Report of Project #2

Lab class 1

Reporter: 12012514李文锦，12013016焦傲

Contributions:

- **2.2 Functional Requirements**李文锦，焦傲
- **3. Advanced Requirements:** 李文锦， 焦傲
- **the percentages of contributions:** 李文锦: 50%, 焦傲: 50%

Part One: APIs and related triggers and functions in Database

- ### APIs and functions:

  In this part, we provide multiple APIs to let user finish manipulations on database.

  - In provided java program, we provide multiple data manipulations including insert, delete, update and select and so on. What' more there are some functions and triggers are related to these APIs, and they will be introduced in next part.
  - For each different manipulation, if your input data is illegal, it will return null.

    ```
    public static void initialize();
    ```

    This API can initialize the database, which can put all original data such as staff, center, model and enterprise into target database.

    ```
    public static String truncate();
    ```

    This API can clear target database' s data. This API is related to a created function in our database

    ```
    create or replace function truncate()
    returns varchar
    ```

    This is the related function in database, it contains multiple truncate manipulations to clear data in created tables.

    ```
    public static void centerInsert(int center_id, String supply_center);
    public static String centerSelect();
    public static void centerDelete(int center_id);
    public static void centerUpdate(int center_id, String supply_center);
    ```

These four APIs provide basic manipulations of select, insert, delete and update on center table in the database. You can delete and update data in center table by its id. You can view all centers by using select and also you can insert center with new center id and center name.

```
public static void stockInSingle(......);
public static void stockIn();
```

These two APIs can put data of inventory into target database. And it related to a trigger, which can prevent loading dirty data into target database. And ***stockInSingle*** is a function use to insert single data of target file, ***stockIn()*** help it to insert all data into database in target file.

```
public static void placeOrderSingle(String contract_num, String enterprise, String product_model, int quantity, int contract_manager, Date contract_date, Date estimated_delivery_date, Date lodgement_date, int salesman_num, String contract_type);
public static void updateOrderSingle(String contract_number, String product_model, int salesman_number, int quantity, Date esDate, Date loDate);
public static void deleteOrderSingle(String contract_number, int salesman_number, int seq);
public static void placeOrder();
public static void updateOrder();
public static void deleteOrder();
```

These APIs can put data of **orders** and **contracts** into target database and can also update and delete target order by input values. These API allowed you do these manipulations one by one (use "single") or just all in. What' s more, when you use ***placeOrder*** function, there is a trigger and it can update related model_sold in inventory and for all these three manipulations, they can also **handle dirty data** and return null. And the parameters of ***placeOrderSingle*** means every data of a new line of orders. And the parameters of ***updateOrderSingle*** means you can update target line' s order_quantity, estimated_delivery_date and lodgement_date in orders by its contract_number, product_model and salesman_number. And the parameters of ***deleteOrderSingle*** means you can delete target order by its contract_number, salesman_number and another parameter called seq, it mean sequence numeber and it is calculated by sort the record in the orders table by estimate_delivery_date and product_model (a-z, A-Z).

```
public static String getAllStaffCount();
```

This API can get different types of staff and the number of them.

```
create or replace function getAllStaffCount()
returns
    table(
        staff_type varchar,
        count integer
    )
```

This is related function in database, it can return a table contain different types of staff and the number of them.

```java
public static String getContractCount();
```

This API can get the count of contracts in target database.

```sql
create or replace function getContractCount()
returns integer
```

This is the related function in database, it can return the count of contracts by sql.

```java
public static String getOrderCount();
```

This API can get the count of orders in target database.

```sql
create or replace function getOrderCount()
returns integer
```

This is the related function in database, it can return the count of orders by sql.

```java
public static String getNeverSoldProductCount();
```

This API can get the count of product that were never sold.

```sql
create or replace function getNeverSoldProductCount()
returns integer
```

This is the related function in database, it can return the count of product that were never sold by sql.

```java
public static String getFavoriteProductModel();
```

This API can get the most quantity of sold product and the number of it.

```sql
create or replace function getFavoriteProductModel()
returns
    table(
        model_name varchar,
        quantity integer
    )
```

This is related function in database, it can return a table contain the most quantity of sold product and the number.

```java
public static String getAvgStockByCenter();
```

This API can get the average quantity of the remaining product models of every supply centers.

```java
public static String getProductByNumber(String product_code);
```

This API can get the current inventory capacity of each product model in each supply center by its code.

```java
public static String getContractInfo(String contract_number);
```

This API can get the information of target contract by its number and every orders' information in that contract.

```java
public static String getOrderListByContractNum(String contract_number);
```

This API can get target contract by its number.

```java
public static String getOrderListByThree(String contract_number,int
salesman_number, String model);
```

This API can get target contract by its number and salesman number and product model.

```java
public static String getOrderListByTwo(String contract_number,int
salesman_number);
```

This API can get target contract by its number and salesman number.

```java
public static String getOrderListBySalesman(int salesman_number);
```

This API can get target contract by its salesman number.

```java
public static String getBestSalesman();
```

This API can get the information of salesman with most orders.

```java
public static String getMVPModel();
```

This API can get the model with highest profit.

```java
public static String getPureProfits();
```

This API can get the pure profits of every supply center .

```
public static String getBill();
```

This API can get the pure profits and cost and profit of every supply center .

- ## Triggers:

In this part we will explain those triggers used in our project

```
create trigger stockIn_trigger before insert on inventory
for each row execute procedure stockIn_check();
```

This, trigger related to API ***stockInSingle*** and ***stockIn***. It can handle five types of dirty data with: **Supply staff do not match**, **Wrong type of the staff**, **Supply center does not exist**, **Product does not exist** and **Staff does not exist**. And only data with correct information can be inserted into target table.

```
create trigger placeOrder_trigger before insert on orders
for each row execute procedure  placeOrder_check();
```

This, trigger related to API ***placeOrderSingle*** and ***placeOrder***. It can handle dirty data that **Quantity in an order larger than the stock** and **Wrong type of the staff**. And this trigger can **update** the column model_sold in table inventory at the same time if input information is reliable. Only data with correct information can be inserted into target table.

```
create trigger updateOrder_trigger before update on orders
for each row execute procedure updateOrder_check();
```

This, trigger related to API ***updateOrderSingle*** and ***updateOrder***. It can handle dirty data that **Wrong order with target salesman**. And this trigger can **update** the column model_sold in table inventory at the same time if input information is reliable. Meanwhile if updated order with zero in column order_quantity, this order will be delete by this trigger and if after updating these data, a contract do not have any order, the contract will not be deleted. Only data with correct information can be inserted into target table.

```
create trigger deleteOrder_trigger before delete on orders
for each row execute procedure deleteOrder_check();
```

This, trigger related to API ***deleteOrderSingle*** and ***deleteOrder***. It can handle dirty data that **Wrong order with target salesman**. And this trigger can **update** the column model_sold in table inventory at the same time if input information is reliable. If after updating these data, a contract do not have any order, the contract will not be deleted. Only data with correct information can be inserted into target table.

```
create trigger placeContract_trigger before insert on contract
for each row execute procedure  placeContract_check();
```

This, trigger related to API ***placeOrderSingle*** and ***placeOrder***. Since the information of every contract and every order are in the same file, so in order to select distinct contract, we make such a trigger to check before insert if the contract has been inserted.

Part Two: Advanced Requirements

1. ### More APIs and tools

   This part have been talked in above part, including bill part, more parameters API, some other useful APIs, triggers and so on.

2. ### Implement a true back-end server

   The project uses the Spring Boot framework as the back-end framework, uses the basic HTML and CSS as the front-end, and realizes the function of the back-end server running method and displaying in the front-end page.

   The front-end display interface is decorated with CSS to achieve a dynamic and natural display effect, both beautiful and practical.

   Project structure is shown above.

```
project_2
├── HELP.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── com.example.project
    │   │       ├── apis // part one
    │   │       ├── index
    │   │       └── ProjectApplication
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       │   ├── css
    │       │   ├── img
    │       │   └── js
    │       └── templates
    └── test
```

```
        └── java
            └── com.example.project
                └── ProjectApplicationTests.java
```

3. **MultiThreading and Concurrent**

Implement all methods of multithreaded access

- The following classes implement multithreaded access to methods.

```java
public static class getAllStaffCountThread implements Runnable{}
public static class getOrderCountThread implements Runnable{}
public static class getContractCountThread implements Runnable{}
public static class getNeverSoldProductCountThread implements Runnable{}
public static class getFavoriteProductModelThread implements Runnable{}
...
```

- The following classes implement multithreaded access to four main methods: stock in, place order, update order, delete order

  The following methods implement parallelism when changing a single piece of data in a database, using `ThreadPoolExecutor` internally as a thread pool and `ArrayBlockingQueue` as a blocking queue.

  The multithreaded approach using thread pools has a significant speed advantage over the common single-thread insertion approach, use `stockin` as example and use `System.nanoTime()` to trace time useage.

  Due to the small size of the test sample data, multi-threaded insertion has no obvious advantage compared with single thread, but it has great performance advantage in the case of large data sets.

  (CPU: apple silicon M1 pro, Memory: LPDDR5 16GB universal memory)

Methods	1	2	3	4	5
stockInConcurrent()	0.084926913s	0.071540190s	0.089824167s	0.080535041s	0.0977465s
sockIn()	0.09266525s	0.11408375s	0.096175708s	0.097039s	0.099048584s

6	7	8	9	10
0.075099041s	0.060185459s	0.074962208s	0.07588475s	0.0856655s
0.080551167s	0.0958465s	0.084830709s	0.073129667	0.087053792s

```java
public static void stockInConcurrent() throws SQLException, IOException,
InterruptedException {}
public static void updateOrderConcurrent() throws IOException,
InterruptedException {}
public static void placeOrderConcurrent() throws IOException,
InterruptedException {}
public static void deleteOrderConcurrent() throws IOException,
InterruptedException {}
```