Lab Manual for Triggers and Check Constraints (September 2024)

Allan O. Omondi
School of Computing and Engineering Sciences,
Strathmore University, Nairobi, Kenya
aomondi@strathmore.edu

Learning Outcomes

By the end of this lab, you will be able to:

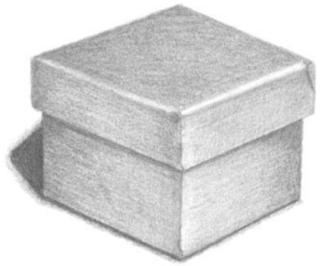
1. **Create** a trigger based on the time when the trigger is set to be executed (BEFORE or AFTER) and the manipulation action that automatically fires the trigger (INSERT, UPDATE, or DELETE)
2. **List** all the triggers in a database
3. **Show** the DDL statements used to create a database object
4. **Record** a timestamp with a precision of a hundredth of a second
5. **Change** the default delimiter from a semicolon (;) to any other symbol
6. **Implement** an IF statement inside a trigger
7. **Apply** an SQL SIGNAL to specify a user-defined error
8. **Implement** static integrity constraints
9. **Implement** dynamic integrity constraints

Original GitHub Repository

https://github.com/course-files/BBT3104-Lab2of6-TriggersandCheckConstraints

Software

1. VS Code: link
2. Git and Git Bash: link
3. GitHub Desktop: link
4. WSL (for Windows OS): link
5. Docker: link
6. DBeaver database tool: link
7. VS Code extensions: Docker, WSL (for Windows OS), YAML, Code Spell Checker



Approximate Time Required

3 hours

Prerequisites

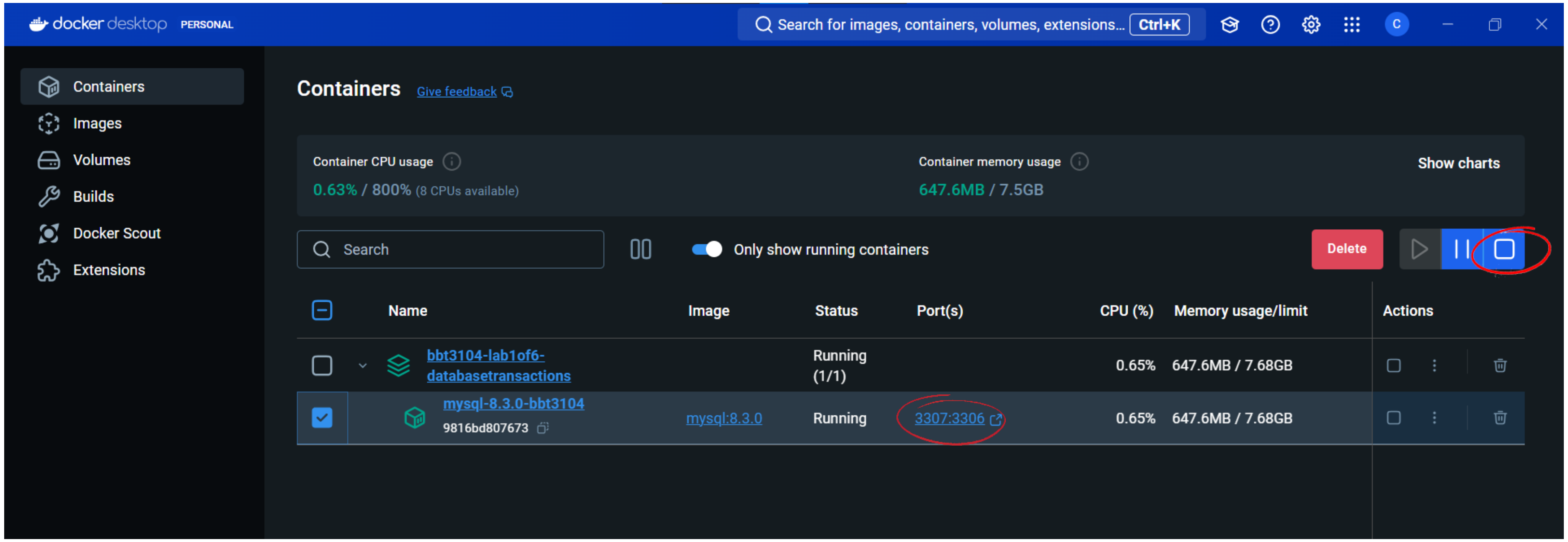
1. You need to have completed Lab 1 on Database Transactions

Recommended citation:
Omondi, A. O. (2024). *BBT3104 – Advanced Database Systems – Lab Manual for Triggers and Check Constraints [PDF Lab Manual]*. Retrieved from the Strathmore University eLearning website https://elearning.strathmore.edu/

Practical Steps

STEP 1. Stop the MySQL Docker container used in the previous lab

Stop the `mysql-8.3.0-bbt3104` container we used in Lab 1. This is to avoid conflicts when creating another container in lab 2 that is using the same port in the host, i.e., port 3307. Remember that we can specify a port number in the host such that any data transmitted through this port is relayed to the container. For example, if data is sent to the host through port 3307, this data is relayed to the container through the container's port 3306.



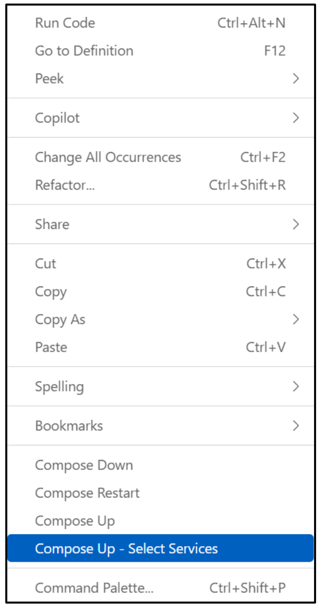
Container CPU usage: 0.63% / 800% (8 CPUs available)

Container memory usage: 647.6MB / 7.5GB

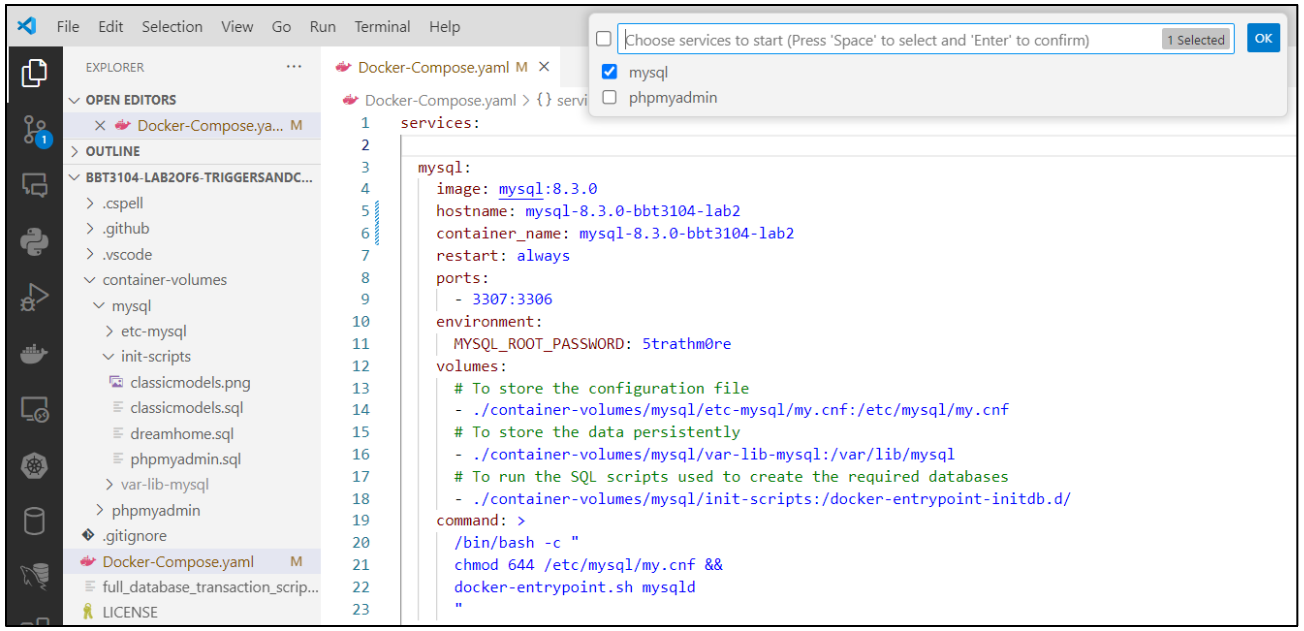
	Name	Image	Status	Port(s)	CPU (%)	Memory usage/limit	Actions
☐	bbt3104-lab1of6-databasetransactions		Running (1/1)		0.65%	647.6MB / 7.68GB	
☑	mysql-8.3.0-bbt3104 9816bd807673	mysql:8.3.0	Running	3307:3306	0.65%	647.6MB / 7.68GB	

STEP 2. Start the MySQL Docker container for Lab 2

The `Docker-Compose.yaml` code is available here: https://github.com/course-files/BBT3104-Lab2of6-TriggersandCheckConstraints/blob/main/Docker-Compose.yaml

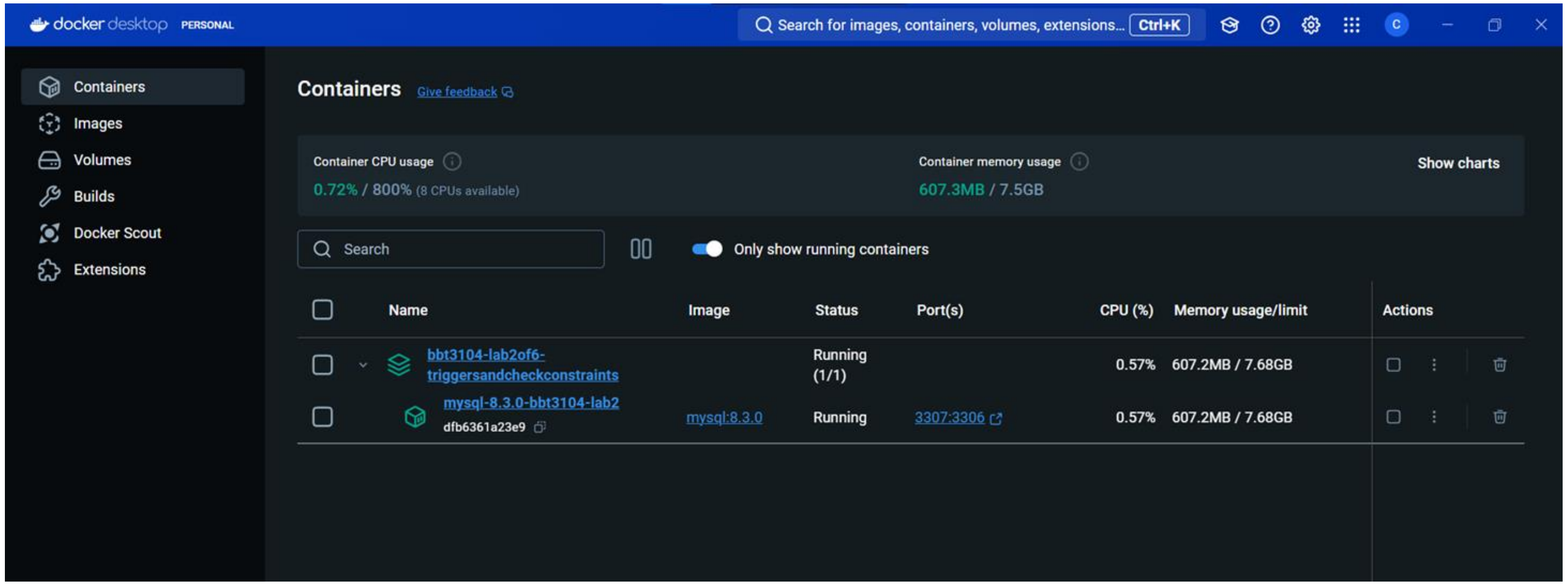


Right-Click anywhere inside the `` `Docker-Compose.yaml` `` file in VS Code and select "**Compose Up – Selected Services**"



```yaml
services:

  mysql:
    image: mysql:8.3.0
    hostname: mysql-8.3.0-bbt3104-lab2
    container_name: mysql-8.3.0-bbt3104-lab2
    restart: always
    ports:
      - 3307:3306
    environment:
      MYSQL_ROOT_PASSWORD: 5trathm0re
    volumes:
      # To store the configuration file
      - ./container-volumes/mysql/etc-mysql/my.cnf:/etc/mysql/my.cnf
      # To store the data persistently
      - ./container-volumes/mysql/var-lib-mysql:/var/lib/mysql
      # To run the SQL scripts used to create the required databases
      - ./container-volumes/mysql/init-scripts:/docker-entrypoint-initdb.d/
    command: >
      /bin/bash -c "
      chmod 644 /etc/mysql/my.cnf &&
      docker-entrypoint.sh mysqld
      "
```

Ensure that Docker is running on your computer then select "mysql" and click "OK" to create the MySQL Docker container.



Container CPU usage 0.72% / 800% (8 CPUs available)

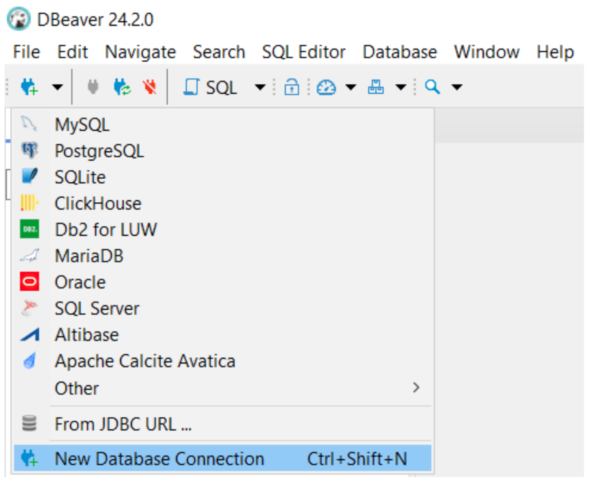
Container memory usage 607.3MB / 7.5GB

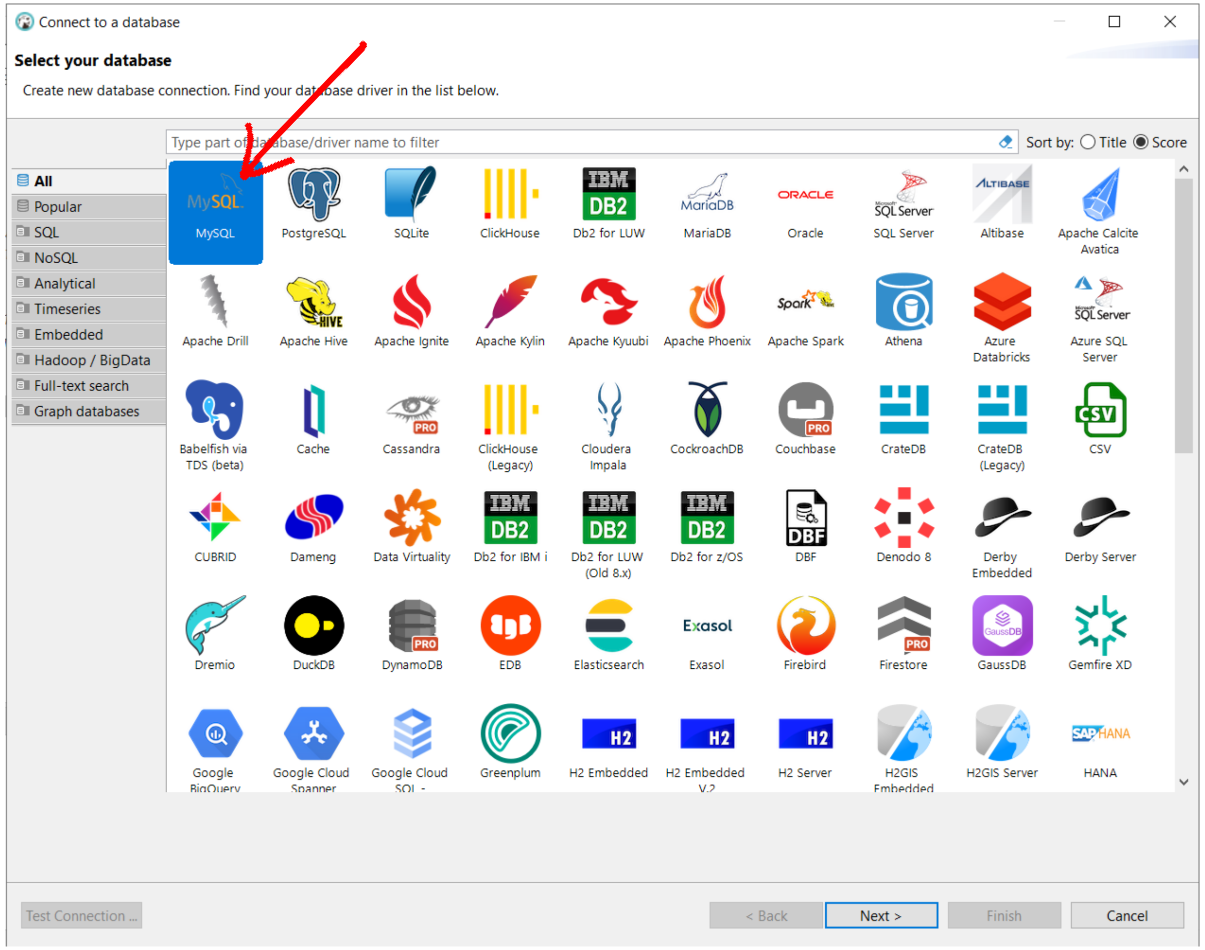
Name	Image	Status	Port(s)	CPU (%)	Memory usage/limit	Actions
bbt3104-lab2of6-triggersandcheckconstraints		Running (1/1)		0.57%	607.2MB / 7.68GB	
mysql-8.3.0-bbt3104-lab2 dfb6361a23e9	mysql:8.3.0	Running	3307:3306	0.57%	607.2MB / 7.68GB	

The assigned name of the MySQL Docker container is “`mysql-8.3.0-bbt3104-lab2`” and it should appear as “**Running**” in the list of containers in Docker Desktop as shown here. Note that it is listed under the cluster called “`bbt3104-lab2of6-triggersandcheckconstraints`”. The name of the cluster is assigned based on the folder name that is storing the `Docker-Compose.yaml` file.

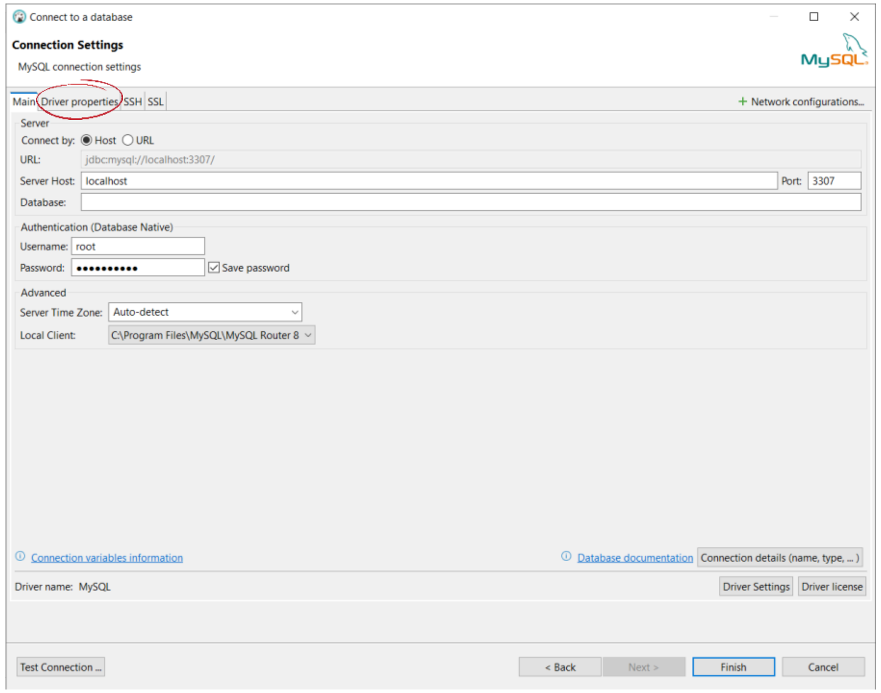
STEP 3. Create a connection to the MySQL container using DBeaver database tool

Alternatively, you can use the existing connection you created in Lab 1.





Open DBeaver and create a new connection. Select the MySQL standard driver and click "Next".



Connect to a database

Connection Settings

MySQL connection settings

Main | Driver properties | SSH | SSL — + Network configurations...

Server

Connect by: (●) Host ( ) URL

URL: jdbc:mysql://localhost:3307/

Server Host: localhost Port: 3307

Database:

Authentication (Database Native)

Username: root

Password: •••••••••• [✓] Save password

Advanced

Server Time Zone: Auto-detect

Local Client: C:\Program Files\MySQL\MySQL Router 8

Connection variables information — Database documentation — Connection details (name, type, ...)

Driver name: MySQL — Driver Settings — Driver license

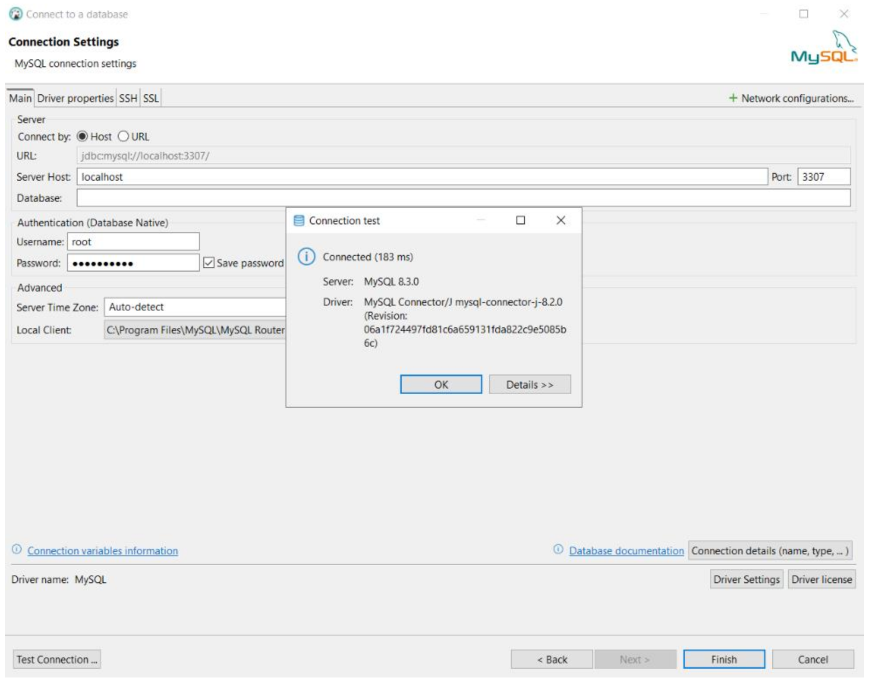
Test Connection ... — < Back — Next > — Finish — Cancel

Specify the server's IP address (localhost), port number (3307), username (root or student), and password (5trathm0re).

Set the value of "**`allowPublicKeyRetrieval`**" property under the "Driver Properties" tab to TRUE. Create the property if it does not exist.

Remember to open the ports in the firewall if you are using Linux, i.e., execute the following in the VS Code terminal:

```
sudo ufw allow 3307/tcp
```



Connection test

Connected (183 ms)

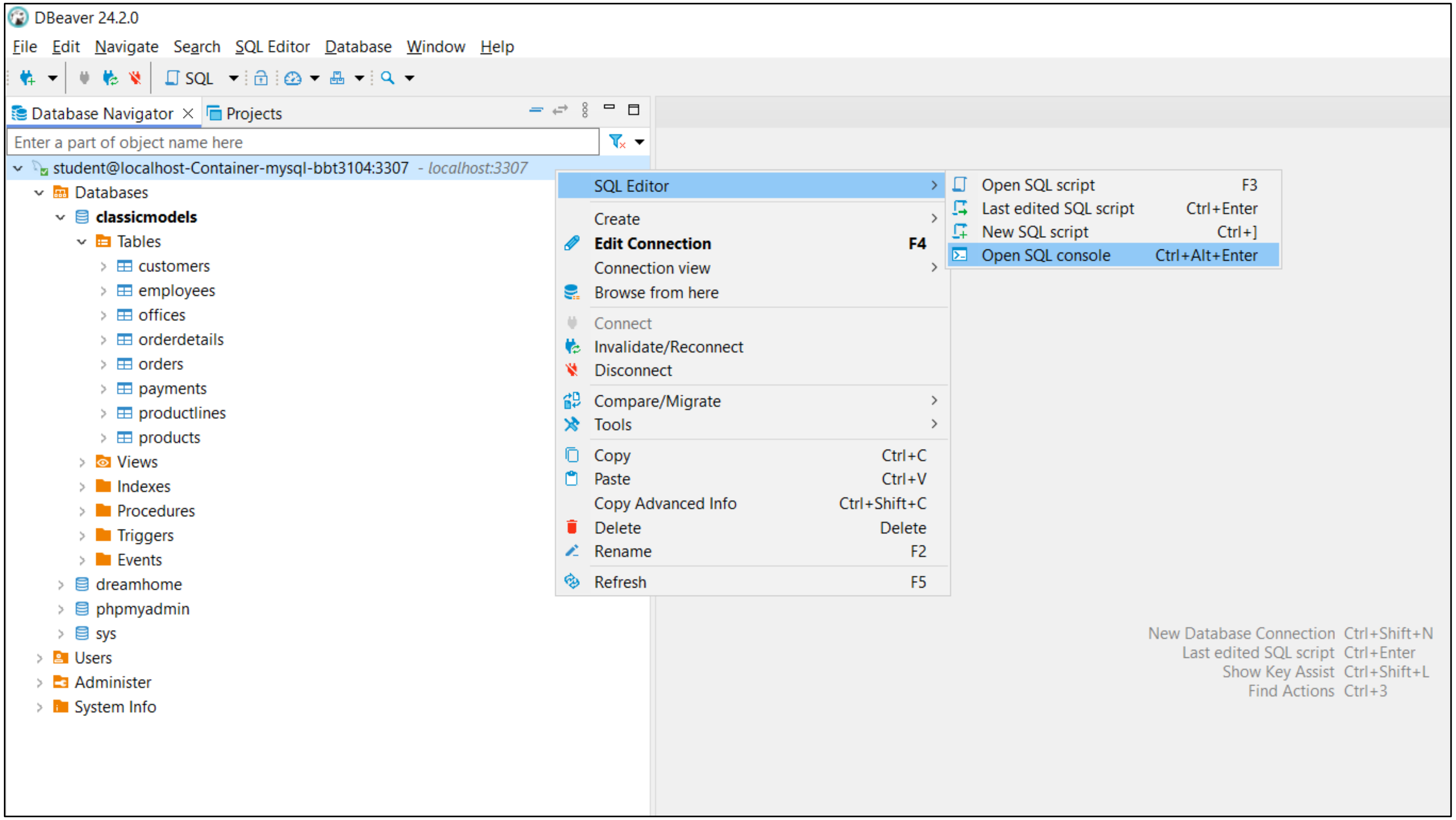
Server: MySQL 8.3.0

Driver: MySQL Connector/J mysql-connector-j-8.2.0 (Revision: 06a1f724497fd81c6a659131fda822c9e5085b6c)

OK — Details >>

Test the connection settings and click "Finish" if the test is successful. You do not need to change any other settings.

Open an SQL console using the connection:



Narrative

The "classicmodels" sample database contains data of a retail business. The retail business, in this case, is engaged in the sale of **models of** classic cars, motorcycles, planes, ships, trains, trucks and buses, and vintage cars. It contains typical business data such as customers, products, product lines (categories of products), orders, order line items (details of an order), payments received, employees, and branch details. The database schema is presented here:
https://github.com/course-files/BBT3104-Lab2of6-TriggersandCheckConstraints/blob/main/container-volumes/mysql/init-scripts/classicmodels.png

Models of classic Ford vehicles

A trigger is a named database object which contains one or more SQL statements which are executed automatically in response to DML actions (i.e., actions that manipulate data using the **D**ata **M**anipulation **L**anguage section of SQL: **C**reate, **U**pdate, **D**elete).

As opposed to over-relying on the programming team, triggers provide DBAs with a way to:

(i) Validate the data before manipulation
(ii) Handle errors at the database tier
(iii) Audit the data manipulation that has occurred in the database (like an alarm that can be set off (triggered) by the actions of a thief)

A trigger can be executed **BEFORE** each row is manipulated or **AFTER** each row has been manipulated. In this case, the manipulation can be either an **INSERT, UPDATE, or DELETE**. We can, therefore, set a trigger to be executed either:

(i) **BEFORE INSERT:** The trigger is executed before a new tuple is inserted into the relation
(ii) **AFTER INSERT:** The trigger is executed after a new tuple has been inserted into the relation
(iii) **BEFORE UPDATE:** The trigger is executed before an existing tuple in the relation is updated

(iv) **AFTER UPDATE:** The trigger is executed after an existing tuple in the relation has been updated
(v) **BEFORE DELETE:** The trigger is executed before an existing tuple in the relation is deleted
(vi) **AFTER DELETE:** The trigger is executed after an existing tuple in the relation has been deleted

Within the trigger body, you can refer to columns (attributes) in the subject table (the table associated with the trigger) by using the aliases **OLD** and **NEW**, that is:

(i) **OLD.column_name** refers to a column of an existing tuple before it is updated or deleted.
(ii) **NEW.column_name** refers to the column of a new tuple to be inserted or it can refer to the column of an existing tuple after it is updated.

The following table illustrates when the OLD and NEW options can be used depending on the data manipulation action.

Data Manipulation Action	**OLD**	**NEW**
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

Suppose that the Human Resource manager has approached the IT department with a request to customize the HR system that they currently own. The customization requires that:

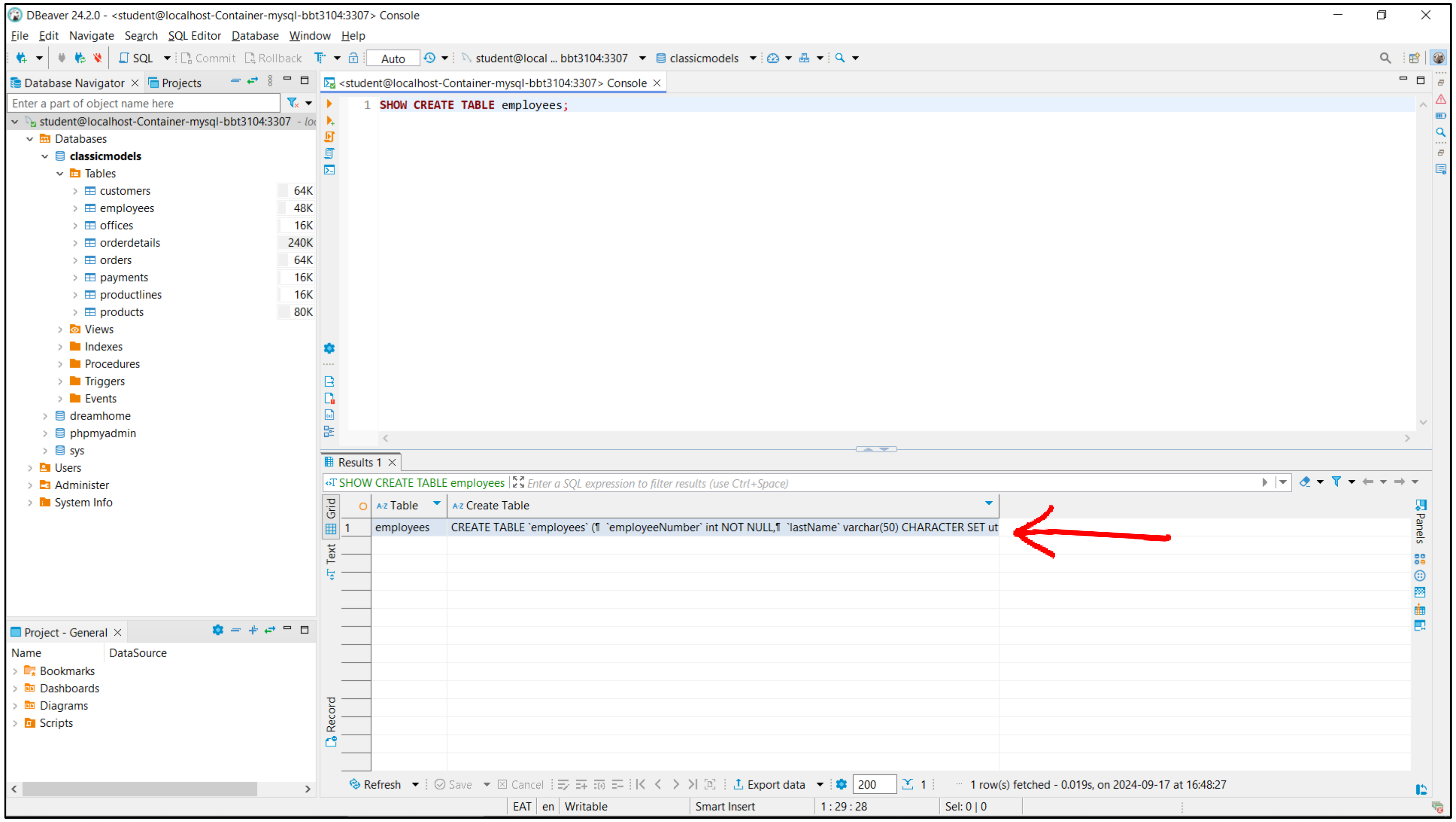
(i) **The system should** have an "Undo" functionality to revert changes that have been made to employee data when necessary

The IT team that has been assigned this project notes that it is necessary to store the old data **before an update** changes the old data. As opposed to implementing the whole project as a programming project, you realize that you can create a BEFORE UPDATE trigger that is executed before any employee record is updated. When the trigger is executed, it stores the old data in a different relation before the update is done. The "Undo" functionality can then refer to the old data stored in a different relation to perform its function as required.

STEP 4. Create a relation that will store the employees' historical data

Execute the following command to view the DDL statement that was used to create the "employees" table:

```
SHOW CREATE TABLE employees;
```


DBeaver 24.2.0 - <student@localhost-Container-mysql-bbt3104:3307> Console
File Edit Navigate Search SQL Editor Database Window Help
SQL Commit Rollback Auto student@local ... bbt3104:3307 classicmodels
Database Navigator Projects
Enter a part of object name here
student@localhost-Container-mysql-bbt3104:3307
Databases
classicmodels
Tables
customers 64K
employees 48K
offices 16K
orderdetails 240K
orders 64K
payments 16K
productlines 16K
products 80K
Views
Indexes
Procedures
Triggers
Events
dreamhome
phpmyadmin
sys
Users
Administer
System Info
Project - General
Name DataSource
Bookmarks
Dashboards
Diagrams
Scripts
<student@localhost-Container-mysql-bbt3104:3307> Console
Results 1
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
Record
Panels
Refresh Save Cancel Export data 200 1
EAT en Writable Smart Insert 1 : 29 : 28 Sel: 0 | 0

```
SHOW CREATE TABLE employees;
```

SHOW CREATE TABLE employees

	Table	Create Table
1	employees	CREATE TABLE `employees` (¶ `employeeNumber` int NOT NULL,¶ `lastName` varchar(50) CHARACTER SET ut

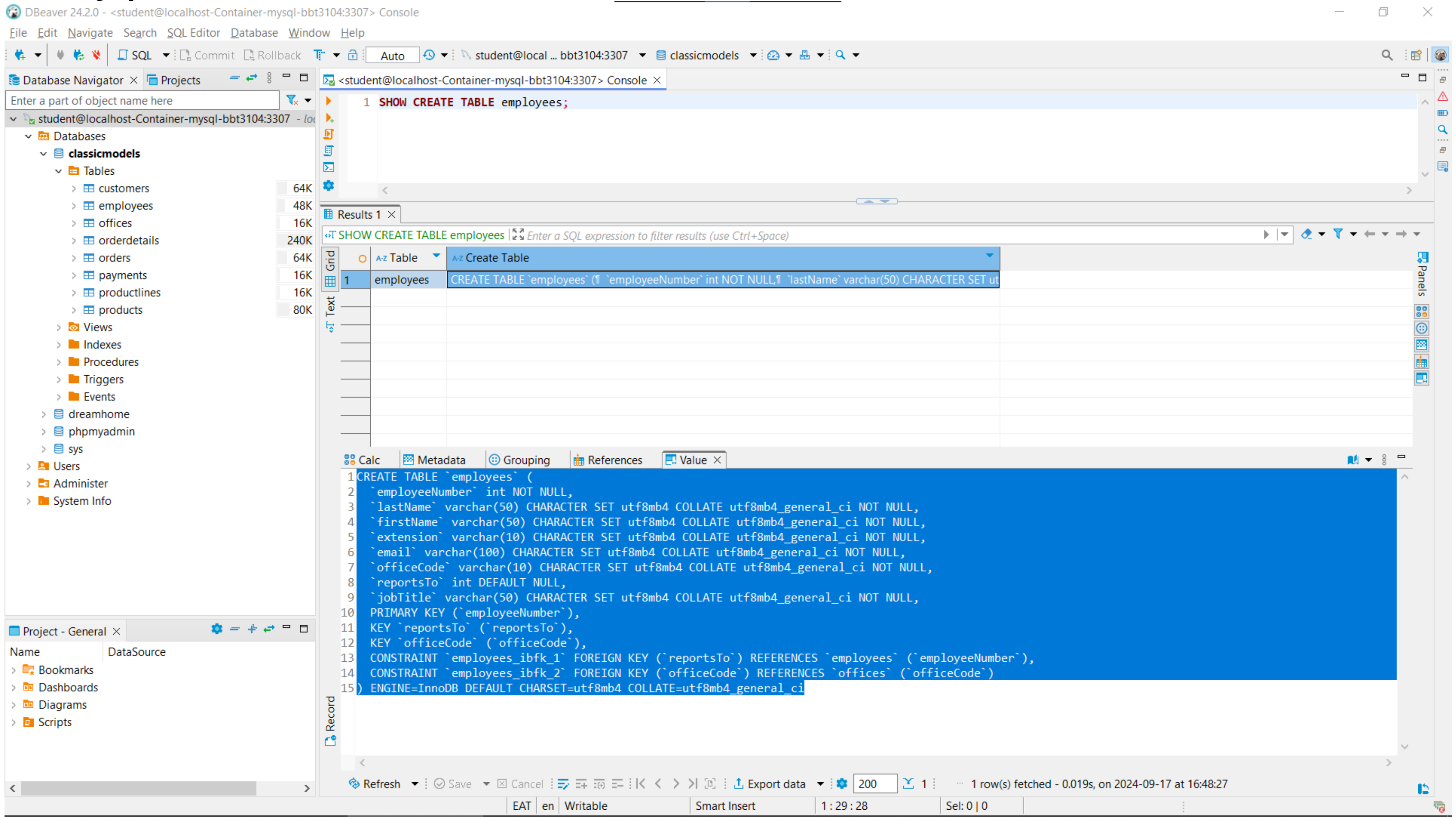
1 row(s) fetched - 0.019s, on 2024-09-17 at 16:48:27

As shown in the previous screenshot, it is difficult to view long text without the “Word Wrap” option. The alternative is:

(i) Select the cell in the result set that has the long text

(ii) Press **F7** on the keyboard

This will display on the text in the cell as shown below:



```sql
SHOW CREATE TABLE employees;
```

Table	Create Table
employees	CREATE TABLE `employees` (¶ `employeeNumber` int NOT NULL,¶ `lastName` varchar(50) CHARACTER SET ut

```sql
CREATE TABLE `employees` (
  `employeeNumber` int NOT NULL,
  `lastName` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `firstName` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `extension` varchar(10) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `email` varchar(100) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `officeCode` varchar(10) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `reportsTo` int DEFAULT NULL,
  `jobTitle` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  PRIMARY KEY (`employeeNumber`),
  KEY `reportsTo` (`reportsTo`),
  KEY `officeCode` (`officeCode`),
  CONSTRAINT `employees_ibfk_1` FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`),
  CONSTRAINT `employees_ibfk_2` FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci
```

1 row(s) fetched - 0.019s, on 2024-09-17 at 16:48:27

Edit the DDL statement used to create the “employees” relation so that the new DDL statement has:

(i) The name of the new relation as “**employees_undo**”
(ii) A new attribute “**date_of_change**” to record when the data was manipulated. This should also be set as the relation’s primary key
(iii) A new attribute “**change_type**” to record the type of data manipulation that has been done, i.e., an insertion, an update, or a deletion.
(iv) All attributes except “**date_of_change**” and “**change_type**” set to allow NULL values

The subsequent DDL statement to create the new relation is as follows:

```
CREATE TABLE `employees_undo` (
  `date_of_change` timestamp(2) NOT NULL DEFAULT CURRENT_TIMESTAMP(2) COMMENT 'Records the date and time when the data was
manipulated. This will help to keep track of the changes made. The assumption is that no 2 users will change the exact same
record at the same time (with a precision of a hundredth of a second, e.g., 4.26 seconds).',
  `employeeNumber` int NOT NULL,
  `lastName` varchar(50) DEFAULT NULL,
  `firstName` varchar(50) DEFAULT NULL,
  `extension` varchar(10) DEFAULT NULL,
  `email` varchar(100) DEFAULT NULL,
  `officeCode` varchar(10) DEFAULT NULL,
  `reportsTo` int DEFAULT NULL,
  `jobTitle` varchar(50) DEFAULT NULL,
  `change_type` varchar(50) NOT NULL COMMENT 'Records the type of data manipulation that was done, for example an insertion,
an update, or a deletion.',
  PRIMARY KEY (`date_of_change`),
  UNIQUE KEY `date_of_change_UNIQUE` (`date_of_change`)
) ENGINE = InnoDB
```

STEP 5. Create a trigger that is executed before updating an employee's data

Execute the following command to create the trigger:

```
CREATE

    TRIGGER TRG_BEFORE_UPDATE_ON_employees
    BEFORE
UPDATE
    ON
    employees FOR EACH ROW

    INSERT
    INTO
    `employees_undo` SET
      `date_of_change` = CURRENT_TIMESTAMP(2),
      `employeeNumber` = OLD.`employeeNumber` ,
      `lastName` = OLD.`lastName` ,
      `firstName` = OLD.`firstName` ,
      `extension` = OLD.`extension` ,
      `email` = OLD.`email` ,
      `officeCode` = OLD.`officeCode` ,
      `reportsTo` = OLD.`reportsTo` ,
      `jobTitle` = OLD.`jobTitle` ,
      `change_type` = 'An update DML operation was executed';
```

- `TRIGGER TRG_BEFORE_UPDATE_ON_employees` – Used to specify the name of the trigger. Notice the use of the prefix "TRG" to identify the object as a trigger.

  Although optional, other prefixes and suffixes that can be used include:

  i.) "PK_" represents a primary key
  ii.) "FK_" represents a foreign key
  iii.) "CHK_" represents a check constraint
  iv.) "IDX_" represents an index
  v.) "_UNIQUE" represents a unique constraint (this can be included as a suffix if the unique constraint is implemented as an index, e.g., IDX_chosen_name_UNIQUE)
  vi.) "FUNC_" represents a function

vii.) “PROC_” represents a procedure
viii.) “EVN_” represents a scheduled event, and so on.

A DBA can also set the name in such a way that it allows one to know the time when the trigger is set to be executed (BEFORE or AFTER) and the manipulation action that automatically fires the trigger (INSERT, UPDATE, or DELETE)

- `BEFORE UPDATE ON employees FOR EACH ROW` – Allows a DBA to specify when the trigger is set to be executed (BEFORE or AFTER) and the manipulation action that automatically fires the trigger (INSERT, UPDATE, or DELETE)
- `BEFORE UPDATE ON employees FOR EACH ROW` – A trigger must be associated with an existing relation so that it is executed when a manipulation action is performed on the data in that associated relation. “`ON employees`” allows us to associate the trigger with the table “employees”.
- `BEFORE UPDATE ON employees FOR EACH ROW` – A **row-level trigger** is activated for each row that is inserted, updated, or deleted. For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically executed 100 times for each of the 100 rows affected. On the other hand, a **statement-level trigger** is executed once for each database transaction regardless of how many rows are inserted, updated, or deleted. MySQL and MariaDB DBMSs support **only** row-level triggers.
- The “`BEFORE UPDATE ON employees FOR EACH ROW`” line is followed by the SQL statement that is executed when the trigger is executed. In this case, it is an INSERT SQL statement, however, it can also be an UPDATE or a DELETE SQL statement.

- The CREATE TRIGGER statement can also have a PRECEDES or FOLLOW clause as shown below:

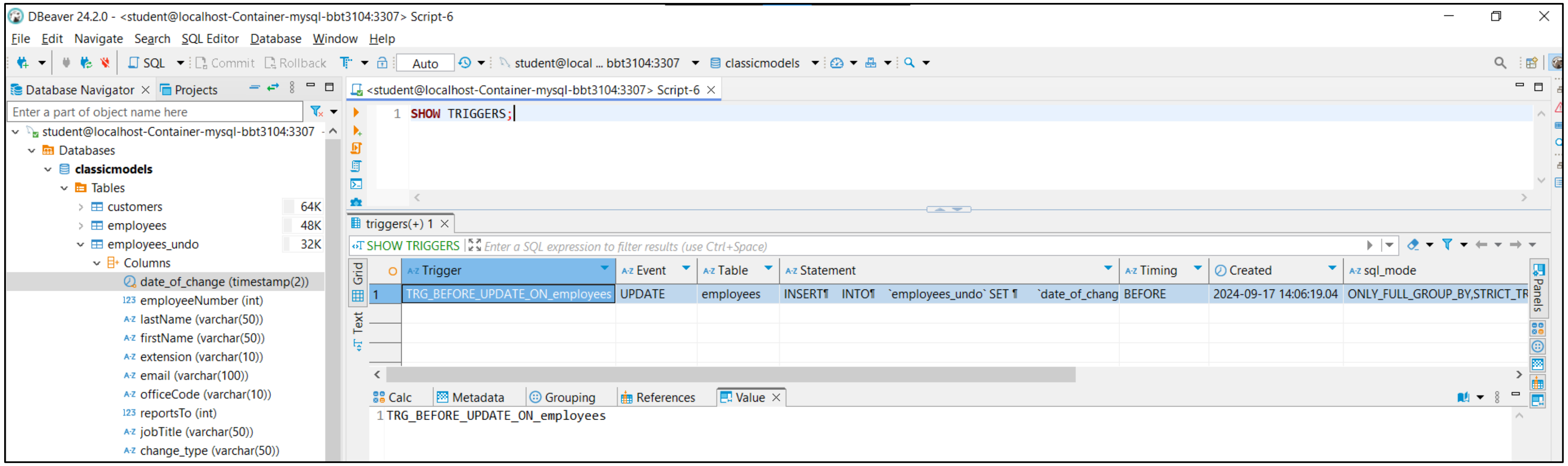
```
CREATE
    TRIGGER TRG_BEFORE_UPDATE_ON_employees_TRG2
    BEFORE
UPDATE
    ON
    employees FOR EACH ROW
    FOLLOWS TRG_BEFORE_UPDATE_ON_employees
```

- This means that **`TRG_BEFORE_UPDATE_ON_employees_TRG2`** is executed **after** **`TRG_BEFORE_UPDATE_ON_employees`**. It is used in a case where there are multiple triggers that are executed at the same time (BEFORE or AFTER) and are also executed by the same manipulation action (INSERT, UPDATE, or DELETE) on the same relation. For example, two BEFORE UPDATE triggers on the relation employees requires the DBA to tell the DBMS which of the two triggers should be executed first (the order of firing the triggers).

STEP 6. Confirm that the trigger exists in the database

You can confirm that the trigger has been created by executing the following command:

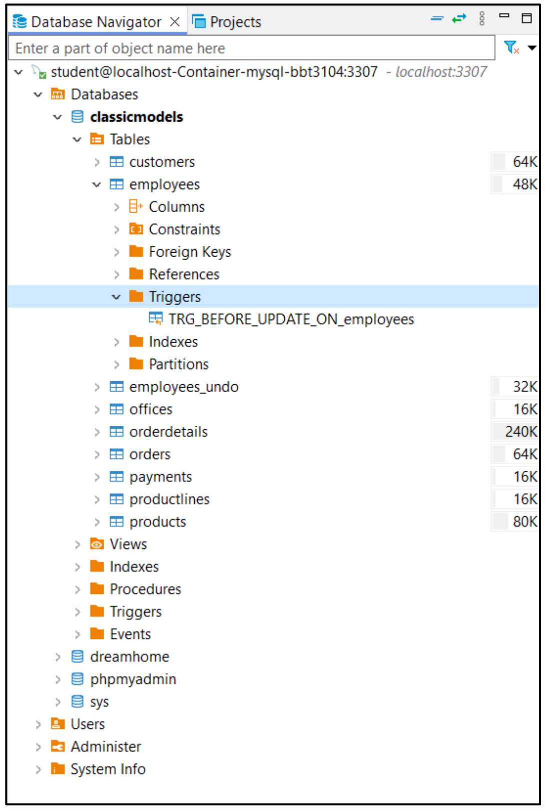
```
SHOW TRIGGERS;
```


DBeaver 24.2.0 - <student@localhost-Container-mysql-bbt3104:3307> Script-6
File Edit Navigate Search SQL Editor Database Window Help
SQL Commit Rollback Auto student@local ... bbt3104:3307 classicmodels
Database Navigator Projects
Enter a part of object name here
student@localhost-Container-mysql-bbt3104:3307
Databases
classicmodels
Tables
customers 64K
employees 48K
employees_undo 32K
Columns
date_of_change (timestamp(2))
employeeNumber (int)
lastName (varchar(50))
firstName (varchar(50))
extension (varchar(10))
email (varchar(100))
officeCode (varchar(10))
reportsTo (int)
jobTitle (varchar(50))
change_type (varchar(50))
<student@localhost-Container-mysql-bbt3104:3307> Script-6
triggers(+) 1
SHOW TRIGGERS
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
Calc Metadata Grouping References Value
Panels

```
SHOW TRIGGERS;
```

	Trigger	Event	Table	Statement	Timing	Created	sql_mode
1	TRG_BEFORE_UPDATE_ON_employees	UPDATE	employees	INSERT¶ INTO¶ `employees_undo` SET ¶ `date_of_chang	BEFORE	2024-09-17 14:06:19.04	ONLY_FULL_GROUP_BY,STRICT_TR

```
1 TRG_BEFORE_UPDATE_ON_employees
```



Or by expanding the trigger option in DBeaver.

STEP 7. Confirm that the trigger is executed when an update is performed on the “employees” relation

Execute the following command to update Mary’s (employee number 1056) last name:

```
UPDATE
    `employees`
SET
    `lastName` = 'Muiruri'
WHERE
    `employeeNumber` = '1056';
```

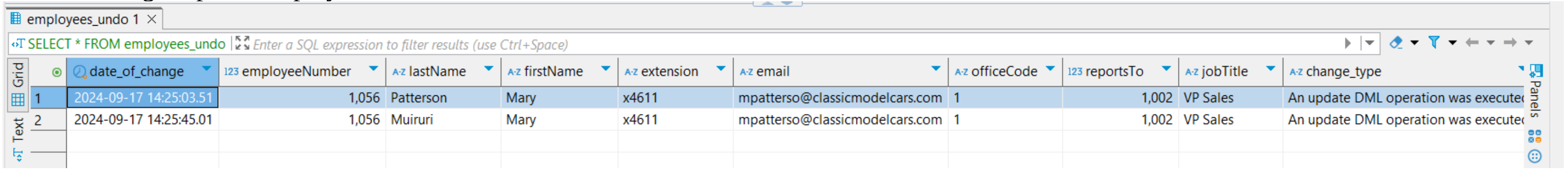
Execute the following command to update Mary’s (employee number 1056) email address:

```
UPDATE
    `employees`
SET
    `email` = 'mmuiruri@classicmodelcars.com'
WHERE
    `employeeNumber` = '1056';
```

Execute the following command to view the updated data:

```
SELECT
    *
FROM
    employees_undo;
```

The following output is displayed:



employees_undo 1

SELECT * FROM employees_undo | Enter a SQL expression to filter results (use Ctrl+Space)

	date_of_change	employeeNumber	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle	change_type
1	2024-09-17 14:25:03.51	1,056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1,002	VP Sales	An update DML operation was execute
2	2024-09-17 14:25:45.01	1,056	Muiruri	Mary	x4611	mpatterso@classicmodelcars.com	1	1,002	VP Sales	An update DML operation was execute

The first row shows how the data was before Mary's last name was changed whereas the second row shows how the data was before Mary's email address was changed. The programming team in the IT department can then use this data to implement an undo functionality in the GUI that shows the user options of different instances of Mary's data that the user can revert to.

BEGIN END Blocks

A DBA can specify that the trigger body should contain **more than one SQL statements** that execute when the trigger is executed. To make this possible, you must use a BEGIN END block to **enclose** the multiple SQL statements.

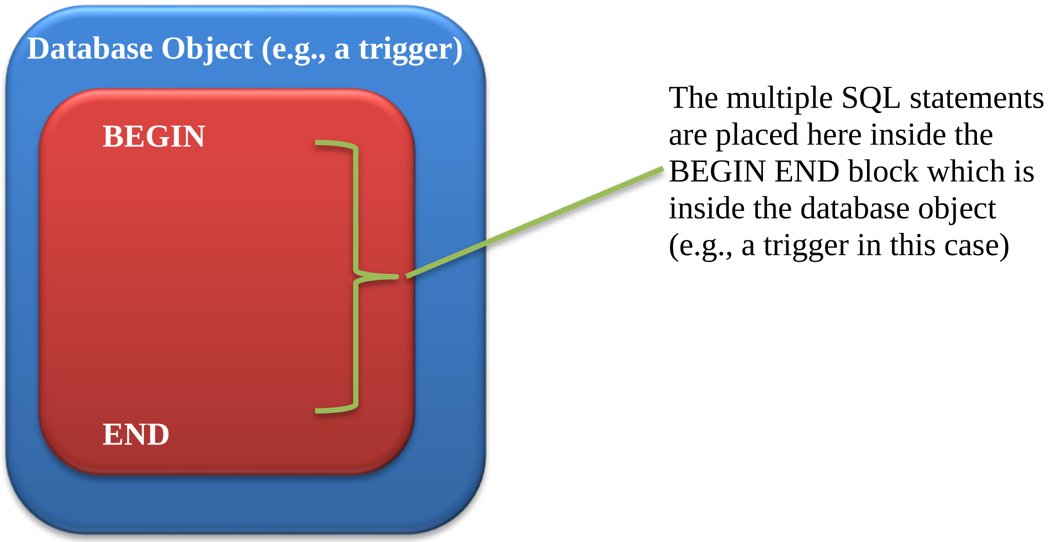

Database Object (e.g., a trigger)
BEGIN
END
The multiple SQL statements are placed here inside the BEGIN END block which is inside the database object (e.g., a trigger in this case)

Figure 1: Analogy of enclosed chicks under a hen's wings

MySQL DBMS (and many other programming languages) use a delimiter to separate multiple statements (or multiple lines of code). Each statement is then executed separately. The most common delimiter symbol is a semicolon (;). For example:

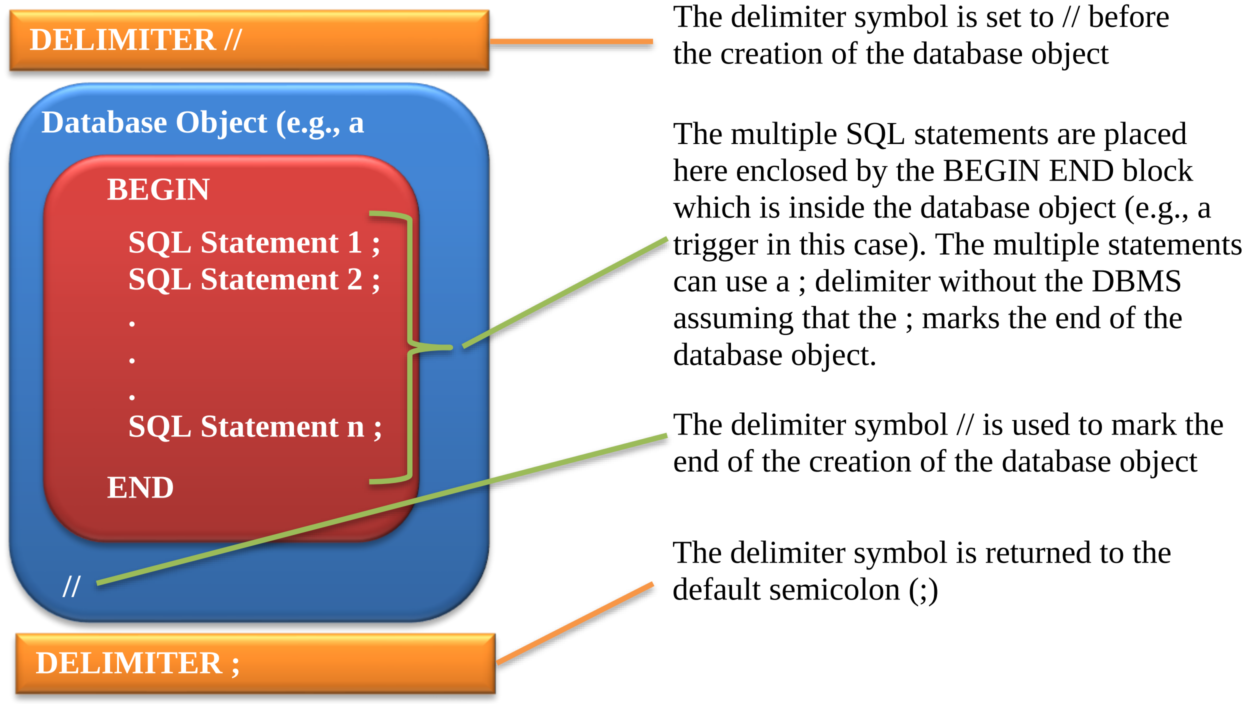
```
SELECT * FROM `employees`;

SELECT * FROM `customers`;
```

The issue is that the database object itself (e.g. a trigger in this case) also ends with the semicolon delimiter. We can, therefore, use a different delimiter **to differentiate between the end of the SQL statements inside the database object and the end of the database object itself.** To implement this, the SQL statements inside the database object can end with a semicolon delimiter whereas the database object can end with a different delimiter. This different delimiter can be any symbol, however, it is advisable to use the symbol twice. For example, instead of a semicolon (;) delimiter, we can use a dollar sign, e.g. $$ (notice the dollar sign is written twice). Writing the symbol twice avoids cases where the symbol is interpreted by the DBMS to have a different meaning. For example, a > symbol in SQL is interpreted as a "greater than" comparison operator, but the symbol >> has no pre-defined meaning in SQL.

The following code is used to redefine a delimiter from the default (;) to a new symbol (in this case, //). Notice that it does not end with a semicolon (;).

```
DELIMITER //
```

It is necessary to change the delimiter back to a semicolon (;) once we have created the database object. This allows the DBA to continue using a semicolon (;) to mark the end of statements. For example:


DELIMITER //
Database Object (e.g., a
BEGIN
SQL Statement 1 ;
SQL Statement 2 ;
.
.
.
SQL Statement n ;
END
//
DELIMITER ;
The delimiter symbol is set to // before the creation of the database object
The multiple SQL statements are placed here enclosed by the BEGIN END block which is inside the database object (e.g., a trigger in this case). The multiple statements can use a ; delimiter without the DBMS assuming that the ; marks the end of the database object.
The delimiter symbol // is used to mark the end of the creation of the database object
The delimiter symbol is returned to the default semicolon (;)

Suppose that the customer service manager notices that the employee doing the data entry does not include all the data when a new client's record is stored in the database. This could be because the client does not have the full required data at the point of registration. The customer service manager requires that **the database should** allow the data entry clerk to record missing values but remind him/her to prompt the client to provide the missing data in future when it is available. The programming team can then create a user interface that shows a pop-up message whenever the client's record is accessed by a customer service officer at the front-office or when the client calls the customer care.

STEP 8. Create a table to store reminders

Execute the following command to create a table to store reminders. The reminders should enable the customer service officers to know what data is missing in a client’s record.

```
CREATE TABLE `customers_data_reminders` (
  `customerNumber` int NOT NULL COMMENT 'Identifies the customer whose data is partly missing',
  `customers_data_reminders_timestamp` timestamp(2) NOT NULL DEFAULT CURRENT_TIMESTAMP(2) COMMENT 'Records the time when the
missing data was detected',
  `customers_data_reminders_message` varchar(100) NOT NULL COMMENT 'Records a message that helps the customer service
personnel to know what data is missing from the customer\' s record',
  `customers_data_reminders_status` tinyint NOT NULL DEFAULT ' 0 ' COMMENT ' Used TO record the status OF a reminder (0 IF it
has NOT yet been addressed
AND 1 IF it has been addressed)',
  PRIMARY KEY
(`customerNumber`,`customers_data_reminders_timestamp`,`customers_data_reminders_message`,`customers_data_reminders_status`),
  CONSTRAINT `FK_1_customers_TO_M_customers_data_reminders` FOREIGN KEY (`customerNumber`) REFERENCES `customers`
(`customerNumber`)
  ON DELETE CASCADE
  ON UPDATE CASCADE
) ENGINE=InnoDB COMMENT=' Used TO remind the customer service personnel about a client\'s missing data. This enables them to
ask the client to provide the data during the next interaction with the client.'
```

STEP 9. Create a trigger that has multiple statements

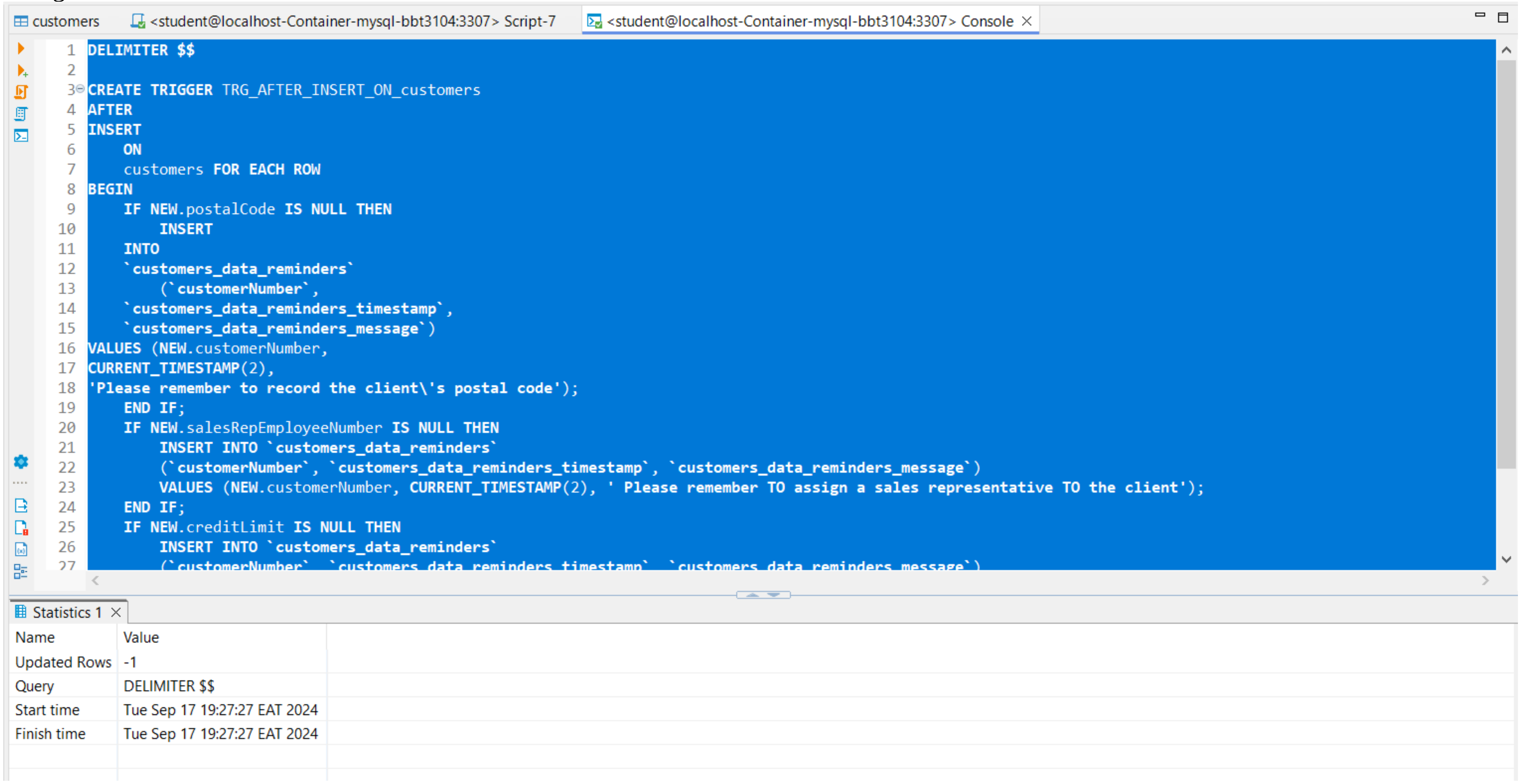
Execute the following command to create the trigger:

```
DELIMITER $$

CREATE TRIGGER TRG_AFTER_INSERT_ON_customers
AFTER
INSERT
    ON
    customers FOR EACH ROW
BEGIN
    IF NEW.postalCode IS NULL THEN
        INSERT
    INTO
    `customers_data_reminders`
        (`customerNumber`,
    `customers_data_reminders_timestamp`,
    `customers_data_reminders_message`)
VALUES (NEW.customerNumber,
CURRENT_TIMESTAMP(2),
'Please remember to record the client\'s postal code');
    END IF;
    IF NEW.salesRepEmployeeNumber IS NULL THEN
        INSERT INTO `customers_data_reminders`
        (`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`)
        VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), 'Please remember TO assign a sales representative TO the client');
    END IF;
    IF NEW.creditLimit IS NULL THEN
        INSERT INTO `customers_data_reminders`
        (`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`)
        VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), 'Please remember TO SET the client\'s credit limit');
END IF;
END$$
DELIMITER ;
```

STEP 10. Use other interfaces to connect to the MySQL server and confirm that the trigger has been created

It is standard practice to always confirm that the changes you have made have been implemented. You may notice that Graphical User Interfaces such as DBeaver, phpMyAdmin, etc. may not always work as expected. Hence the reason why some DBAs prefer to execute the code in Command Line.

Using DBeaver:

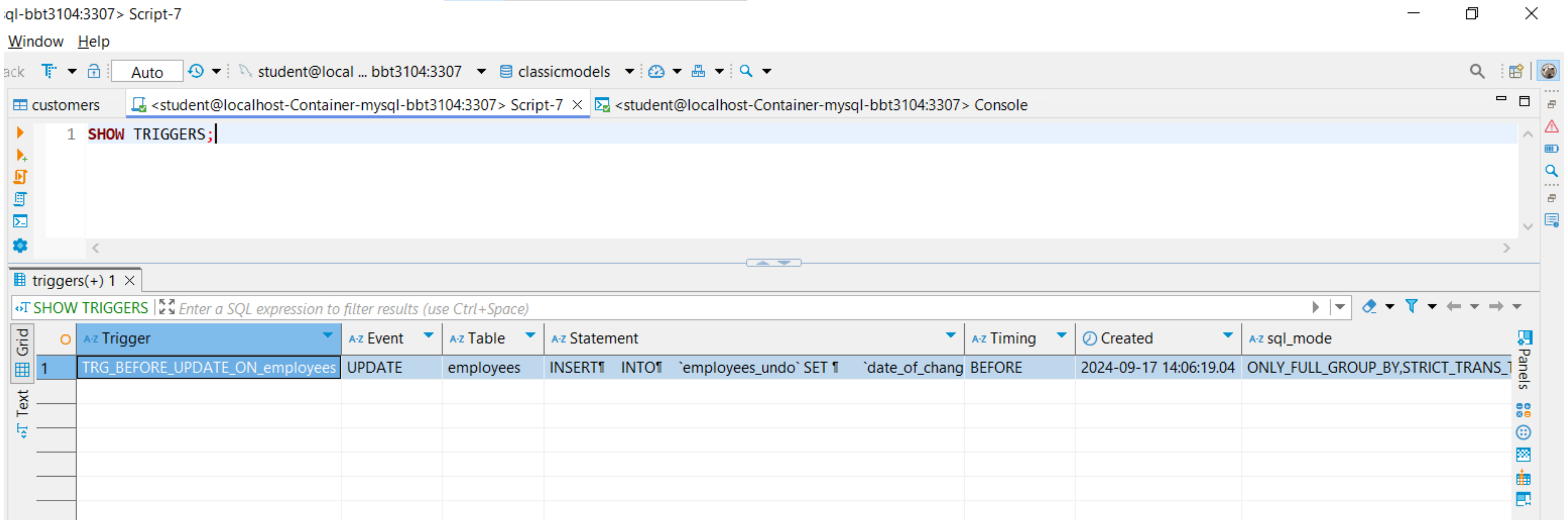

customers
<student@localhost-Container-mysql-bbt3104:3307> Script-7
<student@localhost-Container-mysql-bbt3104:3307> Console
Statistics 1

```sql
DELIMITER $$

CREATE TRIGGER TRG_AFTER_INSERT_ON_customers
AFTER
INSERT
    ON
    customers FOR EACH ROW
BEGIN
    IF NEW.postalCode IS NULL THEN
        INSERT
    INTO
    `customers_data_reminders`
        (`customerNumber`,
    `customers_data_reminders_timestamp`,
    `customers_data_reminders_message`)
VALUES (NEW.customerNumber,
CURRENT_TIMESTAMP(2),
'Please remember to record the client\'s postal code');
    END IF;
    IF NEW.salesRepEmployeeNumber IS NULL THEN
        INSERT INTO `customers_data_reminders`
        (`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`)
        VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), ' Please remember TO assign a sales representative TO the client');
    END IF;
    IF NEW.creditLimit IS NULL THEN
        INSERT INTO `customers_data_reminders`
        (`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`)
```

Name	Value
Updated Rows	-1
Query	DELIMITER $$
Start time	Tue Sep 17 19:27:27 EAT 2024
Finish time	Tue Sep 17 19:27:27 EAT 2024

```
SHOW TRIGGERS;
```

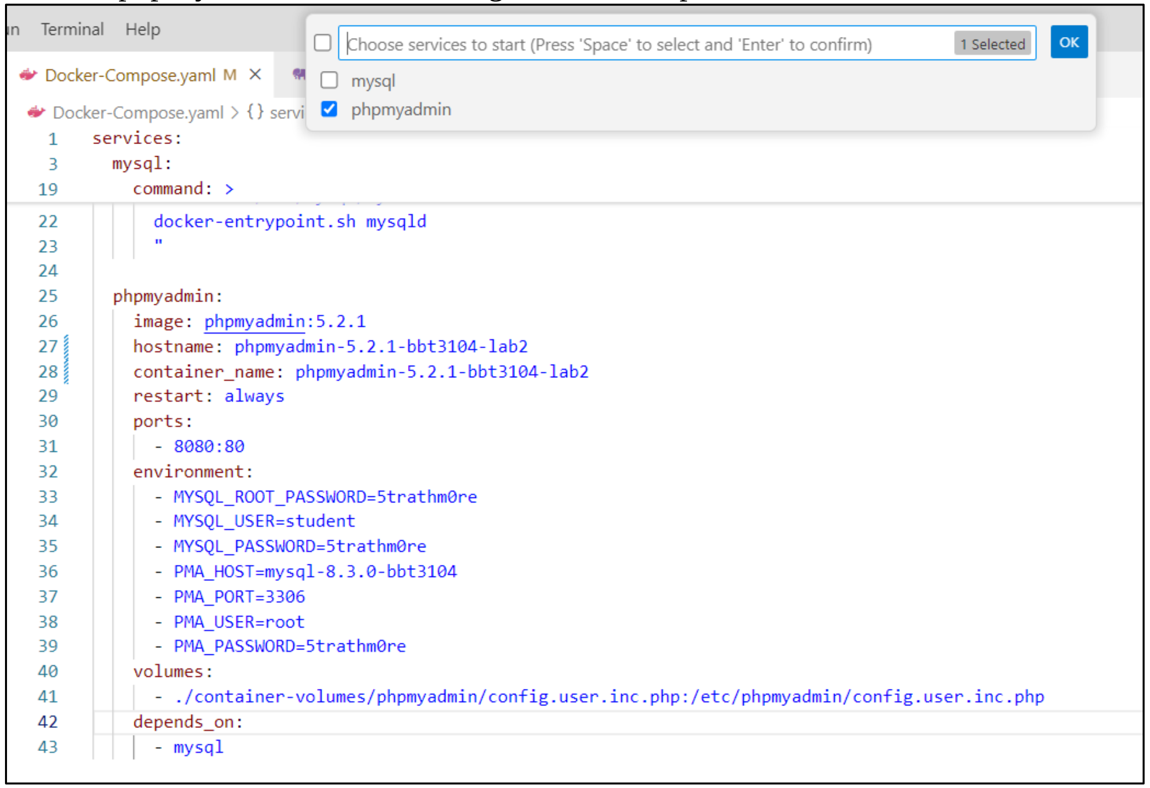
The trigger has not been created:



```
SHOW TRIGGERS;
```

Trigger	Event	Table	Statement	Timing	Created	sql_mode
TRG_BEFORE_UPDATE_ON_employees	UPDATE	employees	INSERT¶ INTO¶ `employees_undo` SET ¶ `date_of_chang	BEFORE	2024-09-17 14:06:19.04	ONLY_FULL_GROUP_BY,STRICT_TRANS_1

Execute the same command in phpMyAdmin.
Start the phpMyAdmin container using Docker-Compose:

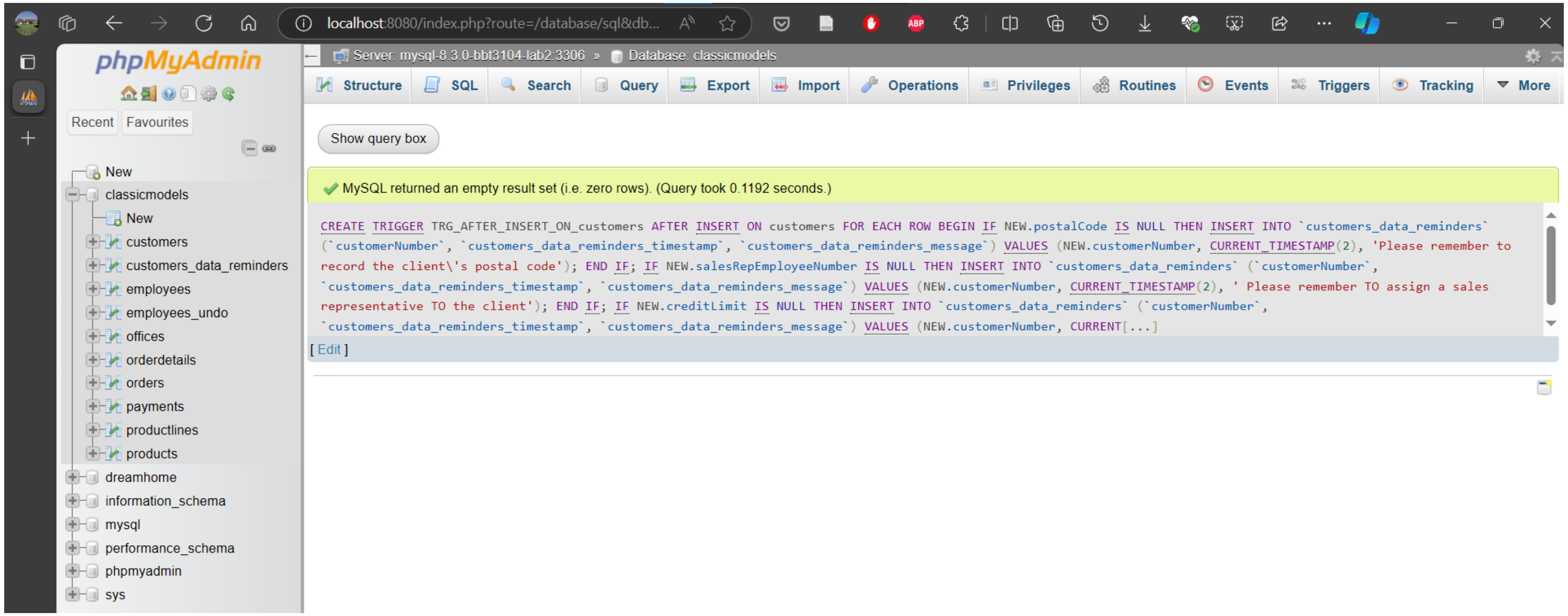

Terminal
Help
Docker-Compose.yaml M
Docker-Compose.yaml > {} servi
Choose services to start (Press 'Space' to select and 'Enter' to confirm)
1 Selected
OK

[ ] mysql
[✓] phpmyadmin

```yaml
services:
  mysql:
    command: >
      docker-entrypoint.sh mysqld
      "

  phpmyadmin:
    image: phpmyadmin:5.2.1
    hostname: phpmyadmin-5.2.1-bbt3104-lab2
    container_name: phpmyadmin-5.2.1-bbt3104-lab2
    restart: always
    ports:
      - 8080:80
    environment:
      - MYSQL_ROOT_PASSWORD=5trathm0re
      - MYSQL_USER=student
      - MYSQL_PASSWORD=5trathm0re
      - PMA_HOST=mysql-8.3.0-bbt3104
      - PMA_PORT=3306
      - PMA_USER=root
      - PMA_PASSWORD=5trathm0re
    volumes:
      - ./container-volumes/phpmyadmin/config.user.inc.php:/etc/phpmyadmin/config.user.inc.php
    depends_on:
      - mysql
```

Then login using the URL http://localhost:8080/ Username student and Password 5trathm0re

Execute the same command provided in STEP 9.



✓ MySQL returned an empty result set (i.e. zero rows). (Query took 0.1192 seconds.)

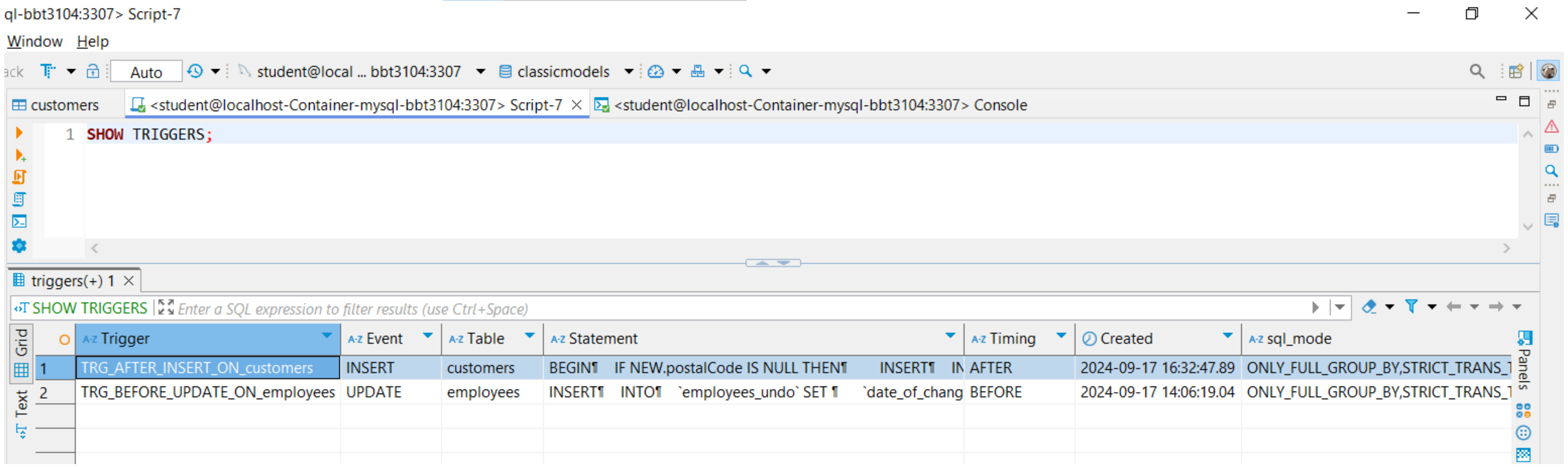
```
CREATE TRIGGER TRG_AFTER_INSERT_ON_customers AFTER INSERT ON customers FOR EACH ROW BEGIN IF NEW.postalCode IS NULL THEN INSERT INTO `customers_data_reminders`
(`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`) VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), 'Please remember to
record the client\'s postal code'); END IF; IF NEW.salesRepEmployeeNumber IS NULL THEN INSERT INTO `customers_data_reminders` (`customerNumber`,
`customers_data_reminders_timestamp`, `customers_data_reminders_message`) VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), ' Please remember TO assign a sales
representative TO the client'); END IF; IF NEW.creditLimit IS NULL THEN INSERT INTO `customers_data_reminders` (`customerNumber`,
`customers_data_reminders_timestamp`, `customers_data_reminders_message`) VALUES (NEW.customerNumber, CURRENT[...]
```

[Edit]

Confirm if the trigger has been created. Use DBeaver to execute the following command again:

```
SHOW TRIGGERS;
```

The result shows that the same code works in phpMyAdmin but does not work in DBeaver:

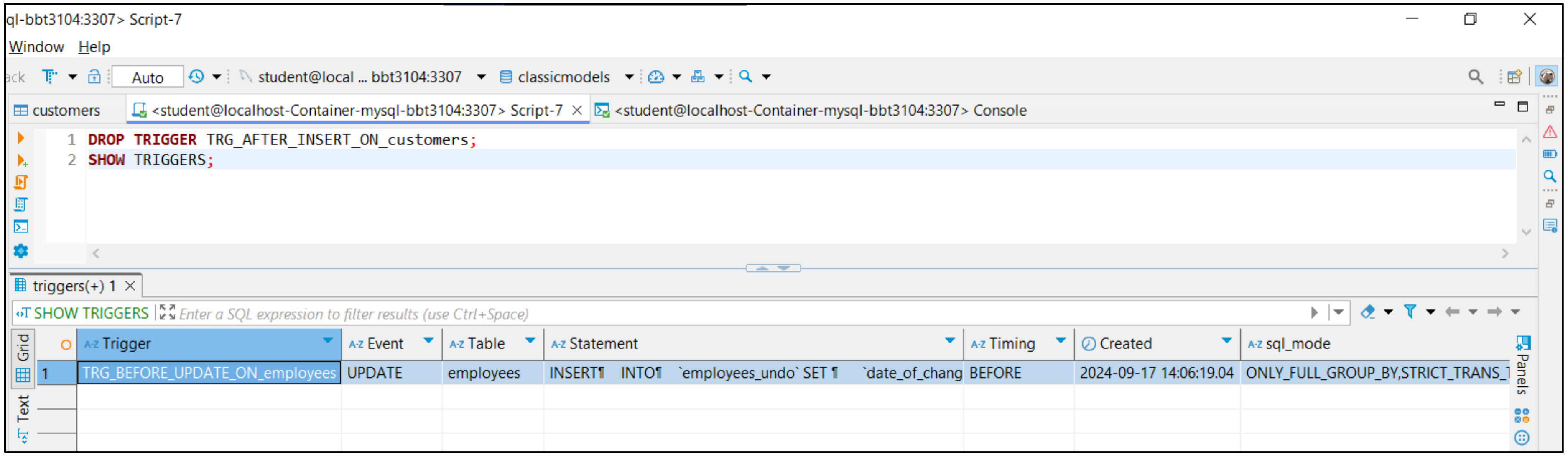


```
SHOW TRIGGERS;
```

	Trigger	Event	Table	Statement	Timing	Created	sql_mode
1	TRG_AFTER_INSERT_ON_customers	INSERT	customers	BEGIN¶ IF NEW.postalCode IS NULL THEN¶ INSERT¶ IN	AFTER	2024-09-17 16:32:47.89	ONLY_FULL_GROUP_BY,STRICT_TRANS_T
2	TRG_BEFORE_UPDATE_ON_employees	UPDATE	employees	INSERT¶ INTO¶ `employees_undo` SET ¶ `date_of_chang	BEFORE	2024-09-17 14:06:19.04	ONLY_FULL_GROUP_BY,STRICT_TRANS_T

Drop the trigger and create it (again) using the MySQL command line client. Execute the following in DBeaver:

```
DROP TRIGGER TRG_AFTER_INSERT_ON_customers;
SHOW TRIGGERS;
```



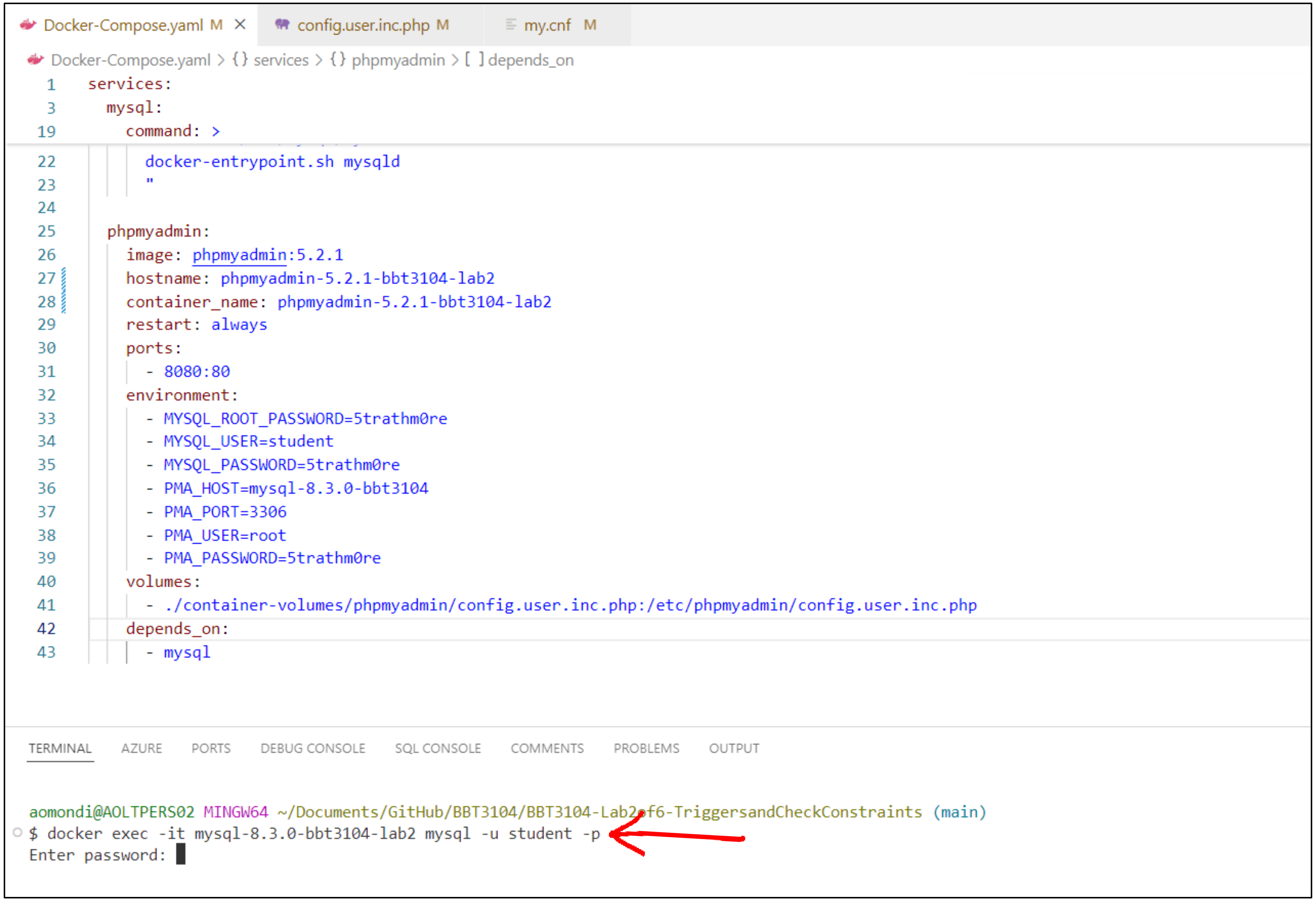
```
DROP TRIGGER TRG_AFTER_INSERT_ON_customers;
SHOW TRIGGERS;
```

	Trigger	Event	Table	Statement	Timing	Created	sql_mode
1	TRG_BEFORE_UPDATE_ON_employees	UPDATE	employees	INSERT¶ INTO¶ `employees_undo` SET ¶ `date_of_chang	BEFORE	2024-09-17 14:06:19.04	ONLY_FULL_GROUP_BY,STRICT_TRANS_T

Connect to the MySQL container's command line by executing the following in Git Bash using VS Code:

```
docker exec -it mysql-8.3.0-bbt3104-lab2 mysql -u student -p
```

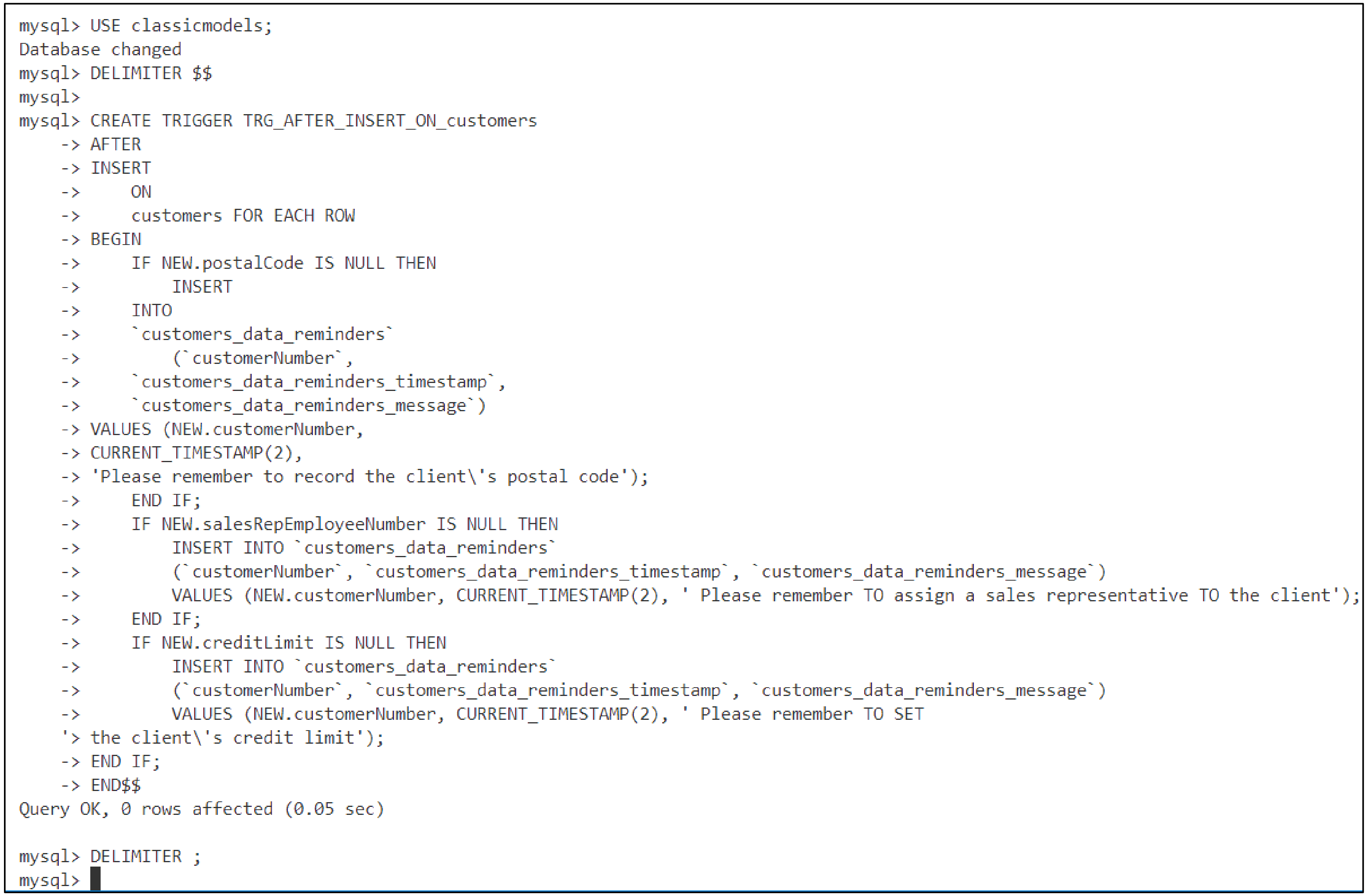
The password is "5trathm0re".



```yaml
services:
  mysql:
    command: >
      docker-entrypoint.sh mysqld
      "

  phpmyadmin:
    image: phpmyadmin:5.2.1
    hostname: phpmyadmin-5.2.1-bbt3104-lab2
    container_name: phpmyadmin-5.2.1-bbt3104-lab2
    restart: always
    ports:
      - 8080:80
    environment:
      - MYSQL_ROOT_PASSWORD=5trathm0re
      - MYSQL_USER=student
      - MYSQL_PASSWORD=5trathm0re
      - PMA_HOST=mysql-8.3.0-bbt3104
      - PMA_PORT=3306
      - PMA_USER=root
      - PMA_PASSWORD=5trathm0re
    volumes:
      - ./container-volumes/phpmyadmin/config.user.inc.php:/etc/phpmyadmin/config.user.inc.php
    depends_on:
      - mysql
```

```
aomondi@AOLTPERS02 MINGW64 ~/Documents/GitHub/BBT3104/BBT3104-Lab2of6-TriggersandCheckConstraints (main)
$ docker exec -it mysql-8.3.0-bbt3104-lab2 mysql -u student -p
Enter password:
```

Use the same code provided in STEP 9 to create the trigger in the command line interface provided by MySQL client.

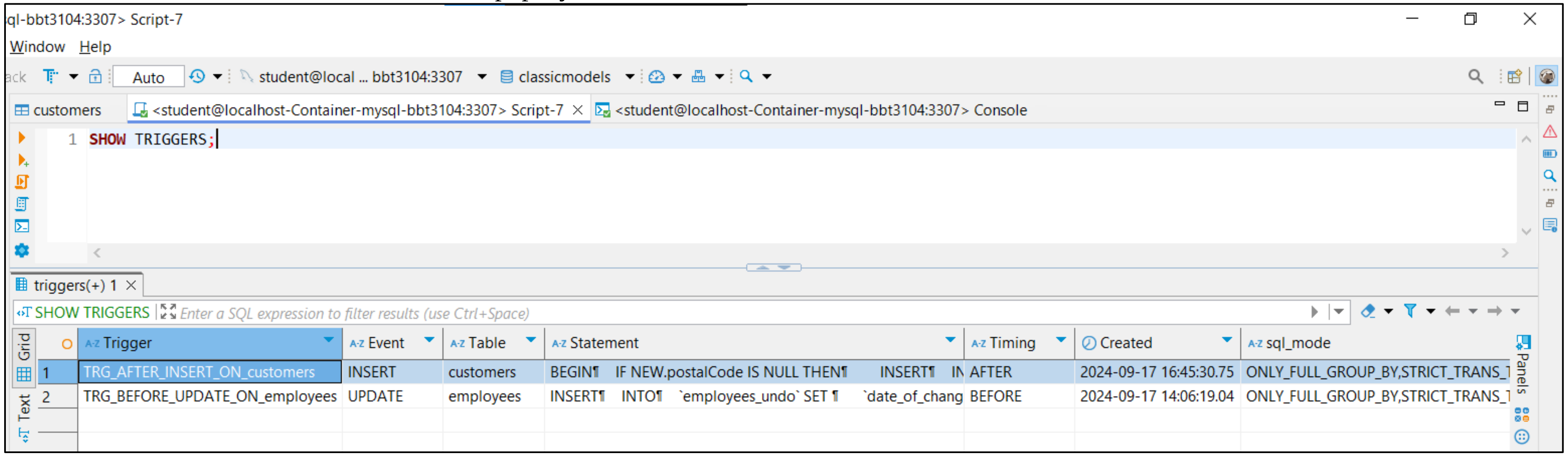


```
mysql> USE classicmodels;
Database changed
mysql> DELIMITER $$
mysql>
mysql> CREATE TRIGGER TRG_AFTER_INSERT_ON_customers
    -> AFTER
    -> INSERT
    ->     ON
    ->     customers FOR EACH ROW
    -> BEGIN
    ->     IF NEW.postalCode IS NULL THEN
    ->         INSERT
    ->     INTO
    ->     `customers_data_reminders`
    ->         (`customerNumber`,
    ->     `customers_data_reminders_timestamp`,
    ->     `customers_data_reminders_message`)
    -> VALUES (NEW.customerNumber,
    -> CURRENT_TIMESTAMP(2),
    -> 'Please remember to record the client\'s postal code');
    ->     END IF;
    ->     IF NEW.salesRepEmployeeNumber IS NULL THEN
    ->         INSERT INTO `customers_data_reminders`
    ->         (`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`)
    ->         VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), ' Please remember TO assign a sales representative TO the client');
    ->     END IF;
    ->     IF NEW.creditLimit IS NULL THEN
    ->         INSERT INTO `customers_data_reminders`
    ->         (`customerNumber`, `customers_data_reminders_timestamp`, `customers_data_reminders_message`)
    ->         VALUES (NEW.customerNumber, CURRENT_TIMESTAMP(2), ' Please remember TO SET
    '> the client\'s credit limit');
    -> END IF;
    -> END$$
Query OK, 0 rows affected (0.05 sec)

mysql> DELIMITER ;
mysql>
```

Confirm if the trigger has been created. Use DBeaver to execute the following command again:

```
SHOW TRIGGERS;
```

The result shows that the same code works in phpMyAdmin but does not work in DBeaver:



```
SHOW TRIGGERS;
```

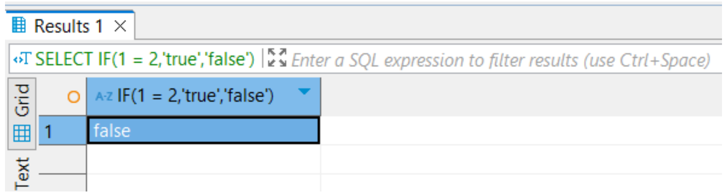
	Trigger	Event	Table	Statement	Timing	Created	sql_mode
1	TRG_AFTER_INSERT_ON_customers	INSERT	customers	BEGIN¶ IF NEW.postalCode IS NULL THEN¶ INSERT¶ IN	AFTER	2024-09-17 16:45:30.75	ONLY_FULL_GROUP_BY,STRICT_TRANS_1
2	TRG_BEFORE_UPDATE_ON_employees	UPDATE	employees	INSERT¶ INTO¶ `employees_undo` SET ¶ `date_of_chang	BEFORE	2024-09-17 14:06:19.04	ONLY_FULL_GROUP_BY,STRICT_TRANS_1

Conclusion: In cases where a GUI database tool is not working as expected, you can default to using the Command Line Interface.

IF Function versus IF Statement

IF(expr, if_true_expr, if_false_expr) is a control flow function in the MySQL DBMS that returns a value based on a condition. If the **expr** (expression) evaluates to **TRUE**, i.e., **expr** is **NOT NULL (expr <> NULL)** and **expr is not 0 (expr <> 0)**, the **IF** function returns the **if_true_expr** , otherwise, it returns **if_false_expr**. For example, the following statement will return false:

```
SELECT IF(1 = 2,'true','false');
```



Results 1

SELECT IF(1 = 2,'true','false')

	IF(1 = 2,'true','false')
1	false

```
IF(1 = 2,'true','false')|
------------------------+
false                   |
```

Another example of the IF () function is as follows. Executing the following command will display “No data provided” instead of displaying NULL (the lack of a value):

```
SELECT
    customerNumber,
    customerName,
    IF(state IS NULL,
    'No data provided',
    state) state,
    country
FROM
    customers;
```

Result set:

```
customerNumber|customerName                      |state           |country     |
--------------+----------------------------------+----------------+------------+
           103|Atelier graphique                 |No data provided|France      |
           112|Signal Gift Stores                |NV              |USA         |
           114|Australian Collectors, Co.        |Victoria        |Australia   |
           119|La Rochelle Gifts                 |No data provided|France      |
           121|Baane Mini Imports                |No data provided|Norway      |
           124|Mini Gifts Distributors Ltd.      |CA              |USA         |
           125|Havel & Zbyszek Co                |No data provided|Poland      |
           128|Blauer See Auto, Co.              |No data provided|Germany     |
           129|Mini Wheels Co.                   |CA              |USA         |
           131|Land of Toys Inc.                 |NY              |USA         |
```

The IF function can also be used in other creative ways, for example:

```sql
SELECT
    SUM(IF(status = 'Shipped', 1, 0)) AS Shipped,
    SUM(IF(status = 'Cancelled', 1, 0)) AS Cancelled
FROM
    orders;
```

Which returns the following result set:

```
Shipped|Cancelled|
-------+---------+
    303|        6|
```

Other control flow functions include:

Name	Description
CASE	Case operator
IFNULL ()	Null if/else construct
NULLIF ()	Return NULL if expression 1 = expression 2

<u>Note:</u> The **`IF ()`** function is different from the **`IF statement`**. The **`IF statement`** has 3 possible forms **(notice the lack of brackets in the `IF statement`)**:

```
/* (1) IF THEN Statement */
IF CONDITION THEN
    statements;
END IF;

/* (2) IF THEN ELSE Statement */
IF CONDITION THEN
    statements;
ELSE
    else-statements;
END IF;

/* (3) IF THEN ELSEIF ELSE Statement */
IF CONDITION THEN
    statements;
ELSEIF elseif-condition THEN
    else-statements;
ELSEIF elseif-condition THEN
    else-statements;
...
ELSE
    else-statements
END IF;
```

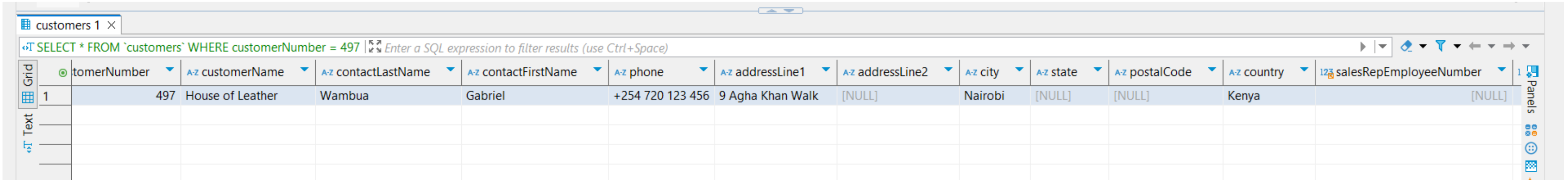
The **`IF statement`** allows a DBA to specify more than one statement separated by a semicolon. Also, notice that the **`IF statement`** ends with a semicolon. We used an **`IF statement`** and not an **`IF () function`** in STEP 9.

STEP 11. Confirm that the trigger is executed when an insertion is performed on the “customers” relation

Execute the following command to create a new client’s record. The client is House of Leather gift shop in Nairobi:

```
INSERT
    INTO
    `customers`
(`customerNumber`,
    `customerName`,
    `contactLastName`,
    `contactFirstName`,
    `phone`,
    `addressLine1`,
    `city`,
    `country`)
VALUES
(497,
'House of Leather',
'Wambua',
'Gabriel',
'+254 720 123 456',
'9 Agha Khan Walk',
'Nairobi',
'Kenya');

SELECT * FROM `customers` WHERE customerNumber = 497;
```

As shown below, some attributes are NULL (NULL represents the absence of a value).



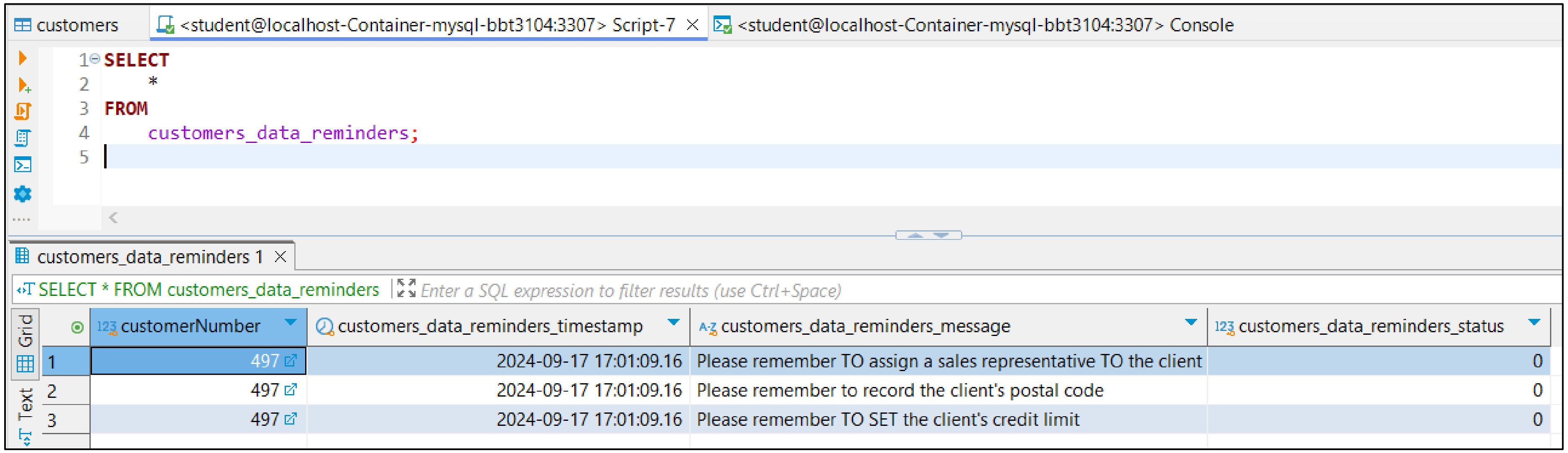
customers 1

SELECT * FROM `customers` WHERE customerNumber = 497

	tomerNumber	customerName	contactLastName	contactFirstName	phone	addressLine1	addressLine2	city	state	postalCode	country	salesRepEmployeeNumber
1	497	House of Leather	Wambua	Gabriel	+254 720 123 456	9 Agha Khan Walk	[NULL]	Nairobi	[NULL]	[NULL]	Kenya	[NULL]

Execute the following statement to confirm that the insertion executed the trigger which then performed an insertion in the "`customer_data_reminders`" relation:

```
SELECT
    *
FROM
    customers_data_reminders;
```



customers | <student@localhost-Container-mysql-bbt3104:3307> Script-7 | <student@localhost-Container-mysql-bbt3104:3307> Console

```
SELECT
    *
FROM
    customers_data_reminders;
```

customers_data_reminders 1

SELECT * FROM customers_data_reminders | Enter a SQL expression to filter results (use Ctrl+Space)

	customerNumber	customers_data_reminders_timestamp	customers_data_reminders_message	customers_data_reminders_status
1	497	2024-09-17 17:01:09.16	Please remember TO assign a sales representative TO the client	0
2	497	2024-09-17 17:01:09.16	Please remember to record the client's postal code	0
3	497	2024-09-17 17:01:09.16	Please remember TO SET the client's credit limit	0

The Check Constraint Syntax

STEP 12. Create a new table to store parts data

Parts of a model for example, the tyres of a “1958 Chevy Corvette Limited Edition”, can be used to replace old or damage parts. Execute the following command to view the product:

```
SELECT * FROM products WHERE products.productName LIKE '1958 Chevy Corvette Limited Edition';
```

Execute the following command to create a relation to store the parts of a model:

```
CREATE TABLE part (
    part_no VARCHAR(18) PRIMARY KEY,
    part_description VARCHAR(255),
    part_supplier_tax_PIN VARCHAR (11) CHECK (part_supplier_tax_PIN REGEXP '^[A-Z]{1}[0-9]{9}[A-Z]{1}$'),
    part_supplier_email VARCHAR (55),
    part_buyingprice DECIMAL(10, 2 ) NOT NULL CHECK (part_buyingprice >= 0),
    part_sellingprice DECIMAL(10, 2) NOT NULL,
    CONSTRAINT CHK_part_sellingprice_GT_buyingprice CHECK (part_sellingprice >= part_buyingprice),
    CONSTRAINT CHK_part_valid_supplier_email CHECK (part_supplier_email REGEXP '^[a-zA-Z0-9]{3,}@[a-zA-Z0-9]{1,}\\.[a-zA-Z0-
9.]{1,}$')
);
```

In this case, we use a CHECK constraint while creating the table. The CHECK CONSTRAINT syntax is as follows:

```
[CONSTRAINT [constraint_name]] CHECK (expression) [[NOT] ENFORCED]
```

- `[CONSTRAINT [constraint_name]]` – Allows you to specify the name for the check constraint that you want to create. If you omit the constraint name, MySQL automatically generates a name for you based on an auto-increment number. It is better to specify the name yourself so that you can know which constraint has been violated. However, the naming of check constraints works only in MySQL because MariaDB does not recognize user-defined check constraint names unless they are defined as a **table constraint** (explained in the next paragraph).

- `CHECK (expression)` – `expression` specifies the constraint condition as a Boolean expression that must evaluate to `TRUE` or `UNKNOWN` (for NULL values) for each row of the table. If the condition evaluates to FALSE, it fails, and a constraint violation occurs.

- `[[NOT] ENFORCED]` – Although optional, a DBA can specify the enforcement clause to indicate whether the check constraint is enforced:
  - Use **`ENFORCED`** or simply omit the **`ENFORCED`** clause to create and enforce the constraint.
  - Use **`NOT ENFORCED`** to create the constraint without enforcing it.

  Note that this works in MySQL and not in MariaDB because MariaDB does not recognize the **`ENFORCED`** keyword in check constraints.

A CHECK constraint can be specified as either **a table constraint** or **column constraint**:

- **Table constraint:** A table constraint does not appear within a column definition and can refer to any column(s). Forward references are permitted to columns appearing later in the table definition. In the example above, the following 2 statements create table constraints:
  - `CONSTRAINT CHK_part_sellingprice_GT_buyingprice CHECK (part_sellingprice >= part_buyingprice) ENFORCED`

  - `CONSTRAINT CHK_part_validation_supplier_email CHECK (part_supplier_email REGEXP '^[a-zA-Z0-9]{3,}@[a-zA-Z0-9]{1,}\\.[a-zA-Z0-9]{1,}$') ENFORCED`

- **Column constraint:** A column constraint appears within a column definition and can refer to only that column. In the example above, the following statement creates a column constraint. Notice that the constraint can only refer to the column called "**`part_buyingprice`**" and the constraint is not assigned any name in MariaDB. We also omit the keyword "**`CONSTRAINT`**".
  - `part_buyingprice DECIMAL(10,2 ) NOT NULL CHECK (part_buyingprice >= 0)`
  - `part_supplier_tax_PIN VARCHAR (11) CHECK (part_supplier_tax_PIN REGEXP '^[A-Z]{1}[0-9]{9}[A-Z]{1}$')`

It is not easy to remember all the possible regular expression rules, therefore, various resources can be used to support the process of creating a regular expression. Below are websites that can be useful:

- A tutorial on regular expressions: https://regexone.com/
- To test your regular expression: https://regexr.com/
- To understand how regular expressions are used in MySQL (MySQL>=8.4) and above: https://dev.mysql.com/doc/refman/8.4/en/regexp.html

You can also test your regular expressions in SQL as follows:

```
SELECT 'toysRus@example.co.ke' REGEXP '^[a-zA-Z0-9]{3,}@[a-zA-Z0-9]{1,}\\.[a-zA-Z0-9.]{1,}$';
```

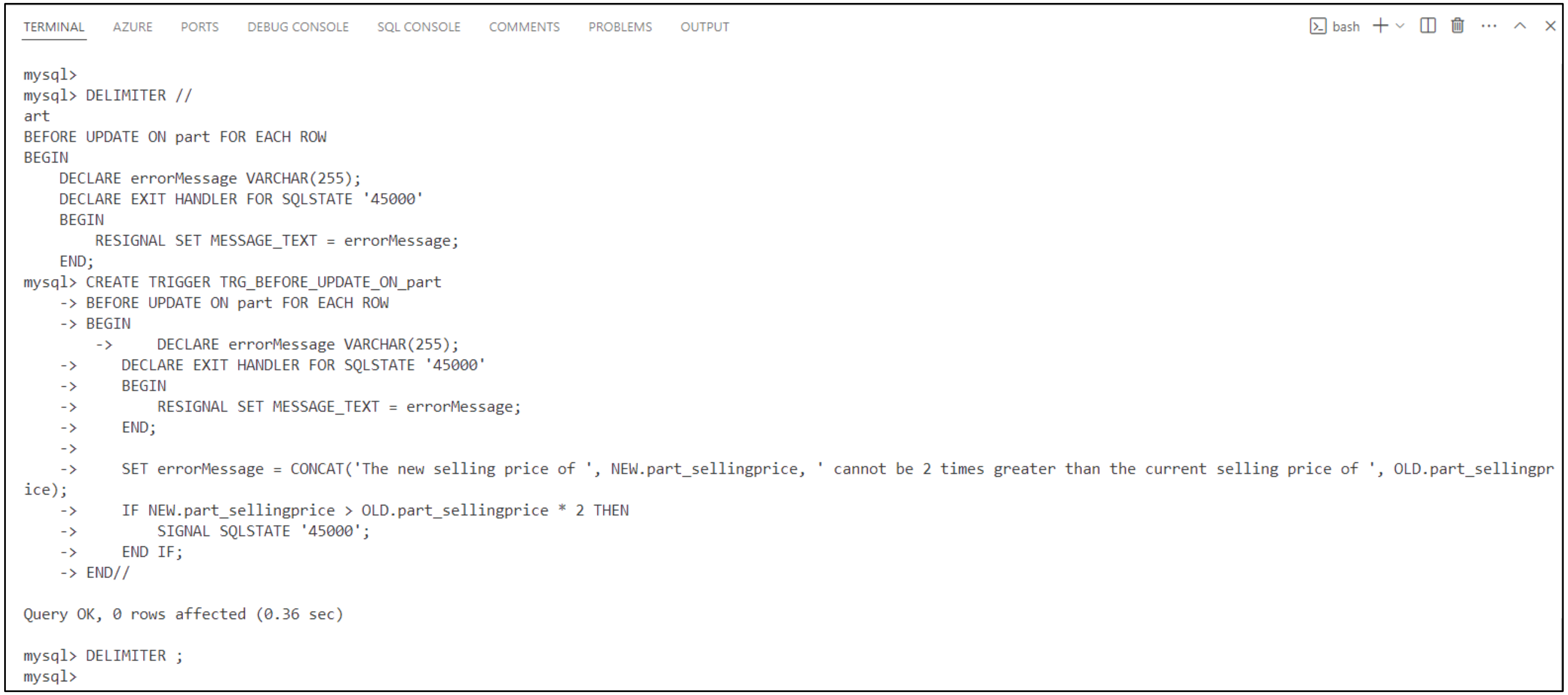
If it returns '1', then the text provided **does not** violate the regular expression.

Error Handling

STEP 13. Create the following user-defined error in a trigger

Execute the following statement (via the MySQL Command Line Interface not using a GUI like DBeaver/phpMyAdmin) to create a trigger that is executed when the data of the selling price is being changed. The rule of the business is that the selling price cannot be increased by more than double at once.

```
DELIMITER //
CREATE TRIGGER TRG_BEFORE_UPDATE_ON_part
BEFORE UPDATE ON part FOR EACH ROW
BEGIN
    DECLARE errorMessage VARCHAR(255);
    DECLARE EXIT HANDLER FOR SQLSTATE '45000'
    BEGIN
        RESIGNAL SET MESSAGE_TEXT = errorMessage;
    END;

    SET errorMessage = CONCAT('The new selling price of ', NEW.part_sellingprice, ' cannot be 2 times greater than the current
selling price of ', OLD.part_sellingprice);
    IF NEW.part_sellingprice > OLD.part_sellingprice * 2 THEN
        SIGNAL SQLSTATE '45000';
    END IF;
END//
DELIMITER ;
```

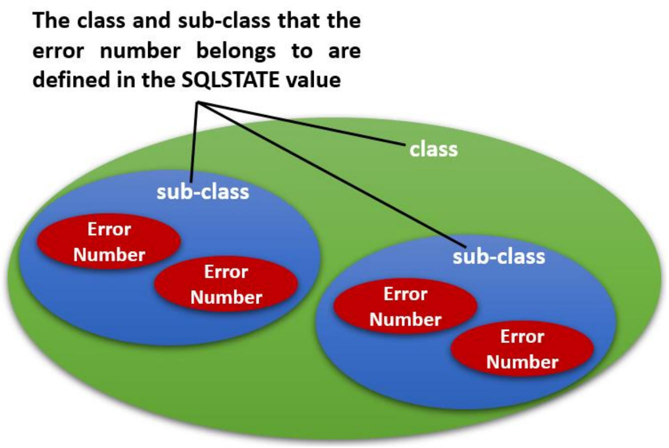


```
mysql>
mysql> DELIMITER //
art
BEFORE UPDATE ON part FOR EACH ROW
BEGIN
    DECLARE errorMessage VARCHAR(255);
    DECLARE EXIT HANDLER FOR SQLSTATE '45000'
    BEGIN
        RESIGNAL SET MESSAGE_TEXT = errorMessage;
    END;
mysql> CREATE TRIGGER TRG_BEFORE_UPDATE_ON_part
    -> BEFORE UPDATE ON part FOR EACH ROW
    -> BEGIN
        ->     DECLARE errorMessage VARCHAR(255);
    ->     DECLARE EXIT HANDLER FOR SQLSTATE '45000'
    ->     BEGIN
    ->         RESIGNAL SET MESSAGE_TEXT = errorMessage;
    ->     END;
    ->
    ->     SET errorMessage = CONCAT('The new selling price of ', NEW.part_sellingprice, ' cannot be 2 times greater than the current selling price of ', OLD.part_sellingpr
ice);
    ->     IF NEW.part_sellingprice > OLD.part_sellingprice * 2 THEN
    ->         SIGNAL SQLSTATE '45000';
    ->     END IF;
    -> END//

Query OK, 0 rows affected (0.36 sec)

mysql> DELIMITER ;
mysql>
```

The SIGNAL command is used to return an error to the caller. There are 3 categories of information that are returned in a MySQL error:

(i) **A numeric error number:** Error codes from **1,000 to 1,800** are shared by MySQL and MariaDB whereas error codes from **1,900 and above** are specific to MariaDB.

(ii) **An SQLSTATE value:** An SQLSTATE is a code which categorizes SQL error numbers. The **first 2** characters of an SQLSTATE specify the class whereas **the last 3** characters specify the sub-class of the error. This implies that multiple errors can be grouped into a sub-class, and multiple sub-classes can be grouped into a class. This is all defined in the SQLSTATE value that an error number belongs to as depicted in the diagram below.


The class and sub-class that the error number belongs to are defined in the SQLSTATE value
class
sub-class
Error Number
Error Number
sub-class
Error Number
Error Number

Both MySQL and MariaDB recommend using the "45000" SQLSTATE for user-defined errors.

(iii) **Error message:** The third category of information returned in the error message is a string describing the error.

The following syntax can be used to specify user-defined errors:

```
SIGNAL SQLSTATE 'sql_state_value' SET MESSAGE_TEXT = errorMessage;
```

Notice that it is adequate to specify only the **SQLSTATE** and **MESSAGE_TEXT** values when specifying a user-defined error. A list of MySQL error numbers and **SQLSTATE** values are available here (https://dev.mysql.com/doc/mysql-errors/8.4/en/server-error-reference.html) and a list of MariaDB error numbers and **SQLSTATE** values are available here (https://mariadb.com/kb/en/mariadb-error-code-reference/).

- `DECLARE EXIT HANDLER FOR SQLSTATE '45000'` Handling an error can involve either a continuation of the execution of the current code block (syntax: `DECLARE CONTINUE HANDLER`) or an exit of the current code block (syntax `DECLARE EXIT HANDLER`). This is then later followed by an appropriate error message that informs the user of what happened, why it happened, and what they can do next.
- Handler declarations must appear **after** variable declarations.
- The first SET keyword must come **after** the last DECLARE keyword.

The CHECK CONSTRAINT syntax, e.g., `CHK_validation_email` can be used to enforce a static integrity constraint (at the point of data insertion). On the other hand, a trigger, e.g. `TRG_BEFORE_UPDATE_ON_part` can be used to enforce a dynamic integrity constraint (at the point of updating data that already exists in the database).

A trigger, e.g., `TRG_BEFORE_UPDATE_ON_part` can also be used to enforce a semantic constraint. In this case, the constraint that states that **the selling price cannot increase by more than double its current price in one update**, is both a dynamic integrity constraint (concerned with how existing data can be updated) and a semantic integrity constraint (concerned with implementing a business rule). It constrains transactions that update the table so that the data in the table is consistent with the real-world business environment of Classic Models.

STEP 14. Confirm that the static, dynamic, and semantic constraints are working

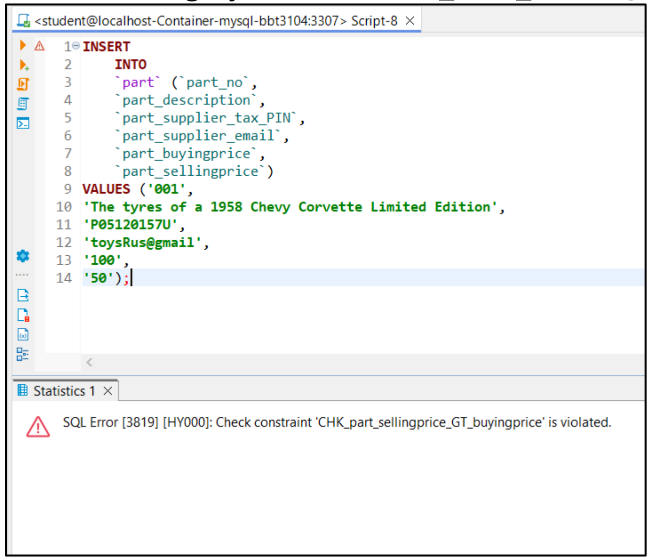
Note: The order of the errors displayed may vary depending on whether you are using MySQL or MariaDB. For example, MariaDB may start by indicating a violation of the KRA PIN format whereas MySQL may start by indicating a violation of the buying price and selling price figures.

Execute the following command to insert data into the relation "part":

```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P05120157U',
'toysRus@gmail',
'100',
'50');
```

User-Defined Error 1

The static integrity constraint '**`CHK_part_sellingprice_GT_buyingprice`**' requires that selling_price >= buying_price.



```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P05120157U',
'toysRus@gmail',
'100',
'50');
```

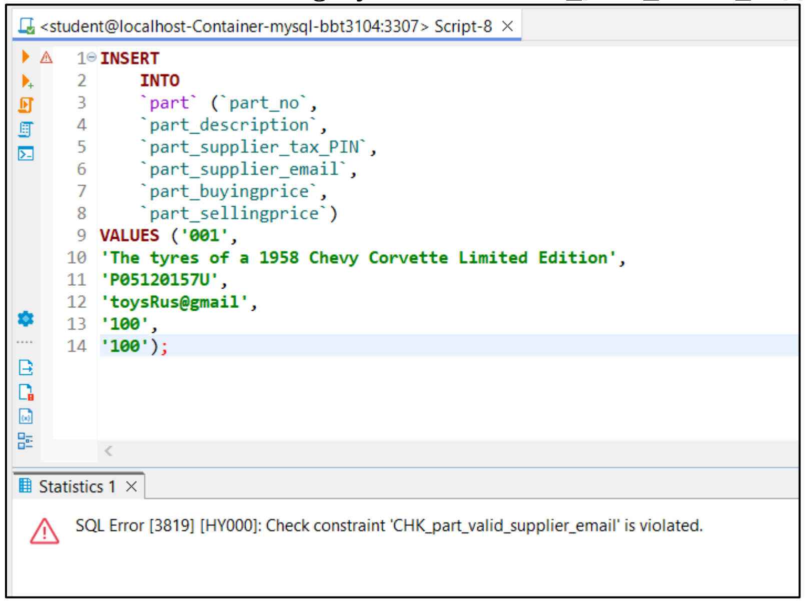
```
SQL Error [3819] [HY000]: Check constraint 'CHK_part_sellingprice_GT_buyingprice' is violated.
```

Execute the following to address the selling price >= buying price business rule.

```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P05120157U',
'toysRus@gmail',
'100',
'100');
```

User-Defined Error 2

This time, the static integrity constraint '**`CHK_part_valid_supplier_email`**' requires that the format of the email address should be valid.



<student@localhost-Container-mysql-bbt3104:3307> Script-8

```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P05120157U',
'toysRus@gmail',
'100',
'100');
```

Statistics 1

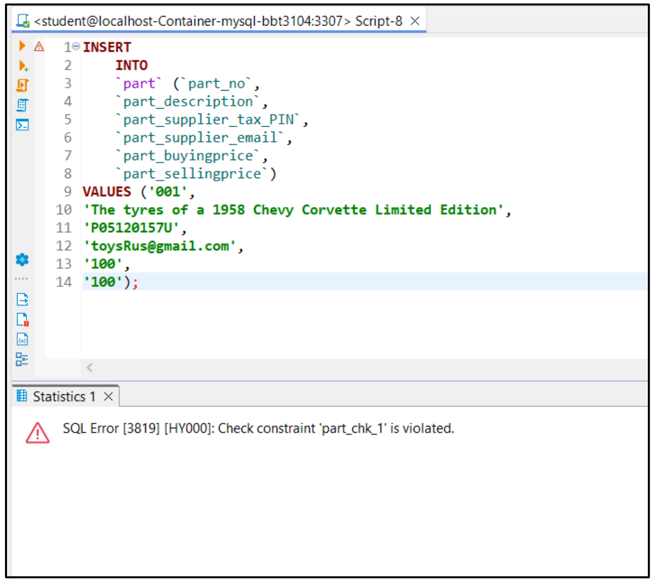
```
SQL Error [3819] [HY000]: Check constraint 'CHK_part_valid_supplier_email' is violated.
```

Execute the following to address the email address format violation:

```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P05120157U',
'toysRus@gmail.com',
'100',
'100');
```

User-Defined Error 3

This time the '`part_chk_1`' static integrity constraint (defined as an unnamed column constraint), which corresponds to the correct KRA PIN format, is violated:



<student@localhost-Container-mysql-bbt3104:3307> Script-8

```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P05120157U',
'toysRus@gmail.com',
'100',
'100');
```

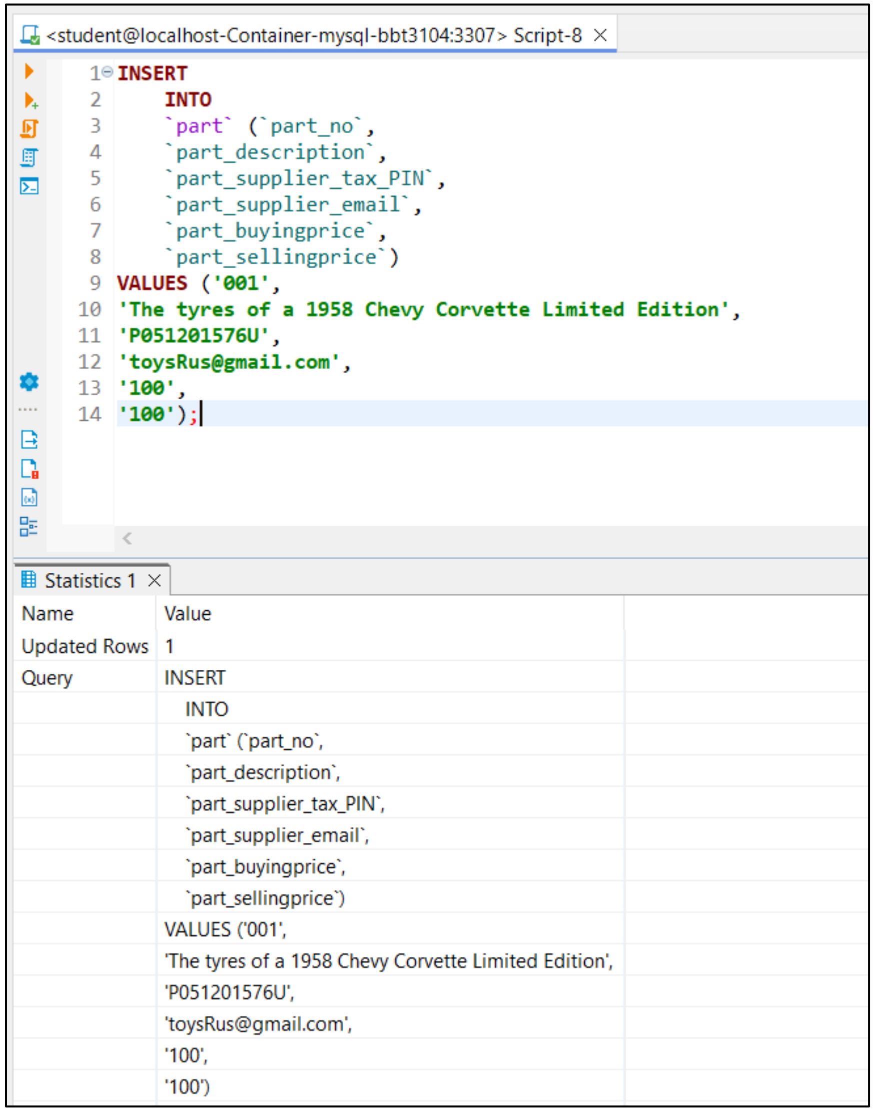
Statistics 1

```
SQL Error [3819] [HY000]: Check constraint 'part_chk_1' is violated.
```

Execute the following to address the KRA PIN format violation, i.e., `'P051201576U'` instead of `'P05120157U'`:

```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P051201576U',
'toysRus@gmail.com',
'100',
'100');
```

This time, the insertion is successful because no violation of a check constraint has occurred:



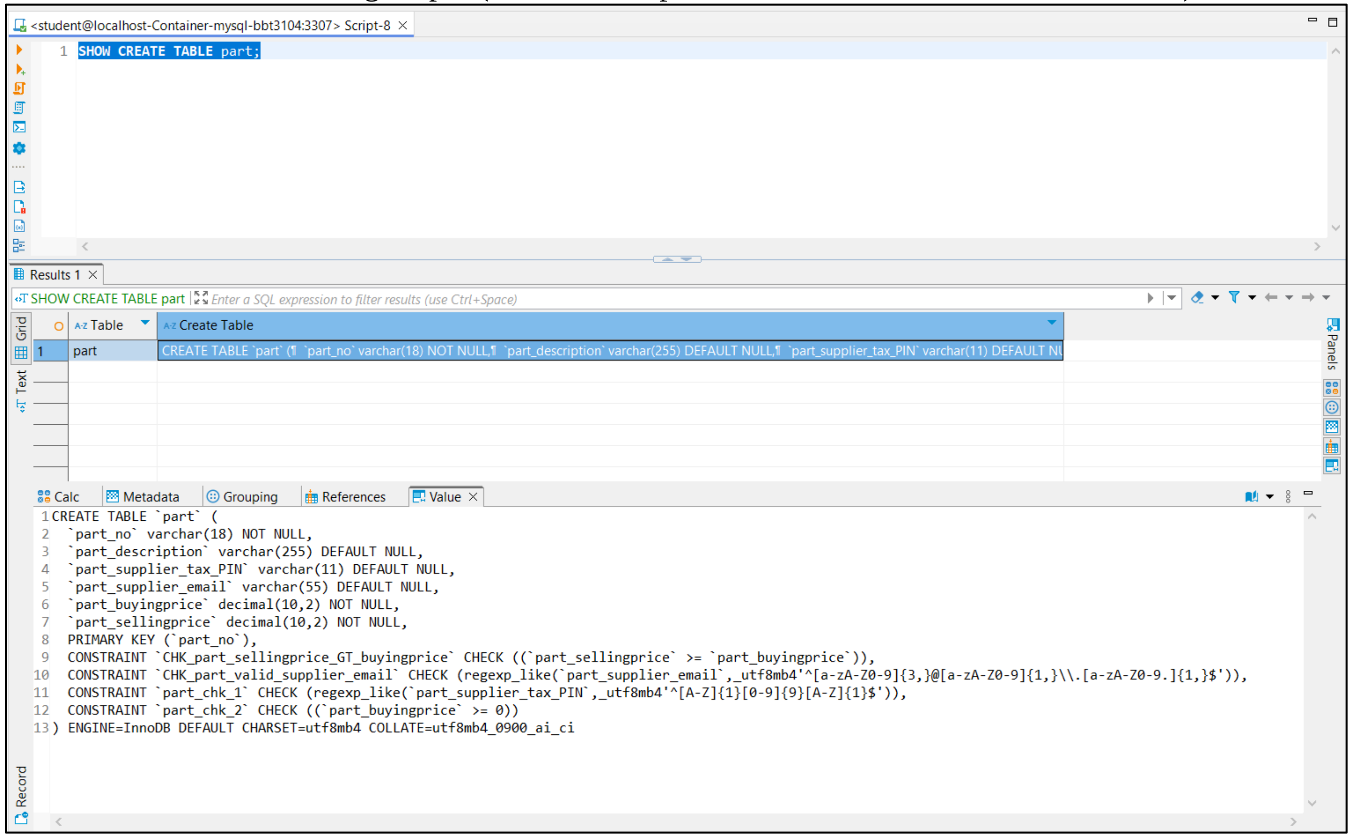
```
INSERT
    INTO
    `part` (`part_no`,
    `part_description`,
    `part_supplier_tax_PIN`,
    `part_supplier_email`,
    `part_buyingprice`,
    `part_sellingprice`)
VALUES ('001',
'The tyres of a 1958 Chevy Corvette Limited Edition',
'P051201576U',
'toysRus@gmail.com',
'100',
'100');
```

Name	Value
Updated Rows	1
Query	INSERT
	INTO
	\`part\` (\`part_no\`,
	\`part_description\`,
	\`part_supplier_tax_PIN\`,
	\`part_supplier_email\`,
	\`part_buyingprice\`,
	\`part_sellingprice\`)
	VALUES ('001',
	'The tyres of a 1958 Chevy Corvette Limited Edition',
	'P051201576U',
	'toysRus@gmail.com',
	'100',
	'100')

There are cases where some DBMSs provide an auto-generated constraint name, e.g. ‘part_chk_3’ To retrieve more information about the exact constraint that has been violated, you need to execute the following command when troubleshooting. This will help you to know what the 3rd CHECK constraint could be in such a case:

```
SHOW CREATE TABLE part;
```

You should see the following output (remember to press F7 to view the full value of the cell):



```
<student@localhost-Container-mysql-bbt3104:3307> Script-8

SHOW CREATE TABLE part;

Results 1
SHOW CREATE TABLE part
Table | Create Table
1 part | CREATE TABLE `part` (¶  `part_no` varchar(18) NOT NULL,¶  `part_description` varchar(255) DEFAULT NULL,¶  `part_supplier_tax_PIN` varchar(11) DEFAULT NI
```

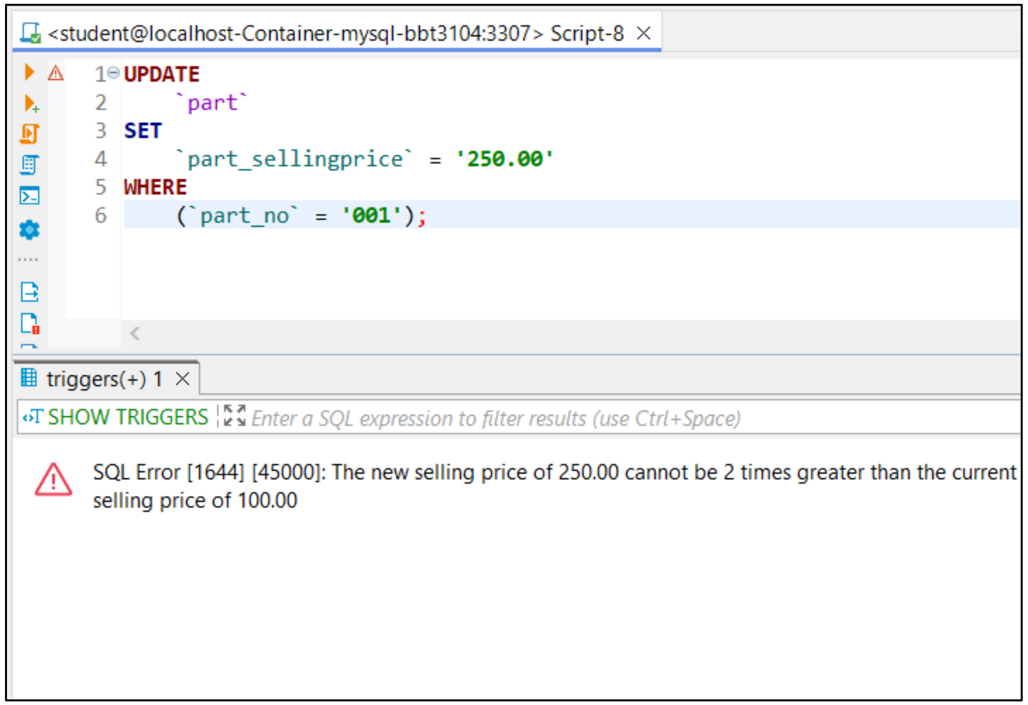
```
CREATE TABLE `part` (
  `part_no` varchar(18) NOT NULL,
  `part_description` varchar(255) DEFAULT NULL,
  `part_supplier_tax_PIN` varchar(11) DEFAULT NULL,
  `part_supplier_email` varchar(55) DEFAULT NULL,
  `part_buyingprice` decimal(10,2) NOT NULL,
  `part_sellingprice` decimal(10,2) NOT NULL,
  PRIMARY KEY (`part_no`),
  CONSTRAINT `CHK_part_sellingprice_GT_buyingprice` CHECK ((`part_sellingprice` >= `part_buyingprice`)),
  CONSTRAINT `CHK_part_valid_supplier_email` CHECK (regexp_like(`part_supplier_email`,_utf8mb4'^[a-zA-Z0-9]{3,}@[a-zA-Z0-9]{1,}\\.[a-zA-Z0-9.]{1,}$')),
  CONSTRAINT `part_chk_1` CHECK (regexp_like(`part_supplier_tax_PIN`,_utf8mb4'^[A-Z]{1}[0-9]{9}[A-Z]{1}$')),
  CONSTRAINT `part_chk_2` CHECK ((`part_buyingprice` >= 0))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Note that the CHECK Constraint syntax also works when the data is being updated. It can, therefore, be used to implement **both a static and a dynamic integrity constraint**.

Execute the following command to update the selling price of part number 001 from 100 to 250. This is an increase of more than double the current price.

```
UPDATE
    `part`
SET
    `part_sellingprice` = '250.00'
WHERE
    (`part_no` = '001');
```

User-Defined Error 4
This time the output displays the customized error message defined in the BEFORE UPDATE trigger named `TRG_BEFORE_UPDATE_ON_part` in STEP 13:


<student@localhost-Container-mysql-bbt3104:3307> Script-8
triggers(+) 1
SHOW TRIGGERS
Enter a SQL expression to filter results (use Ctrl+Space)

```
UPDATE
    `part`
SET
    `part_sellingprice` = '250.00'
WHERE
    (`part_no` = '001');
```

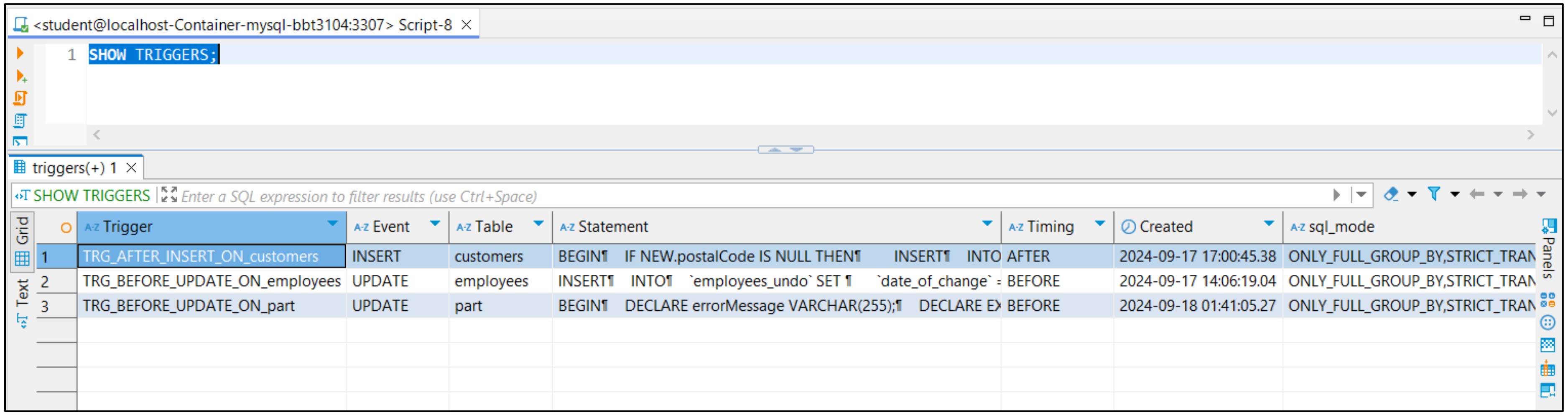
```
SQL Error [1644] [45000]: The new selling price of 250.00 cannot be 2 times greater than the current
selling price of 100.00
```

At this point, the programming team needs to make use of the programming language they are using (e.g., Python, Java, C++, PHP, etc.) to perform additional error handling. For example, the GUI created by the programming team can listen for a specific error number from the MySQL DBMS and respond appropriately. An appropriate response should **tell the user what has happened, why it has happened, and what they need to do as the next step**. Do not simply display the error number for a non-technical user to wonder what it means.

STEP 15. Display all the triggers created so far

Execute the following command to show all the triggers that have been created so far:

```
SHOW TRIGGERS;
```



<student@localhost-Container-mysql-bbt3104:3307> Script-8

```
SHOW TRIGGERS;
```

triggers(+) 1

SHOW TRIGGERS | Enter a SQL expression to filter results (use Ctrl+Space)

	Trigger	Event	Table	Statement	Timing	Created	sql_mode
1	TRG_AFTER_INSERT_ON_customers	INSERT	customers	BEGIN¶ IF NEW.postalCode IS NULL THEN¶ INSERT¶ INTO	AFTER	2024-09-17 17:00:45.38	ONLY_FULL_GROUP_BY,STRICT_TRAN
2	TRG_BEFORE_UPDATE_ON_employees	UPDATE	employees	INSERT¶ INTO¶ `employees_undo` SET ¶ `date_of_change` =	BEFORE	2024-09-17 14:06:19.04	ONLY_FULL_GROUP_BY,STRICT_TRAN
3	TRG_BEFORE_UPDATE_ON_part	UPDATE	part	BEGIN¶ DECLARE errorMessage VARCHAR(255);¶ DECLARE EX	BEFORE	2024-09-18 01:41:05.27	ONLY_FULL_GROUP_BY,STRICT_TRAN

Lab Submission Requirements

In addition to the previous completed steps based on the lab manual:

(i) Create a trigger called "**`TRG_AFTER_UPDATE_ON_customers`**" that is executed after an update on the table customers. This trigger should delete (in the "**`customers_data_reminders`**" table) the previous records associated with the customer whose records are being updated. Reminders to fill in the missing data should then be inserted as part of the actions performed by the same "**`TRG_AFTER_UPDATE_ON_customers`**" trigger.

(ii) Update the check constraint in the table "**`part`**" to implement a static and dynamic integrity constraint that states that "**`part_sellingprice > part_buyingprice`**" (not greater than or equal to).

Submit your answer by committing and pushing your changes to your team's GitHub repository that was created for the lab. The answers (i.e., the code used to create "**`TRG_AFTER_UPDATE_ON_customers`**" and the code used to update table "**`part`**") should be in the repository's "**`lab_submission.sql`**" file.

Further Reading

Elmasri, R., & Navathe, S., B. (2016). Chapter 7: More SQL: Complex Queries, Triggers, Views, and Schema Modification. In *Fundamentals of Database Systems* (7th ed., pp. 207–238). Pearson Education, Inc. https://kaloleni.su.ezproxy.library.strathmore.edu/read/4629/pdf