

How to Use the JSON Data Type with PostgreSQL on Ubuntu 20.04

Learn how to effectively use the JSON data type in PostgreSQL on Ubuntu 20.04, including setup, querying, indexing, and best practices for JSON data storage.

Contents

01	Introduction	3
02	Prerequisites	3
03	1. Set Up the PostgreSQL Database	3
04	2. Populate the PostgreSQL JSON Columns	5
05	3. Query the PostgreSQL JSON Columns	7
06	Conclusion	11

Introduction

JavaScript Object Notation (JSON) is a modern data exchange format often used in API-based services. JSON relies on key-value pairs that make it suitable for humans and machines to read and write. The PostgreSQL database server supports the JSON data type to store semi-structured data.

Depending on the complexity of your application, you can choose from dozens of PostgreSQL inbuilt functions and operators to manipulate JSON data.

This guide takes you through implementing the JSON data type with the PostgreSQL database on Ubuntu 20.04 server.

Prerequisites

To proceed with this guide:

- [Deploy an Ubuntu 20.04 server.](#)
- [Create a non-root `sudo` user.](#)
- [Install the PostgreSQL database server and configure a super-user password.](#)

1. Set Up the PostgreSQL Database

The first step in this guide is setting up a database. Then, create a sample table that implements the JSON data type in a few columns. Execute the steps below to initialize the database:

1. Log in to the PostgreSQL server as a `postgres` user.

```
$ sudo -u postgres psql
```

2. Enter the `postgres` user password and press Enter to proceed. Then, create a sample `online_shop` database.

```
postgres=# CREATE DATABASE online_shop;
```

Output.

```
CREATE DATABASE
```

3. Connect to the new `online_shop` database.

```
postgres=# connect online_shop;
```

Output.

```
You are now connected to database "online_shop" as user "postgres".
```

4. Create a new `customers` table with five columns. Assign a unique identifier to the customers using the `customer_id` `PRIMARY KEY`. Use the `SERIAL` keyword to instruct PostgreSQL to automatically assign a new `customer_id` for each customer during the `INSERT` statement. Define the `profile` and `address` columns using the JSON data type. This guide later shows you how to use the two columns to store the customers' profiles and addresses using the JSON format.

```
online_shop=# CREATE TABLE customers (  
    customer_id SERIAL PRIMARY KEY,  
    first_name VARCHAR (50),  
    last_name VARCHAR (50),  
    profile JSON,  
    address JSON  
);
```

Output.

```
CREATE TABLE
```

Your sample database and table are now in place. Proceed to the next step and populate the table.

2. Populate the PostgreSQL JSON Columns

When inserting data to a PostgreSQL JSON column, you must:

- Enclose the JSON keys in double quotes (`"`).
- Separate the **key** from the **value** using a colon (`:`).
- Separate key-value pairs with a comma (`,`).
- Use the supported JSON data types: strings, numbers, JSON objects, booleans, and null.

The following sample illustrates the above JSON rules.

```
{
  "first_name": "JOHN",
  "last_name": "DOE",
  "gender": "M",
  "dob": 1991,
  "account_balance": 4480.90,
  "is_active" : true,
  "phone_number": null
}
```

To fully understand the PostgreSQL JSON data type, follow the steps below to insert three sample records into the `customers` table. In the following SQL statements, you're inserting the customers `gender`, date of birth (`dob`), and their remaining `account_credit` under the JSON `profile` column. Then, you insert the customers' addresses such as `address_line_1`, `address_line_2`, `town`, `state`, and `zip` under the JSON address column.

```
online_shop=# INSERT INTO customers (
              first_name,
              last_name,
              profile,
```

```
        address
    ) VALUES (
        'JOHN',
        'DOE',
        '{
            "gender": "M",
            "dob": 1991,
            "account_credit": 4480.90
        }',
        '{
            "address_line_1": "1010",
            "address_line_2": "485",
            "town": "NEY YORK",
            "state": "NJ",
            "zip": "2020"
        }'
    );

INSERT INTO customers (
    first_name,
    last_name,
    profile,
    address
) VALUES (
    'MARY',
    'ROE',
    '{
        "gender": "F",
        "dob": 1983,
        "account_credit": 9750.26}',
    '{
        "address_line_1": "1515",
        "address_line_2": "979",
        "town": "MIAMI",
        "state": "FL",
        "zip": "2020"
    }'
);

INSERT INTO customers (
    first_name,
    last_name,
    profile,
    address
) VALUES (
```

```
'STEVE',
'JAMES',
'{'
  "gender": "M",
  "dob": 1963,
  "account_credit": 1340.75}',
'{'
  "address_line_1": "777",
  "address_line_3": "495",
  "town": "LAS VEGAS",
  "state": "NV",
  "zip": "7319"
}'
);
```

Output.

```
..
INSERT 0 1
```

The sample JSON data is now in place. Proceed next to work with some PostgreSQL JSON functions.

3. Query the PostgreSQL JSON Columns

PostgreSQL supports different functions and operators that you can use to query the data. Use these functions to filter records, generate reports, and more. Run the following `SELECT` statements to understand how these functions and operators work:

1. Use the `->>` operator to get a specific JSON object field. For instance, run the SQL statements below to retrieve the customers' gender from the `profile` column.

```
online_shop=# SELECT
               customer_id,
               first_name,
               last_name,
```

```
profile ->> 'gender' AS gender
FROM customers;
```

Output.

```
customer_id | first_name | last_name | gender
-----+-----+-----+-----
          1 | JOHN      | DOE       | M
          2 | MARY      | ROE       | F
          3 | STEVE     | JAMES     | M
(3 rows)
```

2. Use the `->>` operator and the SQL `WHERE` clause to filter records:

- Retrieve a list of all women from the `customers` table.

```
online_shop=# SELECT
               customer_id,
               first_name,
               last_name,
               profile ->> 'gender' AS gender
FROM customers
WHERE profile ->> 'gender' = 'F';
```

Output.

```
customer_id | first_name | last_name | gender
-----+-----+-----+-----
          2 | MARY      | ROE       | F
(1 row)
```

- List all males (`M`) from the `customers` table.

```
online_shop=# SELECT
               customer_id,
               first_name,
               last_name,
               profile ->> 'gender' AS gender
FROM customers
WHERE profile ->> 'gender' = 'M';
```

Output.

```
customer_id | first_name | last_name | gender
-----+-----+-----+-----
          1 | JOHN      | DOE       | M
          3 | STEVE     | JAMES     | M
(2 rows)
```

3. Use the PostgreSQL math functions to compute results from JSON columns. For instance, run the SQL command below to get the total customers' account balance from the `account_credits` key under the `profile` column.

```
online_shop=# SELECT
              SUM(CAST(profile ->> 'account_credit' AS FLOAT)) as
total_customers_credit
              FROM customers;
```

Output.

```
total_customers_credit
-----
          15571.91
(1 row)
```

4. Retrieve the age of customers by computing the difference between the customers' date of birth (`dob`) and this year (`date_part('year', NOW())`).

```
online_shop=# SELECT
              customer_id,
              first_name,
              last_name,
              profile ->> 'dob' as dob,
              date_part('year', NOW()) as current_year,
              (date_part('year', NOW()) - CAST(profile ->> 'dob' AS
INTEGER)) AS age
              FROM customers;
```

Output.

```
customer_id | first_name | last_name | dob | current_year | age
-----+-----+-----+---+-----+--
          1 | JOHN      | DOE       | 1991 |          2022 | 31
          2 | MARY      | ROE       | 1983 |          2022 | 39
          3 | STEVE     | JAMES     | 1963 |          2022 | 59
(3 rows)
```

5. Run the `json_typeof` function to get the data type of a JSON column.

```
online_shop=# SELECT
              customer_id,
              json_typeof(profile)
            FROM customers;
```

Output.

```
customer_id | json_typeof
-----+-----
          1 | object
          2 | object
          3 | object
(3 rows)
```

6. Use the `json_each_text` function to expand a JSON column into a set of key-value pairs.

```
online_shop=# SELECT
              customer_id,
              json_each_text (profile)
            FROM customers;
```

Output.

```
customer_id | json_each_text
-----+-----
          1 | (gender,M)
          1 | (dob,1991)
          1 | (account_credit,4480.90)
          2 | (gender,F)
          2 | (dob,1983)
```

```
2 | (account_credit,9750.26)
3 | (gender,M)
3 | (dob,1963)
3 | (account_credit,1340.75)
(9 rows)
```

7. Run the `json_object_keys` function to return all keys from a JSON column.

```
online_shop=# SELECT
               json_object_keys (profile)
FROM customers;
```

Output.

```
 json_object_keys
-----
gender
dob
account_credit
gender
dob
account_credit
gender
dob
account_credit
(9 rows)
```

Conclusion

This guide implements the JSON data type with the PostgreSQL database server on Ubuntu 20.04 server. Use the above JSON syntax and examples when working on your next JSON project.

For a complete list of PostgreSQL JSON functions and operators, visit the official link below:

- [JSON Functions and Operators.](#)



VULTR

