**Database design and API implementation**

1. **Database design**
   1. **Design explanation**

In our design, we decided that each business object should have a table on its own, since the API expects us to be able to add or delete a business object like this from our database(except dish). In addition to that, the API also have functions that create a specific link between each of the business objects and must also be kept in our database. To support this functionality, we also create tables that keeps these links(indexes 4-5-6 further).

We also decided that views were necessary where it was useful in multiple API functions, in order to avoid code duplication or to simplify a query. We explain in each view for which function we need to use it.

* 1. **Tables**
     1. **Customers(cust\_id: int, full\_name: string, age: int, phone: string):** table who is based on the business object Customer, and contains all of its attributes, with all of them as NOT NULL, cust\_id positive, age between 18 and 120 and phone of 10 characters.

The attribute cust\_id is a primary key.

* + 1. **Orders(order\_id: int, date: timestamp, delivery\_fee: float, delivery\_adress: string):** table who is based on the business object Order, and contains all of its attributes, with all of them as NOT NULL, order\_id > 0, delivery\_fee >= 0, length of delivery\_adress >= 5.

The attribute order\_id is a primary key.

* + 1. **Dishes(dish\_id: int, name: string, price: float, is\_active: bool):** table who is based on the business object Dish, and contains all of its attributes, with all of them as NOT NULL, dish\_id > 0, price > 0, length of name >= 4.

The attribute dish\_id is a primary key.

* + 1. **CustomerPlacesOrder(cust\_id: int, order\_id: int):** table who keeps all the customer who placed an order, when declared with the API.

The attribute order\_id is a primary key, and a foreign key of Orders so that deletion in Orders delete it here, and cust\_id is a foreign key of Customers so that deletion in Customers sets it NULL here.

* + 1. **OrderContainsDish(order\_id: int, dish\_id: int, amount: int, price: float):** table who keeps the dishes that belong to an order, when declared with the API.

We have to check amount >= 0.

The attributes order\_id-dish\_id are the primary key, to ensure each order can contain multiple dishes and each dish can be associated with multiple orders.

The attribute order\_id is a foreign key of Orders so that deletion in Orders delete it here, and dish\_id is a foreign key of Dishes, who cannot be deleted there.

* + 1. **CustomerRatedDish(cust\_id: int, dish\_id: int, rating: int):** table who keep the ratings put on a dish by a customer, when declared with the API.

rating is NOT NULL, 1<= rating <= 5.

The attributes cust\_id-order\_id are the primary key, to ensure a customer can places multiple ratings on multiple dishes and a dish can be rated by multiple customers.

The attribute cust\_id is a foreign key of Customers so that deletion in Customers delete it here, and dish\_id is a foreign key of Dishes, who cannot be deleted there.

* 1. **Views**
     1. **OrderTotalPrice(order\_id: int, total\_price: float, cust\_id):** this view is calculating the total price of an order with the price of its dishes and it s delivery fee, and keeps the customer associated with the order if exist. It is useful in API functions like get\_order\_total\_price, get\_customers\_spent\_max, or get\_cumulative\_profit.
     2. **RatingDish(dish\_id: int, avg\_rating: float):** this view is calculating the average rating for the dishes as the average of all the ratings given by the customers. If there is no rating on a dish, set its rating to 3. It is useful in API functions like did\_customer\_order\_top\_rated\_dishes or get\_customer\_rated\_but\_not\_ordered.
     3. **CustomerOrderedDish**
     4. **AvergeProfitPerOrderPerPrice**

1. **API**
   1. **Create/clear/drop\_table**
      1. **create\_table:** this function create all of the tables and the view we mentioned earlier, by creating first the table corresponding to the business objects, then the tables who make the link between them, then by declaring the view, so that each query that uses an other table has it declared beforehand.
      2. **clear\_table:** clear our table from all the data using the DELETE FROM in SQL. It is to note that only the tables needs clearance and not the views, who are only temporary query and don t have existence in the database.
      3. **drop\_table:** drop all the tables and the views in reverse order from the order of creation, that ensure that a view cant exist if the table it uses in no longer there, and the same thing apply to the tables between the business objects.
   2. **Crud API**
      1. **add\_customer**
      2. **get\_customer**
      3. **delete\_customer**
      4. **add\_order:** add an order to the table Orders with INSERT. Any order who does not stand by the rules of the business object Order or already exists will be spotted by the verifications in the table Orders, and an error will be returned.
      5. **get\_order:** return an order by its order\_id from the table Orders. If the order in not found and the row\_affected is different than 1, we return BadOrder.
      6. **delete\_order:** delete an order from the table Orders by its order\_id with DELETE if exist, and if if it doesn’t exist, spot it by the row\_affected who is 0 and return NOT\_EXISTS
      7. **add\_dish**
      8. **get\_dish**
      9. **update\_dish\_price**
      10. **update\_dish\_active\_status:** change the attribute active\_status for the dish corresponding to the dish\_id of the function in the table Dishes to the status required using UPDATE Dishes SET… . If row\_affected is 0, it means that there is no dish like that and we return NOT\_EXISTS, same thing if there is an illegal id that throw an exception.
      11. **customer\_placed\_order:** use INSERT to add the customer and the order associated in the table CustomerPlacesOrder, who verify by the foreign keys that the customer and the order actually exist, and if they throw an exception, we return NOT\_EXISTS. If the order already exists in the table CustomerPlacesOrder, it means it already have a customer affected and it returns then ALREADY\_EXISTS.
      12. **get\_customer\_that\_placed\_order:** search in the table CustomerPlacesOrder if the order we look for has been deposed by a customer, and return the customer from the table Customer if exist. If the query doesn’t find a customer, then row\_affected will be 0 and we will return BadCustomer.
      13. **order\_contains\_dish**
      14. **order\_does\_not\_contains\_dish**
      15. **get\_all\_order\_item:** search in the table OrderContainsDish for all the tuples with the corresponding order\_id, and return a list of all the dishes that are related to him without the active status, in a special object OrderDish. If the order doesn’t exist or does not have dishes, the query will return an empty table and will then not add any member to the list, and the list returned will be empty.
      16. **customer\_rated\_dish**
      17. **customer\_deleted\_rating\_on\_dish**
      18. **get\_all\_customer\_ratings:** get from the table CustomerRatedDish all the dish and rating deposed by the customer we’re looking for, with an ORDER BY dish\_id. If the customer doesn’t exist or didn’t rated any dishes, the query will return an empty table and will then not add any member to the list, and the list returned will be empty.
   3. **Basic API**
      1. **get\_order\_total\_price**
      2. **get\_customers\_spent\_max\_avg\_amount\_money**
      3. **get\_most\_purchased\_dish\_amoung\_anonymous\_order:** in this query, we are going to use a subquery to get from CustomerPlaceOrder all the order\_id that are associated to a customer who still exist, and then select from the table OrderContainsDish all the orders that do not belong to those, in fact the anonymous orders that contains dishes. Then we group by dish\_id all the orders in OrderContainsDish and make a sum of theirs amount, to finally get the one with the higher sum who is in fact, the most purchased one. Since we know there exists at least one anonymous order with a dish, we are sure that our query will return something.
      4. **did\_customer\_order\_top\_rated\_dishes:** in this query, we use our view RatingDish in a subquery that gets only the 5 better ratings dishes, who we are going to compare with the dishes in the view CustomerOrderedDish to get from there all the cust\_id from the customers who ordered those top-rated dishes. If there are customers like this, we return true, else we return false.
   4. **Advanced API**
      1. **get\_customers\_rated\_but\_not\_ordered**
      2. **get\_non\_worth\_price\_increase**
      3. **get\_cumulative\_profit\_per\_month**
      4. **get\_potential\_dish\_recommandations**