Error Handling

This lesson introduces error handing in MySQL with the help of various examples.

We'll cover the following

- Syntax

Error Handling

When an error occurs during stored procedure execution, the error is returned to the calling program and the stored procedure terminates. Error handling is necessary in order to avoid abnormal termination of stored procedures.

A MySQL error handler has three main parts: error condition, action and statements. An error handler specifies an error condition and the statements to execute followed by an action to be taken if the error occurs.

The error condition is the reason why the handler is invoked and can be one of the three: a MySQL error code, a SQLSTATE code or a user defined named condition associated with an error code or SQLSTATE value. MySQL has error codes that are unique to the MySQL Server. On the other hand ANSI has also defined error codes which are database independent. These are referred to as SQLSTATE error codes.

The action can either be to **CONTINUE** execution or **EXIT** the block or stored procedure. This action is taken after the statements defined in the handler are executed.

The last part of the handler is the statements to be executed once it is activated. Most of the time the statement is simply setting a variable value to

be checked within the stored procedure. But in some cases multiple lines of code may be written within a **BEGIN END** block.

The **DECLARE HANDLER** statement is used to declare a handler. MySQL also has the option to define a name for an error condition. This facility makes the code readable because instead of using error codes that are difficult to remember, we can use easy to understand names. The **DECLARE CONDITION** statement is used to declare a named error condition by specifying a condition_name for a condition_value which can be a MySQL error code or SQLSTATE value.

Syntax

> DECLARE CONTINUE | EXIT HANDLER FOR
>
> **MySQL error code| SQLSTATE code| condition_name**
>
> **statements;**

> DECLARE **condition_name** CONDITION FOR **condition_value**;

> Connect to the terminal below by clicking in the widget. Once connected, the command line prompt will show up. Enter or copy and paste the command **./DataJek/Lessons/57lesson.sh** and wait for the MySQL prompt to start-up.

```
-- The lesson queries are reproduced below for convenient copy/paste into the terminal.

-- Query 1
DELIMITER **
CREATE PROCEDURE InsertDigitalAssets(
    IN Id INT,
    IN Asset VARCHAR (100),
    IN Type VARCHAR (25))
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
    SELECT 'Duplicate key error occurred' AS Message;
    END;

    INSERT INTO DigitalAssets(URL, AssetType, ActorID) VALUES(Asset, Type, Id);

    SELECT COUNT(*) AS AssetsOfActor
```

```
    FROM DigitalAssets
    WHERE ActorId = Id;
END**

DELIMITER ;

-- Query 2
CALL InsertDigitalAssets(10, 'https://instagram.com/iamsrk','Instagram');
CALL InsertDigitalAssets(10, 'https://instagram.com/iamsrk','Instagram');

-- Query 3
DROP PROCEDURE InsertDigitalAssets;
DELIMITER **
CREATE PROCEDURE InsertDigitalAssets(
    IN Id INT,
    IN Asset VARCHAR (100),
    IN Type VARCHAR (25))
BEGIN
    DECLARE EXIT HANDLER FOR 1062
    BEGIN
    SELECT 'Duplicate key error occurred' AS Message;
    END;

    INSERT INTO DigitalAssets(URL, AssetType, ActorID) VALUES(Asset, Type, Id);

    SELECT COUNT(*) AS AssetsOfActor
    FROM DigitalAssets
    WHERE ActorId = Id;
END**
DELIMITER ;

CALL InsertDigitalAssets(10, 'https://instagram.com/iamsrk','Instagram');

-- Query 4
CREATE PROCEDURE abc()
BEGIN END;

CREATE PROCEDURE abc()
BEGIN END;

-- Query 5
DELIMITER **
CREATE PROCEDURE HandlerScope( )
BEGIN
    BEGIN -- inner block
       DECLARE CONTINUE HANDLER FOR 1048
        SELECT 'Value cannot be NULL' AS Message;
    END;
    INSERT INTO DigitalAssets(URL) VALUES (NULL);
END**
DELIMITER ;

CALL HandlerScope();

-- Query 6
DELIMITER **
CREATE PROCEDURE proc1()
BEGIN
   DECLARE CONTINUE HANDLER FOR 1048
    SELECT 'Value cannot be NULL' AS Message;
   CALL proc2();
END**
DELIMITER ;
```

```
DELIMITER **
CREATE PRoCEDURE proc2()
BEGIN
    INSERT INTO DigitalAssets(URL) VALUES (NULL);
END**
DELIMITER ;

CALL proc1();
```

Terminal

1. We have discussed an example of error handler in the lesson on cursors where an error is raised when the cursor tries to fetch a row that does not exist. To handle this error, the **NOT FOUND** condition needs to be addressed.

   ```
   DECLARE CONTINUE HANDLER FOR NOT FOUND
   SET RowNotFound = 1;
   ```

   This handler executes when the **NOT FOUND** condition is raised. It sets a variable **RowNotFound** to 1 and this variable is checked in every iteration of the loop. The action of the handler is to let the stored procedure **CONTINUE** execution to close the cursor and then exit the stored procedure.

2. If an exception occurs during stored procedure execution, we can choose to rollback the previous operation and then exit the stored procedure as follows:

   ```
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
   BEGIN
       ROLLBACK;
       SELECT 'An error has occurred, operation rolled back and the stor
   ed procedure was terminated';
   END;
   ```

   This handler executes when it encounters **SQLEXCEPTION** which is a SQLSTATE value. It rolls back the last action performed and then creates an error message for the user. The **EXIT** action causes the block being currently executed to terminate. In case the block with the **EXIT** handler

is enclosed within another block then the inner block exits and control goes to the outer block.

3. One of the most common errors when inserting data is the duplicate key error. We can use the MySQL error code 1062 to define a handler for this error as follows:

```
DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SELECT 'Duplicate key error occurred' AS message;
    END;
```

This handler is executed when duplicate key error 1062 is encountered. The statement to execute is enclosed in the **BEGIN END** block. After the statement executes, the action is to **CONTINUE** execution.

4. Now we will see how to use an error handler in a stored procedure. Consider the following stored procedure that inserts a digital asset for an actor in the **DigitalAssets** table:

```
DELIMITER **
CREATE PROCEDURE InsertDigitalAssets(
    IN Id INT,
    IN Asset VARCHAR (100),
    IN Type VARCHAR (25))
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
    SELECT 'Duplicate key error occurred' AS Message;
    END;

    INSERT INTO DigitalAssets(URL, AssetType, ActorID) VALUES(Asset,
Type, Id);

    SELECT COUNT(*) AS AssetsOfActor
    FROM DigitalAssets
    WHERE ActorId = Id;
END**
DELIMITER ;
```

The procedure takes the actor Id, URL and asset type as IN parameters and inserts a record in the **DigitalAssets** table. It then displays the total number of digital assets owned by the actor.

```
CALL InsertDigitalAssets(10, 'https://instagram.com/iamsrk','Instagra
m');
CALL InsertDigitalAssets(10, 'https://instagram.com/iamsrk','Instagra
m');
```

The first call to the procedure is successful. A new digital asset is inserted for the actor taking his asset count to three. When we try to insert the same record again in the second call, the duplicate key handler is executed. The action specified in the handler is **CONTINUE** which means the stored procedure continues execution and the last SELECT statement to display the total digital assets of the actor is executed. If we specify EXIT as action, then after creating the error message the stored procedure will terminate without executing the last SELECT statement as shown below:

```
DROP PROCEDURE InsertDigitalAssets;
DELIMITER **
CREATE PROCEDURE InsertDigitalAssets(
    IN Id INT,
    IN Asset VARCHAR (100),
    IN Type VARCHAR (25))
BEGIN
    DECLARE EXIT HANDLER FOR 1062
    BEGIN
    SELECT 'Duplicate key error occurred' AS Message;
    END;

    INSERT INTO DigitalAssets(URL, AssetType, ActorID) VALUES(Asset,
Type, Id);

    SELECT COUNT(*) AS AssetsOfActor
    FROM DigitalAssets
    WHERE ActorId = Id;
END**
DELIMITER ;

CALL InsertDigitalAssets(10, 'https://instagram.com/iamsrk','Instagra
m');
```

As it can be seen from the output, this time the stored procedure terminated when the handler was executed and control did not reach the last SELECT statement.

5. To create a named condition, let's take the example of MySQL error code 1322 which is issued when a cursor statement is not a SELECT statement. Instead of using 1322 in the handler we can first give it a name that describes the error and then use that name in the handler to make the code readable.

```
DECLARE WrongCursorStatement CONDITION for 1322 ;

DECLARE EXIT HANDLER FOR WrongCursorStatement
    SELECT 'Provide a SELECT statement for the cursor' Message;
```

The **DECLARE CONDITION** statement must appear before the **DECLARE HANDLER** statement.

6. When MySQL reports an error, it provides both the MySQL error code as well as the SQLSTATE value of the error code. For example, attempting to use a procedure name that already exists will result in an error:

```
CREATE PROCEDURE abc()
BEGIN END;

CREATE PROCEDURE abc()
BEGIN END;
```

The MySQL error code is 1304 while the SQLSTATE code 42000 is mentioned within parenthesis. All MySQL error codes have corresponding SQLSTATE values but these values may not be unique. Many times more than one MySQL error codes map to one SQLSTATE value.

7. Consider the following error handler statements which are for the same scenario but defined in terms of MySQL error code, SQLSTATE value and the NOT FOUND condition which is a shorthand for the class of SQLSTATE values:

```
DECLARE CONTINUE HANDLER FOR 1329
SET LastRow =1;

DECLARE CONTINUE HANDLER FOR SQLSTATE ‘02000’
SET LastRow =1;

DECLARE CONTINUE HANDLER FOR NOT FOUND
SET LastRow =1;
```

If all these are written in a stored procedure, they all are eligible to execute when a cursor fails to fetch a row. The handler precedence rules state that the most specific handler will execute. The MySQL error code takes the highest precedence followed by SQLSTATE which is followed by the generic SQLEXCEPTION, SQL WARNING and NOT FOUND handlers. So in our case the first handler will execute.

The benefit of defining multiple handlers for the same condition is that we can handle specific errors in terms of MySQL error codes as well as write generic handlers for conditions that may occur unexpectedly.

8. In the end we will discuss the scope of an error handler. The handler is relevant to the block in which it is defined. Consider the following example where the CONTINUE handler is not invoked because it is not in the same block as the statement which produces the error.

```
DELIMITER **
CREATE PROCEDURE HandlerScope( )
BEGIN
    BEGIN -- inner block
       DECLARE CONTINUE HANDLER FOR 1048
           SELECT 'Value cannot be NULL' AS Message;
    END;

    INSERT INTO DigitalAssets(URL) VALUES (NULL);
END**
DELIMITER ;

CALL HandlerScope();
```

The scope of this handler is the BEGIN END of the inner block. The

INSERT statement is outside the block in which the handler is defined so it is invoked when error condition arises.

The handler scope covers any calls to a stored procedure contained within the same block as the handler declaration. Consider the following code:

```
DELIMITER **
CREATE PROCEDURE proc1()
BEGIN
   DECLARE CONTINUE HANDLER FOR 1048
            SELECT 'Value cannot be NULL' AS Message;
   CALL proc2();
END**
DELIMITER ;


DELIMITER **
CREATE PRoCEDURE proc2()
BEGIN
    INSERT INTO DigitalAssets(URL) VALUES (NULL);
END**
DELIMITER ;


CALL proc1();
```

The handler declared in **proc1** is invoked when the **INSERT** statement in **proc2** is executed. If we define a separate handler to handle error 1048 in proc2, then that handler will take precedence over the one defined in proc1.