Task0 report:

Design and implementation of a JAVA application connecting to relational DB

Group members: Federico Balestri, Nicola Barsanti, Riccardo Bertini, Mirco Quintavalla

>AGGIUNGERE INDICE

Introduction

The following paper describes the steps we have taken in order to create a realistic Java application based on a relational database. Our idea for the application consists in the development of a software solution designed to meet the data management requirements of a fictional tech company named *Innovative Solutions*. The company’s business consists in the retailing of electronical IoT-oriented products assembled in-house by teams of employees from their constituting components, which in turn are bought from various suppliers.

In the first part of the paper we will deal with the application requirements (functional and not functional), that represent the starting point for our design work. We will then analyse the requirements in order to identify actors and use cases, provide a simple dataflow diagram and we will decide the software architecture for our application.

In the second part of the paper we will provide a class diagram based on the decisions we made in the first part, and we will discuss the theoretical and logical aspects of the relational database that will be used by the application.

Application requirements

The company requires a software solution which on one hand allows the management to oversee and control all the information concerning the company’s production process, and on the other enables customers to directly purchase the products on offer and keep track of their orders.

The design of the software solution that will be proposed to the company is based upon the following working hypotheses:

* *Customers* may purchase any *Products* up to their available quantity
* Each *Employee* in the company may belong to up to one *Team*
* *Teams* are assigned to the assembly of the *Products* offered by the company, where each *Product* is assembled by a single team. Each team has an employee who acts as leader.
* *Products* are composed of one or more *Components*
* *Components* are purchased from a list of *Suppliers*, where different suppliers may offer the same component at different prices

More specifically, the company requires the presence of a *System Administrator* for the application, who will be able to add and remove users, and update an employee salary. *Team leaders* should also be able to use the application in order to visualize information about their team (components of the team, products the team is working on) and change the availability of the products.

On the other hand, the application is required to be available also for the company customers. A customer should be able to see the list of available products, purchase products and keep track of all his orders.

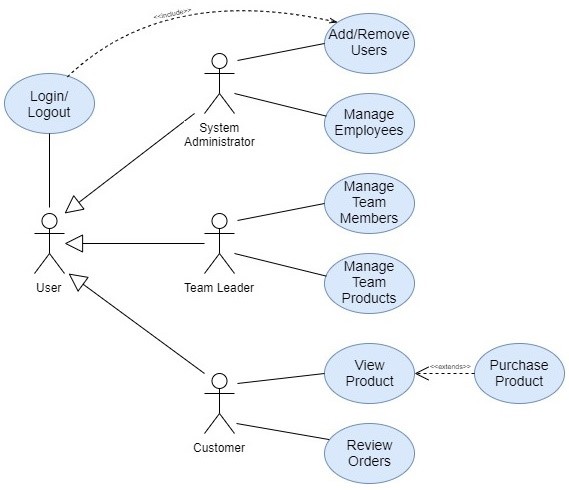
Regarding not functional requirements, we design our application in order to be as fast and efficient as possible. Since it is expected to be used by customers and team leaders, who may not be familiar with programming languages, we want to introduce a simple graphical interface in order to make the application very easy and intuitive to use.

Use cases

Considering the application requirements, we manage to identify the following actors:

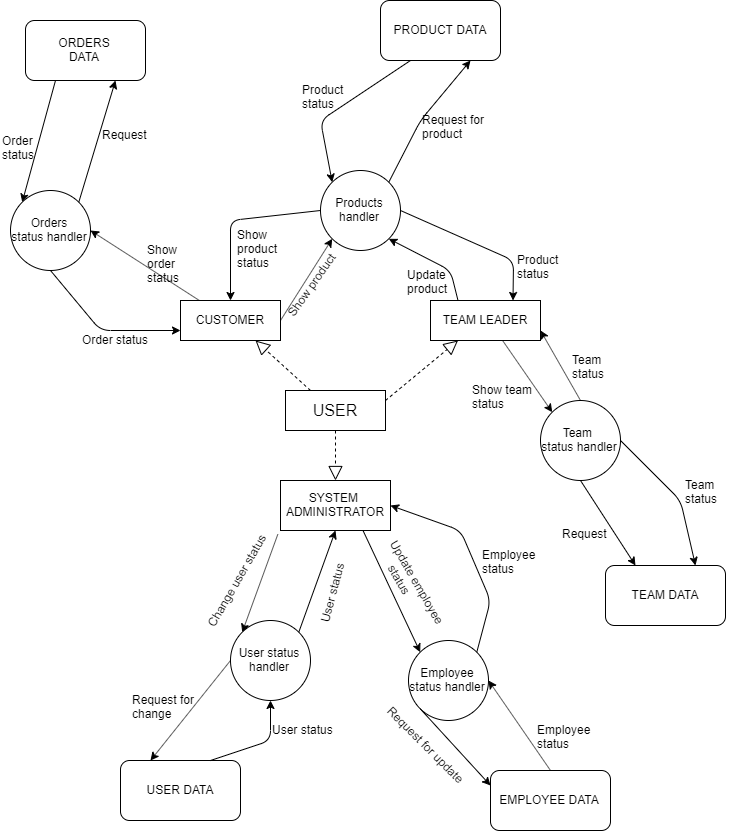
* The company’s system administrator, who is permitted to perform a number of actions a high level of privilege.
* The team leaders, who are allowed to manage the members and the products assigned to their teams
* The customers, who may view and purchase the products on offer, as well as review their past orders.

The three actor work on different privilege levels, which results in the introduction of a login/logout system based on username/password paradigm for our application. We don’t want to permit a low level user (the customers) to be able to perform the same operation as the system administrator, who is an high level user.



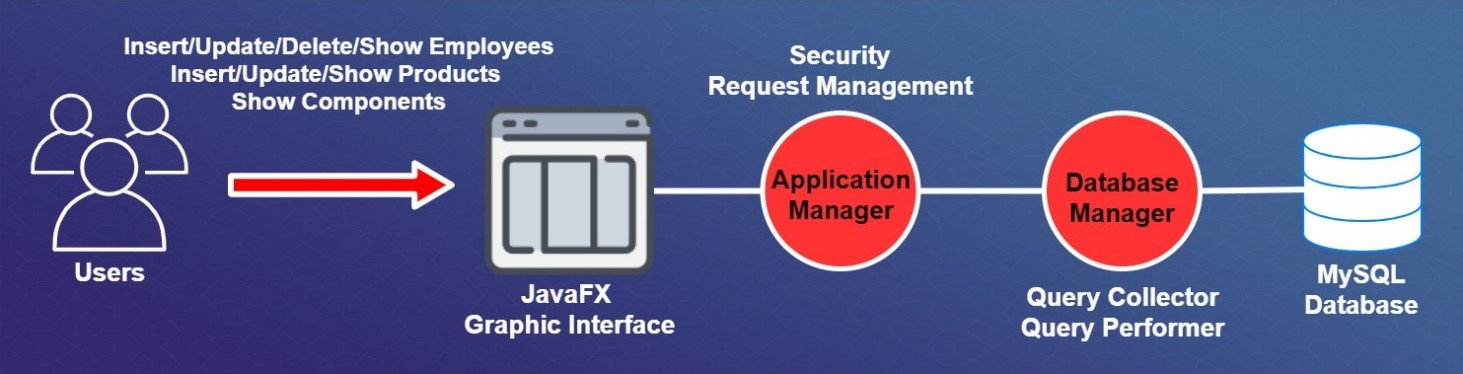
Application dataflow

The projected dataflow of the application is outlined below:



Software architecture

The software will consist in a front-end module written in Java providing a graphical interface through which users following an authentication process can connect to an underlying MySQL database, where the actions they can perform on the latter depend on their level of privilege.



Class diagram

>INSERIRE

Relational database design

In order to implement our application, we have to produce a relational database using MySQL. The database was designed by analysing the use cases and the class diagram, in order for the application to be as fast and efficient as possible.

**CONCEPTUAL DESIGN**

**Entities**

Considering the application requirements, we identify the following entities:

* **USER:** represent the application user
* **EMPLOYEE**: represents the different employees that work within the company.
* **TEAM:** the various employees work in teams.
* **ITEM:** represents the various objects the company work with
* **PRODUCT:** the different type of products that are assembled in the company by the teams.
* **PRODUCT\_STOCK:** represents the products stored in the warehouse
* **COMPONENT:** each product is made by different type of components
* **COMPONENT\_STOCK:** represents the components stored in the warehouse
* **SUPPLIER:** represents the various suppliers from which the company buys the components to assemble the products.
* **CUSTOMER:** represents the company clients. Clients can buy products.

**Attributes and identifiers**

The following table shows the various attributes related to the entities. Each attribute is to be considered having (1,1) cardinality with the entity, unless otherwise specified.

|  |  |
| --- | --- |
| **ENTITY** | **ATTRIBUTES** |
| USER | IDuser, name, surname, username, password, mail |
| EMPLOYEE | IDemployee, role, salary |
| TEAM | IDteam, location |
| ITEM | itemType, itemName, itemDescription,itemAvailability |
| PRODUCT | productType, productPrice |
| PRODUCT\_STOCK | productID,productType |
| COMPONENT | componentType |
| COMPONENT\_STOCK | componentID,componentType |
| SUPPLIER | IDsupplier, companyName, supplierMail |
| CUSTOMER | IDcostumer, address |

The entity USER is uniquely identified by the IDuser attribute, which is unique for each user. We also report the user name, surname, username, password and mail. Username are not necessarily unique for the users.

The entity EMPLOYEE is uniquely identified by the IDemployee attribute, which is unique for each employee. The other attributes are the employee’s role in his team and the salary.

The entity TEAM is uniquely identified by the IDteam attribute, which is unique for each team. We also have an attribute location which represents the physical location where the team is currently working.

The entity ITEM is uniquely identified by the attribute itemType. The other attributes are the item name, a brief description of the item and its availability.

The entity PRODUCT is uniquely identified by the productType attribute, which is unique for each product. We also report the product price.

The entity PRODUCT\_STOCK is uniquely identified by the combination of productID and productType.

The entity COMPONENT is uniquely identified by the componentType attribute.

The entity COMPONENT\_STOCK is uniquely identified by the combination of componentID and componentType.

The entity SUPPLIER is uniquely identified by the iDSupplier attribute. We also have an attribute for the company for which the supplier works and its mail address.

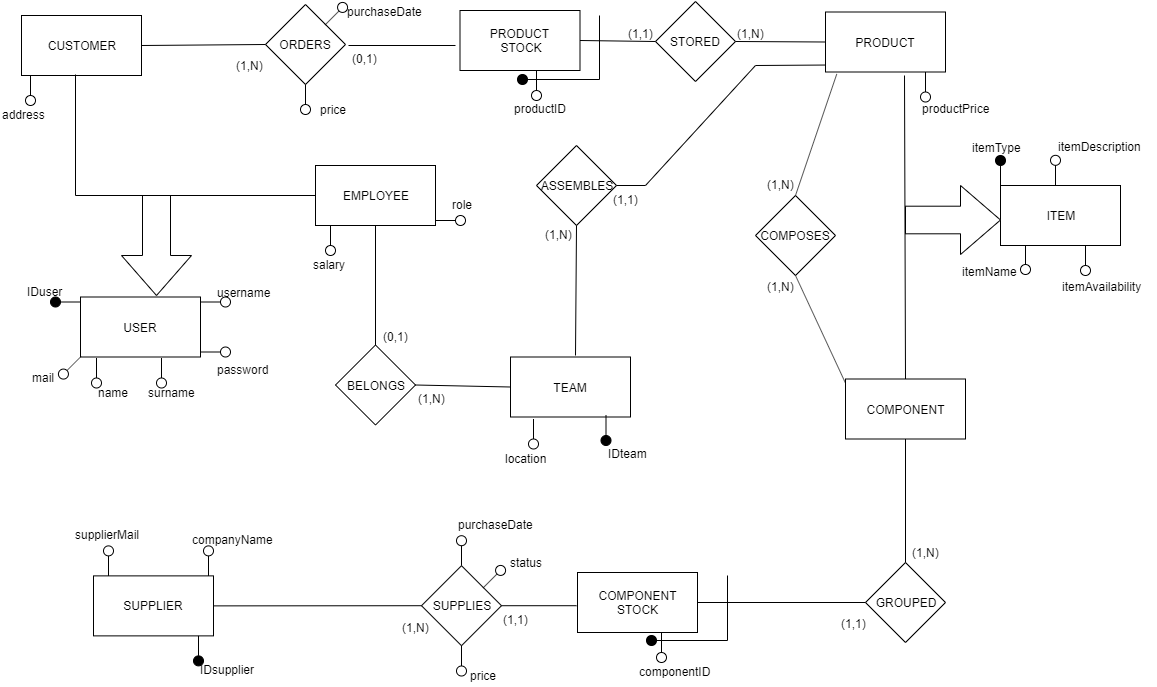
The entity COSTUMER is uniquely identified by the iDCostumer attribute. We also report the customer’s address.

**Relations between entities**

The following table shows the different relations between the entities in the database. Each relation is matched with the appropriate cardinality. We propose a brief explanation for each match.

|  |  |  |  |
| --- | --- | --- | --- |
| **RELATIONS** | **ENTITY A**  **(Cardinality with A)** | **ENTITY B**  **(Cardinality with B)** | **MOTIVATION** |
| BELONGS | EMPLOYEE  (0,1) | TEAM  (1,N) | An employee can work in up to one team, and a team needs at least one member |
| ASSEMBLES | TEAM  (1,N) | PRODUCT  (1,1) | Each team can work at same time on one or more type of products. A product can be assembled by only one team |
| ORDERS | CUSTOMER  (1,N) | PRODUCT\_STOCK  (0,1) | Each product can be purchased by only one client. A customer can of course by more than one product |
| STORED | PRODUCT\_STOCK  (1,1) | PRODUCT  (1,N) | More than one product for type can be stored in the warehouse |
| COMPOSES | PRODUCT  (1,N) | COMPONENT  (0,1) | A product can be made using one or more components. A component can be used in only one product. |
| GROUPED | COMPONENT\_STOCK  (1,1) | COMPONENT  (1,N) | More than one component for type can be stored in the warehouse |
| SUPPLIES | COMPONENT\_STOCK  (1,1) | SUPPLIER  (1,N) | A supplier can sell more than one component |

The ER-Diagram appears as follows:

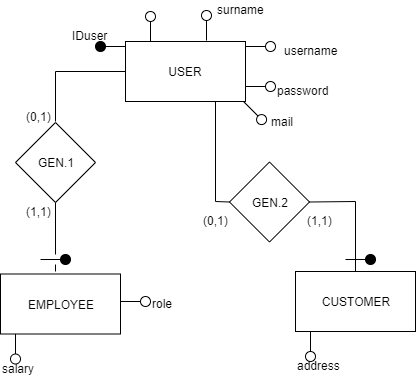


**Generalization analysis**

As we can see, there are two generalization in the ER diagram.

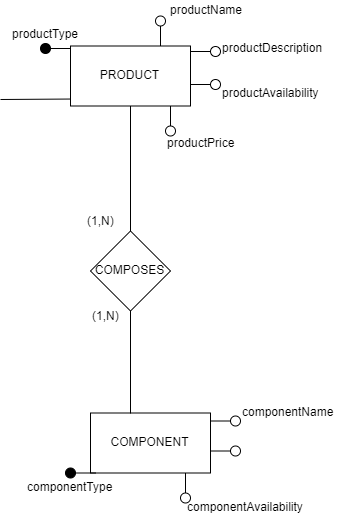
1. The entity USER is actually a generalization of the entities EMPLOYEE and CUSTOMER. We decided to solve using the “no aggregation” approach. We maintain in the scheme both the father and the children, so that we still have three entities: USER, EMPLOYEE and CUSTOMER. This approach may result in an increase in the number of access, but doesn’t introduce NULL values in the database. The “no aggregation” approach is preferable when we want to distinguish the occurrences on the father entity (USER) and the children entities (EMPLOYEE and CUSTOMER).

The following picture shows the resolved generalization



1. The entity ITEM is a generalization of the entities PRODUCT and COMPONENT. We dediced to solve using the “aggregation into children” approach, so that we maintained only the two children entities and insert into them all the attributes that used to belong to the father.

The following picture shows the resolved generalization



**Operations on the Database**

Considering the application requirements we defined before, the following operations are expected:

1. ADD USER

This operation allows the administrator to add a user to the database

Expected frequency: 3 times a day

1. REMOVE USER

This operation allows the administrator to remove a user from the database

Expected frequency: 1 times a day

1. UPDATE SALARY

This operation allows the administrator to modify an employee’ salary

Expected frequency: 1 times a day

1. VISUALIZE ORDER STATUS

This operation allows a customer to see the status of his orders

Expected frequency: 100 times a day

1. VISUALIZE PRODUCTS

This operation allows a customer to see the list of the available products

Expected frequency: 200 times a day

1. ADD ORDER

This operation allows a customer to buy a product from the ones available

Expected frequency: 20 times a day

1. VISUALIZE TEAM PRODUCTS

This operation allows a team leader to see the different product his team it’s working on.

Expected frequency: 60 times a day

1. VISUALIZE TEAM COMPONENTS

This operation allows a team leader to see the components of his team

Expected frequency: 6 times a day.

1. CHANGE AVAILABILITY

This operation allow a team leader to change the availability of a product

Expected frequency: 60 times a day

**Volume table**

In the following table we report the number of instances from the various entities and relations in the E-R diagram. These are examples based on how many people we expect to interact with the application.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **E/R** | **Number of instances** | **Motivation** |
| USER | E | 100 | Initial hypothesis |
| GEN.1 | R | 30 | 30% of the users are employees |
| EMPLOYEE | E | 30 | (1,1) cardinality with Gen.1 |
| GEN.2 | R | 70 | 70% of the users are customers |
| CUSTOMER | E | 70 | (1,1) cardinality with Gen.2 |
| BELONGS | R | 70 | (0,1) cardinality with Employee |
| TEAM | E | 6 | There are 6 teams in the company |
| ASSEMBLES | R | 30 | Each team assembles 5 type of products on average |
| PRODUCT | E | 30 | (1,1) cardinality with Assembles |
| STORED | R | 600 | There are on average 10 product for each type in the warehouse |
| PRODUCT\_STOCK | E | 600 | (1,1) cardinality with Stored |
| ORDERS | R | 350 | On average each customers make 5 orders (5x70=350) |
| COMPOSES | R | 150 | On average 5 components are needed to assemble a product |
| COMPONENT | E | 150 | (0,1) cardinality with Composes |
| GROUPED | R | 3000 | On average there are 20 components for each type in the warehouse |
| COMPONENT\_STOCK | E | 3000 | (1,1) cardinality with Grouped |
| SUPPLIES | R | 3000 | (1,1) cardinality with Component |
| SUPPLIER | E | 10 | Each supplier can supply 300 components (3000/300 = 10) |

**Operations analysis and redundancy introduction**

Let’s now analyse operation 8 (VISUALIZE COMPONENTS) in terms of the number of accesses required each time the operation is called. We would like to introduce a redundant attribute called “team” for the entity EMPLOYEE. The “team” represents the team where the employee is currently working.

Operation 8: VISUALIZE TEAM COMPONENTS

This operation allows a team leader to see the components of his team

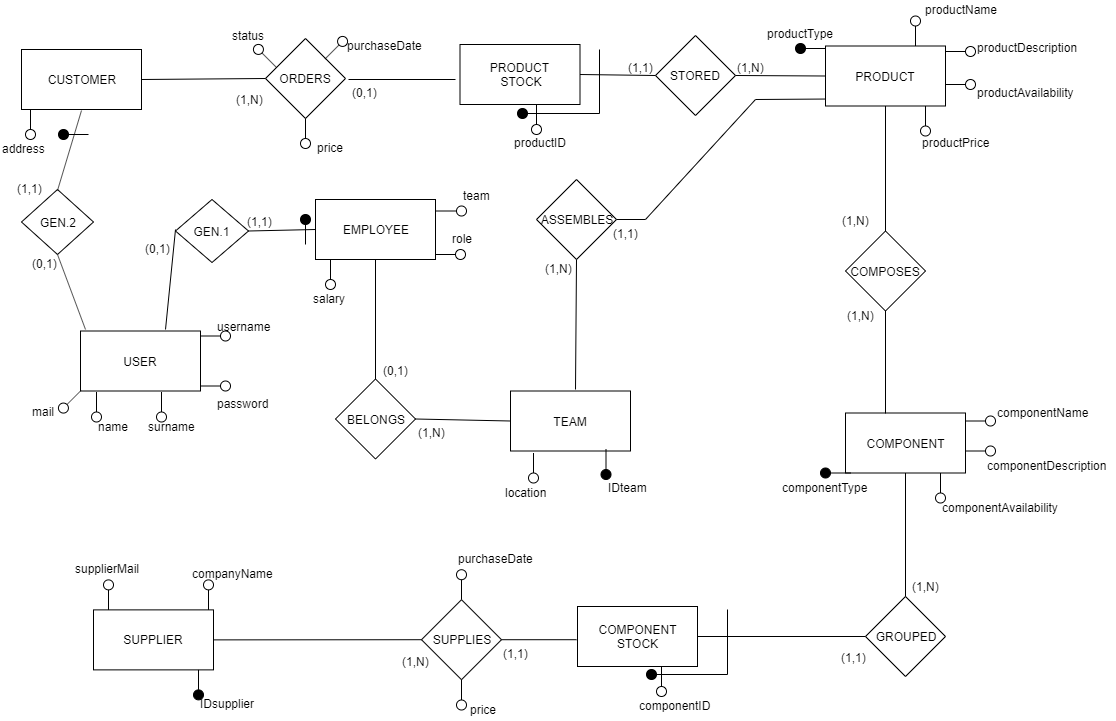
|  |  |  |
| --- | --- | --- |
| **Without redundancy “team”** | | |
| **Elementary operations number** | **Type of operation (read/write)** | **Motivation** |
| 1 | R | Access to Employee |
| 1 | R | Access to Team |
| 70x6 = 420 | R | Join of the entities Employee and Team |
| 422 | Total number of operations | |
| 422x6 = 2532 | Total number of operations in a day | |

|  |  |  |
| --- | --- | --- |
| **With redundancy “team”** | | |
| **Elementary operations number** | **Type of operation (read/write)** | **Motivation** |
| 70 | R | Access to Employee. I can directly read in which team each employee is working |
| 70 | Total number of operations | |
| 70x6 = 420 | Total number of operations in a day | |

Since the attribute “team” makes the operations more than 6 times faster, we decide to add the redundancy in the database.

**Final ER diagram**

The definitive version of the ER diagram appears as follow:



**LOGICAL DESIGN**

**Logical scheme**

Since we have the final ER diagram, we can now translate it into tables obtaining the following logical scheme:

* USER(IDuser, name, surname, username, password, mail)
* EMPLOYEE(IDemployee, salary, role, team)
* CUSTOMER(IDcustomer, address)
* TEAM(IDteam, location)
* ASSEMBLES(team, product)
* PRODUCT(productType, productName, productDescription, productPrice, productAvailability)
* PRODUCT\_STOCK(productID, productType)
* COMPONENT(componentType, componentName, componentDescription, componentAvailability)
* COMPONENT\_STOCK(componentID, componentType)
* COMPOSES(component,product)
* ORDERS(customer, product, purchaseDate, status)
* SUPPLIER(IDsupplier, companyName, supplierMail)
* SUPPLIES(component, supplier, purchaseDate, price)

Referential integrity constraint:

* There is a referential integrity constraint between the attribute IDuser of the table USER and the attribute IDemployee of the table EMPLOYEE
* There is a referential integrity constraint between the attribute IDuser of the table USER and the attribute IDcustomer of the table CUSTOMER
* There is a referential integrity constraint between the attribute IDteam of the table EMPLOYEE and the attribute IDteam of the table TEAM
* There is a referential integrity constraint between the attribute IDteam of the table TEAM and the attribute team of the table ASSEMBLES
* There is a referential integrity constraint between the attribute productType of the table PRODUCT and the attribute product of the table ASSEMBLES
* There is a referential integrity constraint between the attribute componentType of the table COMPONENT and the attribute component of the table COMPOSES
* There is a referential integrity constraint between the attribute productType of the table PRODUCT and the attribute product of the table COMPOSES
* There is a referential integrity constraint between the attribute IDcustomer of the table CUSTOMER and the attribute customer of the table ORDERS
* There is a referential integrity constraint between the attribute productID of the table PRODUCT\_STOCK and the attribute product of the table ORDERS
* There is a referential integrity constraint between the attribute productType of the table PRODUCT\_STOCK and the attribute productType of the table PRODUCT
* There is a referential integrity constraint between the attribute componentType of the table COMPONENT\_STOCK and the attribute product componentType of the table COMPONENT
* There is a referential integrity constraint between the attribute componentID of the table COMPONENT\_STOCK and the attribute component of the table SUPPLIES
* There is a referential integrity constraint between the attribute IDsupplier of the table SUPPLIER and the attribute supplier of the table SUPPLIES

**Normalization**

All the tables in our design respect the Boyce-Codd normal form because for every one of the dependencies X🡪Y in a table it’s true that X is a superkey for the table. There are not redundancy based on functional dependency.

* USER: IDuser🡪name, surname, username, password, mail
* EMPLOYEE: IDemployee🡪 salary, role, team
* CUSTOMER: IDcustomer🡪 address
* TEAM: IDteam🡪location
* ASSEMBLES: team, product
* PRODUCT: productType🡪productName, productDescription, productPrice, productAvailability
* COMPONENT: componentType🡪componentName, componentDescription, componentAvailability
* COMPOSES: component,product
* ORDERS: customer, product 🡪purchaseDate, price, status
* PRODUCT\_STOCK: productID, productType
* COMPONENT\_STOCK: componentID, componentType
* SUPPLIER: IDsupplier🡪companyName, supplierMail
* SUPPLIES: component, supplier-->purchaseDate, price

**Database implementation**

Terminated our theoretical study for the database, we can implement it in MySQL.

>INSERIRE IMMAGINE DELLE TABELLE FATTA CON MYSQL (QUELLA DI RICCARDO)

Final class diagram

>INSERIRE