# **PostgreSQL 17 Authentication: How**pg\_hba.conf**Controls Access Like a Firewall**

Press enter or click to view image in full size



PostgreSQL is one of the most powerful open-source relational databases, but with great power comes great responsibility — especially when it comes to security. 🔒

In PostgreSQL 17, ****internal authentication**** is managed primarily through one file:  
👉 pg\_hba.conf (PostgreSQL Host-Based Authentication)

In this guide, we’ll explore:

* ✅ What pg\_hba.conf does
* ✅ How PostgreSQL authentication works
* ✅ Types of authentication: trust, md5, and scram-sha-256
* ✅ Best practices for securing PostgreSQL access

Let’s dive in 👇

## **📂 What is**pg\_hba.conf**?**

You can think of pg\_hba.conf as ****PostgreSQL’s built-in firewall and access control list (ACL)****.

Whenever any client tries to connect to your PostgreSQL database, PostgreSQL reads this file to answer four questions:

1️⃣ ****Who is trying to connect?**** → (User account)  
2️⃣ ****Which database are they trying to access?**** → (Database name)  
3️⃣ ****Where are they connecting from?**** → (Host or IP address)  
4️⃣ ****How should they authenticate?**** → (Authentication method)

If the connection request matches a rule in pg\_hba.conf, PostgreSQL proceeds to authenticate the user accordingly. If no matching rule exists, the connection is refused.

## **🔍 Where is**pg\_hba.conf**Located in PostgreSQL?**

When you’re managing a PostgreSQL server, one of the most critical files you’ll encounter is pg\_hba.conf — the Host-Based Authentication configuration file that controls who can connect, from where, and how.

But where exactly is this file located? Let’s break it down for you:

## **📂 Default Location of**pg\_hba.conf

By default, PostgreSQL stores the pg\_hba.conf file inside the server’s ****data directory****. This data directory is where PostgreSQL keeps all of its internal files, configurations, and database data files.

On most Linux-based installations (such as RHEL, CentOS, or Rocky Linux), if you’re using PostgreSQL 17installed from the official PostgreSQL repository, the typical path will be:

/var/lib/pgsql/17/data/pg\_hba.conf

✅ Here:

* /var/lib/pgsql/17/data/ is your data directory.
* pg\_hba.conf is located right inside that folder.

## **🧑‍💻 How to View the File**

To view or edit the file, you typically need to switch to the postgres user (the dedicated PostgreSQL service account) and use any text viewer or editor:

sudo su - postgres  
cat /var/lib/pgsql/17/data/pg\_hba.conf

Or to edit:

sudo vi /var/lib/pgsql/17/data/pg\_hba.conf

## **🔎 Verify the Exact Path on Your Server**

Since different Linux distributions and custom PostgreSQL installations might store data in different directories, you can safely retrieve the actual path PostgreSQL is using by running:

SHOW hba\_file;

Example output:

/var/lib/pgsql/17/data/pg\_hba.conf

✅ This ensures you’re always editing the correct file.

## **🔍 Checking Active Authentication Methods**

If you want to quickly see which authentication methods are currently configured, you can filter the file using grep:

cat /var/lib/pgsql/17/data/pg\_hba.conf | grep -i method

Or more simply:

grep -i method /var/lib/pgsql/17/data/pg\_hba.conf

This will return lines that mention the word “method”, helping you focus directly on the authentication configurations (like trust, md5, scram-sha-256, etc.).

## **🛑 Don’t Forget: Reload Configuration After Changes**

After editing pg\_hba.conf, PostgreSQL won't automatically pick up your changes until you reload the configuration:

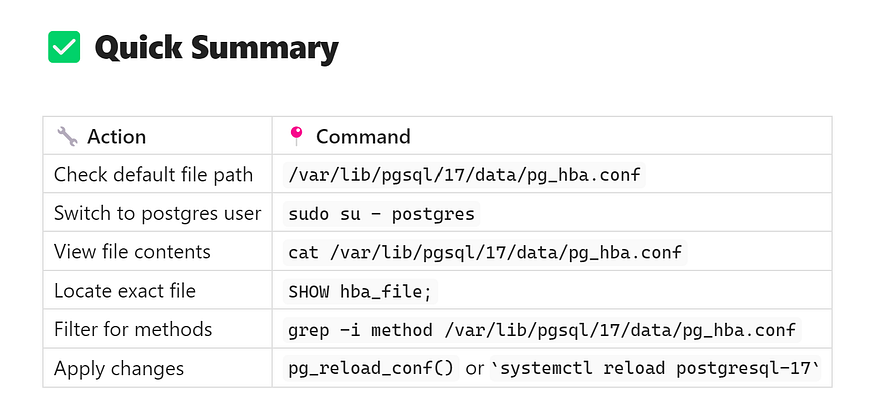
SELECT pg\_reload\_conf();

Or from shell:

sudo systemctl reload postgresql-17

🔄 This safely applies your changes without requiring a full server restart.

Press enter or click to view image in full size



*🚀******Pro Tip******: Always double-check your file path using SHOW hba\_file; — especially on custom installations, Docker containers, or managed cloud PostgreSQL services.*

## **📊 The Structure of**pg\_hba.conf**in PostgreSQL: How Access Rules Work**

In PostgreSQL, your first line of defense for controlling who can access your database is a single file: pg\_hba.conf. But to use it effectively, you need to fully understand its structure and how each column defines the rules for database access.

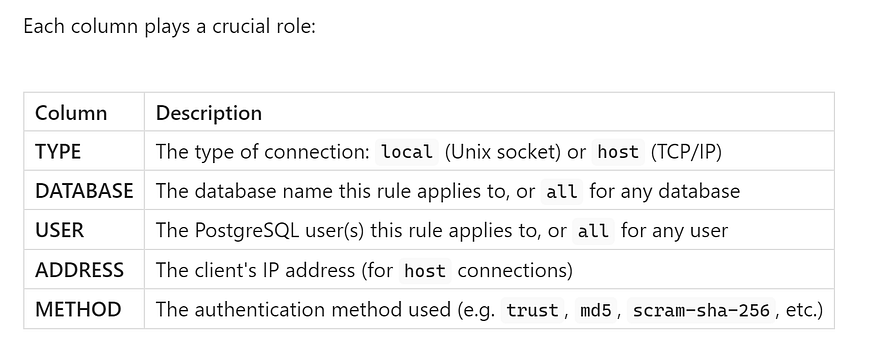
Let’s break it down in simple terms:

## **🔎 The 5 Columns of**pg\_hba.conf

Every rule inside pg\_hba.conf follows a consistent five-column structure:

# TYPE DATABASE USER ADDRESS METHOD

Press enter or click to view image in full size



## **🧩 Let’s Look at Each Column in Detail**

## **1️⃣ TYPE — Connection Type**

* local: Unix-domain socket connections (typically for connections from the same server).
* host: TCP/IP connections (for remote or network-based connections).
* hostssl: TCP/IP connections encrypted with SSL.
* hostnossl: TCP/IP connections without SSL encryption.

👉 ****Example:****

host all all 127.0.0.1/32 md5

This rule applies to any client connecting via TCP/IP (host).

## **2️⃣ DATABASE — Which Databases the Rule Applies To**

* You can specify:
* A specific database name.
* The keyword all to apply the rule to every database.
* pguser to match databases named the same as the connecting user.
* replication to control replication connections.

👉 ****Example:****

host dvdrental all 127.0.0.1/32 md5

This rule allows connections only to the dvdrental database.

## **3️⃣ USER — Who Can Connect**

* You can list:
* A single PostgreSQL username.
* Multiple usernames separated by commas.
* The keyword all for any user.

👉 ****Example:****

host all appuser,readonlyuser 127.0.0.1/32 md5

This allows only pguser and readonlyuser to connect.

## **4️⃣ ADDRESS — The Client’s IP Address or Subnet**

* For host rules, this defines:
* The client IP address (127.0.0.1 for localhost).
* The subnet (192.168.11.0/24 for entire network ranges).
* 0.0.0.0/0 for all IPv4 addresses.
* ::1/128 for IPv6 localhost.

👉 ****Example:****

host all all 192.168.11.0/24 md5

This allows connections from any client in the 192.168.11.x subnet.

## **5️⃣ METHOD — Authentication Method**

* ****trust**** — No password required (not secure for production).
* ****reject**** — Always deny access.
* ****md5**** — Use MD5 password-based authentication.
* ****scram-sha-256**** — More secure, recommended for PostgreSQL 10+.
* ****peer**** — Use system user identity (for local connections).
* ****password**** — Send password in clear text (rarely used anymore).

👉 ****Example:****

host all all 127.0.0.1/32 scram-sha-256

Requires SCRAM-based authentication.

## **🔨 Full Example Rule**

Let’s look at a full rule:

host all all 127.0.0.1/32 md5

📝 This means:

* ✅ Allow connections over TCP/IP (host)
* ✅ For ****all databases****
* ✅ From ****all users****
* ✅ Only from ****localhost (127.0.0.1)****
* ✅ Using ****MD5 password authentication****

## **🧑‍💻 Important Notes**

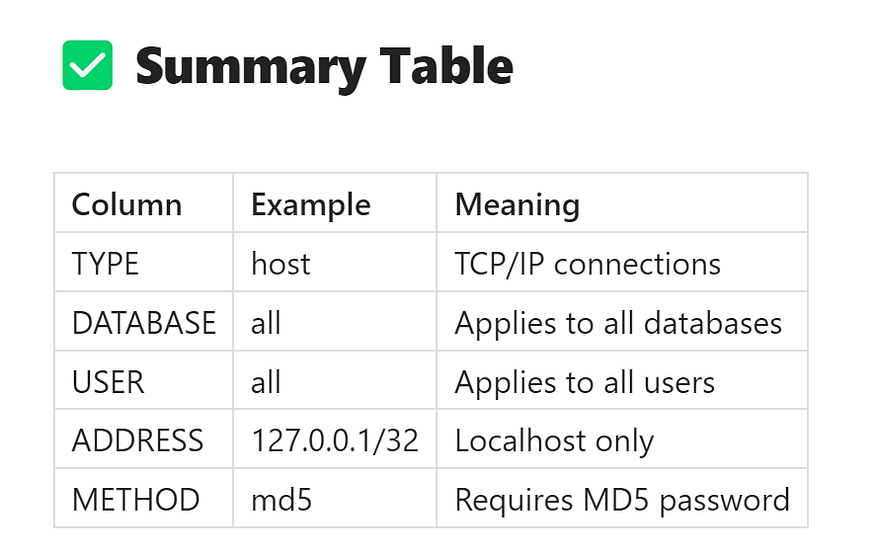
* Rules are evaluated ****top-down**** — PostgreSQL applies the first rule that matches.
* Always place more restrictive rules first.
* After modifying pg\_hba.conf, reload the configuration:

SELECT pg\_reload\_conf();

Or via shell:

sudo systemctl reload postgresql-17

Press enter or click to view image in full size



🔐 **Pro tip:** pg\_hba.conf is one of the most powerful access control tools in PostgreSQL. Mastering its structure is key to securing your database.

## **🔑 PostgreSQL Authentication Methods Explained (for Beginners & DBAs)**

PostgreSQL gives you full control over who can access your database, from where, and with which credentials. This flexibility is managed using the pg\_hba.conf file, where you configure your ****authentication methods****.

Let’s explore the most common authentication methods in PostgreSQL — when to use them, how to configure them, and how to test connections.

## **🔓 1️⃣ Trust Authentication — “No Password Mode”**

****Summary:****

* ✅ No password required
* 🚫 Extremely risky for production — use only in local development or isolated environments

****How it works:****  
With trust authentication, PostgreSQL simply trusts any client connecting that matches the rule. No password is ever requested.

****Example Configuration in pg\_hba.conf:****

host all all 127.0.0.1/32 trust

* This rule allows *any* user to connect to *any* database from 127.0.0.1 (localhost), without providing a password.

****Testing the Connection:****

psql postgres -h 127.0.0.1 -U postgres

👉 You will not be prompted for a password.

****What happens if remote clients try?****

psql postgres -h <remote\_IP> -U postgres

👉 Result:

FATAL: pg\_hba.conf rejects connection for host "<remote\_IP>", user "postgres", database "postgres"

🔒 ****Important Note:****  
Never use trust in production environments unless you're 100% certain the machine is isolated and fully protected.

## **🔑 2️⃣ MD5 Authentication — “Password with MD5 Hashing”**

****Summary:****

* ✅ Password required
* ✅ Uses MD5 hashing to store & verify passwords
* 🔒 Safer than trust, but older and less secure than modern alternatives

****How it works:****  
PostgreSQL stores user passwords hashed with MD5, and verifies them on login.

****Example Configuration in pg\_hba.conf:****

host all all 127.0.0.1/32 md5

* Requires clients to supply a password.

****Testing the Connection:****

psql postgres -h <remote\_IP> -U postgres

👉 Result:  
You’ll be prompted for a password:

Password for user postgres:

****⚠ MD5 Considerations:****

* MD5 is widely supported but considered outdated.
* Modern security best practices prefer stronger hashing methods like SCRAM.

## **🔐 3️⃣ SCRAM-SHA-256 Authentication — “Modern, Recommended Approach”**

****Summary:****

* ✅ Strong, salted password hashing
* ✅ Supported since PostgreSQL 10 (default in PostgreSQL 13+)
* ✅ Recommended for production use

****Why SCRAM?****  
SCRAM-SHA-256 (Salted Challenge Response Authentication Mechanism) uses modern hashing algorithms, adding stronger protection against password leaks and attacks.

## **🔧 How to Set Up SCRAM Authentication**

****Step 1️⃣: Enable SCRAM Password Encryption****

Inside your PostgreSQL session:

SET password\_encryption = 'scram-sha-256';

✅ This tells PostgreSQL to store future passwords using SCRAM-SHA-256.

****Step 2️⃣: Set (or reset) the User Password****

For example, to update the postgres user:

ALTER USER postgres WITH PASSWORD 'your\_password';

👉 The new password is now hashed with SCRAM-SHA-256.

****Step 3️⃣: Update pg\_hba.conf to Require SCRAM****

host all all <your\_IP>/32 scram-sha-256

* Replace <your\_IP> with the client's IP address or subnet.

****Step 4️⃣: Reload Configuration****

SELECT pg\_reload\_conf();

or from shell:

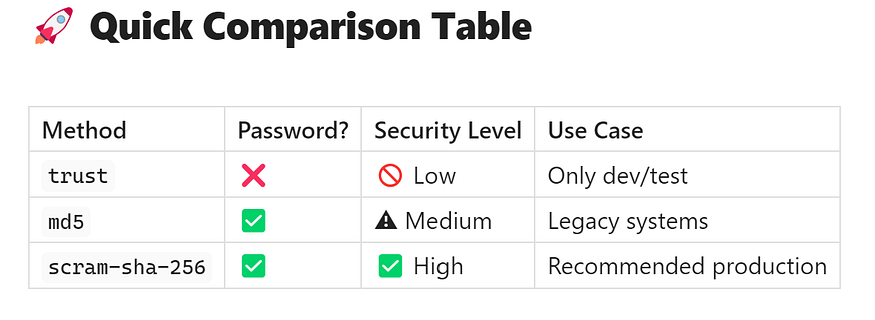
sudo systemctl reload postgresql-17

****Testing SCRAM Authentication:****

psql postgres -h <remote\_IP> -U postgres

👉 You’ll be prompted for a password just like MD5 — but now using strong SCRAM-based verification.

Press enter or click to view image in full size



🔐 **Best Practice:** Always use scram-sha-256 for production systems on PostgreSQL 17 for better password security.

## **🚩 Best Practices for PostgreSQL Authentication**

* 🔐 Always prefer scram-sha-256 for production.
* 🚫 Avoid trust outside of isolated dev machines.
* 🔎 Review pg\_hba.conf regularly as part of security audits.
* 🔄 Reload configuration after changes (no downtime required).
* 🚪 Restrict IP addresses carefully in the ADDRESS field.

## **🔐 Why PostgreSQL Authentication Is So Powerful**

Unlike many databases, PostgreSQL gives you full flexibility to:

* 🔒 Control every incoming connection
* 🔒 Define granular user/database rules
* 🔒 Combine with SSL/TLS encryption
* 🔒 Prevent unauthorized access at the connection level

Properly configured pg\_hba.conf is your ****first line of defense**** before authentication even happens.

## **🏁 Conclusion**

PostgreSQL’s internal authentication via pg\_hba.conf is simple, powerful, and incredibly flexible — but only when used correctly.

* ✅ Use scram-sha-256 for production-grade password security.
* ✅ Keep pg\_hba.conf clean, specific, and locked down.
* ✅ Reload configuration after edits, no restart needed.
* ✅ Remember: Database security starts before the connection even begins!

💡 **PostgreSQL 17 continues to deliver serious enterprise-grade security — but only if you configure it properly. Master pg\_hba.conf and you master your database perimeter.**