VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY

Database Assignment

Author:

Name	Student ID	Class
Nguyễn Nam Kha	2052515	CC03
Nguyễn Ngọc Hòa	2052485	CC03
Châu Nguyễn An Khang	1952757	CC03

Ho Chi Minh city, 12/2022

CONTRIBUTIONS

Name	Student ID	Task
Nguyễn Nam Kha	2052515	Part 1.II, Part 2.I, Part 3
Nguyễn Ngọc Hòa	2052485	Part 1.I, Part 2.II, Part 3
Châu Nguyễn An Khang	1952757	Part 1.I, Part 2.I, Part 3

Contents

PART 1: CREATE DATABASE

To implement this assignment, we use MySQL – a free and open-source relational database management system.

All of our source code can be found here:

https://github.com/namkha1032/databaseAssignment

The code to create database, create stored procedures, functions, triggers and insert data are all stored in `script.sql` file.

I. Create relational table

1. branch (BranchID, Province, Address, PhoneNum, Email)

This table stores the data of each branch. In this table we set *BranchID* as the PRIMARY KEY. We also set the attributes *Address*, *PhoneNum*, *Email* as UNIQUE KEYS since those attributes cannot be duplicated.

```
CREATE TABLE branch_seq(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT);
CREATE TABLE `branch` (
  `BranchID` varchar(6) NOT NULL,
  `Province` varchar(50) NOT NULL,
  `Address` varchar(100) NOT NULL,
  `PhoneNum` varchar(12) NOT NULL,
  `Email` varchar(64) NOT NULL,
  PRIMARY KEY (`BranchID`),
  UNIQUE KEY `Address_UNIQUE` (`Address`),
  UNIQUE KEY `PhoneNum_UNIQUE` (`PhoneNum`),
  UNIQUE KEY `Email_UNIQUE` (`Email`));
```

We also set a TRIGGER that defines a regex to ensure that the incoming *BranchIDs* always start with CN followed by an auto-increment integer.

```
DELIMITER $$
CREATE TRIGGER generate_branch_id
BEFORE INSERT ON `branch`
FOR EACH ROW
BEGIN
```

```
  INSERT INTO branch_seq VALUES (NULL);
  SET NEW.BranchID = CONCAT('CN', LAST_INSERT_ID());
END$$
DELIMITER ;
```

2. image_branch (BranchID, Image)

This table stores the data of the hotel's image. In this table we set both *BranchID* and *Image* as the PRIMARY KEY. *BranchID* is the FOREIGN KEY reference *branch(branchID)*

```
CREATE TABLE `image_branch` (
  `BranchID` varchar(6) NOT NULL,
  `Image` varchar(255) NOT NULL,
  PRIMARY KEY (`BranchID`,`Image`),
  CONSTRAINT `FK_branch_image` FOREIGN KEY (`BranchID`) REFERENCES
`branch` (`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

3. zone (BranchID, ZoneName)

This table stores the data of each zone. In this table we set both *BranchID* and *ZoneName* as the PRIMARY KEY. *BranchID* is the FOREIGN KEY reference *branch(branchID)*.

```
CREATE TABLE `zone` (
  `BranchID` varchar(6) NOT NULL,
  `ZoneName` varchar(20) NOT NULL,
  PRIMARY KEY (`BranchID`,`ZoneName`),
  CONSTRAINT `FK_branch_zone` FOREIGN KEY (`BranchID`) REFERENCES
`branch` (`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE
) ;
```

4. room_type (RoomID, RoomName, Area, NumGuest, Description)

This table stores the data of each zone. In this table we set *RoomID* as the PRIMARY KEY. *BranchID* is the FOREIGN KEY reference *branch(branchID)*. We also set constraints 'LimitGuest' ensuring that the number of guest will be between 1 and 10

```
CREATE TABLE `room_type` (
  `RoomID` int unsigned NOT NULL AUTO_INCREMENT,
```

```
  `RoomName` varchar(45) NOT NULL,
  `Area` int unsigned NOT NULL,
  `NumGuest` int unsigned NOT NULL,
  `Description` TEXT DEFAULT NULL,
  PRIMARY KEY (`RoomID`),
  CONSTRAINT `LimitGuest` CHECK(`NumGuest` BETWEEN 1 AND 10) ) ;
```

5. bed_info (RoomID, Size, Quantity)

This table stores the information about the bed. in this table we set both *RoomID* and *Size* as the PRIMARY KEY. *RoomID* is the FOREIGN KEY reference *room_type(roomID)*. We also set constraints '*LimitBed*' ensuring that Quantity will be between 1 and 10

```
CREATE TABLE `bed_info` (
  `RoomID` int unsigned NOT NULL,
  `Size` DECIMAL(2,1) NOT NULL,
  `Quantity` int NOT NULL DEFAULT '1',
  PRIMARY KEY (`RoomID`,`Size`),
  CONSTRAINT `FK_roomtype_bed` FOREIGN KEY (`RoomID`) REFERENCES
`room_type` (`RoomID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `LimitBed` CHECK(`Quantity` BETWEEN 1 AND 10)) ;
```

6. roomtype_branch (RoomID, BranchID, RentalPrice)

This table stores the data of each special room type of each branch. In this table we set both *RoomID* and *BranchID* as the PRIMARY KEY. While *BranchID* is the FOREIGN KEY reference *branch(branchID)*, *RoomID* is the FOREIGN KEY reference *room_type(RoomID)*

```
CREATE TABLE `roomtype_branch` (
  `RoomID` int unsigned NOT NULL,
  `BranchID` varchar(6) NOT NULL,
  `RentalPrice` int unsigned NOT NULL,
  PRIMARY KEY (`RoomID`,`BranchID`),
  KEY `BranchID2_idx` (`BranchID`),
  CONSTRAINT `BranchID2` FOREIGN KEY (`BranchID`) REFERENCES `branch`
(`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `RoomID2` FOREIGN KEY (`RoomID`) REFERENCES `room_type`
(`RoomID`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

7. room(<u>BranchID, RoomNumber</u>, RoomID, ZoneName)

This table stores the data of each room. In this table we set both *RoomNumber* and *BranchID* as the PRIMARY KEY. While *BranchID* is the FOREIGN KEY reference *zone(branchID)*, *RoomID* is the FOREIGN KEY reference *room_type(RoomID)* and *(`BranchID`, `ZoneName`)* is the FOREIGN KEY reference *zone(`BranchID`, `ZoneName`)*

```
CREATE TABLE `room` (
  `BranchID` varchar(6) NOT NULL,
  `RoomNumber` varchar(3) NOT NULL,
  `RoomID` int unsigned NOT NULL,
  `ZoneName` varchar(20) NOT NULL,
  PRIMARY KEY (`BranchID`,`RoomNumber`),
  KEY `ZoneName_idx` (`ZoneName`),
  KEY `RoomID3_idx` (`RoomID`),
  CONSTRAINT `FK_branch_room` FOREIGN KEY (`BranchID`) REFERENCES `zone`
(`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_roomtype_room` FOREIGN KEY (`RoomID`) REFERENCES
`room_type` (`RoomID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_zone_room` FOREIGN KEY (`BranchID`, `ZoneName`)
REFERENCES `zone` (`BranchID`, `ZoneName`) ON UPDATE CASCADE ON DELETE
CASCADE) ;
```

8. supply_type (<u>SupplyID</u>, SppName)

This table stores the data of each supply type. In this table we set *SppID* as the PRIMARY KEY.

```
CREATE TABLE `supply_type` (
  `SppID` varchar(6) NOT NULL PRIMARY KEY ,
  `SppName` varchar(50) NOT NULL UNIQUE) ;
```

We also set a TRIGGER that defines a regex to ensure that the incoming *sppID* is always started with VT followed by a chain of six numbers [0-9][0-9][0-9][0-9][0-9][0-9]. If the incoming *SppID* does not follow that pattern, we will get an error 'Wrong Format!!!'

```
DELIMITER $$
CREATE TRIGGER SppId_check BEFORE INSERT ON `supply_type`
FOR EACH ROW
```

```
BEGIN
IF (NEW.SppID REGEXP '^VT[0-9]{4}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
     SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
END$$
DELIMITER ;
```

9. room_type_supply_type (SppID, RoomID, Quantity)

This table stores the data of unique supply type(s) in each room type. In this table we set *(`SppID`, `RoomID`)* as the PRIMARY KEY. *RoomID* is the FOREIGN KEY reference *room_type(RoomID).* Likewise, *SppID* is the FOREIGN KEY reference *supply_type(SppID)*

```
CREATE TABLE `room_type_supply_type` (
  `SppID` varchar(8) NOT NULL,
  `RoomID` int unsigned NOT NULL,
  `Quantity` int unsigned NOT NULL DEFAULT '1',
  PRIMARY KEY (`SppID`,`RoomID`),
  KEY `RoomID4_idx` (`RoomID`),
  CONSTRAINT `FK_room` FOREIGN KEY (`RoomID`) REFERENCES `room_type`
(`RoomID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_supply_type` FOREIGN KEY (`SppID`) REFERENCES
`supply_type` (`SppID`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

10. supplies (BranchID, SppID, SupplyIndex, Condition, RoomNumber)

This table stores the data of each supply. In this table we set *(`BranchID`, `SupplyIndex`, `SppID`)* as the PRIMARY KEY. *BranchID* is the FOREIGN KEY reference for both *branch(branchID)* and *room(BranchID).* While *(`BranchID`, `RoomNumber`)* is the FOREIGN KEY reference *room(`BranchID`, `RoomNumber`),* *SuppID* is the FOREIGN KEY reference *SuppID('SppID')*

```
CREATE TABLE `supplies` (
  `BranchID` varchar(6) NOT NULL,
  `SppID` varchar(6) NOT NULL,
  `SupplyIndex` int unsigned NOT NULL,
  `Condition` varchar(45) DEFAULT 'Good',
```

```
  `RoomNumber` varchar(3) DEFAULT NULL,
  PRIMARY KEY (`BranchID`,`SupplyIndex`,`SppID`),
  KEY `SppID2_idx` (`SppID`),
  CONSTRAINT `FK_branch_supplies` FOREIGN KEY (`BranchID`) REFERENCES
`branch` (`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `BranchID51` FOREIGN KEY (`BranchID`) REFERENCES `room`
(`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_room_supplies` FOREIGN KEY (`BranchID`,`RoomNumber`)
REFERENCES `room` (`BranchID`,`RoomNumber`) ON UPDATE CASCADE ON DELETE
CASCADE,
  CONSTRAINT `FK_supply_type_supplies` FOREIGN KEY (`SppID`) REFERENCES
`supply_type` (`SppID`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

11. supplier (SupplierID, SupplierName, Mail, Address)

This table stores the data of each supplier. In this table we set *SupplierID* as the PRIMARY KEY.

```
CREATE TABLE `supplier` (
  `SupplierID` varchar(8) NOT NULL,
  `SupplierName` varchar(50) NOT NULL,
  `Mail` varchar(45) NOT NULL,
  `Address` varchar(50) NOT NULL,
  PRIMARY KEY (`SupplierID`)) ;
```

We also set a TRIGGER that defines a regex to ensure that the incoming SuppierIDs always start with NCC followed by a chain of six numbers [0-9][0-9][0-9][0-9][0-9][0-9]. If the incoming SppID does not follow that pattern, we will get an error 'Wrong Format!!!'

```
DELIMITER $$
CREATE TRIGGER SupplierID_check BEFORE INSERT ON `supplier`
FOR EACH ROW
BEGIN
IF (NEW.SupplierID REGEXP '^NCC[0-9]{4}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
     SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
END$$
DELIMITER ;
```

12. supplying (SppID, BranchID, SupplierID)

This table stores the data of each zone. In this table we set *('SppID', 'BranchID')* as the PRIMARY KEY. While *BranchID* is the FOREIGN KEY reference *branch(branchID)*, *SppID* is the FOREIGN KEY reference *supply_type(SppID)* and *SupplierID* is the FOREIGN KEY reference *supplier(SupplierID)*.

```
CREATE TABLE `supplying` (
  `SppID` varchar(8) NOT NULL,
  `BranchID` varchar(6) NOT NULL,
  `SupplierID` varchar(8) NOT NULL,
  PRIMARY KEY (`SppID`,`BranchID`),
  KEY `SupplierID_idx` (`SupplierID`),
  KEY `BranchID6_idx` (`BranchID`),
  CONSTRAINT `FK_branch_supplying` FOREIGN KEY (`BranchID`) REFERENCES
`branch` (`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_supply_type_supplying` FOREIGN KEY (`SppID`) REFERENCES
`supply_type` (`SppID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_supplier_supplying` FOREIGN KEY (`SupplierID`)
REFERENCES `supplier` (`SupplierID`) ON UPDATE CASCADE ON DELETE CASCADE)
;
```

13. customer (CustomerID, CitizenID, FullName, Phone, Email, Username, Password, Point, CustomerType)

This table stores the data of each customer. In this table we set *CustomerID* as the PRIMARY KEY. We also set the attributes *CitizenID*, *Phone*, *Email*, *Username* as UNIQUE KEYS since those attributes cannot be duplicated.

```
CREATE TABLE `customer` (
  `CustomerID` varchar(10) NOT NULL,
  `CitizenID` varchar(12) NOT NULL,
  `FullName` varchar(45) NOT NULL,
  `Phone` varchar(12) NOT NULL,
  `Email` varchar(45) DEFAULT NULL,
  `Username` varchar(45) DEFAULT NULL,
  `Password` varchar(45) DEFAULT NULL,
  `Point` int unsigned DEFAULT '0',
  `CustomerType` int unsigned DEFAULT '1',
```

```
  PRIMARY KEY (`CustomerID`),
  UNIQUE KEY `CitizenID_UNIQUE` (`CitizenID`),
  UNIQUE KEY `Phone_UNIQUE` (`Phone`),
  UNIQUE KEY `Email_UNIQUE` (`Email`),
  UNIQUE KEY `UserName_UNIQUE` (`Username`)) ;
```

We also set a TRIGGER that defines a regex to ensure that the incoming *CustomerIDs* always start with KH followed by a chain of six numbers [0-9][0-9][0-9][0-9][0-9][0-9]. If the incoming *CustomerID* does not follow that pattern, we will get an error 'Wrong Format!!!'

```
DELIMITER $$
CREATE TRIGGER CustomerID_check BEFORE INSERT ON `customer`
FOR EACH ROW
BEGIN
IF (NEW.CustomerID REGEXP '^KH[0-9]{6}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
     SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
END$$
DELIMITER ;
```

14. service_packet (PackageName, DayNum, GuestNum, Price)

This table stores the data of the packets that the hotel offers. We set *PackageName* as the PRIMARY KEY. We also set constraints specifying that *DayNum* must be between 1 and 100; and *GuestNum* must be between 1 and 10.

```
CREATE TABLE `service_packet` (
  `PackageName` varchar(50) NOT NULL,
  `DayNum` int unsigned NOT NULL,
  `GuestNum` int unsigned NOT NULL,
  `Price` int NOT NULL,
  PRIMARY KEY (`PackageName`),
  CONSTRAINT `LimitDay` CHECK(`DayNum` BETWEEN 1 AND 100),
  CONSTRAINT `LimitGuestPacket` CHECK(`GuestNum` BETWEEN 1 AND 10)) ;
```

15. packet_bill (CustomerID, PackageName, PurchaseDate, StartDate, TotalPay, RemainDay)

This table stores the data of the packets that are purchased by customers. We set *(CustomerID, PackageName, PurchaseDate)* as the PRIMARY KEY of the table. *CustomerID* is the FOREIGN KEY reference *customer(CustomerID)*. Likewise, *PackageName* is the FOREIGN KEY reference to *service_packet (PackageName)*. CONSTRAINT *CheckDate* is set to ensure that *StartDate> PurchaseDate.*

```
CREATE TABLE `packet_bill` (
  `CustomerID` varchar(10) NOT NULL,
  `PackageName` varchar(50) NOT NULL,
  `PurchaseDate` datetime NOT NULL,
  `StartDate` datetime NOT NULL,
  `TotalPay` int DEFAULT NULL,
  `RemainDay` int unsigned DEFAULT NULL,
  PRIMARY KEY (`CustomerID`,`PackageName`,`PurchaseDate`),
  KEY `PName1_idx` (`PackageName`),
  CONSTRAINT `FK_customer_bill` FOREIGN KEY (`CustomerID`) REFERENCES
`customer` (`CustomerID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_packet_bill` FOREIGN KEY (`PackageName`) REFERENCES
`service_packet` (`PackageName`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `CheckDate` CHECK(StartDate > PurchaseDate)) ;
```

16. booking (BookingID, BookingDate, GuestNum, CheckIn, CheckOut, Status, TotalPay, CustomerID, PackageName)

This table stores the data of each booking. *BookingID* is the PRIMARY KEY of the table. *CustomerID* and *PackageName* are the FOREIGN KEYs reference *customer(CustomerID)* and *service_package(PackageName)* respectively. Constraints are set to ensure that *CheckIn > BookingDate* and *CheckOut > CheckIn.*

```
CREATE TABLE `booking` (
  `BookingID` varchar(20) NOT NULL,
  `BookingDate` datetime NOT NULL,
  `GuestNum` int unsigned NOT NULL,
```

```
  `CheckIn` datetime NOT NULL,
  `CheckOut` datetime NOT NULL,
  `Status` int unsigned NOT NULL DEFAULT '0',
  `TotalPay` int unsigned NOT NULL DEFAULT '0',
  `CustomerID` varchar(10) NOT NULL,
  `PackageName` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`BookingID`),
  KEY `CusID3_idx` (`CustomerID`),
  KEY `PName_idx` (`PackageName`),
  CONSTRAINT `FK_customer_booking` FOREIGN KEY (`CustomerID`) REFERENCES
`customer` (`CustomerID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `FK_package_booking` FOREIGN KEY (`PackageName`) REFERENCES
`service_packet` (`PackageName`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `CheckDate2` CHECK(CheckIn > BookingDate),
  CONSTRAINT `CheckDate3` CHECK(CheckOut > CheckIn)) ;
```

We also used TRIGGER to auto generate *BookingID* for each booking being added to the table. In the TRIGGER, the *BookingID* is defined by the pattern DP[DDMMYYYY][id] with DDMMYYYY is the date of the booking and *id* is an auto increment value created by table *booking_seq*.

```
CREATE TABLE booking_seq(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT
);
DELIMITER $$
CREATE TRIGGER generate_booking_id
BEFORE INSERT ON `booking`
FOR EACH ROW
BEGIN
  INSERT INTO booking_seq VALUES (NULL);
  SET NEW.BookingID = CONCAT('DP', DATE_FORMAT(CAST(NEW.BookingDate as
DATE),'%d%m%Y'), LPAD(LAST_INSERT_ID(), 6, '0'));
END$$
DELIMITER ;
```

17. booking_room (<u>BookingID, BranchID, RoomNumber</u>)

This table stores data of the booking room in each booking. All three attributes (*BookingID*, *BranchID*, *RoomNumber*) are set as PRIMARY KEY. *BookingID* is the FOREIGN KEY reference *booking (BookingID)*. *(BranchID, RoomNumber)* is the FOREIGN KEY reference *room (BranchID, RoomNumber)*.

```
CREATE TABLE `booking_room` (
  `BookingID` varchar(20) NOT NULL,
  `BranchID` varchar(6) NOT NULL,
  `RoomNumber` varchar(3) NOT NULL,
  PRIMARY KEY (`BookingID`,`BranchID`,`RoomNumber`),
  KEY `BranchID_idx` (`BranchID`),
  CONSTRAINT `BookingID` FOREIGN KEY (`BookingID`) REFERENCES `booking`
(`BookingID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `BranchID7` FOREIGN KEY (`BranchID`,`RoomNumber`) REFERENCES
`room` (`BranchID`,`RoomNumber`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

18. booking_bill (BillID, CheckIn, CheckOut, BookingID)

This table stores data about the Bill of each booking. *BillID* is the PRIMARY KEY of the table. *BookingID* is the FOREIGN KEY reference *booking(BookingID)*.

```
CREATE TABLE `booking_bill` (
  `BillID` varchar(20) NOT NULL,
  `CheckIn` TIME NOT NULL,
  `CheckOut` TIME NOT NULL,
  `BookingID` varchar(20) NOT NULL,
  PRIMARY KEY (`BillID`),
  KEY `BookingID_idx` (`BookingID`),
  CONSTRAINT `BookingID2` FOREIGN KEY (`BookingID`) REFERENCES `booking`
(`BookingID`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

TRIGGER is used to auto generate *BillID* for each bill being added to the table. In the TRIGGER, the *BillID* is defined by the pattern HD[DDMMYYYY][id] with DDMMYYYY is the current date and id is an auto increment value created by table *booking_bill_seq*.

```
CREATE TABLE booking_bill_seq(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT
);
DELIMITER $$
CREATE TRIGGER generate_booking_bill_id
BEFORE INSERT ON `booking_bill`
FOR EACH ROW
BEGIN
  INSERT INTO booking_bill_seq VALUES (NULL);
```

```
  SET NEW.BillID = CONCAT('HD', DATE_FORMAT(CAST(NOW() as
DATE),'%d%m%Y'), LPAD(LAST_INSERT_ID(), 6, '0'));
END$$
DELIMITER ;
```

19. enterprise (EnterpriseID, EnterpriseName)

This table stores data about the enterprises that collaborate with the hotel. *EnterpriseID* is the PRIMARY KEY of the table.

```
CREATE TABLE `enterprise` (
  `EnterpriseID` varchar(8) NOT NULL,
  `EnterpriseName` varchar(45) NOT NULL,
  PRIMARY KEY (`EnterpriseID`)) ;
```

We also use a TRIGGER that defines a regex to ensure that the incoming *EnterpriseIDs* always start with DN followed by a chain of four numbers [0-9][0-9][0-9][0-9]. If the incoming *EnterpriseID* does not follow that pattern, we will get an error *'Wrong Format!!!'*

```
DELIMITER $$
CREATE TRIGGER EnterpriseID_check BEFORE INSERT ON `enterprise`
FOR EACH ROW
BEGIN
IF (NEW.EnterpriseID REGEXP '^DN[0-9]{4}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
     SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
END$$
DELIMITER ;
```

20. service (ServiceID, ServiceType, GuestNum, Style, EnterpriseID)

This table stores data about services that the enterprises can offer. *ServiceID* is the PRIMARY KEY of the table. *EnterpriseID* is the FOREIGN KEY reference *enterprise(EnterpriseID)*.

```
CREATE TABLE `service` (
  `ServiceID` varchar(8) NOT NULL,
  `ServiceType` varchar(1) NOT NULL,
  `GuestNum` int unsigned DEFAULT '0',
  `Style` varchar(45) DEFAULT NULL,
```

```
  `EnterpriseID` varchar(10) NOT NULL,
  PRIMARY KEY (`ServiceID`),
  CONSTRAINT `FK_enterprise_service` FOREIGN KEY (`EnterpriseID`)
REFERENCES `enterprise` (`EnterpriseID`) ON UPDATE CASCADE ON DELETE
CASCADE) ;
```

TRIGGER is also used to ensure that the incoming *ServiceID* follows a pattern of *DV[R/S/C/M/B][0-9][0-9][0-9]* with *R* stands for *Restaurant*, *S* stands for *Spa*, *C* stands for *Convenience* store, *M* stands for *Souvenir store* and *B* stands for *Bar*. If the incoming *ServiceID* does not follow that pattern, we will get an error *'Wrong Format!!!'*

```
DELIMITER $$
CREATE TRIGGER ServiceID_check BEFORE INSERT ON `service`
FOR EACH ROW
BEGIN
IF (NEW.ServiceID REGEXP '^DV[RSCMB][0-9]{3}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
IF (SUBSTR(NEW.ServiceID,3,1) <> NEW.ServiceType) THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong ServiceType!!!';
END IF;
END$$
DELIMITER ;
```

21. spa_service (<u>ServiceID</u>, <u>ProvidedService</u>)

This table stores data about the services offered by the spa. *(ServiceID, ProvidedService)* is the PRIMARY KEY of the table. *ServiceID* is the FOREIGN KEY reference *service (ServiceID)*.

```
CREATE TABLE `spa_service` (
  `ServiceID` varchar(10) NOT NULL,
  `ProvidedService` varchar(50) NOT NULL,
  PRIMARY KEY (`ServiceID`,`ProvidedService`),
  CONSTRAINT `FK_service_spa` FOREIGN KEY (`ServiceID`) REFERENCES
`service` (`ServiceID`) ON UPDATE CASCADE ON DELETE CASCADE) ;
```

TRIGGER is also used to ensure that the incoming *ServiceID* follows a pattern of *DV[R/S/C/M/B][0-9][0-9][0-9]*. If the incoming *ServiceID* does not

follow that pattern, we will get an error *'Wrong Format!!!'*. Then it will check if the incoming *ServiceID* follows the pattern of *DV[S][0-9][0-9][0-9]*. If it does not, we will get an error *'Wrong ServiceType!!!'*.

```
DELIMITER $$
CREATE TRIGGER Spa_ServiceID_check BEFORE INSERT ON `spa_service`
FOR EACH ROW
BEGIN
IF (NEW.ServiceID REGEXP '^DV[RSCMB][0-9]{3}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
IF (SUBSTR(NEW.ServiceID,3,1) <> 'S') THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong ServiceType!!!';
END IF;
END$$
DELIMITER ;
```

22. souvenir_category (<u>ServiceID, Category</u>)

This data stores data about the Category of products that souvenir stores offer. *(ServiceID, Category)* is the PRIMARY KEY of the table.

```
CREATE TABLE `souvenir_category` (
  `ServiceID` varchar(10) NOT NULL,
  `Category` varchar(20) NOT NULL,
  PRIMARY KEY (`ServiceID`, `Category`),
  CONSTRAINT `svID2` FOREIGN KEY (`ServiceID`) REFERENCES `service`
(`ServiceID`) ON UPDATE CASCADE ON DELETE CASCADE
) ;
```

TRIGGER is used to ensure that the incoming *ServiceID* follows a pattern of *DV[R/S/C/M/B][0-9][0-9][0-9]*. If the incoming *ServiceID* does not follow that pattern, we will get an error *'Wrong Format!!!'*. Then it will check if the incoming *ServiceID* follows the pattern of *DV[M][0-9][0-9][0-9]*. If it does not, we will get an error *'Wrong ServiceType!!!'*.

```
DELIMITER $$
CREATE TRIGGER Souvenir_ServiceID_check1 BEFORE INSERT ON
`souvenir_category`
FOR EACH ROW
```

```
BEGIN
IF (NEW.ServiceID REGEXP '^DV[RSCMB][0-9]{3}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
IF (SUBSTR(NEW.ServiceID,3,1) <> 'M') THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong ServiceType!!!';
END IF;
END$$
DELIMITER ;
```

23. souvenir_brand (ServiceID, Brand)

This data stores data about the Brand of products that souvenir stores offer. *(ServiceID, Brand)* is the PRIMARY KEY of the table.

```
CREATE TABLE `souvenir_brand` (
  `ServiceID` varchar(10) NOT NULL,
  `Brand` varchar(45) NOT NULL,
  PRIMARY KEY (`ServiceID`, `Brand`),
  CONSTRAINT `svID3` FOREIGN KEY (`ServiceID`) REFERENCES `service`
(`ServiceID`) ON UPDATE CASCADE ON DELETE CASCADE
) ;
```

TRIGGER is used to ensure that the incoming *ServiceID* follows a pattern of *DV[R/S/C/M/B][0-9][0-9][0-9]*. If the incoming *ServiceID* does not follow that pattern, we will get an error *'Wrong Format!!!'*. Then it will check if the incoming *ServiceID* follows the pattern of *DV[M][0-9][0-9][0-9]*. If it does not, we will get an error *'Wrong ServiceType!!!'*.

```
DELIMITER $$
CREATE TRIGGER Souvenir_ServiceID_check2 BEFORE INSERT ON
`souvenir_brand`
FOR EACH ROW
BEGIN
IF (NEW.ServiceID REGEXP '^DV[RSCMB][0-9]{3}$' ) = 0 THEN
  SIGNAL SQLSTATE '12345'
   SET MESSAGE_TEXT = 'Wrong Format!!!';
END IF;
IF (SUBSTR(NEW.ServiceID,3,1) <> 'M') THEN
  SIGNAL SQLSTATE '12345'
```

```
   SET MESSAGE_TEXT = 'Wrong ServiceType!!!';
END IF;
END$$
DELIMITER ;
```

24. block (BranchID, BlockID, Length, Width, RentalPrice, Description, ServiceID, StoreName, Logo)

This table stores data about blocks of each branch. *(BranchID, BlockID)* is the PRIMARY KEY of the table. *BranchID* is a FOREIGN KEY reference *branch (BranchID)*. *ServiceID* is a FOREIGN KEY reference *service (ServiceID)*. CONSTRAINT *CheckBlockID* is set to ensure that values of incoming *BlockID* must be between 1 and 50.

```
CREATE TABLE `block` (
  `BranchID` varchar(6) NOT NULL,
  `BlockID` int unsigned NOT NULL DEFAULT '1',
  `Length` float NOT NULL,
  `Width` float NOT NULL,
  `RentalPrice` int unsigned NOT NULL,
  `Description` text,
  `ServiceID` varchar(10) DEFAULT NULL,
  `StoreName` varchar(50) DEFAULT NULL,
  `Logo` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`BranchID`,`BlockID`),
  KEY `svID4_idx` (`ServiceID`),
  CONSTRAINT `FK_branch_block` FOREIGN KEY (`BranchID`) REFERENCES
`branch` (`BranchID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `svID4` FOREIGN KEY (`ServiceID`) REFERENCES `service`
(`ServiceID`) ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT `CheckBlockID` CHECK(`BlockID` BETWEEN 1 AND 50)
) ;
```

25. store_image(BranchID, BlockID, Image)

This table stores the data about the image of each store. *(BranchID, BlockID, Image)* is the PRIMARY KEY of the table. *(BranchID, BlockID)* is the FOREIGN KEY reference *block (BranchID, BlockID)*. CONSTRAINT

LimitBlockID is set to ensure that values of incoming *BlockID* must be between 1 and 50.

```
CREATE TABLE `store_image` (
  `BranchID` varchar(6) NOT NULL,
  `BlockID` int unsigned NOT NULL DEFAULT '1',
  `Image` varchar(60) NOT NULL,
  PRIMARY KEY (`BranchID`,`BlockID`,`Image`),
  KEY `BlockID_idx` (`BlockID`),
  CONSTRAINT `BlockID2` FOREIGN KEY (`BranchID`,`BlockID`) REFERENCES
`block` (`BranchID`,`BlockID`) ON UPDATE CASCADE ON DELETE CASCADE
) ;
```

26. active_time (BranchID, BlockID, openTime, closeTime)

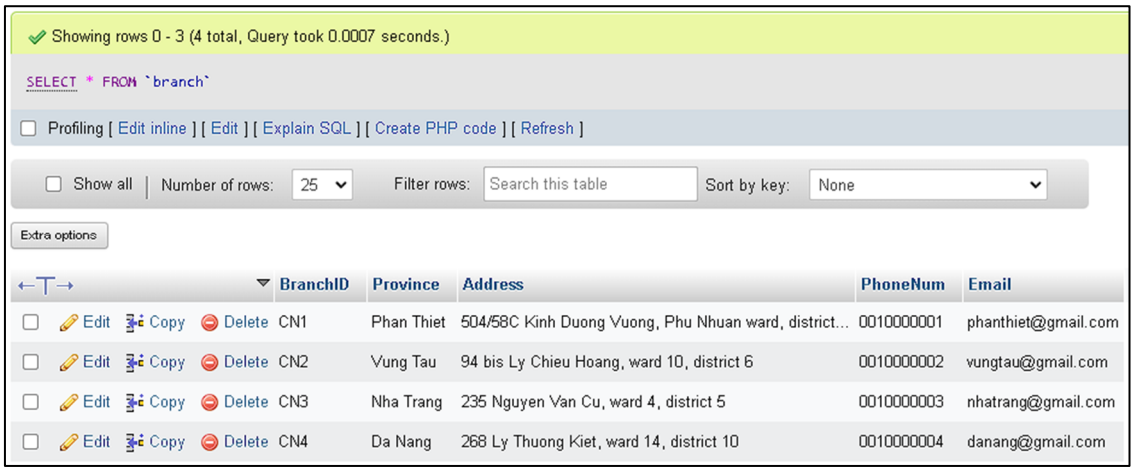
This table stores data about the active time of each block at each branch. *(BranchID, BlockID, openTime)* is the PRIMARY KEY of the table. *(BranchID, BlockID)* is the FOREIGN KEY reference *block (BranchID, BlockID)*.

```
CREATE TABLE `active_time` (
  `BranchID` varchar(6) NOT NULL,
  `BlockID` int unsigned NOT NULL DEFAULT '1',
  `openTime` TIME NOT NULL,
  `closeTime` TIME NOT NULL,
  PRIMARY KEY (`BranchID`,`BlockID`,`openTime`),
  KEY `BlockID_idx` (`BlockID`),
  CONSTRAINT `BlockID3` FOREIGN KEY (`BranchID`,`BlockID`) REFERENCES
`block` (`BranchID`,`BlockID`) ON UPDATE CASCADE ON DELETE CASCADE
) ;
```

II. Insert data

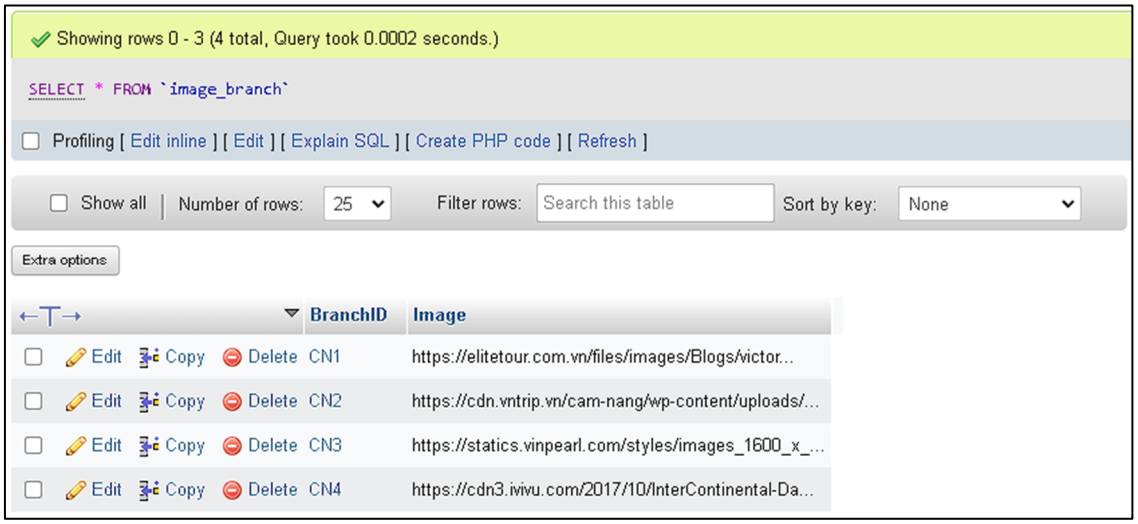
1. branch (BranchID, Province, Address, PhoneNum, Email)

In this table, we add 4 entries.



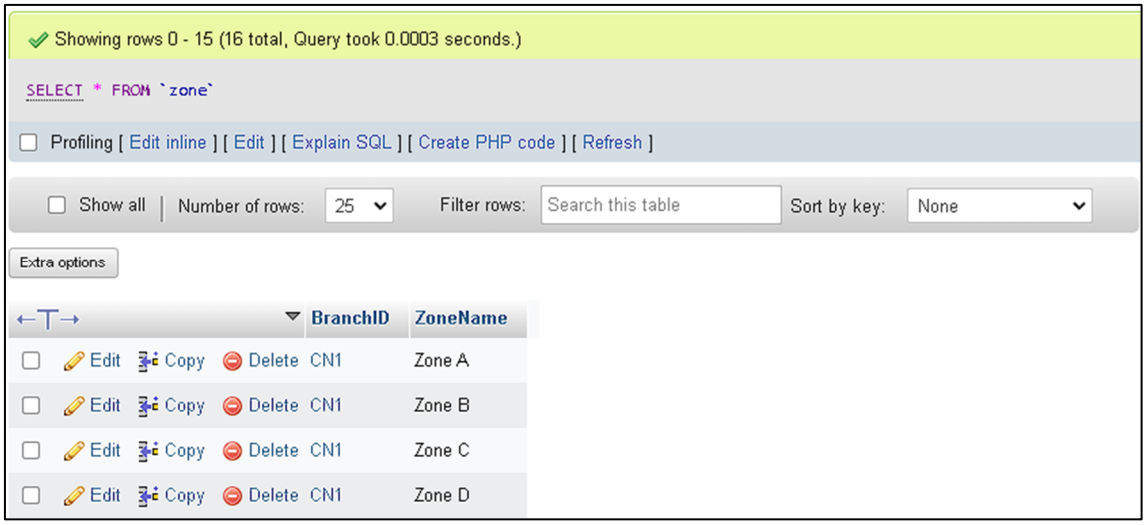
2. image_branch (BranchID, Image)

In this table, we add 4 entries.



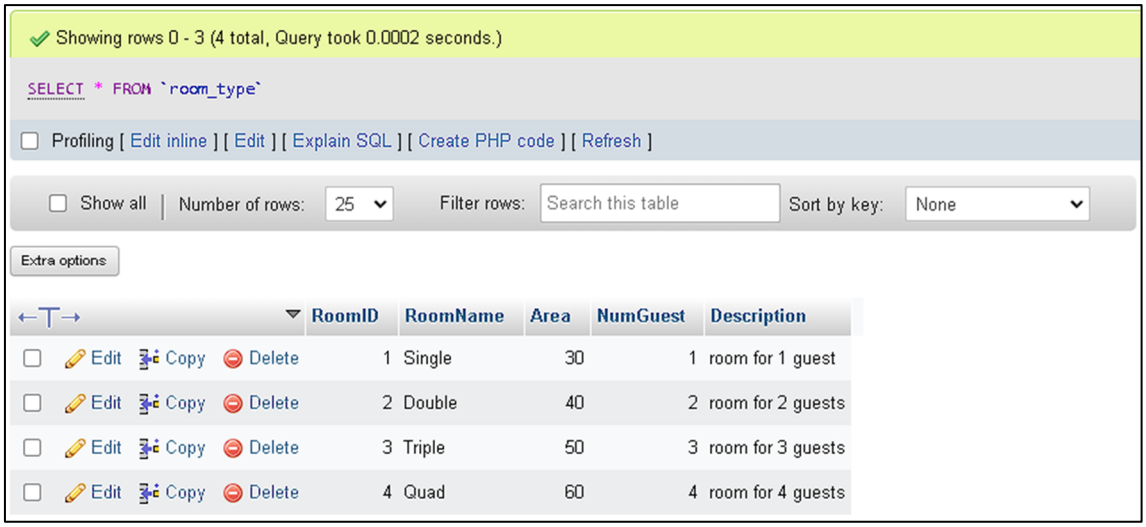
3. zone (<u>BranchID, ZoneName</u>)

In this table, we add 4 entries.



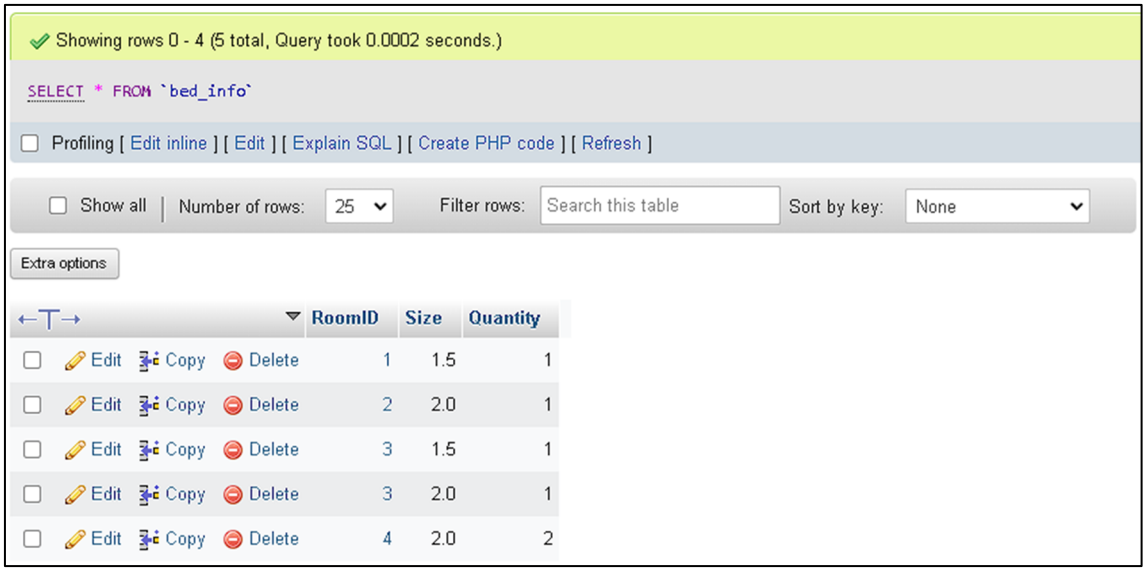
4. room_type (<u>RoomID</u>, RoomName, Area, NumGuest, Description)

In this table, we add 4 entries.



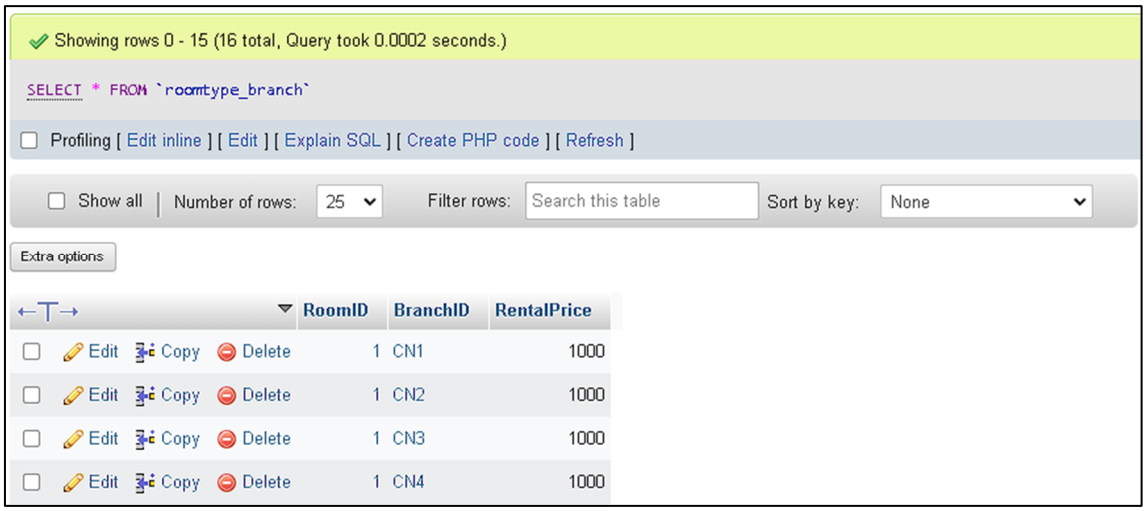
5. bed_info (RoomID, Size, Quantity)

In this table, we add 5 entries.



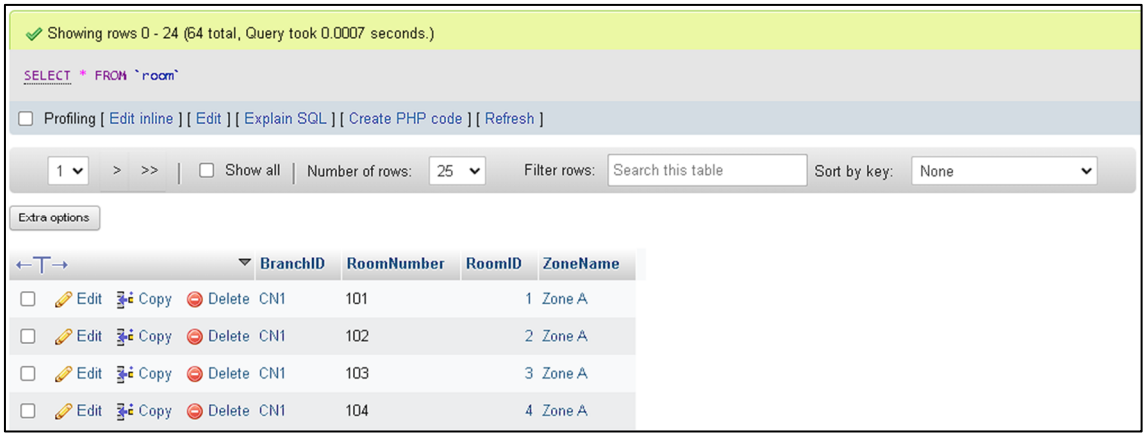
6. roomtype_branch (RoomID, BranchID, RentalPrice)

In this table, we add 16 entries



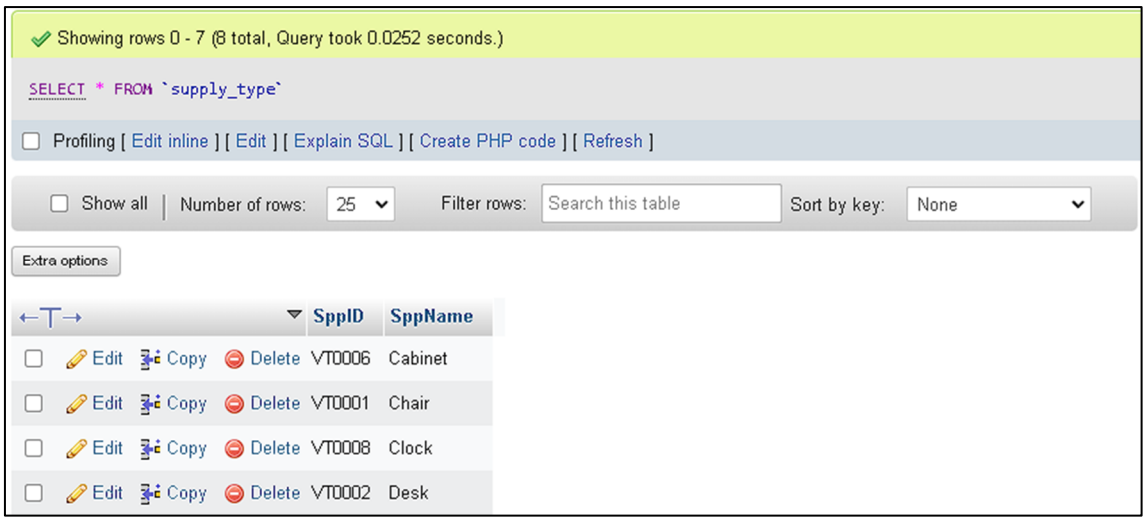
7. room(<u>BranchID, RoomNumber</u>, RoomID, ZoneName)

In this table, we add 64 entries



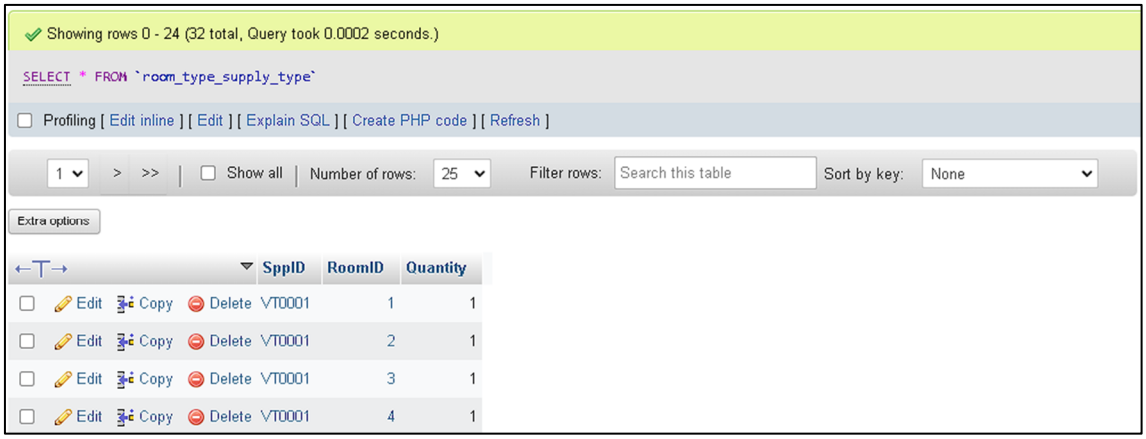
8. supply_type (<u>SupplyID</u>, SppName)

In this table, we add 8 entries



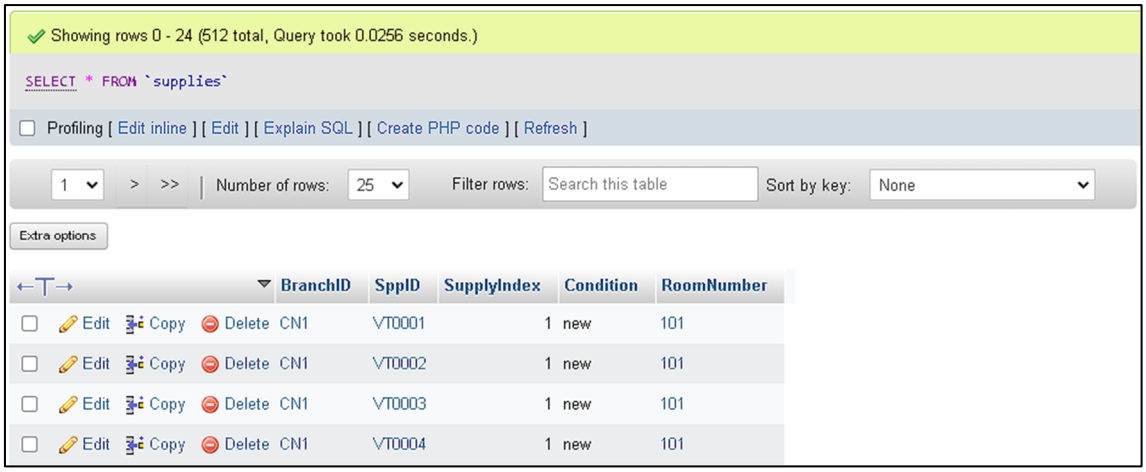
9. room_type_supply_type (<u>SppID, RoomID</u>, Quantity)

In this table, we add 64 entries.



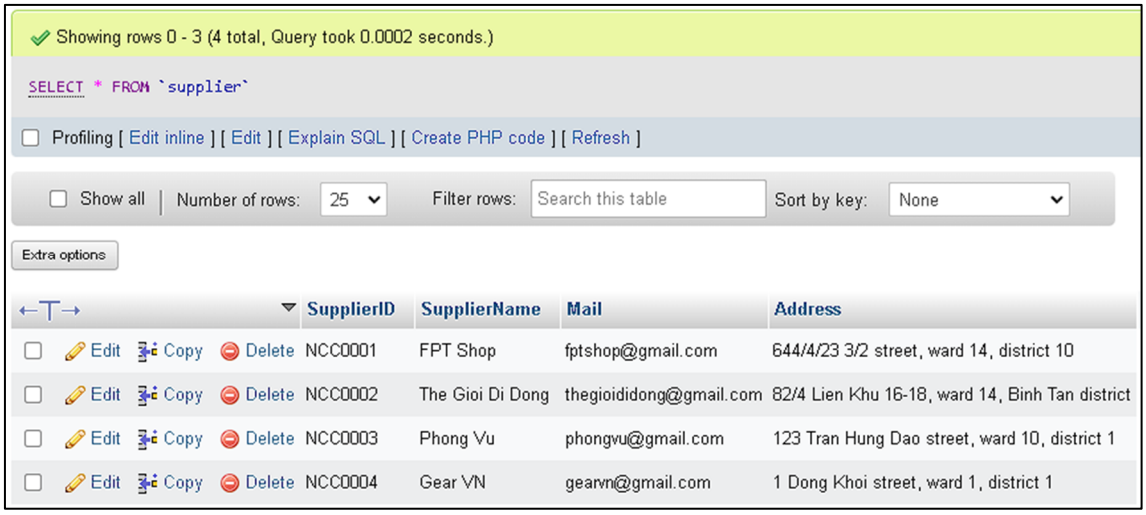
10. supplies (<u>BranchID, SppID, SupplyIndex</u>, Condition, RoomNumber)

In this table, we add 512 entries



11. supplier (SupplierID, SupplierName, Mail, Address)

In this table, we add 4 entries

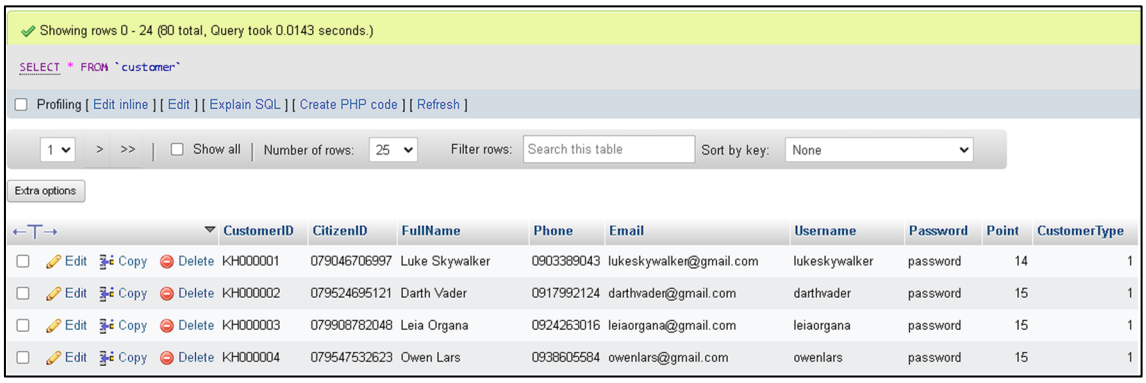


12. supplying (SppID, BranchID, SupplierID)

In this table, we add 32 entries

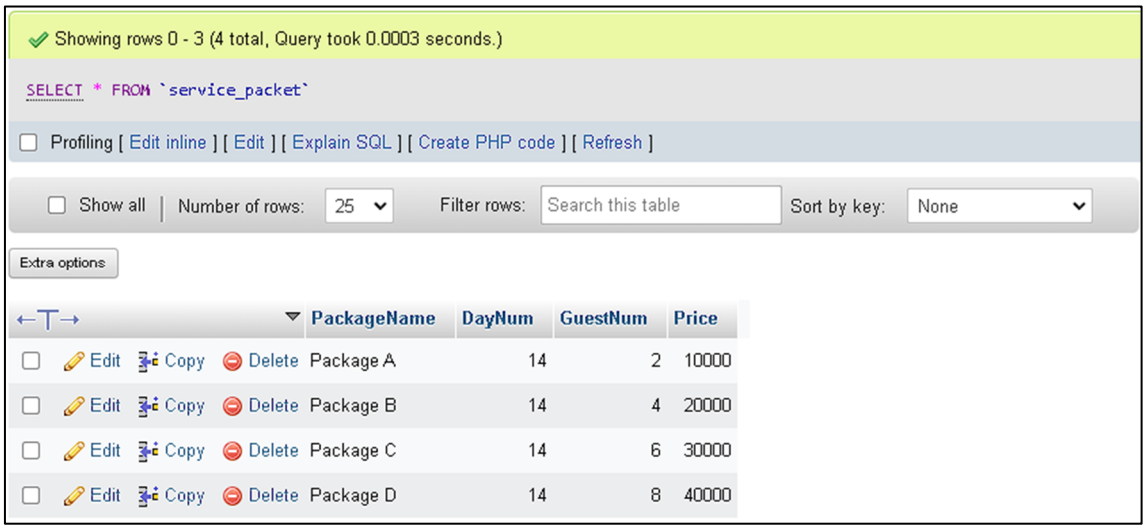
13. customer (CustomerID, CitizenID, FullName, Phone, Email, Username, Password, Point, CustomerType)

In this table, we add 80 entries.



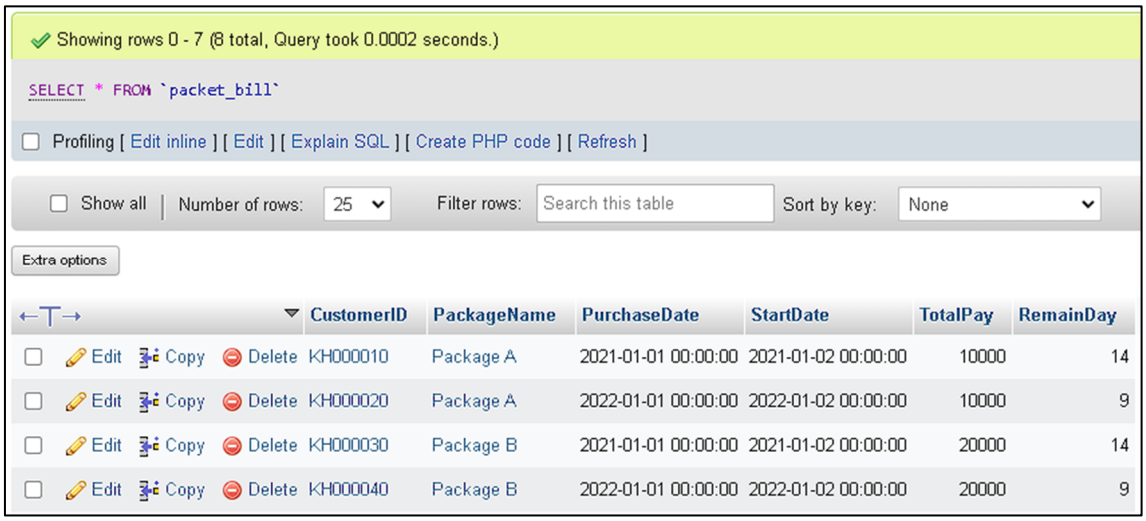
14. service_packet (PackageName, DayNum, GuestNum, Price)

In this table, we add 4 entries.



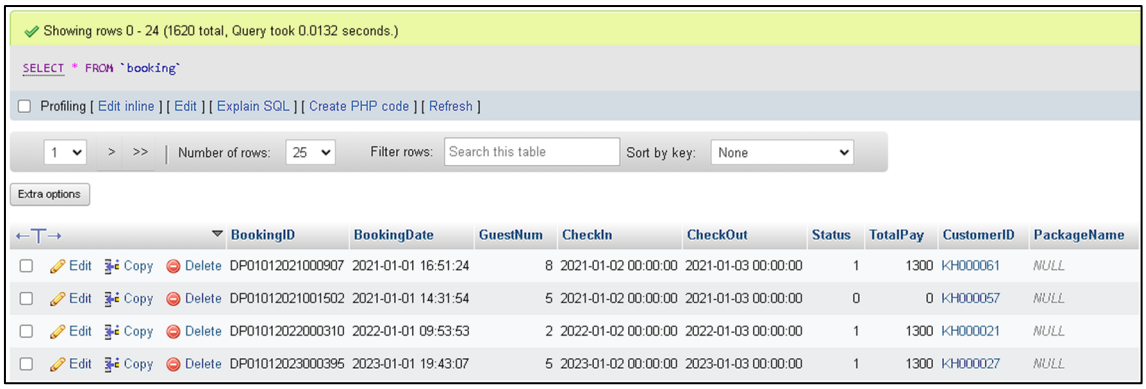
15. packet_bill (CustomerID, PackageName, PurchaseDate, StartDate, TotalPay, RemainDay)

In this table, we add 8 entries



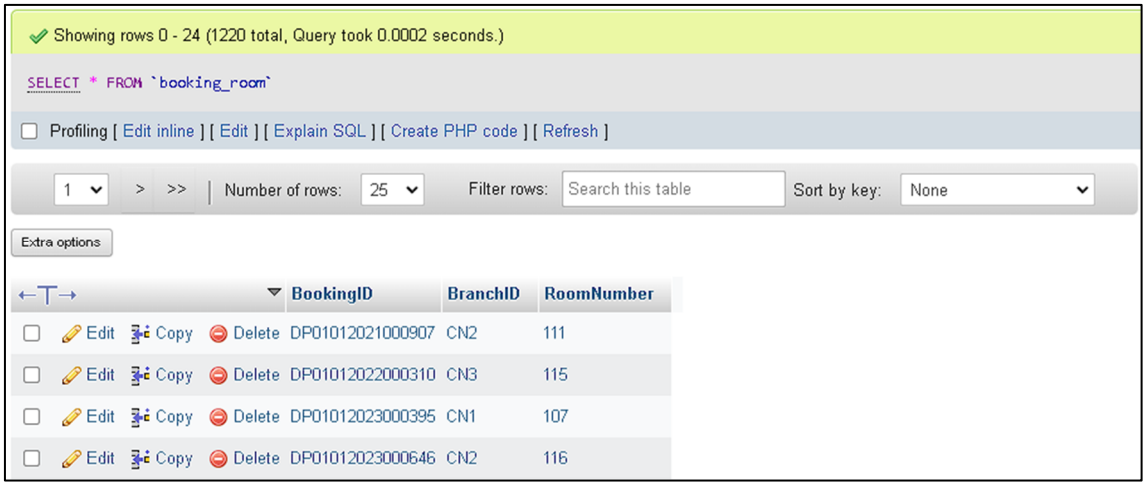
16. booking (BookingID, BookingDate, GuestNum, CheckIn, CheckOut, Status, TotalPay, CustomerID, PackageName)

In this table, we add 1620 entries



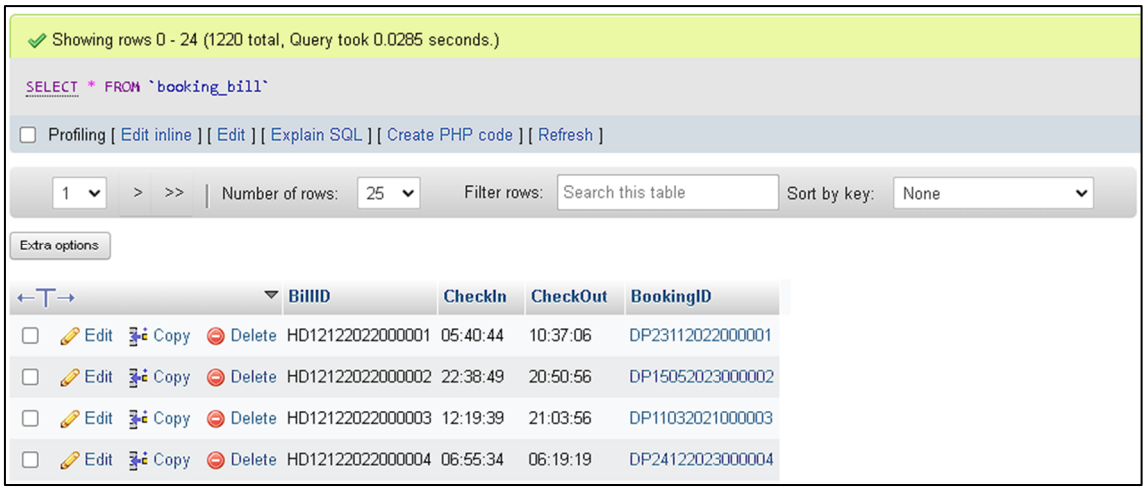
17. booking_room (<u>BookingID, BranchID, RoomNumber</u>)

In this table, we add 1220 entries.



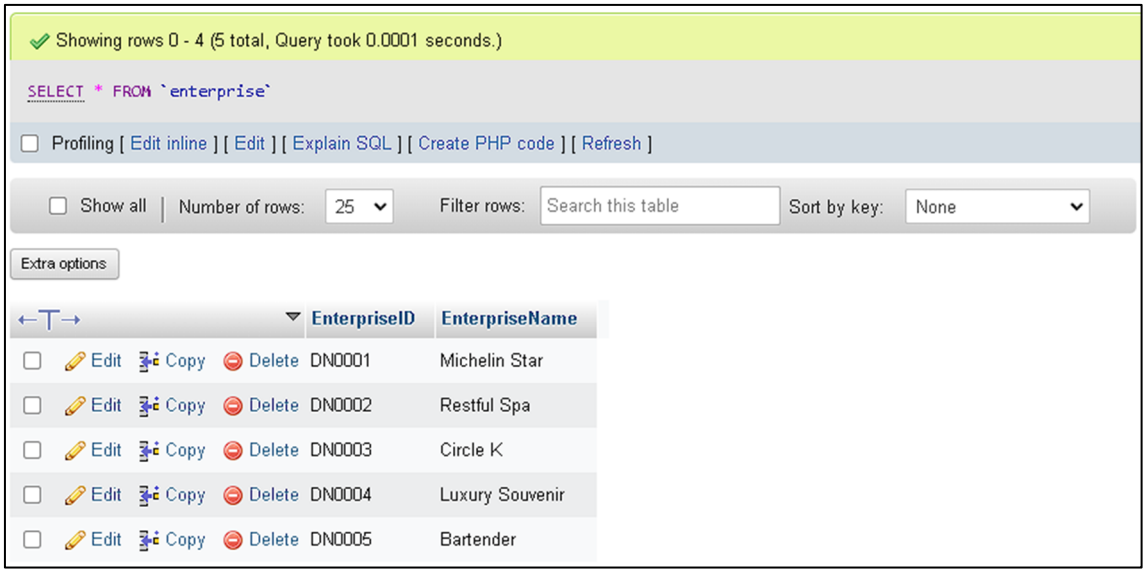
18. booking_bill (<u>BillID</u>, CheckIn, CheckOut, BookingID)

In this table, we add 1220 entries.



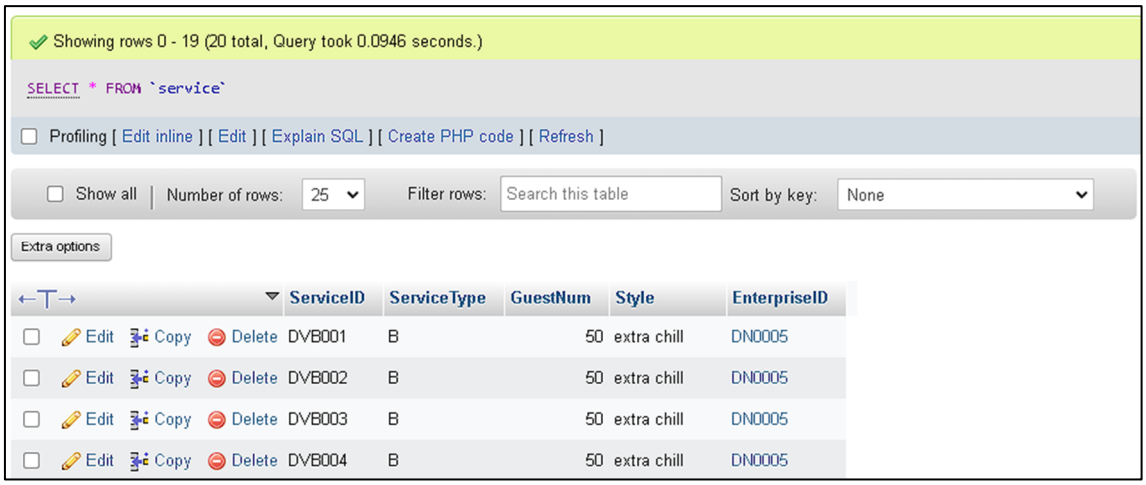
19. enterprise (<u>EnterpriseID</u>, EnterpriseName)

In this table, we add 5 entries



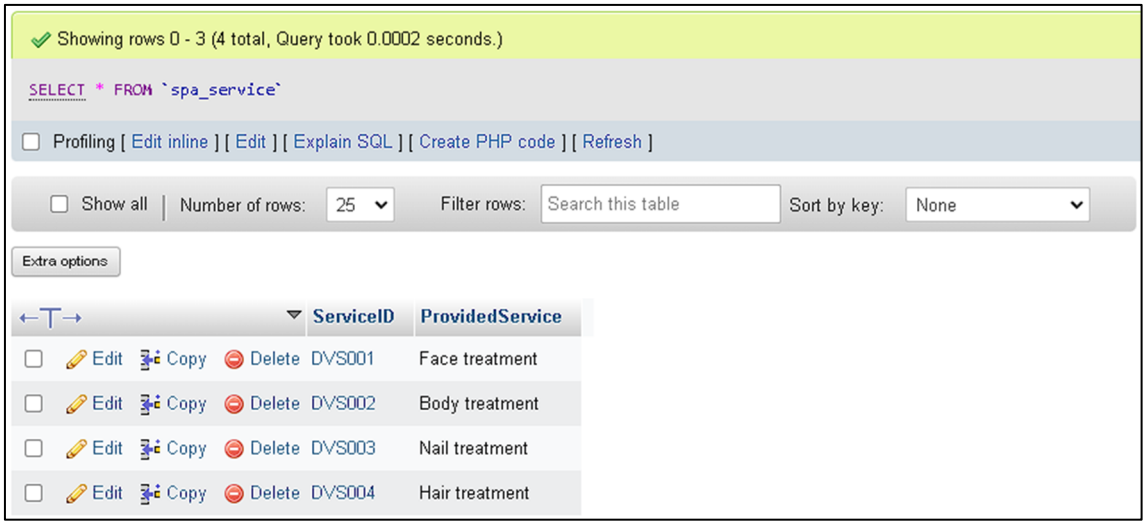
20. service (<u>ServiceID</u>, ServiceType, GuestNum, Style, EnterpriseID)

In this table, we add 20 entries.



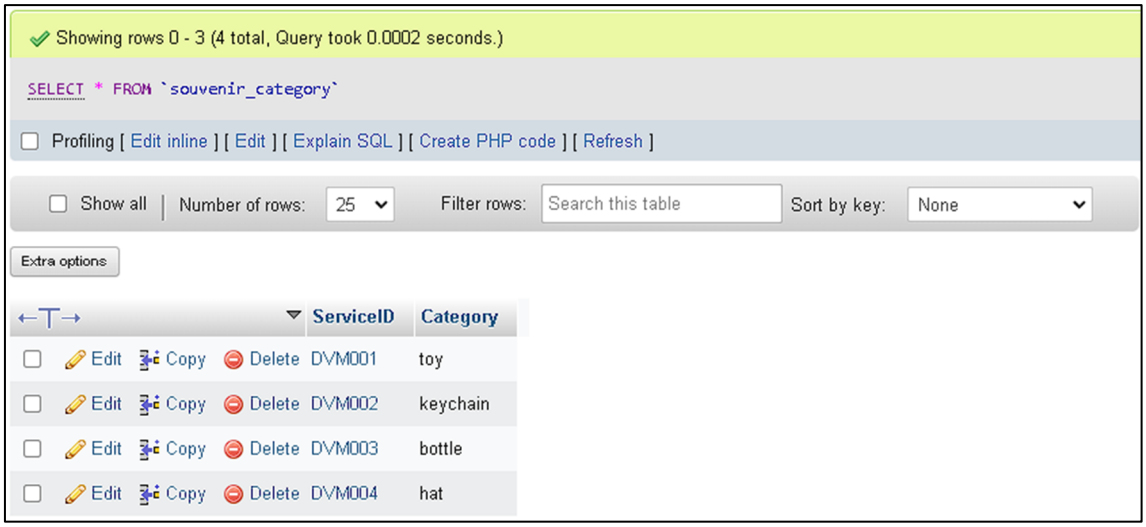
21. spa_service (ServiceID, ProvidedService)

In this table, we add 4 entries



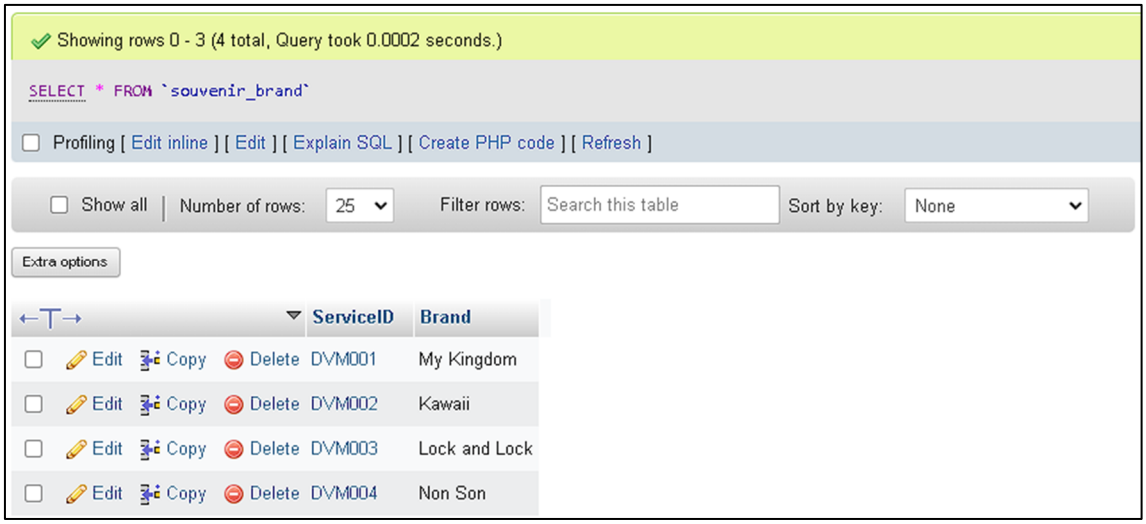
22. souvenir_category (ServiceID, Category)

In this table, we add 4 entries.



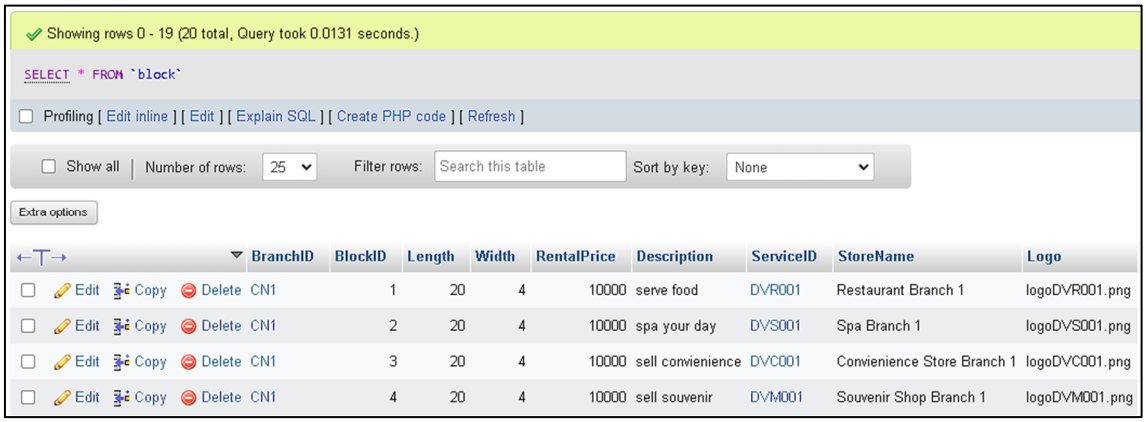
23. souvenir_brand (ServiceID, Brand)

In this table, we add 4 entries



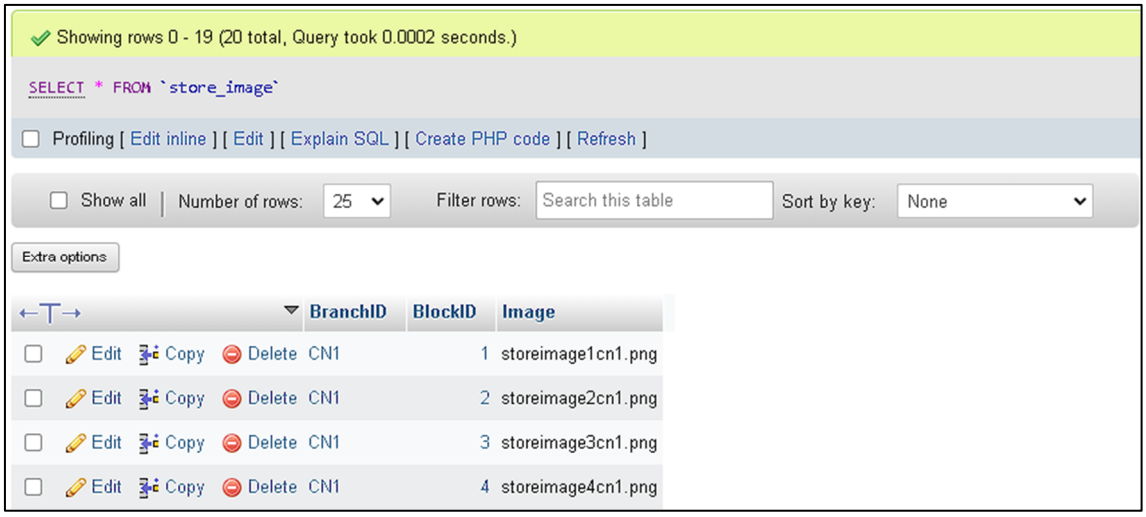
24. block (BranchID, BlockID, Length, Width, RentalPrice, Description, ServiceID, StoreName, Logo)

In this table, we add 20 entries



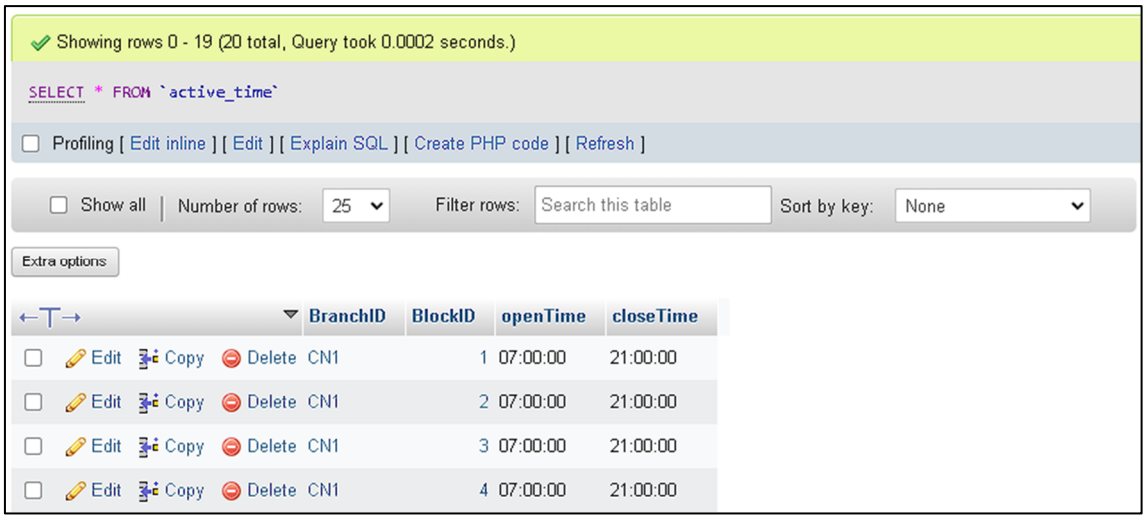
25. store_image(<u>BranchID, BlockID, Image</u>)

In this table, we add 20 entries.



26. active_time (<u>BranchID, BlockID, openTime</u>, closeTime)

In this table, we add 20 entries.



PART 2: STORE PROCEDURE, FUNCTION, TRIGGER

I. Stored Procedure/Function

1. Write procedure/function GoiDichVu to show valid service packets of a customer.

Input: *CustomerID*

Output: *<Package_Name, Number_of_Guest, Start_Date, Expired_Date, Remaining_Day>*

```
DELIMITER //
DROP PROCEDURE IF EXISTS GoiDichVu //
CREATE PROCEDURE GoiDichVu
(
  in customer_id varchar(10)
)
BEGIN

  SELECT packet_bill.PackageName AS Package_Name,
      service_packet.GuestNum AS Number_Of_Guest,
            packet_bill.StartDate AS Start_Date,
            (packet_bill.StartDate + INTERVAL 1 YEAR) AS Expire_Date,
            packet_bill.RemainDay AS Remaining_Day
  FROM packet_bill INNER JOIN service_packet
    ON  packet_bill.PackageName = service_packet.PackageName
  WHERE packet_bill.CustomerID = customer_id
    HAVING Expire_Date > NOW();

END; //
DELIMITER ;
```

The GoiDichVu procedure will be written as above, the procedure has an IN parameter *customer_id* and returns a table which has 5 columns *<Package_Name, Number_Of_Guest, Start_Date, Expire_Date, Remaining_Day>* as a result.

In the procedure, the *Expire_Date* is calculated by taking the *StartDate* and adding 1 Year to it. To retrieve the number of guests in a package, we have to perform an INNER JOIN of *packet_bill* and *service_packet*. Finally, we choose only the rows which satisfy the criteria: Package belongs to the given Customer and isn't expired.

```
655 •  CALL GoiDichVu('KH000020');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Package_Name	Number_Of_Guest	Start_Date	Expire_Date	Remaining_Day
Package A	2	2022-01-02 00:00:00	2023-01-02 00:00:00	14

2. Write procedure/function ThongKeLuotKhach to show statistics of a branch in a year.

Input: *Branch ID, Year*

Output: *<Month, Total_Guest>*

```
DELIMITER //
CREATE PROCEDURE ThongKeLuotKhach
(
  in branch_id varchar(6),
    in chosen_year int unsigned
)
BEGIN

  DROP TABLE IF EXISTS tmp;
    DROP TABLE IF EXISTS all_months;

    CREATE TABLE all_months (
    month_num INT
  );

  INSERT INTO all_months VALUES
(1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12);

  CREATE TEMPORARY TABLE tmp AS
    SELECT booking.*
```

```
        FROM booking INNER JOIN booking_room
        ON booking.BookingID = booking_room.BookingID
        AND booking_room.BranchID = branch_id AND booking.status = 1 AND
YEAR(booking.CheckIn) = chosen_year;

    SELECT
    all_months.month_num AS Month,
    CASE
    WHEN temp.total_guest is NULL THEN 0
    ELSE temp.total_guest
    END AS total_guest
  FROM (
    SELECT DATE_FORMAT(CheckIn, '%m') AS review_month, SUM(GuestNum) AS
total_guest
    FROM tmp
    GROUP BY DATE_FORMAT(CheckIn, '%m')
   ) as temp
  RIGHT JOIN all_months ON all_months.month_num=temp.review_month
  ORDER BY all_months.month_num ASC;

    DROP TABLE IF EXISTS all_months;
    DROP TABLE IF EXISTS tmp;
END; //
DELIMITER ;
```

First, a table named "*all_month*" is created to store all the months of a year. Without this table, we cannot display the month which doesn't have guests.

Next, we need to find "*booking*" which has the status of 1 (paid) and the *CheckIn* year is the same as the input year. However, we cannot identify which branch a booking receipt belongs to using only the "*booking*" table. Therefore, a temporary table "*tmp*" is created by using an INNER JOIN between "*booking*" and "*booking_room*" which has the same *BookingID*.

Finally, the result table is created by using the RIGHT JOIN between "*all_months*" and "*temp*" we just created, group by month of *CheckIn* date. The total number of guests in 1 month will be calculated using the aggregate function *SUM(),* the month which doesn't have any guest will display as 0.

```
589 •  CALL ThongKeLuotKhach('CN1',2022);
```

Result Grid | Filter Rows: | Export:

Month	total_guest
1	17
2	8
3	7
4	15
5	4
6	7
7	19
8	0
9	16
10	9
11	0
12	9

II. Trigger

1. Trigger that update value

a. Update service packet bill

```
DELIMITER $$
CREATE TRIGGER calculate_packet_bill BEFORE INSERT ON `packet_bill`
FOR EACH ROW FOLLOWS check_packet_bill
BEGIN
  DECLARE customer_type INT ;
    DECLARE total_day INT;
```

```
  SET customer_type = (SELECT CustomerType FROM customer
            WHERE customer.CustomerID = NEW.CustomerID);

    SET total_day = (SELECT DayNum FROM service_packet
          WHERE service_packet.PackageName = NEW.PackageName);

  IF (customer_type = 3) THEN
    SET total_day = total_day + 1;
  END IF ;

  IF (customer_type = 4) THEN
    SET total_day = total_day + 2;
  END IF;

    SET NEW.RemainDay = total_day;
END$$
DELIMITER ;
```

Based on the information given on Assignment 1, only VIP (*CustomerType*=3) and SUPER VIP (*CustomerType*=4) customers can earn the promo when buying a service packet. The promo will add 1 or 2 days to the service packet based on *CustomerType* and won't discount the price of the packet.

Therefore, trigger will be created BEFORE INSERT ON "*packet_bill*" and adds 1 day to the *RemainDay* of "*packet_bill*" if *CustomerType* is 3 or 2 days if *CustomerType* is 4. The price will stay the same.

b. Update total price of Booking

```
DELIMITER $$
CREATE TRIGGER calculate_booking_price BEFORE INSERT ON `booking_room`
FOR EACH ROW
BEGIN
  DECLARE customer_type INT ; DECLARE total_price INT; DECLARE
appliedPackage INT;
    DECLARE package_name VARCHAR(45); DECLARE duration INT; DECLARE
booking_date DATETIME;
    DECLARE customer_id VARCHAR(10);
    DROP TEMPORARY TABLE IF EXISTS booking_temp;
    CREATE TEMPORARY TABLE booking_temp AS( SELECT * FROM booking WHERE
booking.BookingID = NEW.BookingID);
    SET booking_date = (SELECT BookingDate FROM booking_temp); SET
customer_id = (SELECT CustomerID FROM booking_temp);
```

```
    SET package_name = (SELECT PackageName from booking_temp);
    SET duration = (SELECT TIMESTAMPDIFF(DAY, booking_temp.CheckIn,
booking_temp.CheckOut) FROM booking_temp);
  SET customer_type = (SELECT CustomerType FROM customer INNER JOIN
booking_temp
            ON customer.CustomerID = booking_temp.CustomerID);

  SET total_price = (SELECT roomtype_branch.RentalPrice
            FROM room INNER JOIN roomtype_branch
                        ON room.RoomID = roomtype_branch.RoomID and
room.BranchID = roomtype_branch.BranchID
                        WHERE room.RoomNumber = NEW.RoomNumber and
room.BranchID =  NEW.BranchID) ; -- get room price
    SET total_price = total_price * duration;
  SET appliedPackage = 0;
     IF NOT(package_name IS NULL OR package_name = '') THEN   -- if
package is applied
    SELECT COUNT(*) INTO appliedPackage
        FROM
      (SELECT  packet_bill.PackageName AS Package_Name,
          service_packet.GuestNum AS Number_Of_Guest,
          packet_bill.StartDate AS Start_Date,
          (packet_bill.StartDate + INTERVAL 1 YEAR) AS Expire_Date,
          packet_bill.RemainDay AS Remaining_Day
      FROM packet_bill INNER JOIN service_packet
      ON  packet_bill.PackageName = service_packet.PackageName
            INNER JOIN booking_temp
      ON packet_bill.CustomerID = booking_temp.CustomerID
           AND booking_temp.BookingDate >= packet_bill.StartDate
                  AND booking_temp.PackageName = packet_bill.PackageName
                  AND booking_temp.GuestNum <=
service_packet.GuestNum         -- check if number of guest is valid
                  AND (packet_bill.RemainDay) >= TIMESTAMPDIFF(DAY,
booking_temp.CheckIn, booking_temp.CheckOut) -- check if remaining day of
a packet is valid
      HAVING Expire_Date > booking_date) AS tmp;  -- check if packet is
expired
    IF (appliedPackage <> 0) THEN                    -- if exist a valid
package
      UPDATE booking SET TotalPay = 0 WHERE booking.BookingID =
NEW.BookingID ;
      UPDATE booking SET Status = 1 WHERE booking.BookingID =
NEW.BookingID;
```

```
            UPDATE packet_bill
       SET packet_bill.RemainDay = packet_bill.RemainDay - duration --
subtract using day from packet
            WHERE CustomerID = customer_id AND PackageName =
package_name
          AND (packet_bill.StartDate + INTERVAL 1 YEAR) > booking_date ;
    ELSE
      SIGNAL SQLSTATE '12345'
        SET MESSAGE_TEXT = 'Cannot applied package';
    END IF ;
    ELSE           -- if package is not applied
       IF(customer_type = 1) THEN
        UPDATE booking SET TotalPay = total_price WHERE booking.BookingID
= NEW.BookingID ;
      END IF;
       IF(customer_type = 2) THEN
        UPDATE booking SET TotalPay = total_price * 0.9 WHERE
booking.BookingID = NEW.BookingID ;
      END IF;
      IF(customer_type = 3) THEN
        UPDATE booking SET TotalPay = total_price * 0.9 WHERE
booking.BookingID = NEW.BookingID ;
      END IF;
       IF(customer_type = 4) THEN
        UPDATE booking SET TotalPay = total_price * 0.9 WHERE
booking.BookingID = NEW.BookingID ;
      END IF;
    END IF;
   DROP TEMPORARY TABLE IF EXISTS booking_temp;
END$$
DELIMITER ;
```

The trigger "*calculate_booking_price*" is created to update the *TotalPay* of "*booking*" when customer books room and applied package and promotion for *CustomerType* (*Loyal, VIP, SUPERVIP*)

The initial price for the booking will be calculated by taking *RentalPrice* of that room type at that branch multiplied by the number of days that customer stays.

This trigger has 2 cases:

- When a packet is applied in booking: In this situation, the trigger will check if the applied package is valid or not: *RemainDay* is enough to use the packet on this booking , the number of guests allowed on this packet is valid and the packet is not expired. If the applied packet is valid, status of this booking is set to 1 (paid) and the customer doesn't have to pay money for this booking so *TotalPay* is set to 0 then *RemainDay* of packet of this customer will be subtracted based on duration between *CheckIn* and *CheckOut* of this booking. If the applied packet is not valid, an error will be announced "Cannot applied package", the current booking room will not be inserted into the "*booking_room*" table and the customer will have to book another room or create another booking without using the package.
- When packet is not applied in booking: In this situation, *CustomerType* of the customer will be checked. If *CustomerType* is 2 (Loyal) , *TotalPay* will be discounted by 10%. If *CustomerType* is 3 (VIP) , *TotalPay* will be discounted by 15%. If *CustomerType* is 4 (SUPERVIP) , *TotalPay* will be discounted by 20%. Then the *TotalPay* of this booking will be updated accordingly.

c. Update customer's point when complete booking payment or when buy service packet

In this situation, we will create 2 trigger to cover all cases:

```
DELIMITER $$
CREATE TRIGGER update_point_packet_bill AFTER INSERT ON `packet_bill`
FOR EACH ROW
BEGIN
  DECLARE new_point INT;
    SET new_point = floor(NEW.TotalPay/1000); -- calculate point

    UPDATE customer
    SET Point = Point + new_point -- update customer's point
    WHERE CustomerID = NEW.CustomerID;
END$$
DELIMITER ;
```

In the first situation, the customer's point will be updated when buying a service packet.

The trigger "*update_point_packet_bill*" is triggered after a *packet_bill* is inserted, it will calculate points based on *TotalPay* of that *packet_bill* and update customer's *Point* accordingly.

```
DELIMITER $$
CREATE TRIGGER update_point_booking AFTER UPDATE ON `booking`
FOR EACH ROW
BEGIN
  DECLARE new_point INT;
  IF (NEW.Status <> OLD.Status AND NEW.Status = 1) THEN
    SET new_point = floor(NEW.TotalPay/1000);
    UPDATE customer
    SET Point = Point + new_point
    WHERE CustomerID = NEW.CustomerID;
  END IF;
END$$
DELIMITER ;
```

In the second situation, customer's point will be updated when customer pay for a booking after the booking room is success (the booking's status is changed to paid)

The trigger "*update_point_booking*" is triggered after a booking updates its status to paid, it will check if the status is updated or not and if it is updated to paid (1), then it will calculate point and add to customer's *Point*.

d. Update Customer's type based on Customer's point

```
DELIMITER $$
CREATE TRIGGER update_customer_type BEFORE UPDATE ON `customer`
FOR EACH ROW
BEGIN
  IF (NEW.Point <> OLD.Point) THEN
    IF (NEW.Point < 50 ) THEN
      SET NEW.CustomerType = 1;
    ELSEIF (NEW.Point < 100) THEN
      SET NEW.CustomerType = 2;
    ELSEIF (NEW.Point < 1000) THEN
      SET NEW.CustomerType = 3;
    ELSE
      SET NEW.CustomerType = 4;
    END IF;
    END IF;
```

```
END$$
DELIMITER ;
```

In this situation, we do not need to check customer's *CustomerType* based on customer's *Point* when a customer is inserted because the *Point* is always 0, so we only need to check when customer's Point is updated.

The trigger "*update_customer_type*" will be triggered if customer's *Point* is updated and will update *CustomerType* if condition is satisfied.

2. Check constraint: Customer can not buy the same service packet if not expired

```
DELIMITER $$
CREATE TRIGGER check_packet_bill BEFORE INSERT ON `packet_bill`
FOR EACH ROW
BEGIN
  DECLARE current_package INT;
  SELECT COUNT(*) INTO current_package
    FROM
    ( SELECT * FROM packet_bill
    WHERE NEW.CustomerID = packet_bill.CustomerID
        AND NEW.PackageName = packet_bill.PackageName
        AND NEW.StartDate < (packet_bill.StartDate + INTERVAL 1 YEAR)
    ) AS tmp;
    IF (current_package <> 0) THEN
    SIGNAL SQLSTATE '12345'
      SET MESSAGE_TEXT = 'Cannot buy the same package when package is not
expire';
    END IF;
END$$
DELIMITER ;
```

The trigger "*check_packet_bill*" will be triggered before a *packet_bill* is inserted. It will check if there is any current package with the same *PackageName*, belongs to the same Customer and is not expired (its *ExpiredDate* is larger than *StartDate* of the newly added packet). If there is a current package still not expired, the customer can not insert the *packet_bill* and an error will be shown "Cannot buy the same package when package is not expired".

PART 3: BUILD WEB APPLICATION

Github link: https://github.com/namkha1032/databaseAssignment.git

Demo video link:

https://drive.google.com/file/d/12OZ9qt0niS5NGghTFi79Q9_uW_xxaunN/view?usp=sharing

In this part, we decide to use HTML/CSS for Front-end and Django for Back-end to design a Web application

We connect our application database with MySQL database in settings.py as below

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'qlks',
        'USER': 'root',
        'HOST': 'localhost',
        'PORT': 3306,
        'PASSWORD': '',
    }
}
```

Also, we create models.py as our application's model to interact with our MySQL database, you can see more about this models.py in our code

```python
from django.db import models
from django.contrib.auth.models import AbstractUser

# Create your models here.

class User(AbstractUser):
    username = models.CharField(max_length=200, null=True, unique = True)
    email = models.EmailField(null =True)

    USERNAME_FIELD = 'username'
    REQUIRED_FIELDS = []



class Branch(models.Model):
    branchid = models.CharField(db_column='BranchID', primary_key=True, max_length=6)  # Field name made lowercase.
    province = models.CharField(db_column='Province', max_length=50)  # Field name made lowercase.
    address = models.CharField(db_column='Address', unique=True, max_length=100)  # Field name made lowercase.
    phonenum = models.CharField(db_column='PhoneNum', unique=True, max_length=12)  # Field name made lowercase.
    email = models.CharField(db_column='Email', unique=True, max_length=64)  # Field name made lowercase.

    class Meta:
        managed = False
        db_table = 'branch'

class Customer(models.Model):
    customerid = models.CharField(db_column='CustomerID', primary_key=True, max_length=10)  # Field name made lowercase.
    citizenid = models.CharField(db_column='CitizenID', unique=True, max_length=12)  # Field name made lowercase.
    fullname = models.CharField(db_column='FullName', max_length=45)  # Field name made lowercase.
    phone = models.CharField(db_column='Phone', unique=True, max_length=12)  # Field name made lowercase.
    email = models.CharField(db_column='Email', unique=True, max_length=45, blank=True, null=True)  # Field name made lowercase.
    username = models.CharField(db_column='Username', unique=True, max_length=45, blank=True, null=True)  # Field name made lowercase.
    password = models.CharField(db_column='Password', max_length=45, blank=True, null=True)  # Field name made lowercase.
    point = models.PositiveIntegerField(db_column='Point', blank=True, null=True)  # Field name made lowercase.
    customertype = models.PositiveIntegerField(db_column='CustomerType', blank=True, null=True)  # Field name made lowercase.

    class Meta:
        managed = False
        db_table = 'customer'


class Booking(models.Model):
    bookingid = models.CharField(db_column='BookingID', primary_key=True, max_length=20)  # Field name made lowercase.
    bookingdate = models.DateTimeField(db_column='BookingDate')  # Field name made lowercase.
    guestnum = models.PositiveIntegerField(db_column='GuestNum')  # Field name made lowercase.
    checkin = models.DateTimeField(db_column='CheckIn')  # Field name made lowercase.
    checkout = models.DateTimeField(db_column='CheckOut')  # Field name made lowercase.
    status = models.PositiveIntegerField(db_column='Status')  # Field name made lowercase.
    totalpay = models.PositiveIntegerField(db_column='TotalPay')  # Field name made lowercase.
    customerid = models.ForeignKey('Customer', models.DO_NOTHING, db_column='CustomerID')  # Field name made lowercase.
    packagename = models.ForeignKey('ServicePacket', models.DO_NOTHING, db_column='PackageName', blank=True, null=True)  # Field name made lowercase.

    class Meta:
```

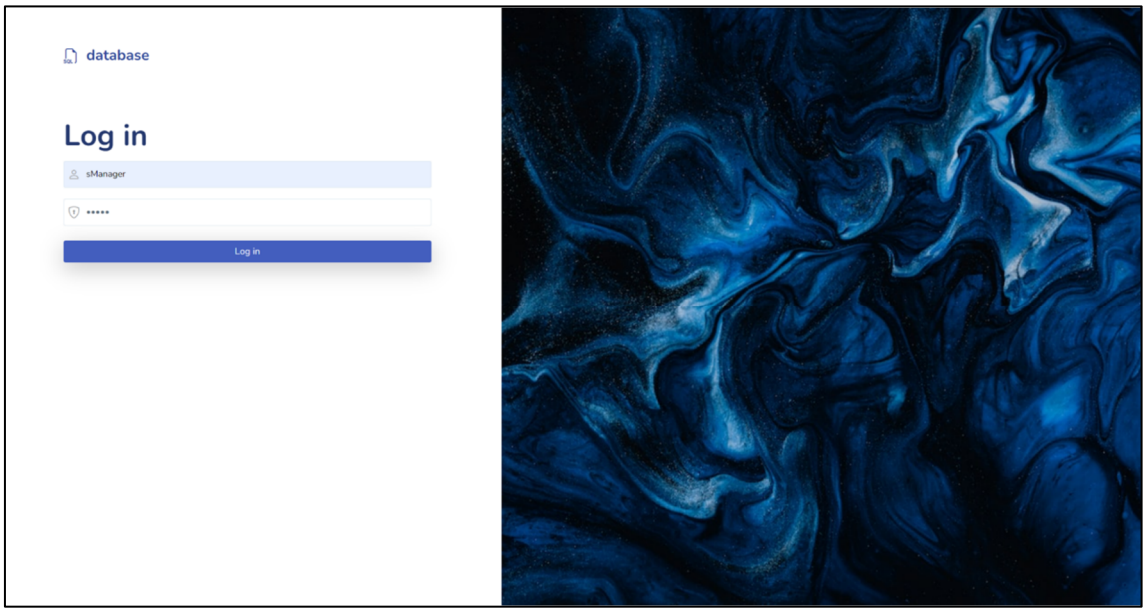
1. Create user

By typing into terminal *: python manage.py createsuperuser*

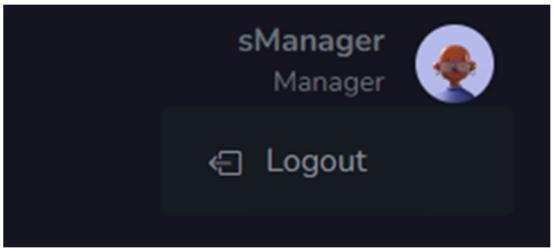
Then input the username and password as below, I have created a user with username sManager and this user has all access to the database.

```
PS C:\Users\Admin\Desktop\DB PROJ\final\project> python manage.py createsuperuser
Username: sManager
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

The picture below shows that the user sManager is created successfully in our database.

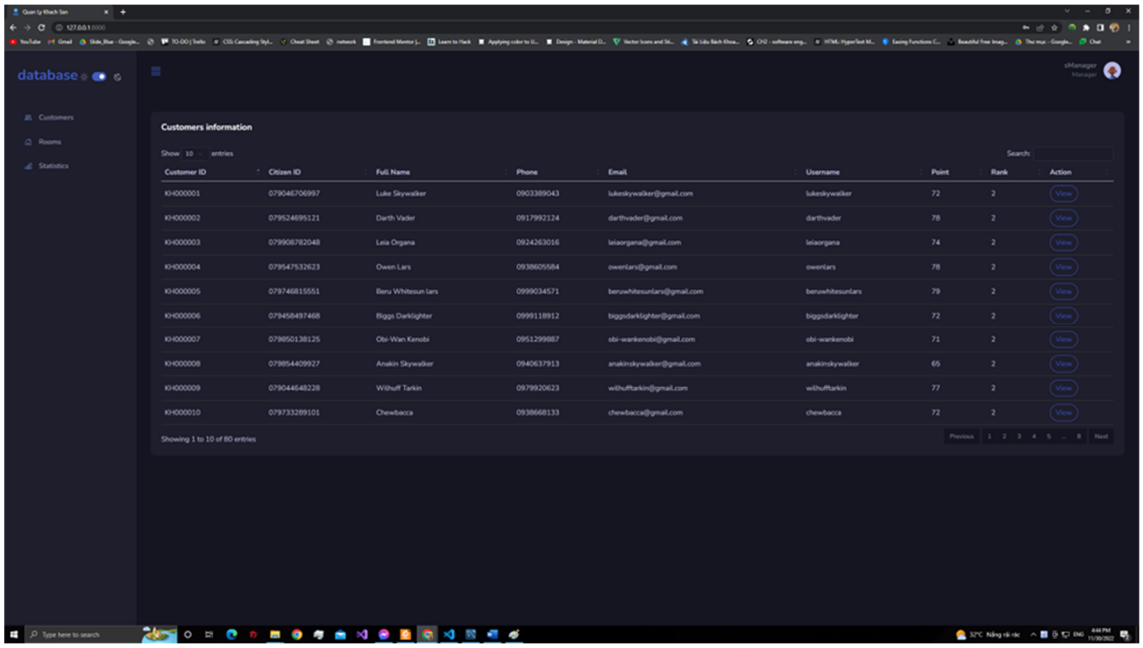


We login into the website using the sManager account that we have created above. When login successfully, we will be redirected to the homepage which shows the information of all customers. On the top-right, we have the avatar of the user, click on the avatar to see the Logout option.



2. Main functionality

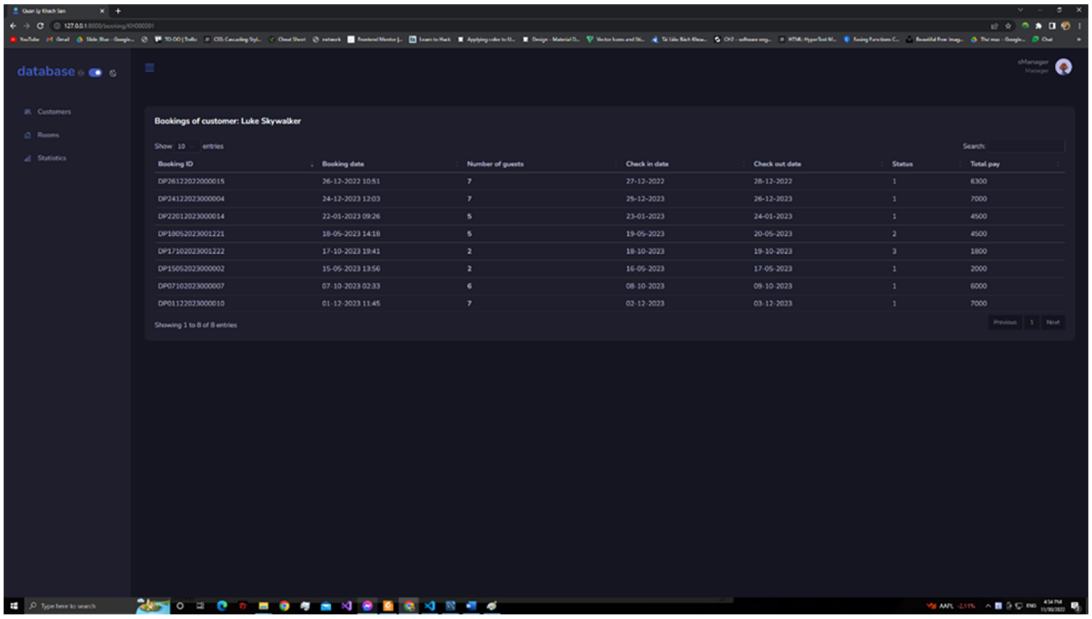
a. View Customer's information



When login successfully, we will be redirected to the homepage which shows the information of all customers. On this page, we can select the number of entries we want to appear.

b. Search Customer and View Booking's information

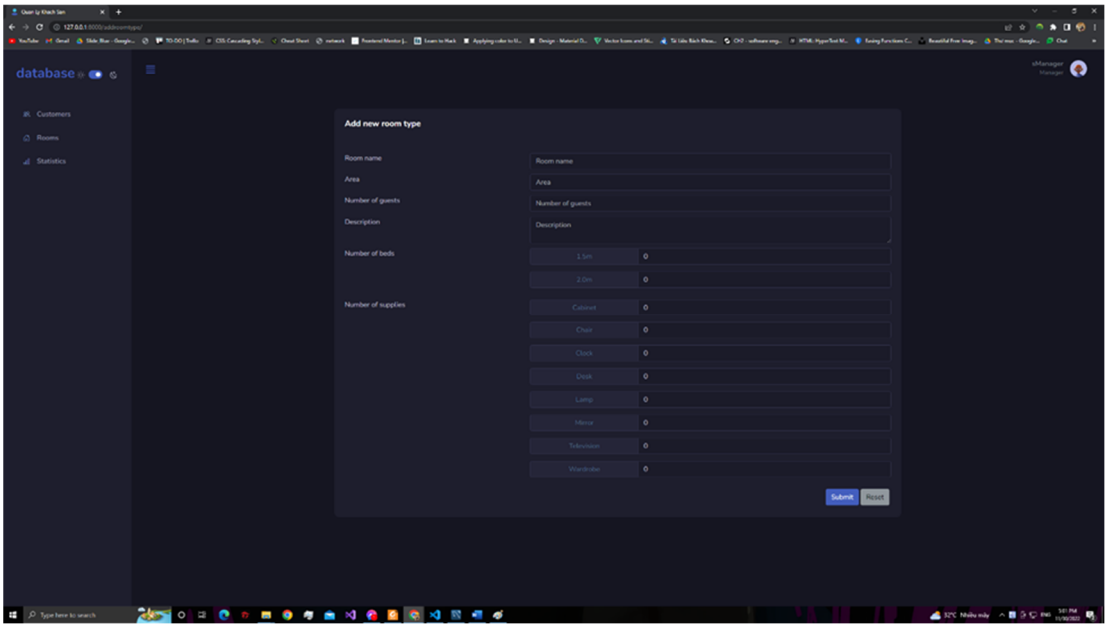
We can type into the search box to search for the Customer we are looking for, and in the column "Action", there is a button "View", click on this button to view customer's booking information.



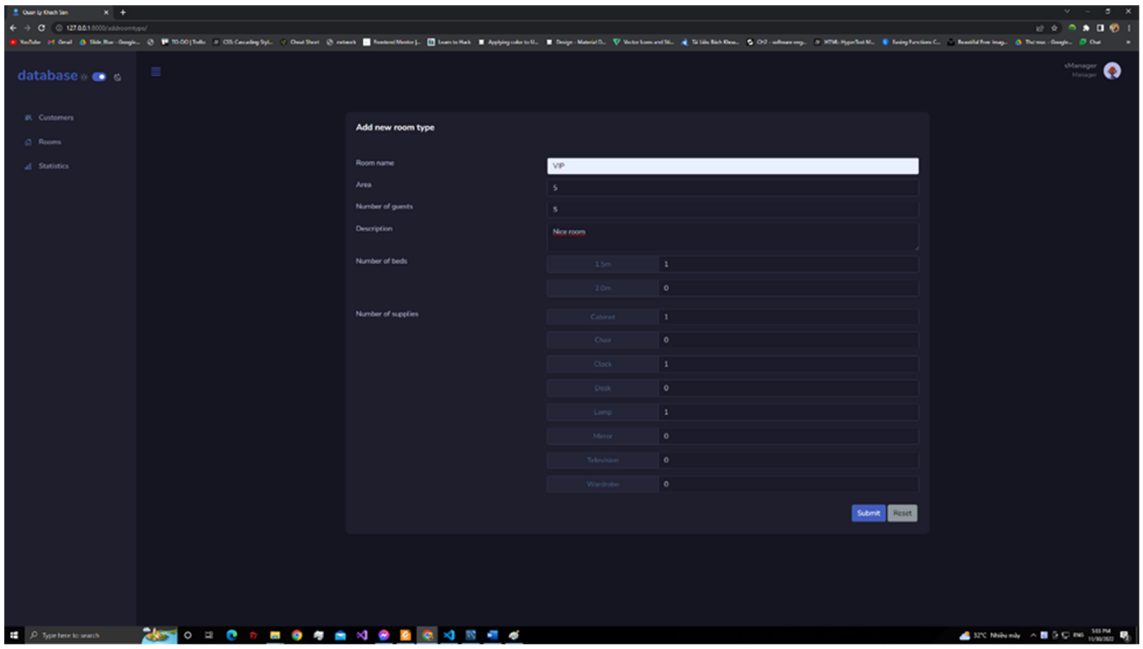
The picture above is the booking information of customer Luke Skywalker, it includes all the necessary information of Customer's booking. And only the booking with the CheckIn date in the future will be shown.

c. Add a new room type

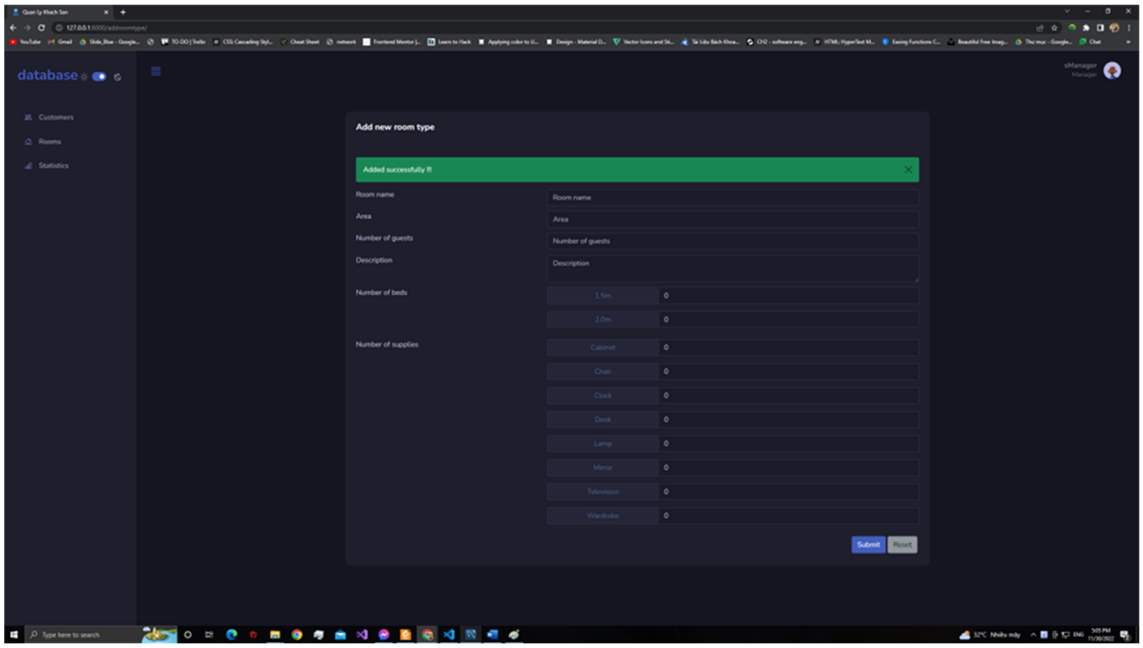
We can click on the "Rooms" button on the left side bar to redirect to the "Add room type" page.



This is the “Add new room type” form that we can use to create a new room type by adding necessary information.



When submitted, it will return a message “Added successfully !!!” and we can see in our database that the new room type has been added.



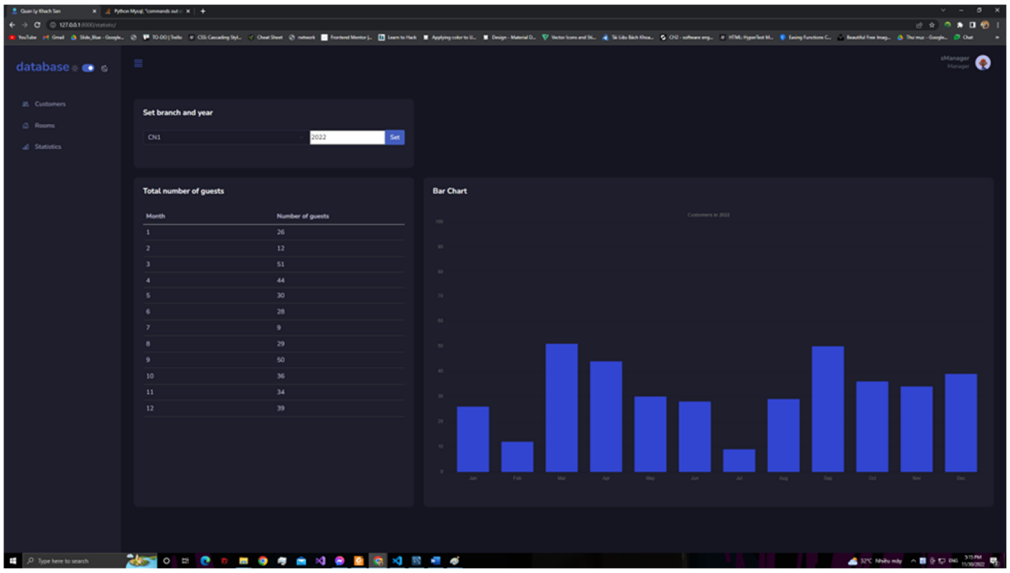
	RoomID	RoomName	Area	NumGuest	Description
▶	1	Single	30	1	room for 1 guest
	2	Double	40	2	room for 2 guests
	3	Triple	50	3	room for 3 guests
	4	Quad	60	4	room for 4 guests
	5	VIP	5	5	Nice room
*	NULL	NULL	NULL	NULL	NULL

d. Customer's Statistics

```
@login_required(login_url='login')
def statistic(request):
    branches = Branch.objects.all()
    branch_id = 'CN1'
    year = 2022
    data_list = "["
    if request.method == 'POST':
        branch_id = request.POST.get('branch_id')
        year = request.POST.get('year')

    cursor = connection.cursor()
    cursor.callproc("ThongKeLuotKhach", [branch_id, year])#call procedure
    results = cursor.fetchall()

    for result in results:
        data_list += str(result[1]) + ','

    data_list = data_list[:-1];
    data_list += ']';
    cursor.close()

    context = {'branches': branches, 'results': results, 'year': year, 'branch_id': branch_id, 'data_list': data_list}
    return render(request, 'db-statistic.html', context)
```

To implement this functionality, we have to call the procedure ThongKeLuotKhach from our database, the highlighted code above is used to call the procedure without having to write any SQL code.

To go to the Statistics page, we click on "Statistic" button on the left side bar



Here, we can see the statistics illustrated in the bar chart and in the table as in the picture above.

In the “Set branch and year” section, we can choose which year or which branch we want to see the statistics and click “Set”, this will show us the following statistics of the branch and year we had chosen.

This is the statistics of branch CN3 in 2022