POSTGRESQL - TRIGGERS

 Copyright © tutorialspoint.com

PostgreSQL **Triggers** are database callback functions, which are automatically performed/invoked when a specified database event occurs. Following are important points about PostgreSQL triggers:

- PostgreSQL trigger can be specified to fire before the operation is attempted on a row *beforeconstraintsarecheckedandtheINSERT, UPDATEorDELETEisattempted*; or after the operation has completed *afterconstraintsarecheckedandtheINSERT, UPDATE, orDELETEhascompleted*; or instead of the operation *inthecaseofinserts, updatesordeletesonaview*.
- A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.
- Both the WHEN clause and the trigger actions may access elements of the row being inserted, deleted or updated using references of the form **NEW.column-name** and **OLD.column-name**, where column-name is the name of a column from the table that the trigger is associated with.
- If a WHEN clause is supplied, the PostgreSQL statements specified are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the PostgreSQL statements are executed for all rows.
- If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
- The BEFORE, AFTER or INSTEAD OF keyword determines when the trigger actions will be executed relative to the insertion, modification or removal of the associated row.
- Triggers are automatically dropped when the table that they are associated with is dropped.
- The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just **tablename**, not **database.tablename**.
- A CONSTRAINT option when specified creates a *constraint trigger*. This is the same as a regular trigger except that the timing of the trigger firing can be adjusted using SET CONSTRAINTS. Constraint triggers are expected to raise an exception when the constraints they implement are violated..

Syntax:

The basic syntax of creating a **trigger** is as follows:

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER|INSTEAD OF] event_name
ON table_name
[
 -- Trigger logic goes here....
];
```

Here, **event_name** could be *INSERT, DELETE, UPDATE,* and *TRUNCATE* database operation on the mentioned table **table_name**. You can optionally specify FOR EACH ROW after table name.

Following is the syntax of creating a trigger on an UPDATE operation on one or more specified columns of a table as follows:

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name
ON table_name
[
 -- Trigger logic goes here....
];
```

Example

Let us consider a case where we want to keep audit trial for every record being inserted in COMPANY table, which we will create newly as follows *DropCOMPANYtableifyoualreadyhaveit*:

```
testdb=# CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

To keep audit trial, we will create a new table called AUDIT where log messages will be inserted whenever there is an entry in COMPANY table for a new record:

```
testdb=# CREATE TABLE AUDIT(
    EMP_ID INT NOT NULL,
    ENTRY_DATE TEXT NOT NULL
);
```

Here, ID is the AUDIT record ID, and EMP_ID is the ID which will come from COMPANY table and DATE will keep timestamp when the record will be created in COMPANY table. So now let's create a trigger on COMPANY table as follows:

```
testdb=# CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY
FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

Where auditlogfunc is a PostgreSQL **procedure** and has the following definition:

```
CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER AS $example_table$
    BEGIN
        INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, current_timestamp);
        RETURN NEW;
    END;
$example_table$ LANGUAGE plpgsql;
```

Now, we will start actual work, let's start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. So let's create one record in COMPANY table as follows:

```
testdb=# INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

This will create one record in COMPANY table, which is as follows:

```
 id | name | age | address      | salary
----+------+-----+--------------+--------
  1 | Paul |  32 | California   |  20000
```

Same time, one record will be created in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation on COMPANY table. Similar way you can create your triggers on UPDATE and DELETE operations based on your requirements.

```
 emp_id |          entry_date
--------+-------------------------------
      1 | 2013-05-05 15:49:59.968+05:30
(1 row)
```

Listing TRIGGERS

You can list down all the triggers in the current database from **pg_trigger** table as follows:

```
testdb=# SELECT * FROM pg_trigger;
```

Above PostgreSQL statement will list down all triggers.

If you want to list down triggers on a particular table, then use AND clause with table name as follows:

```
testdb=# SELECT tgname FROM pg_trigger, pg_class WHERE tgrelid=pg_class.oid AND
relname='company';
```

Above PostgreSQL statement will also list down only one entry as follows:

```
     tgname
-----------------
 example_trigger
(1 row)
```

Dropping TRIGGERS

Following is the DROP command, which can be used to drop an existing trigger:

```
testdb=# DROP TRIGGER trigger_name;
```

Loading [MathJax]/jax/output/HTML-CSS/jax.js