, nullable = false )
 private String productDescription;

 @Column( name = "productAvailability", nullable = false )
 private int productAvailability;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

ProductStock.java

```
@Entity
@Table(name="ProductStock")
public class ProductStock
{
 @Id
 @Column( name = "IDstock" , nullable = false )
 private int IDstock

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Order.java

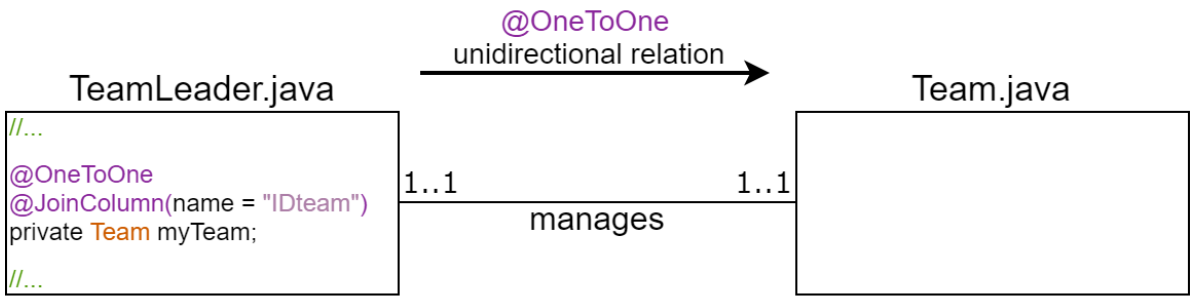
```
@Entity
@Table(name="orders")
public class Order
{
 @Id
 @Column( name = "IDorder", nullable = false )
 private int IDorder;

 @Column( name = "purchaseDate", nullable = false )
 private Timestamp purchaseDate;

 @Column( name = "price", nullable = false )
 private int price;

 @Column( name = "status", length = 45, nullable = false )
 private String status;

 /* Constructors, Getters and Setters (omitted for brevity) */
}
```

Entities Relationships Implementation

Next we present our design choices regarding the implementation of the relationships between the persistence entities:

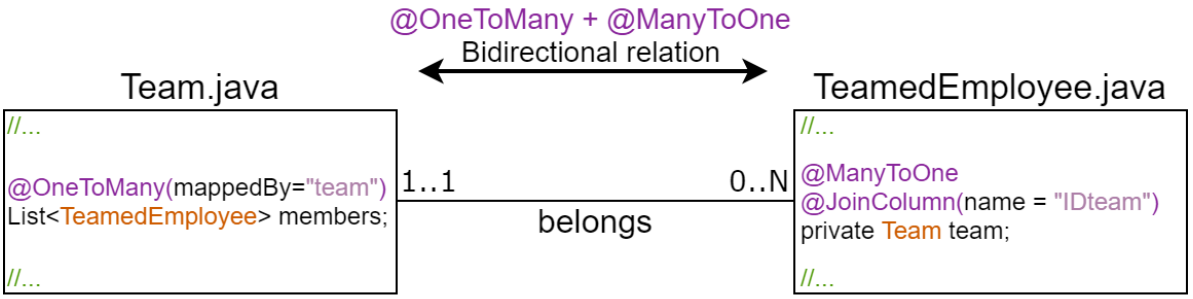
- ### TeamLeader ⇌ Team (One to One)

  This relationship is implemented as a OneToOne unidirectional relationship between the *TeamLeader* and *Team* entities, which is obtained by adding to the former an "IDteam" attribute identifying the Team managed by the TeamLeader.
  The relationship is implemented as unidirectional since from the application's specification, and more precisely from its use cases, it is required to retrieve the Team managed by a TeamLeader, but not vice versa.



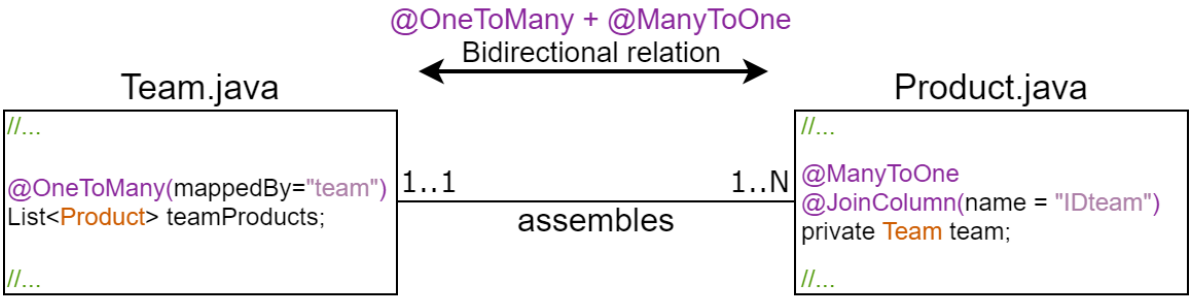
- ## Team ⇌ TeamedEmployee (One to Many)

  This relationship is implemented as a OneToMany bidirectional relationship between the *Team* and *TeamedEmployee* entities, which is obtained by adding to the former a "members" attribute identifying the list of TeamedEmployees that belong to the Team and by adding to the latter an "IDTeam" attribute identifying the Team the TeamedEmployee belongs to.
  The relationship is implemented as bidirectional since in our application it is necessary to retrieve both the list of TeamedEmployees belonging to a Team and the Team a TeamedEmployee belongs to.



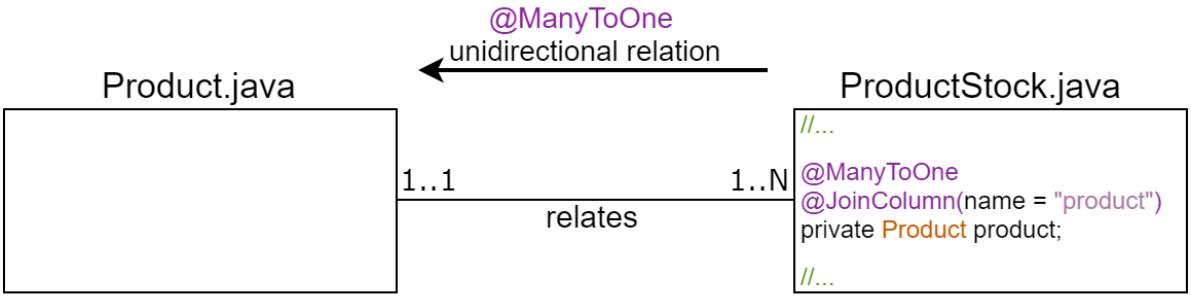
- ## Team ⇌ Product (One to Many)

  This relationship is implemented as a OneToMany bidirectional relationship between the *Team* and *Product* entities, which is obtained by adding to the former a "teamProducts" attribute identifying the list of Products assembled by the Team and by adding to the latter an "IDTeam" attribute identifying the Team in charge of its assembly.
  The relationship is implemented as bidirectional since in our application it is necessary to retrieve both the list of products assembled by a Team and which Team is in charge of the assembly of a Product.



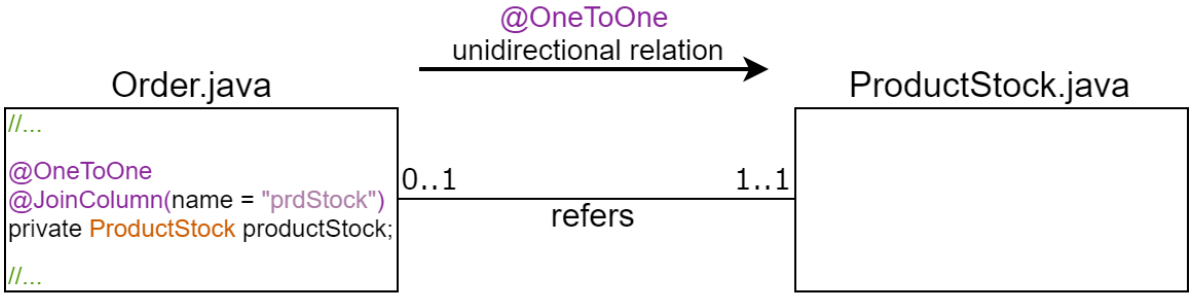
- ## Product ⇌ ProductStock (One to Many)

  This relationship is implemented as a ManyToOne unidirectional relationship between the *ProductStock* and *Product* entities, which is obtained by adding to the former a "product" attribute identifying the category the physical product belongs to.
  The relationship is implemented as unidirectional since from the application's specification it is required to retrieve the category a physical product belongs to, but not vice versa.



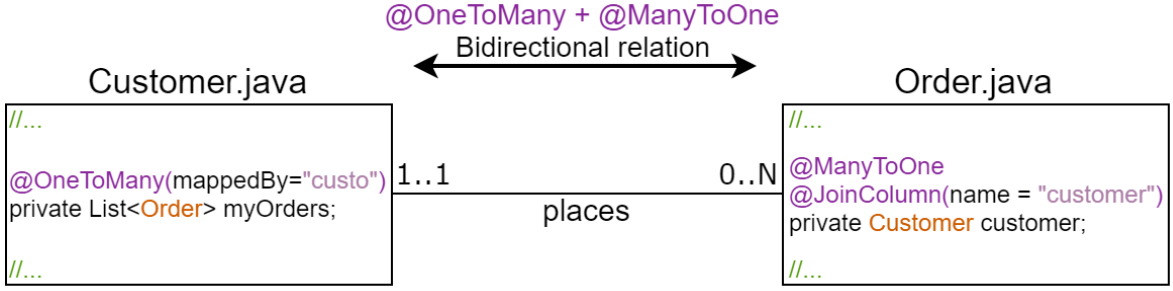
- ## Order ⇌ ProductStock (One to One)

  This relationship is implemented as a OneToOne unidirectional relationship between the *Order* and *Product* entities, which is obtained by adding to the former a "productStock" attribute identifying the IDstock of the physical product related to the Order.
  The relationship is implemented as unidirectional since from the application's specification it is required to retrieve the physical product relative to an Order, but not vice versa.



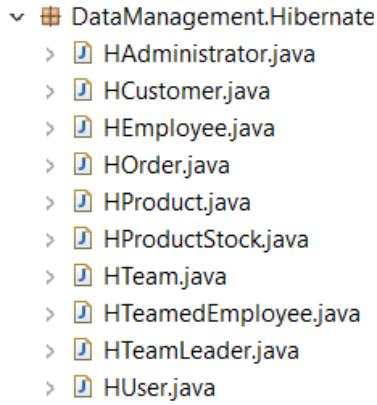
- ## Customer ⇌ Order (One to Many)

  This relationship is implemented as a OneToMany bidirectional relationship between the *Customer* and *Order* entities, which is obtained by adding to the former a "myOrders" attribute identifying the list of Orders relative to the Customer and by adding to the latter a "customer" attribute identifying the Customer the order refers to.
  The relationship is implemented as bidirectional since in our application it is necessary to retrieve both the list of Orders relative to a Customer and which Customer is relative to an Order.



Persistence Entities Implementation (final)

The final implementation of the persistence entities complete with their methods required by the application logic can be found in the attached project grouped under the "DataManagment.Hibernate" package, where each class has been prefixed with an "H" character to prevent naming collisions with other classes in the application:



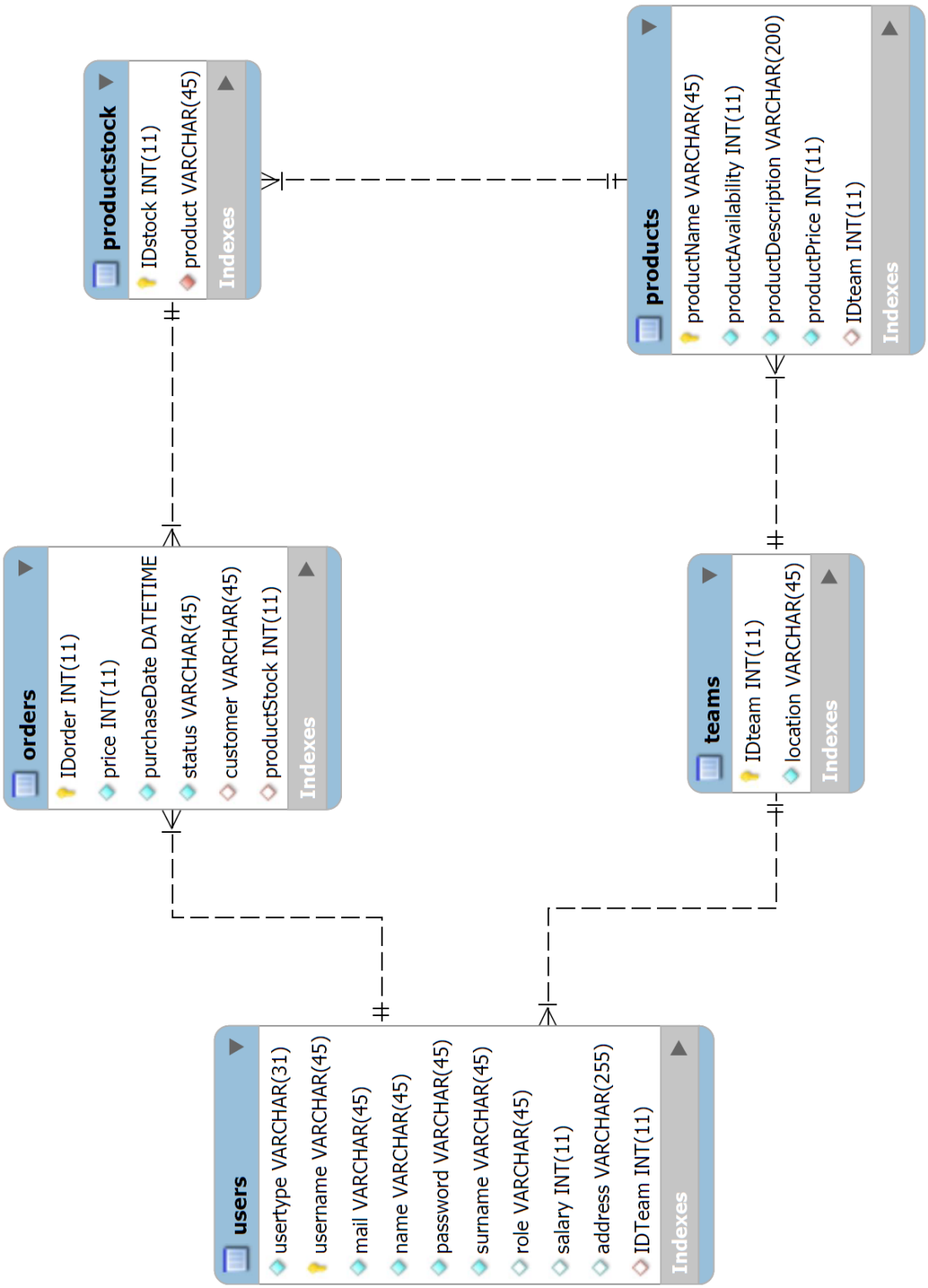
persistence.xml

Here follows the *persistence.xml* configuration file for the JPA used in our application:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- JPA version information -->
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                                 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

<!-- Persistence Unit definition -->
<persistence-unit name="taskOne">
 <properties>

  <!-- General server parameters, such as its IP and port, RDBMS type and schema to use -->
  <property name="javax.persistence.jdbc.url"
            value="jdbc:mysql://localhost:3306/taskone?serverTimezone=UTC"/>

  <!-- JDBC parameters, such as the user and password used to connect to the database -->
  <property name="javax.persistence.jdbc.user" value="root"/>
  <property name="javax.persistence.jdbc.password" value="root"/>
  <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>

  <!-- JPA implementation-specific parameters (in this case, hibernate) -->
  <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
  <property name="hibernate.archive.autodetection" value="class"/>
  <property name="hibernate.hbm2ddl.auto" value="update"/>
  <property name="hibernate.show_sql" value="true"/>
  <property name="hibernate.format_sql" value="true"/>

 </properties>
</persistence-unit>
</persistence>
```

JPA-generated Database

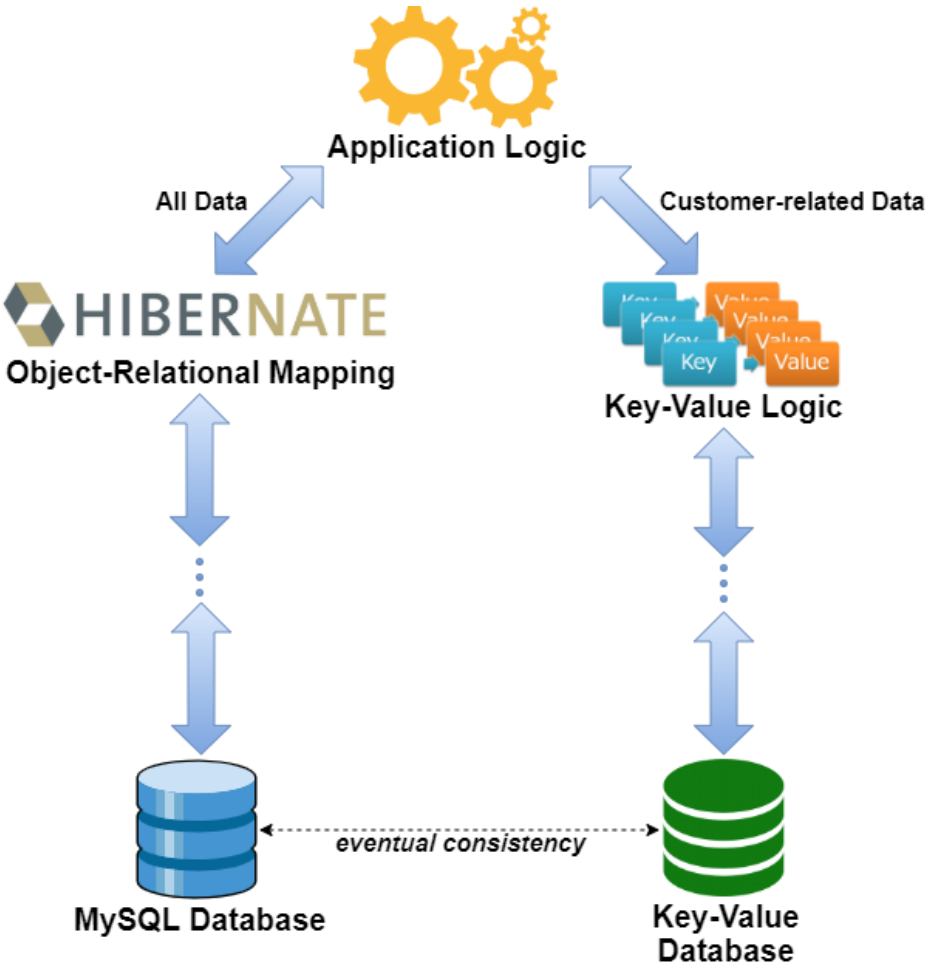
The MySQL database generated by JPA from the persistence entities we have defined is shown below:



Key-Value Database Adoption Study

Key-Value Database Feasibility Study

As the next step of the application's development we now take into consideration the possibility of adding a key-value database in parallel to the JPA-managed MySQL database discussed in the previous section.
Considering the application's requirements, the context in which it will be used, and the intrinsically sensitive nature of the data involved, it is apparent that a structured and consistent approach to data persistence is a necessity, as is ensured by the use of relational databases and their ACID transactions. Nevertheless, the adoption of a NoSQL and in particular a key-value database would greatly improve the access times of the application, especially the ones relative to queries that would translate into many join operations between tables of a relational database, and again, in view of the application's requirements, since a fast response time is critical especially for customers, we have chosen to introduce in our application, in parallel to the existing relational database, a key-value database in which to store the entire customer-related dataset, so that write operations are carried out in both databases, while read operations concerning data relating to the customers are first queried from the key-value database, with the relational database acting as a backup.



Other than improving the access times relative to the customers' operations, this solution also improves the application's fault tolerance, since a failure on the relational side wouldn't cause a service interruption towards the customers, while a failure on the key-value side would only worsen the application's response times, at the cost of the additional logic required to ensure the *eventual consistency* of the data in the two databases.

Key-Value Database Design

The dataset that must be maintained in the key-value database include:

- All customers' *passwords*
- All customers' *orders*
- All *Product* categories offered by the company
- All *IDstock* of the physical products available in stock

The naming strategy of the keys that will be used in the database is shown below:

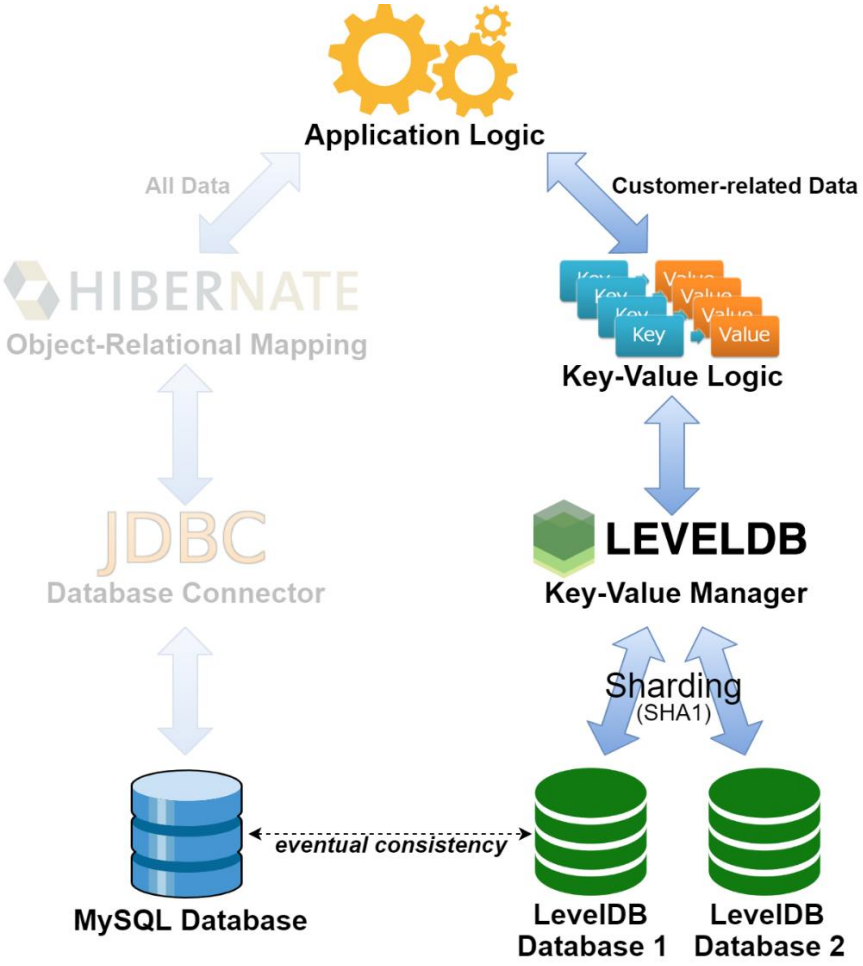
Key	Value
user : *username*	The password of the Customer identified by the *username*
user : *username* : order	The list of Order IDs relative to the Customer identified by the *username*
user : *username* : order : *idorder*	The details of the Order identified by the *idorder* of the Customer identified by the *username*
user : *username* : lastorder	The IDorder of the last order placed by the Customer identified by the *username* (used for consistency rollback purposes, see later)
order	The maximum IDorder value in the application
product	The list of product names of all Product categories offered by the company
prod : *productname*	The details of the Product category identified by *productname*
prod : *productname* : stockindex	The list of the IDstocks of the physical products of the *productname* category available in stock

The values in the database will be stored in JSON format, where Google's *GSON* library will be used to serialize and deserialize the objects of our application to and from the JSON format.

Regarding the choice of the Key-Value implementation to use in our application, among the different open-source possibilities available, we have chosen to adopt a Java port of Google's popular *LevelDB*, which is optimized for small read and write operations and provides an efficient data compression through the use of Google's *Snappy* library.

In addition, in order to minimize the latency of the read/write operations, having a RAID storage configuration in mind, we have also chosen to shard the entire key-value dataset into two separate databases, where as a load balancing technique the target database of each operation will be determined by hashing via the SHA-1 algorithm its key, where all hashes resulting in a value $< 2^{n-1}$ will be directed to the first database, while the ones resulting in a value $\geq 2^{n-1}$ will be directed to the second.

Thus, the final architecture of the Key-Value side of our application appears as shown below:



Data Consistency Management

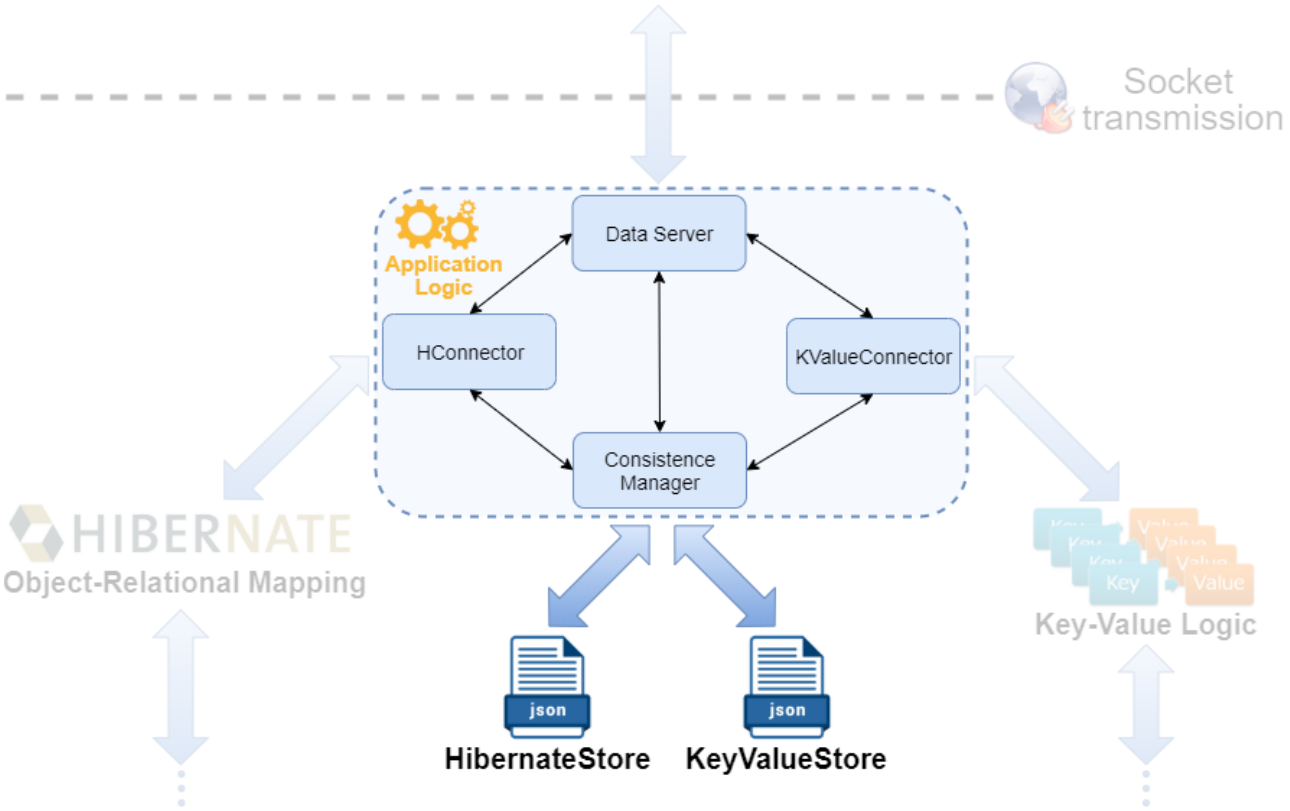
Regarding how the data consistency between the relational and the key-value databases should be handled, the write operations of our application can be divided into the following categories:

- Write operations that do not require consistency, which are given by the write operations whose results must be stored in the MySQL database only.
  In our application the only operation falling into this category is "Update Salary".

- Write operations that require a unilateral consistency, which are given by the write operations whose results, while not directly related to customers, must still be stored in the key-value databases.
  The operations falling into this category in our application are "Add User", "Delete User" and "Add Product".

- Write operations that require a bilateral consistency, which are given by the write operations performed by the customers and whose results must be persisted in both databases.
  In our application the only operation requiring a bilateral consistency is "Add Order".

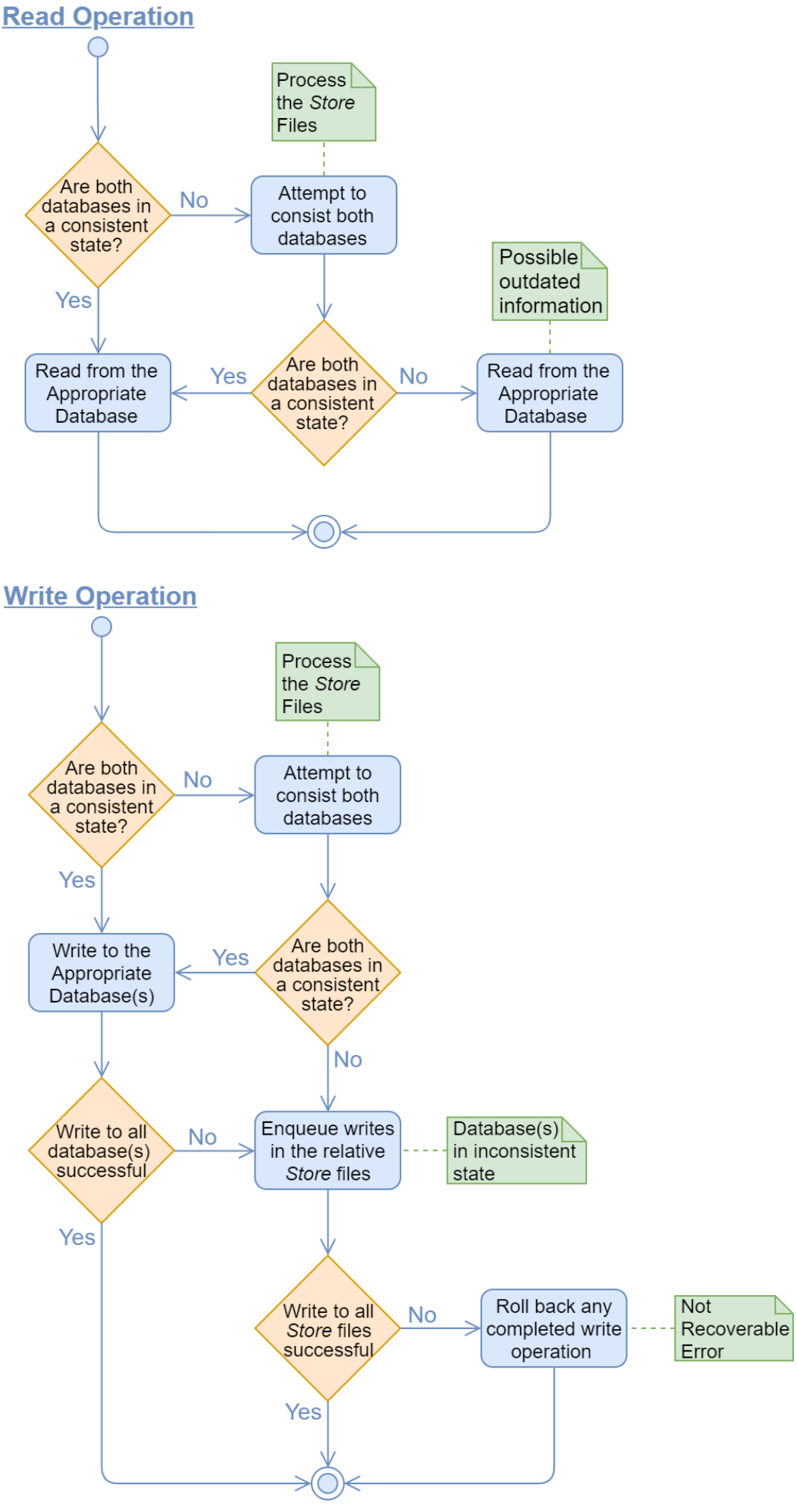
From here, taking into account all the possible scenarios where one or both databases may fail during the application's execution, as an *eventual consistency* mechanism between the two databases a *Consistence Manager* module was added to the application which, every time a write operation into one of the databases fails, serializes such operation in JSON format into a local file relative to such database (*HibernateStore* for the MySQL database and *KeyValueStore* for the LevelDB database, which can also be considered as two additional temporary document databases) for then, each time a new read/write operation is requested in the application, attempting beforehand to execute in the relative databases the operations stored in such files, in order to restore data consistency.



From here, each time a new read/write operation is requested in the application, if at least one of the databases is in an inconsistent state (i.e. The *HibernateStore* and/or the *KeyValueStore* files are not empty), the *Consistence Manager* will attempt beforehand to restore the consistency by executing the write operations stored in both files, where:

- If full consistency is restored, the new read/write operation is executed as normal (where again the latter will be stored in the relative *Store* file, should it fail).
- If full consistency is not restored, a new read operation is carried out normally (even if it could return outdated values), while a new write operation is directly enqueued in the appropriate *Store* file(s).
- Finally, in case a failed operation couldn't be written into its relative *Store* file(s), to ensure consistency the operation is rolled back from the database where it was successfully executed, if any.

As a recap, the complete behaviour of the *Consistence Manager* module for each new read or write operation in the application is best described through the following diagrams:


Read Operation
Process the Store Files
Are both databases in a consistent state?
No
Attempt to consist both databases
Possible outdated information
Yes
Read from the Appropriate Database
Yes
Are both databases in a consistent state?
No
Read from the Appropriate Database
Write Operation
Process the Store Files
Are both databases in a consistent state?
No
Attempt to consist both databases
Yes
Write to the Appropriate Database(s)
Yes
Are both databases in a consistent state?
No
Write to all database(s) successful
No
Enqueue writes in the relative Store files
Database(s) in inconsistent state
Yes
Write to all Store files successful
No
Roll back any completed write operation
Not Recoverable Error
Yes

Application Modules Diagram

The following diagram shows the complete application divided into its component modules: