if the database server machine is a remote machine, you will need to set the PGHOST environment variable to the name of the database server machine. The environment variable PGPORT might also have to be set.

PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

• A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server program is called postgres.

You will need to become the operating system user under which PostgreSQL was installed (usually postgres) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your operating system user name; in that case you need to use the -U switch or set the PGUSER environment variable to specify your PostgreSQL user name.

Use pgadmin4 tool to create a database;

-- DROP DATABASE plascencia;

CREATE DATABASE plascencia

WITH OWNER = plascencia

ENCODING = 'UTF8'

LC\_COLLATE = 'English\_United States.1252'

LC\_CTYPE = 'English\_United States.1252'

TABLESPACE = pg\_default

CONNECTION LIMIT = -1;

COMMENT ON DATABASE plascencia

IS 'plascencia';

C:\>psql --dbname=plascencia --username=plascencia

Password for user plascencia: plascencia

psql (13.0)

WARNING: Console code page (437) differs from Windows code page (1252)

8-bit characters might not work correctly. See psql reference

page "Notes for Windows users" for details.

Type "help" for help.

<https://dsavenko.me/read-write-apex-application-fully-based-on-alien-data/>

[Release v7.0.1 · PostgREST/postgrest · GitHub](https://github.com/PostgREST/postgrest/releases/tag/v7.0.1)

<https://github.com/PostgREST/postgrest/releases/download/v7.0.1/postgrest-v7.0.1-windows-x64.zip>

[PostgREST Documentation — PostgREST 7.0.1 documentation](https://postgrest.org/en/v7.0.0/)

<https://postgrest.org/en/v7.0.0/>

PostgREST is a standalone web server that turns your PostgreSQL database directly into a RESTful API. The structural constraints and permissions in the database determine the API endpoints and operations. Using PostgREST is an alternative to manual CRUD programming.

https://postgrest.org/en/v7.0.0/tutorials/tut0.html

<https://postgrest.org/en/v7.0.0/tutorials/tut1.html#tut1>

<https://postgrest.org/en/v7.0.0/install.html>

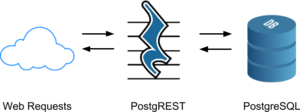
<https://postgrest.org/en/v7.0.0/ecosystem.html#community-tutorials>

# Tutorial 0 - Get it Running

|  |  |
| --- | --- |
| **author:** | [begriffs](https://github.com/begriffs) |

Welcome to PostgREST! In this pre-tutorial we’re going to get things running so you can create your first simple API.

PostgREST is a standalone web server which turns a PostgreSQL database into a RESTful API. It serves an API that is customized based on the structure of the underlying database.



To make an API we’ll simply be building a database. All the endpoints and permissions come from database objects like tables, views, roles, and stored procedures. These tutorials will cover a number of common scenarios and how to model them in the database.

By the end of this tutorial you’ll have a working database, PostgREST server, and a simple single-user todo list API.

## Step 1. Relax, we’ll help

As you begin the tutorial, pop open the project [chat room](https://gitter.im/begriffs/postgrest) in another tab. There are a nice group of people active in the project and we’ll help you out if you get stuck.

## Step 2. Install PostgreSQL

If you’re already familiar with using PostgreSQL and have it installed on your system you can use the existing installation(see [PostgreSQL dependency](https://postgrest.org/en/v7.0.0/install.html#pg-dependency) for minimum requirements). For this tutorial we’ll describe how to use the database in Docker because database configuration is otherwise too complicated for a simple tutorial.

If Docker is not installed, you can get it [here](https://www.docker.com/community-edition#download). Next, let’s pull and start the database image:

sudo docker run --name tutorial -p 5433:5432 **\**

-e POSTGRES\_PASSWORD=mysecretpassword **\**

-d postgres

This will run the Docker instance as a daemon and expose port 5433 to the host system so that it looks like an ordinary PostgreSQL server to the rest of the system.

## Step 3. Install PostgREST

PostgREST is distributed as a single binary, with versions compiled for major distributions of Linux/BSD/Windows. Visit the [latest release](https://github.com/PostgREST/postgrest/releases/latest) for a list of downloads. In the event that your platform is not among those already pre-built, see [Build from Source](https://postgrest.org/en/v7.0.0/development.html#build-source) for instructions how to build it yourself. Also let us know to add your platform in the next release.

The pre-built binaries for download are .tar.xz compressed files (except Windows which is a zip file). To extract the binary, go into the terminal and run

*# download from https://github.com/PostgREST/postgrest/releases/latest*

tar xJf postgrest-<version>-<platform>.tar.xz

The result will be a file named simply postgrest (or postgrest.exe on Windows). At this point try running it with

./postgrest

If everything is working correctly it will print out its version and information about configuration. You can continue to run this binary from where you downloaded it, or copy it to a system directory like /usr/local/bin on Linux so that you will be able to run it from any directory.

**Note**

PostgREST requires libpq, the PostgreSQL C library, to be installed on your system. Without the library you’ll get an error like “error while loading shared libraries: libpq.so.5.” Here’s how to fix it:

Ubuntu or DebianFedora, CentOS, or Red HatOS XWindows

## Step 4. Create Database for API

Connect to the SQL console (psql) inside the container. To do so, run this from your command line:

sudo docker exec -it tutorial psql -U postgres

You should see the psql command prompt:

Or if not using docker

c:\Program Files\PostgreSQL>set PGPASSWORD=epfy44yqe

c:\Program Files\PostgreSQL>psql -U postgres

Password for user postgres:

psql (13.0)

WARNING: Console code page (437) differs from Windows code page (1252)

8-bit characters might not work correctly. See psql reference

page "Notes for Windows users" for details.

Type "help" for help.

psql (9.6.3)

Type "help" **for** help.

postgres=*#*

The first thing we’ll do is create a [named schema](https://www.postgresql.org/docs/current/static/ddl-schemas.html) for the database objects which will be exposed in the API. We can choose any name we like, so how about “api.” Execute this and the other SQL statements inside the psql prompt you started.

**create** **schema** api;

Our API will have one endpoint, /todos, which will come from a table.

**create** **table** api.todos (

id serial **primary** **key**,

done boolean **not** **null** **default** **false**,

task text **not** **null**,

due timestamptz

);

**insert** **into** api.todos (task) **values**

('finish tutorial 0'), ('pat self on back');

Next make a role to use for anonymous web requests. When a request comes in, PostgREST will switch into this role in the database to run queries.

**create** **role** web\_anon nologin;

**grant** usage **on** **schema** api **to** web\_anon;

**grant** **select** **on** api.todos **to** web\_anon;

The web\_anon role has permission to access things in the api schema, and to read rows in the todos table.

It’s a good practice to create a dedicated role for connecting to the database, instead of using the highly privileged postgres role. So we’ll do that, name the role authenticator and also grant him the ability to switch to the web\_anon role :

**create** **role** authenticator noinherit login **password** 'mysecretpassword';

**grant** web\_anon **to** authenticator;

Now quit out of psql; it’s time to start the API!

\q

## Step 5. Run PostgREST

PostgREST uses a configuration file to tell it how to connect to the database. Create a file tutorial.conf with this inside:

db-uri = "postgres://authenticator:mysecretpassword@localhost:5433/postgres"

db-schema = "api"

db-anon-role = "web\_anon"

The configuration file has other [options](https://postgrest.org/en/v7.0.0/configuration.html#configuration), but this is all we need. Now run the server:

./postgrest tutorial.conf

You should see

Listening on port 3000

Attempting to connect to the database...

Connection successful

c:\Program Files\PostgreSQL\13\postgrest>type tutorial.conf

db-uri = "postgres://authenticator:mysecretpassword@127.0.0.1:5432/postgres"

db-schema = "api"

db-anon-role = "web\_anon"

server-port = 3000

c:\Program Files\PostgreSQL\13\postgrest>run\_Web\_server.bat

c:\Program Files\PostgreSQL\13\postgrest>cd "c:\Program Files\PostgreSQL\13\postgrest"

c:\Program Files\PostgreSQL\13\postgrest>"c:\Program Files\PostgreSQL\13\postgrest" tutorial.conf

Listening on port 3000

Attempting to connect to the database...

Connection successful

===========

It’s now ready to serve web requests. There are many nice graphical API exploration tools you can use, but for this tutorial we’ll use curl because it’s likely to be installed on your system already. Open a new terminal (leaving the one open that PostgREST is running inside). Try doing an HTTP request for the todos.

curl http://localhost:3000/todos

The API replies:

[

{

**"id"**: 1,

**"done"**: **false**,

**"task"**: "finish tutorial 0",

**"due"**: **null**

},

{

**"id"**: 2,

**"done"**: **false**,

**"task"**: "pat self on back",

**"due"**: **null**

}

]

With the current role permissions, anonymous requests have read-only access to the todos table. If we try to add a new todo we are not able.

curl http://localhost:3000/todos -X POST **\**

-H "Content-Type: application/json" **\**

-d '{"task": "do bad thing"}'

Response is 401 Unauthorized:

{

**"hint"**: **null**,

**"details"**: **null**,

**"code"**: "42501",

**"message"**: "permission denied for relation todos"

}

There we have it, a basic API on top of the database! In the next tutorials we will see how to extend the example with more sophisticated user access controls, and more tables and queries.

Now that you have PostgREST running, try the next tutorial, [Tutorial 1 - The Golden Key](https://postgrest.org/en/v7.0.0/tutorials/tut1.html#tut1)

# Tutorial 1 - The Golden Key

|  |  |
| --- | --- |
| **author:** | [begriffs](https://github.com/begriffs) |

In [Tutorial 0 - Get it Running](https://postgrest.org/en/v7.0.0/tutorials/tut0.html#tut0) we created a read-only API with a single endpoint to list todos. There are many directions we can go to make this API more interesting, but one good place to start would be allowing some users to change data in addition to reading it.

## Step 1. Add a Trusted User

The previous tutorial created a web\_anon role in the database with which to execute anonymous web requests. Let’s make a role called todo\_user for users who authenticate with the API. This role will have the authority to do anything to the todo list.

*-- run this in psql using the database created*

*-- in the previous tutorial*

**create** **role** todo\_user nologin;

**grant** todo\_user **to** authenticator;

**grant** usage **on** **schema** api **to** todo\_user;

**grant** **all** **on** api.todos **to** todo\_user;

**grant** usage, **select** **on** **sequence** api.todos\_id\_seq **to** todo\_user;

## Step 2. Make a Secret

Clients authenticate with the API using JSON Web Tokens. These are JSON objects which are cryptographically signed using a password known to only us and the server. Because clients do not know the password, they cannot tamper with the contents of their tokens. PostgREST will detect counterfeit tokens and will reject them.

Let’s create a password and provide it to PostgREST. Think of a nice long one, or use a tool to generate it. **Your password must be at least 32 characters long.**

**Note**

Unix tools can generate a nice password for you:

*# Allow "tr" to process non-utf8 byte sequences*

export LC\_CTYPE=C

*# read random bytes and keep only alphanumerics*

< /dev/urandom tr -dc A-Za-z0-9 | head -c32

Open the tutorial.conf (created in the previous tutorial) and add a line with the password:

*# PASSWORD MUST BE AT LEAST 32 CHARS LONG*

*# add this line to tutorial.conf:*

jwt-secret = "<the password you made>"

If the PostgREST server is still running from the previous tutorial, restart it to load the updated configuration file.

## Step 3. Sign a Token

Ordinarily your own code in the database or in another server will create and sign authentication tokens, but for this tutorial we will make one “by hand.” Go to [jwt.io](https://jwt.io/#debugger-io) and fill in the fields like this:



*How to create a token at*[*https://jwt.io*](https://jwt.io/)

*eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyb2xlIjoidG9kb191c2VyIn0.bgDPxJbkWzF0plVRM2DdNLXekzLi3a9Rgqh8ZFPC2b8*

**Remember to fill in the password you generated rather than the word “secret”.** After you have filled in the password and payload, the encoded data on the left will update. Copy the encoded token.

**Note**

While the token may look well obscured, it’s easy to reverse engineer the payload. The token is merely signed, not encrypted, so don’t put things inside that you don’t want a determined client to see.

## Step 4. Make a Request

Back in the terminal, let’s use curl to add a todo. The request will include an HTTP header containing the authentication token.

export TOKEN="<paste token here>"

curl http://localhost:3000/todos -X POST **\**

-H "Authorization: Bearer $TOKEN" **\**

-H "Content-Type: application/json" **\**

-d '{"task": "learn how to auth"}'

And now we have completed all three items in our todo list, so let’s set done to true for them all with a PATCH request.

curl http://localhost:3000/todos -X PATCH **\**

-H "Authorization: Bearer $TOKEN" **\**

-H "Content-Type: application/json" **\**

-d '{"done": true}'

A request for the todos shows three of them, and all completed.

curl http://localhost:3000/todos

[

{

**"id"**: 1,

**"done"**: **true**,

**"task"**: "finish tutorial 0",

**"due"**: **null**

},

{

**"id"**: 2,

**"done"**: **true**,

**"task"**: "pat self on back",

**"due"**: **null**

},

{

**"id"**: 3,

**"done"**: **true**,

**"task"**: "learn how to auth",

**"due"**: **null**

}

]

## Step 4. Add Expiration

Currently our authentication token is valid for all eternity. The server, as long as it continues using the same JWT password, will honor the token.

It’s better policy to include an expiration timestamp for tokens using the exp claim. This is one of two JWT claims that PostgREST treats specially.

| **Claim** | **Interpretation** |
| --- | --- |
| role | The database role under which to execute SQL for API request |
| exp | Expiration timestamp for token, expressed in “Unix epoch time” |

**Note**

Epoch time is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), January 1st 1970, minus the number of leap seconds that have taken place since then.

To observe expiration in action, we’ll add an exp claim of five minutes in the future to our previous token. First find the epoch value of five minutes from now. In psql run this:

**select** **extract**(epoch **from** now() + '5 minutes'::interval) :: integer;

Go back to jwt.io and change the payload to

{

**"role"**: "todo\_user",

**"exp"**: 123456789

}

**NOTE**: Don’t forget to change the dummy epoch value 123456789 in the snippet above to the epoch value returned by the psql command.

Copy the updated token as before, and save it as a new environment variable.

export NEW\_TOKEN="<paste new token>"

Try issuing this request in curl before and after the expiration time:

curl http://localhost:3000/todos **\**

-H "Authorization: Bearer $NEW\_TOKEN"

After expiration, the API returns HTTP 401 Unauthorized:

{**"message"**:"JWT expired"}

## Bonus Topic: Immediate Revocation

Even with token expiration there are times when you may want to immediately revoke access for a specific token. For instance, suppose you learn that a disgruntled employee is up to no good and his token is still valid.

To revoke a specific token we need a way to tell it apart from others. Let’s add a custom email claim that matches the email of the client issued the token.

Go ahead and make a new token with the payload

{

**"role"**: "todo\_user",

**"email"**: "disgruntled@mycompany.com"

}

Save it to an environment variable:

export WAYWARD\_TOKEN="<paste new token>"

PostgREST allows us to specify a stored procedure to run during attempted authentication. The function can do whatever it likes, including raising an exception to terminate the request.

First make a new schema and add the function:

**create** **schema** auth;

**grant** usage **on** **schema** auth **to** web\_anon, todo\_user;

**create** **or** **replace** **function** auth.check\_token() **returns** void

**language** plpgsql

**as** $$

**begin**

**if** current\_setting('request.jwt.claim.email', **true**) =

'disgruntled@mycompany.com' **then**

**raise** insufficient\_privilege

**using** hint = 'Nope, we are on to you';

**end** **if**;

**end**

$$;

Next update tutorial.conf and specify the new function:

*# add this line to tutorial.conf*

pre-request = "auth.check\_token"

Restart PostgREST for the change to take effect. Next try making a request with our original token and then with the revoked one.

*# this request still works*

curl http://localhost:3000/todos -X PATCH **\**

-H "Authorization: Bearer $TOKEN" **\**

-H "Content-Type: application/json" **\**

-d '{"done": true}'

*# this one is rejected*

curl http://localhost:3000/todos -X PATCH **\**

-H "Authorization: Bearer $WAYWARD\_TOKEN" **\**

-H "Content-Type: application/json" **\**

-d '{"task": "AAAHHHH!", "done": false}'

The server responds with 403 Forbidden:

{

**"hint"**: "Nope, we are on to you",

**"details"**: **null**,

**"code"**: "42501",

**"message"**: "insufficient\_privilege"

}

[Next](https://postgrest.org/en/v7.0.0/api.html)[Previous](https://postgrest.org/en/v7.0.0/tutorials/tut0.html)

**DO $$**

**DECLARE**

quantity integer := 30;

**BEGIN**

RAISE NOTICE 'Quantity here is %', quantity; ***-- 30***

quantity := 50;

***-- Create a subblock***

**DECLARE**

quantity integer := 80;

**BEGIN**

RAISE NOTICE 'Quantity here is %', quantity; ***-- 80***

**END**;

RAISE NOTICE 'Quantity here is %', quantity; ***-- 50***

**END**

**$$**;

**DECLARE**

curs1 refcursor;

curs2 **CURSOR FOR SELECT** \* **FROM** tenk1;

curs3 **CURSOR** (**key** integer) **FOR SELECT** \*

**FROM** tenk1

**WHERE** unique1 = **key**;

• Functions can return a result set

• Use SETOF

• Use RETURN NEXT

• RETURN NEXT does not actually return from the function

• Successive RETURN NEXT commands build a result set

• A final RETURN exits the function

• All tables and views automatically have corresponding type

definitions so they can be used as return types

**CREATE FUNCTION** get\_oldest\_session()

**RETURNS** pg\_stat\_activity **AS $$**

**DECLARE**

r record;

**BEGIN**

**SELECT** \*

**INTO** r

**FROM** pg\_stat\_activity

**WHERE** usename = **SESSION\_USER**

**ORDER BY** backend\_start **DESC**

**LIMIT** 1;

**RETURN** r;

**END**

**$$ LANGUAGE** plpgsql;

**CREATE FUNCTION** running\_queries(p\_rows int, p\_len int **DEFAULT** 50)

**RETURNS SETOF** running\_queries **AS**

**$$**

**BEGIN**

**RETURN** QUERY **SELECT** runtime, pid, usename, waiting,

**substring**(query,1,p\_len) **as** query

**FROM** running\_queries

**ORDER BY** 1 **DESC**

**LIMIT** p\_rows;

**END**

**$$ LANGUAGE** plpgsql;

Code Name

22000 data\_exception

22012 division\_by\_zero

2200B escape\_character\_conflict

22007 invalid\_datetime\_format

22023 invalid\_parameter\_value

2200M invalid\_xml\_document

2200S invalid\_xml\_comment

23P01 exclusion\_violation

**CREATE OR REPLACE FUNCTION** get\_connection\_count()

**RETURNS** integer **AS $$**

**DECLARE**

v\_count integer;

**BEGIN**

**SELECT count**(\*)

**INTO** STRICT v\_count

**FROM** pg\_stat\_activity;

**RETURN** v\_count;

**EXCEPTION**

**WHEN** TOO\_MANY\_ROWS **THEN**

RAISE NOTICE 'More than 1 row returned';

**RETURN** -1;

**WHEN** OTHERS **THEN**

RAISE NOTICE 'Unknown Error';

**RETURN** -1;

**END**

**$$ LANGUAGE** plpgsql;

• SQLSTATE Returns the numeric value for the error code.

• SQLERRM Returns the message associated with the error

number.

**DECLARE**

v\_count integer;

err\_num integer;

err\_msg varchar;

**BEGIN**

...

**EXCEPTION**

**WHEN** OTHERS **THEN**

err\_num := **SQLSTATE**;

err\_msg := **SUBSTR**(SQLERRM,1,100);

RAISE NOTICE 'Trapped Error: %', err\_msg;

**RETURN** -1;

**END**

• The details of an error are usually required when handling

• Use GET STACKED DIAGNOSTICS to return the details

GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];

Diagnostic Item

RETURNED\_SQLSTATE

COLUMN\_NAME

CONSTRAINT\_NAME

PG\_DATATYPE\_NAME

MESSAGE\_TEXT

TABLE\_NAME

SCHEMA\_NAME

PG\_EXCEPTION\_DETAIL

PG\_EXCEPTION\_HINT

PG\_EXCEPTION\_CONTEXT

**CREATE TRIGGER name**

{ **BEFORE** | **AFTER** | **INSTEAD OF** }

{ event [ OR ... ] }

**ON table\_name**

[ **FOR** [ **EACH** ] { **ROW** | **STATEMENT** } ]

[ **WHEN** ( condition ) ]

**EXECUTE PROCEDURE** function\_name ( arguments )

• A function with no parameters that returns TRIGGER

**CREATE FUNCTION** trg() **RETURNS trigger AS $$**

**BEGIN**

**RETURN NEW**;

**END**;

**$$ LANGUAGE** plpgsql;

Trigger Events

• Insert

• Update

• Delete

• Truncate

• Before

• The trigger is fired before the change is made to the table

• Trigger can modify NEW values

• Trigger can suppress the change altogether

• After

• The trigger is fired after the change is made to the table

• Trigger sees final result of row

• For Each Row

• The trigger is fired once each time a row is affected

• For Each Statement

• The trigger is fired once each time a statement is executed

• A firing trigger adds overhead to the calling transaction

• The percentage overhead can be found with a simple pgbench

test:

**INSERT INTO** trigger\_test (**value**) **VALUES** (‘hello’);

\**set** keys :scale

\setrandom **key** 1 :keys

**UPDATE** trigger\_test **SET value** = 'HELLO' **WHERE key** = :key;

pgbench -n -t 100000

-f INSERTS.pgbench postgres

pgbench -n -s 100000 -t 10000

-f UPDATES.pgbench postgres

Inserts: 4510 tps

Updates: 4349 tps

**CREATE FUNCTION** empty\_trigger()

**RETURNS trigger AS $$**

**BEGIN**

**RETURN NEW**;

**END**;

**$$ LANGUAGE** plpgsql;

**CREATE TRIGGER** empty\_trigger

**BEFORE INSERT** OR **UPDATE ON** trigger\_test

**FOR EACH ROW EXECUTE PROCEDURE** empty\_trigger();

• NEW

• Variable holding the new row for INSERT/UPDATE operations in row-level triggers

• OLD

• Variable holding the old row for UPDATE/DELETE operations in row-level triggers

**CREATE TABLE** audit (

event\_time **timestamp NOT NULL**,

user\_name varchar **NOT NULL**,

old\_row json,

new\_row json

);

**CREATE OR REPLACE FUNCTION** audit\_trigger()

**RETURNS TRIGGER AS $$**

**BEGIN**

**INSERT INTO** audit

**VALUES** (**CURRENT\_TIMESTAMP**,

**CURRENT\_USER**,

row\_to\_json(**OLD**),

row\_to\_json(**NEW**));

**RETURN NEW**;

**END**;

**$$**

**LANGUAGE** plpgsql;

• TG\_OP

• A string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired

• TG\_NAME

• Variable that contains the name of the trigger actually fired

• TG\_WHEN

• A string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger’s definition

• TG\_LEVEL

• A string of either ROW or STATEMENT depending on the trigger’s Definition

**CREATE TABLE** audit (

event\_time **timestamp NOT NULL**,

user\_name varchar **NOT NULL**,

**operation** varchar **NOT NULL**,

old\_row json,

new\_row json

);

**CREATE OR REPLACE FUNCTION** audit\_trigger() **RETURNS TRIGGER AS $$**

**BEGIN**

IF (TG\_OP = 'DELETE') **THEN**

**INSERT INTO** audit **VALUES**

(**CURRENT\_TIMESTAMP**, **CURRENT\_USER**,TG\_OP, row\_to\_json(**OLD**), **null**);

**RETURN OLD**;

ELSIF (TG\_OP = 'UPDATE') **THEN**

**INSERT INTO** audit **VALUES**

(**CURRENT\_TIMESTAMP**, **CURRENT\_USER**,TG\_OP,

row\_to\_json(**OLD**), row\_to\_json(**NEW**));

**RETURN NEW**;

ELSIF (TG\_OP = 'INSERT') **THEN**

**INSERT INTO** audit **VALUES**

(**CURRENT\_TIMESTAMP**, **CURRENT\_USER**,TG\_OP, **null**, row\_to\_json(**NEW**));

**RETURN NEW**;

**END** IF;

**RETURN NULL**;

**END**;

**$$ LANGUAGE** plpgsql;

• TG\_TABLE\_NAME

• The name of the table that caused the trigger invocation.

• TG\_RELNAME

• The name of the table that caused the trigger invocation

• TG\_RELID

• The object ID of the table that caused the trigger invocation

• TG\_TABLE\_SCHEMA

• The name of the schema of the table that caused the trigger Invocation

**CREATE TABLE** audit (

event\_time **timestamp NOT NULL**,

user\_name varchar **NOT NULL**,

**operation** varchar **NOT NULL**,

**table\_name** varchar **NOT NULL**,

old\_row json,

new\_row json

);

• TG\_NARGS

• The number of arguments given to the trigger procedure in the CREATE TRIGGER statement

• TG\_ARGV[]

• The arguments from the CREATE TRIGGER statement

• Create child tables for each partition

**CREATE TABLE** audit\_2014 (

**CHECK** ( event\_time >= **DATE** '2014-01-01'

AND event\_time < **DATE** '2015-01-01')

) **INHERITS** (audit);

**CREATE TABLE** audit\_2015 (

**CHECK** ( event\_time >= **DATE** '2015-01-01'

AND event\_time < **DATE** '2016-01-01')

) **INHERITS** (audit);

• The trigger function will move the row to the correct child table

**CREATE OR REPLACE FUNCTION** partition\_audit\_trigger()

**RETURNS TRIGGER AS $$**

**BEGIN**

**EXECUTE** 'INSERT INTO audit\_' ||

**to\_char**(NEW.event\_time, 'YYYY') ||

' VALUES ($1, $2, $3, $4, $5, $6)'

**USING** NEW.event\_time, NEW.user\_name, NEW.operation,

NEW.table\_name, NEW.old\_row, NEW.new\_row;

**RETURN NULL**;

**END**;

**$$**

**LANGUAGE** plpgsql;

• A trigger needs to be added to the parent table

**CREATE TRIGGER** partition\_audit\_trigger

**BEFORE INSERT ON** audit

**FOR EACH ROW**

**EXECUTE PROCEDURE**

partition\_audit\_trigger();

• Performance is much better if dynamic SQL is not used

**CREATE OR REPLACE FUNCTION** partition\_audit\_trigger()

**RETURNS TRIGGER AS $$**

**BEGIN**

IF ( NEW.event\_time >= **DATE** '2015-01-01' AND

NEW.event\_time < **DATE** '2016-01-01' ) **THEN**

**INSERT INTO** audit\_2015 **VALUES** (NEW.\*);

ELSIF ( NEW.event\_time >= **DATE** '2014-01-01' AND

NEW.event\_time < **DATE** '2015-01-01' ) **THEN**

**INSERT INTO** audit\_2014 **VALUES** (NEW.\*);

**ELSE**

RAISE **EXCEPTION** 'Date out of range. Fix

partition\_audit\_trigger() function!';

**END** IF;

**RETURN NULL**;

**END**;

**$$ LANGUAGE** plpgsql;

• If the column used for the partition key changes, the row may

need to be moved to a different partition

**CREATE TRIGGER** move\_partition\_audit\_trigger

**BEFORE UPDATE**

**ON** audit\_2014

**FOR EACH ROW EXECUTE PROCEDURE**

move\_partition\_audit\_trigger('2014-01-01', '2015-01-01');

**CREATE TRIGGER** move\_partition\_audit\_trigger

**BEFORE UPDATE**

**ON** audit\_2015

**FOR EACH ROW EXECUTE PROCEDURE**

move\_partition\_audit\_trigger('2015-01-01', '2016-01-01');

**CREATE FUNCTION** move\_partition\_audit\_trigger() **RETURNS TRIGGER AS $$**

**DECLARE**

start\_date **DATE**;

end\_date **DATE**;

**BEGIN**

start\_date := TG\_ARGV[0];

end\_date := TG\_ARGV[1];

IF ( NEW.event\_time **IS DISTINCT FROM** OLD.event\_time ) **THEN**

IF (NEW.event\_time < start\_date OR NEW.event\_time >= end\_date) **THEN**

**EXECUTE** 'DELETE FROM ' || TG\_TABLE\_SCHEMA || '.' || TG\_TABLE\_NAME ||

' WHERE ctid = $1' **USING** OLD.ctid;

**INSERT INTO** audit **VALUES** (NEW.\*);

**RETURN null**;

**END** IF;

**END** IF;

**RETURN NEW**;

**END**;

**$$ LANGUAGE** plpgsql;

**CREATE TRIGGER** move\_partition\_audit\_trigger

**BEFORE UPDATE**

**ON** audit\_2014

**FOR EACH ROW**

**WHEN** (NEW.event\_time **IS DISTINCT FROM** OLD.event\_time)

**EXECUTE PROCEDURE**

move\_partition\_audit\_trigger('2014-01-01', '2015-01-01');

**CREATE TRIGGER** move\_partition\_audit\_trigger

**BEFORE UPDATE**

**ON** audit\_2015

**FOR EACH ROW**

**WHEN** (NEW.event\_time **IS DISTINCT FROM** OLD.event\_time)

**EXECUTE PROCEDURE**

move\_partition\_audit\_trigger('2015-01-01', '2016-01-01');

• Calculate columns

• Calculate complex values

• Extract values from complex structures

• Enforce derived values when using denormalization

• Used to:

• Increase performance

• Simplify queries

$ head -n 5 zips.json

{ ”\_id” : ”01001”, ”city” : ”AGAWAM”,

”loc” : [ -72.622739, 42.070206 ], ”pop” : 15338, ”state” : ”MA” }

{ ”\_id” : ”01002”, ”city” : ”CUSHMAN”,

”loc” : [ -72.51564999999999, 42.377017 ], ”pop” : 36963, ”state” : ”MA” }

{ ”\_id” : ”01005”, ”city” : ”BARRE”,

”loc” : [ -72.10835400000001, 42.409698 ], ”pop” : 4546, ”state” : ”MA” }

{ ”\_id” : ”01007”, ”city” : ”BELCHERTOWN”,

”loc” : [ -72.41095300000001, 42.275103 ], ”pop” : 10579, ”state” : ”MA” }

{ ”\_id” : ”01008”, ”city” : ”BLANDFORD”,

”loc” : [ -72.936114, 42.182949 ], ”pop” : 1240, ”state” : ”MA” }

**CREATE TABLE** zips (

zip\_code varchar **PRIMARY KEY**,

**state** varchar,

**data** json

);

**CREATE OR REPLACE FUNCTION** extract\_data\_trigger()

**RETURNS TRIGGER AS $$**

**BEGIN**

NEW.zip\_code := NEW.data->>'\_id';

NEW.state := NEW.data->>'state';

**RETURN NEW**;

**END**;

**$$ LANGUAGE** plpgsql;

**CREATE TRIGGER** extract\_data\_trigger

**BEFORE UPDATE** OR **INSERT ON** zips

**FOR EACH ROW EXECUTE PROCEDURE** extract\_data\_trigger();

Trigger Use Cases

• Cache invalidation

• Remove stale entries from a cache

• The database tracks all data so is the single source of truth

• Used to:

• Simplify cache management

• Remove application complexity

Note: Foreign Data Wrappers simplify this process significantly

Note: ON (action) CASCADE contraints can simplify this too.

Event Triggers

• Event triggers fire for DML commands (CREATE, ALTER, DROP,

etc)

• They are not tied to a single table

• They are global to a database

**CREATE OR REPLACE FUNCTION** notice\_ddl()

**RETURNS** event\_trigger **AS**

**$$**

**BEGIN**

RAISE NOTICE 'DDL Fired: % %', tg\_event, tg\_tag;

**END**;

**$$ LANGUAGE** plpgsql;

**CREATE** EVENT **TRIGGER** notice\_ddl

**ON** ddl\_command\_start

**EXECUTE FUNCTION** notice\_ddl();

• ddl\_command\_start

• ddl\_command\_end

• table\_rewrite

• sql\_drop

ddl\_command\_start

• Fired just before the command starts

• This is before any information is known about the command

• Fires for all event trigger command tags

• Does not fire for shared objects such as databases and roles

• Does not fire for commands involving event triggers

ddl\_command\_end

• Fired after the command ends

• Fires for all event trigger command tags

• The objects have been affected so the details of the command

can be obtained

sql\_drop

• Fired just before ddl\_command\_end fires

• The objects have already been removed so they are not accessible

• Only fired for commands that drop an object

• The objects have been affected so the details of the command

can be obtained

table\_rewrite

• Fired just before the table is rewritten by the command

• Only fired for commands that rewrites an object

• CLUSTER and VACUUM FULL do not fire the event

Event Tags

• ALTER POLICY

• ALTER SCHEMA

• ALTER SEQUENCE

• ALTER TABLE

• CREATE EXTENSION

• CREATE FUNCTION

• CREATE INDEX

• CREATE SEQUENCE

• CREATE TABLE

• CREATE TABLE AS

• CREATE VIEW

• DROP INDEX

• DROP TABLE

• DROP VIEW

• GRANT

• REVOKE

The full list is available in the documentation

<https://www.postgresql.org/docs/current/event-trigger-matrix.html>

Event Trigger Functions

A set of functions to help retrieve information from event triggers

• pg\_event\_trigger\_ddl\_commands

• pg\_event\_trigger\_dropped\_objects

• pg\_event\_trigger\_table\_rewrite\_oid

• pg\_event\_trigger\_table\_rewrite\_reason

Understanding pg\_event\_trigger\_ddl\_commands

• Returns a line for each DDL command executed

• Only valid inside a ddl\_command\_end trigger

Column Type

classid oid

objid oid

objsubid integer

command\_tag text

object\_type text

schema\_name text

object\_identity text

in\_extension bool

command pg\_ddl\_command

Understanding pg\_event\_trigger\_dropped\_objects

• Returns a line for each object dropped by the DDL executed

• Only valid inside a sql\_drop trigger

Column Type

classid oid

objid oid

objsubid integer

original bool

normal bool

is\_temporary bool

Column Type

object\_type text

schema\_name text

object\_name text

object\_identity text

address\_names text[]

address\_args text[]

Understanding rewrite functions

• Only valid inside a table\_rewrite trigger

• pg\_event\_trigger\_table\_rewrite\_oid

• Returns the OID of the table about to be rewritten

• pg\_event\_trigger\_table\_rewrite\_reason

• Returns the reason code of why the table was rewritten

Using Event Trigger Functions

**CREATE OR REPLACE FUNCTION** stop\_drops()

**RETURNS** event\_trigger **AS**

**$$**

**DECLARE**

l\_tables varchar[] := '{sales, inventory}';

**BEGIN**

IF **EXISTS**(**SELECT** 1

**FROM** pg\_event\_trigger\_dropped\_objects()

**WHERE** object\_name = **ANY** (l\_tables)) **THEN**

RAISE **EXCEPTION** 'Drops of critical tables are not permitted';

**END** IF;

**END**;

**$$ LANGUAGE** plpgsql;

**CREATE** EVENT **TRIGGER** stop\_drops

**ON** sql\_drop

**EXECUTE FUNCTION** stop\_drops();

Things to Remember

• Triggers are part of the parent transaction

• The trigger fails, the main transaction fails

• The main transaction rolls back, the trigger call never happened

• If the trigger takes a long time, the whole transaction timing is

affected

• Triggers can be difficult to debug

• Especially cascaded triggers