9-CHAPTER COURSE

MySQL Installation & Administration

Install, harden, configure, back up, and monitor MySQL on Ubuntu and Debian — from first apt install to phpMyAdmin, PHP integration, and performance monitoring.

1 Installing MySQL on Ubuntu/Debian
2 Initial Security Hardening
3 Users and Privileges
4 MySQL Configuration
5 Backup and Restore
6 Securing Remote Access
7 Monitoring and Performance Basics
8 MySQL with Apache and PHP
9 phpMyAdmin

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 1

Installing MySQL on Ubuntu/Debian

Chapter 1 — Installing MySQL on Ubuntu / Debian

MySQL is the database engine that runs behind the overwhelming majority of the web — WordPress, Joomla, phpBB, custom PHP applications — all store their data in MySQL tables. Before you can create databases, run queries, or connect web applications, you need a running MySQL server. This chapter gets it installed, started, and verified on Ubuntu and Debian systems.

What this chapter covers: What MySQL is and how it fits into a web stack. The MySQL vs MariaDB distinction — knowing which one you're about to install. Installing MySQL on Ubuntu (straightforward via apt). Installing MySQL on Debian (requires the official MySQL APT repository, since the default Debian packages install MariaDB instead). Starting and enabling the service. First login via Unix socket authentication. Verifying the installation and exploring the MySQL client. A tour of the files and directories MySQL creates. The four default databases explained.

What MySQL Is — and Where It Fits

The LAMP stack — where MySQL lives: Browser ⟶ Apache (web server) ⟶ PHP (application logic) ⟶ MySQL (data) │ ┌──────────┘ └──────────┐ │ Tables/Rows │ │ books │ │ users │ │ orders │ └──────────┘ MySQL's job: store data persistently, retrieve it fast, enforce relationships, and handle multiple simultaneous users without data corruption.

A **relational database management system** (RDBMS) stores data in tables — rows and columns, like a spreadsheet, but with strict structure and the ability to link tables together. MySQL is the world's most widely deployed open-source RDBMS. It handles everything from a personal blog's twenty posts to a billion-row e-commerce catalogue.

MySQL runs as a background service (`mysqld`). Your PHP code, Python scripts, or admin tools connect to it over a socket or TCP connection, send SQL commands, and receive results. MySQL itself never touches your web files — it only speaks SQL.

MySQL vs MariaDB — Know Before You Install

MariaDB was created in 2009 as a community fork of MySQL after Oracle acquired MySQL AB. The two share the same SQL syntax, the same client tools, and are largely compatible — but they are different software with diverging internals. On **Debian**, the default package `default-mysql-server` installs MariaDB, not MySQL. On **Ubuntu**, `mysql-server` installs MySQL.

MySQL (Oracle)
Maintained by Oracle. **Packages:** `mysql-server` on Ubuntu, or the MySQL APT repository on Debian.

Used throughout this course. Newer features: JSON columns, window functions (8.0+), InnoDB improvements.

Check you have it: `mysqld --version` should say `MySQL Community Server`.
MariaDB (Community)
Open-source fork. **Package:** `default-mysql-server` or `mariadb-server`.

Default on Debian. Uses `mysqld` binary and `mysql` client — the commands look identical, which causes confusion.

Check you have it: `mysqld --version` says `mariadb` or `MariaDB`.

The SQL you'll write in this course runs on both. The *administration* details (service names, config file locations, some system variables) can differ slightly. The instructions in this chapter install MySQL specifically on both distros, so you'll be on the same version regardless of which OS you're running.

Installing MySQL

Ubuntu (20.04 / 22.04 / 24.04)

Ubuntu's official repositories ship MySQL Community Server — `apt install mysql-server` gets you the real thing with no extra steps.

Update the package index first $ sudo apt update # Install MySQL server (includes the client tools automatically) $ sudo apt install mysql-server -y # Verify the installed version $ mysql --version mysql Ver 8.0.36 Distrib 8.0.36, for Linux (x86_64) using EditLine wrapper # Check the service started automatically $ sudo systemctl status mysql ● mysql.service - MySQL Community Server Loaded: loaded (/lib/systemd/system/mysql.service; enabled) Active: active (running) since Mon 2026-06-15 10:00:01 BST

On Ubuntu, MySQL starts and enables itself automatically during installation. You don't need to run `systemctl start` or `systemctl enable` — it's already running and set to start on boot.

Debian (11 Bullseye / 12 Bookworm)

Debian's default repositories don't include MySQL — `default-mysql-server` installs MariaDB instead. To get MySQL specifically, you need to add Oracle's official MySQL APT repository first.

Setup Walkthrough · Debian — Adding the MySQL APT Repository
Install MySQL on Debian by adding Oracle's official APT source.
1
Download the MySQL APT configuration package. This adds the MySQL repository and its signing key in one step.

$ sudo apt install wget lsb-release gnupg -y $ wget https://dev.mysql.com/get/mysql-apt-config_0.8.29-1_all.deb
Visit **dev.mysql.com/downloads/repo/apt/** to get the latest version number if the filename above has changed.
2
Install the configuration package. A menu appears — select **MySQL Server & Cluster** and choose the version you want (MySQL 8.0 or 8.4 LTS). Then select **OK**.

$ sudo dpkg -i mysql-apt-config_0.8.29-1_all.deb # An ncurses menu appears — select MySQL 8.0 (stable) or 8.4 (LTS), then OK
3
Update apt and install MySQL.

$ sudo apt update $ sudo apt install mysql-server -y # You will be prompted to set a root password during installation on Debian
4
Start and enable the service.

$ sudo systemctl start mysql $ sudo systemctl enable mysql $ sudo systemctl status mysql Active: active (running)
5
Clean up the downloaded package.

$ rm mysql-apt-config_0.8.29-1_all.deb
The MySQL APT repository also keeps MySQL up to date via `apt upgrade` — you don't need to manually download new versions in the future.

Managing the MySQL Service

Start MySQL (if not already running) $ sudo systemctl start mysql # Stop MySQL (flushes data to disk, closes connections gracefully) $ sudo systemctl stop mysql # Restart MySQL (stop then start — disconnects all active clients) $ sudo systemctl restart mysql # Reload configuration without restarting (not all changes take effect this way) $ sudo systemctl reload mysql # Enable MySQL to start on boot (usually already set after install) $ sudo systemctl enable mysql # Disable auto-start on boot $ sudo systemctl disable mysql # Detailed service status (shows recent log lines) $ sudo systemctl status mysql ● mysql.service - MySQL Community Server Loaded: loaded (/lib/systemd/system/mysql.service; enabled; ...) Active: active (running) since Mon 2026-06-15 10:00:01 BST; 5min ago Process: ExecStartPre=/usr/share/mysql/mysql-systemd-start pre (code=exited, status=0/SUCCESS) Main PID: 1234 (mysqld) Tasks: 38 (limit: 4678) Memory: 386.2M CPU: 1.843s

`systemctl stop mysql` before maintenance — not `kill`. Killing the mysqld process abruptly risks leaving data in an inconsistent state. Always use `systemctl stop`, which triggers a clean shutdown that flushes the InnoDB buffer pool and closes all open transactions.

Your First Login — Unix Socket Authentication

On a fresh Ubuntu install, MySQL's root account uses **Unix socket authentication** instead of a password. This means you log in as the MySQL root user by being the Linux `root` user (or using `sudo`) — no password required over the terminal. This is intentional and secure: root database access is only possible from the server itself, by a user who already has sudo.

Connect to MySQL as root (Ubuntu default — no password prompt) $ sudo mysql Welcome to the MySQL monitor. Commands end with ; or \g. Your MySQL connection id is 8 Server version: 8.0.36 MySQL Community Server - GPL Type 'help;' or '\h' for help. Type '\c' to clear the current input statement. mysql> # You are now inside the MySQL client. The mysql> prompt is waiting for SQL. # All SQL statements end with a semicolon (;) — without it, MySQL waits for more input. # Show the server version mysql> SELECT VERSION(); +-----------+ | VERSION() | +-------

----+ | 8.0.36 | +-----------+ 1 row in set (0.00 sec) # List all databases mysql> SHOW DATABASES; +--------------------+ | Database | +--------------------+ | information_schema | | mysql | | performance_schema | | sys | +--------------------+ 4 rows in set (0.00 sec) # Exit the MySQL client mysql> EXIT; Bye

Debian difference: On Debian, you set a root password during installation. Log in with: `mysql -u root -p` — you'll be prompted for that password. If you need to connect from the terminal without a password on Debian, use `sudo mysql` — this also bypasses the password via the Unix socket.

What Got Installed — Files and Directories

/etc/mysql/ ← main config directory ├── mysql.conf.d/ │ └── mysqld.cnf ← primary server config (Ubuntu) ├── conf.d/ │ └── mysql.cnf ← client config (charset etc.) └── my.cnf ← top-level include file /var/lib/mysql/ ← data directory — your actual databases live here ├── mysql/ ← the system mysql database ├── performance_schema/ ├── sys/ ├── ibdata1 ← InnoDB shared tablespace ├── ib_logfile0 ← InnoDB redo log └── mysql.sock ← Unix socket (how local connections work) /var/log/mysql/ ← log files └── error.log ← startup errors, crashes, warnings — check here first /usr/bin/ ├── mysql ← the MySQL command-line client ├── mysqldump ← backup tool (Chapter 5) ├── mysqladmin ← admin operations from the shell └── mysqlcheck ← table check/repair/optimise

Never edit files directly in `/var/lib/mysql/` — that's where MySQL stores its binary data files. Touching them outside MySQL will corrupt your databases. All data manipulation goes through SQL or the `mysqldump` backup tool.

The Four Default Databases

Every fresh MySQL install comes with four databases. You'll see them in `SHOW DATABASES` — they're not yours to use for application data, but understanding what they are saves confusion later.

mysql
System database
Stores user accounts, privileges, and server settings. The `user` table inside it controls who can connect and what they can do. **You'll interact with this indirectly** — using `CREATE USER` and `GRANT` commands (Chapter 3) rather than editing the table directly.
information_schema
Metadata read-only
A virtual read-only database containing metadata about everything on the server — tables, columns, indexes, views, privileges. **Useful for:** discovering what columns a table has when you don't have the original schema. `SELECT * FROM information_schema.TABLES`.
performance_schema
Monitoring
Runtime performance statistics — query execution times, lock waits, I/O events, memory usage. Data is collected continuously. **Used in Chapter 7** (monitoring). Read-only, memory-resident — no data persisted to disk.
sys
Helper views
A collection of views and stored procedures that make `performance_schema` human-readable. **Example:** `SELECT * FROM sys.processlist` — shows active queries in plain English rather than raw performance counters.

Essential MySQL Client Commands

── Connecting ── # Connect as root via Unix socket (no password on Ubuntu) $ sudo mysql # Connect as a named user with a password prompt $ mysql -u myuser -p # Connect to a specific database directly $ mysql -u myuser -p mydatabase # Connect to a remote server $ mysql -u myuser -p -h 192.168.1.100 # ── Inside the MySQL client ── # Show all databases mysql> SHOW DATABASES; # Switch to a database mysql> USE myDatabase; Database changed # Show all tables in the current database mysql> SHOW TABLES; # Show the columns of a table mysql> DESCRIBE tableName; # Show the CREATE TABLE statement for a table (reveals indexes, constraints) mysql> SHOW CREATE TABLE tableName\G # Check server status summary mysql> STATUS; -------------- mysql Ver 8.0.36 Distrib 8.0.36, for Linux (x86_64) Connection id: 8 Current database: Current user: root@localhost Server version: 8.0.36 MySQL Community Server - GPL Uptime: 15 min 4 sec Threads: 2 Questions: 12 Slow queries: 0 ... # Show all active connections and queries mysql> SHOW PROCESSLIST; # Get help on any SQL command mysql> HELP SELECT; # Clear the current input (abandoned command) — use \c instead of backspace mysql> SELECT * FROM broken_query \c # Exit the client mysql> EXIT; # or: mysql> QUIT;

SQL keywords are case-insensitive — `SELECT`, `select`, and `Select` all work. Convention is to write SQL keywords in UPPERCASE and table/column names in lowercase — it makes queries easier to read at a glance. All examples in this course follow that convention.

The semicolon is not optional. MySQL won't execute a statement until it sees a `;`. If you press Enter without one, you get a continuation prompt (`->`). This is a feature — you can split a long statement across multiple lines. Type `;` on its own line to run what you've written so far.

Troubleshooting

mysql: command not found after installation

The client binary wasn't installed. On Ubuntu, `apt install mysql-server` includes the client. If you installed only `mysql-client` and need the server, run `sudo apt install mysql-server`. On Debian, if you only added the APT repo but didn't install, run `sudo apt update && sudo apt install mysql-server`. Confirm: `which mysql` — should return `/usr/bin/mysql`.

ERROR 2002: Can't connect to local MySQL server through socket

MySQL isn't running. Check: `sudo systemctl status mysql` — look at the Active line. If it says failed, check the error log: `sudo tail -30 /var/log/mysql/error.log`. Common causes: another process is using port 3306 (`sudo ss -tlnp | grep 3306`), or the data directory has wrong permissions (`ls -la /var/lib/mysql` — should be owned by `mysql:mysql`). Fix permissions: `sudo chown -R mysql:mysql /var/lib/mysql`.

ERROR 1045: Access denied for user 'root'@'localhost'

On Ubuntu, root uses socket authentication — run `sudo mysql` (with sudo), not `mysql -u root -p`. Without sudo, MySQL sees you as your regular Linux user, not root, and rejects the connection. On Debian where you set a root password during install, use `mysql -u root -p` and enter that password — without the `sudo` prefix.

apt install mysql-server on Debian installs MariaDB instead

On Debian, `mysql-server` is a virtual package that resolves to `default-mysql-server` → MariaDB. You must add the MySQL APT repository first (see the Debian walkthrough above). After adding the repo and running `apt update`, running `apt install mysql-server` will find the real MySQL package from Oracle's repo and install that instead of MariaDB. Confirm after install: `mysqld --version` should say `MySQL Community Server`, not `MariaDB`.

MySQL is running but `SHOW DATABASES` shows nothing or gives access denied

You're probably connected as a user with no privileges. Connect as root: `sudo mysql` (Ubuntu) or `mysql -u root -p` (Debian). If you're already root and see only `information_schema`, this is correct — a root user with socket auth has access to all databases, but non-root users only see databases they have privileges on. Run `SHOW GRANTS;` inside MySQL to see what the current user can access.

Quick Reference — Chapter 1

Command	Purpose
sudo apt install mysql-server -y	Install MySQL on Ubuntu (includes client tools)
sudo systemctl start mysql	Start the MySQL service
sudo systemctl enable mysql	Enable MySQL to start automatically on boot
sudo systemctl status mysql	Check whether MySQL is running (and see recent log lines)
mysql --version	Print the MySQL client version
mysqld --version	Print the server version — use this to confirm MySQL vs MariaDB
sudo mysql	Connect as root via Unix socket (Ubuntu — no password needed)
mysql -u root -p	Connect as root with a password prompt (Debian)
SHOW DATABASES;	List all databases the current user can see
SELECT VERSION();	Show the MySQL server version from inside the client
STATUS;	Show connection info, server version, and uptime
EXIT; / QUIT;	Leave the MySQL client
sudo tail -f /var/log/mysql/error.log	Watch the MySQL error log — first place to check if something's wrong

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 2

Initial Security Hardening

Chapter 2 — Initial Security Hardening

A fresh MySQL installation ships in a deliberately permissive state — anonymous users can connect without credentials, a world-accessible test database exists, and root can potentially be reached without a password. None of this is malicious; it's to make the initial experience smooth on a development machine. On a server visible to the internet, these defaults are unacceptable. This chapter hardens them in a single step using MySQL's built-in tool, then verifies the result.

What this chapter covers: What a fresh MySQL install leaves open and why it matters. Running `mysql_secure_installation` — a full annotated walkthrough of every prompt and the recommended answer. The Validate Password component and when each policy level is appropriate. The Ubuntu socket-auth decision: whether to keep `sudo mysql` access or switch to password auth. Manual verification queries to confirm the hardening worked. What this script doesn't cover (a bridge to Chapter 3). Troubleshooting.

What a Fresh Install Leaves Open

Before running `mysql_secure_installation`, connect as root and look at the user table. What you find explains exactly what the hardening script fixes.

$ sudo mysql # See every account that can connect to this server mysql> SELECT user, host, plugin, authentication_string != '' AS has_password -> FROM mysql.user; +------------------+-----------+-----------------------+--------------+ | user | host | plugin | has_password | +------------------+-----------+-----------------------+--------------+ | | localhost | mysql_native_password | 0 | | webserver | mysql_native_password | 0 | | mysql.infoschema | localhost | caching_sha2_password | 1 | | mysql.session | localhost | caching_sha2_password | 1 | | mysql.sys | localhost | caching_sha2_password | 1 | | root | localhost | auth_socket | 0 | +------------------+-----------+-----------------------+--------------+ 6 rows in set (0.00 sec) # Check if the test database exists mysql> SHOW DATABASES; +--------------------+ | Database | +--------------------+ | information_schema | | mysql | | performance_schema | | sys | | test | +--------------------+ 5 rows in set (0.00 sec)
Before hardening — security problems
✗ **Anonymous users** — rows with an empty `user` field. Anyone who can reach the MySQL port can connect with no credentials at all.

✗ **test database** — world-accessible by anonymous users. Can be used to probe disk space, test injection payloads, or generate load.

✗ **Root accessible from multiple hosts** — on some installs, a root account exists with `host='%'`, meaning remote login as root is possible.

✗ **No password policy** — nothing stops application users from being created with weak passwords.
After hardening — what changes
✓ **Anonymous users removed** — every connection requires a named account.

✓ **test database dropped** — no more world-accessible scratch space.

✓ **Remote root blocked** — root can only connect from `localhost`.

✓ **Password validation enabled** (optional) — enforces complexity rules on all future `CREATE USER` and `ALTER USER` calls.

The Validate Password Component

The first prompt in `mysql_secure_installation` asks whether to install the **VALIDATE PASSWORD component**. If enabled, it enforces a complexity policy on any password set or changed from that point forward — including passwords you set for application users in Chapter 3.

LOW (policy 0)

- Minimum 8 characters
- No other requirements

Too permissive — skip
MEDIUM (policy 1)

- Minimum 8 characters

- At least 1 uppercase
- At least 1 lowercase
- At least 1 digit
- At least 1 special character

Recommended
STRONG (policy 2)

- All MEDIUM rules, plus:
- Dictionary file check
- Rejects common words

Can cause friction with scripts
STRONG policy and automated scripts: If you use STRONG, ensure your dictionary file (`/usr/share/mysql/english.txt` or similar) exists — if MySQL can't find it, all password attempts fail. MEDIUM is robust without this dependency.

The Ubuntu Decision: Keep Socket Auth or Switch to Password Auth

On Ubuntu, MySQL root uses **Unix socket authentication** — you connect with `sudo mysql` and no password is asked. During `mysql_secure_installation`, you'll be asked whether to change the root password. **Your answer determines how you'll administer MySQL going forward.**

Keep socket auth (recommended for home servers)
Answer **N** when asked "Change the password for root?"

How to connect after: `sudo mysql` (no password, just sudo access required)

Pros: Simpler, no root password to manage or forget, root is already protected by your Linux user password and SSH keys.

Cons: Some GUI tools (phpMyAdmin, MySQL Workbench) can't authenticate via socket — you'll need a separate admin account for those (Chapter 3 covers this).
Switch to password auth (better for GUI tools)
Answer **Y** and set a strong password.

How to connect after: `mysql -u root -p` (you'll be prompted for the password). `sudo mysql` may no longer work.

Pros: phpMyAdmin and MySQL Workbench can authenticate as root directly.

Cons: A root database password to store and manage. If forgotten, recovery requires stopping MySQL and starting in skip-grant mode.
On Debian, you already set a root password during installation. `mysql_secure_installation` may offer to change it — decide whether you want a new one. The socket-auth question doesn't apply in the same way.

Running mysql_secure_installation — Full Annotated Walkthrough

$ sudo mysql_secure_installation
Step 1 of 5 — Validate Password Component
Securing the MySQL server deployment. Connecting to MySQL using a blank password. VALIDATE PASSWORD COMPONENT can be used to test passwords and improve security. It checks the strength of password and allows the users to set only those passwords which are secure enough. Would you like to setup VALIDATE PASSWORD component? Press y|Y for Yes, any other key for No: y There are three levels of password validation policy: LOW Length >= 8 MEDIUM Length >= 8, numeric, mixed case, and special characters STRONG Length >= 8, numeric, mixed case, special characters and dictionary file Please enter 0 = LOW, 1 = MEDIUM and 2 = STRONG: 1
Answer Y to install the component and enter 1 for MEDIUM. This policy (8+ chars, mixed case, digit, special character) covers the vast majority of attack scenarios without causing friction when creating application users.
Step 2 of 5 — Root Password
Estimated strength of the password: 0 Change the password for root ? ((Press y|Y for Yes, any other key for No) : n ... skipping.
Ubuntu with socket auth: Answer N to keep `sudo mysql` working. Your root connection is already secured by your Linux sudo password + SSH keys.

Debian with a password already set: Answer Y only if you want to change the password you set during installation. Otherwise N is fine.

If you answer Y: MySQL switches root from `auth_socket` to `caching_sha2_password`. You must use `mysql -u root -p` from that point forward — `sudo mysql` will no longer work.
Step 3 of 5 — Anonymous Users
Remove anonymous users? (Press y|Y for Yes, any other key for No) : y Success.
Always Y. Anonymous users (the accounts with an empty username you saw in the `mysql.user` table earlier) can connect to MySQL without any credentials. This is never acceptable on a server. Removing them means every connection requires a named account with a password.
Step 4 of 5 — Remote Root Login
Disallow root login remotely? (Press y|Y for Yes, any other key for No) : y Success.
Always Y. This removes any root accounts where `host` is not `localhost`. Root should only ever be used locally, from the server itself. Remote root access, even with a strong password, exposes your entire database server to brute-force attempts on the most privileged account. Application databases use non-root accounts (Chapter 3).
Step 5a of 5 — Test Database
Remove test database and access to it? (Press y|Y for Yes, any other key for No) : y - Dropping test database... Success. - Removing privileges on test database... Success.
Always Y. The test database is accessible by anonymous users and can be used to probe the server — writing large amounts of data to measure disk speed, running expensive queries to generate load, or simply exploring what functions are available before attempting an injection attack. Remove it entirely.
Step 5b of 5 — Reload Privileges
Reload privilege tables now? (Press y|Y for Yes, any other key for No) : y Success. All done! If you've completed all of the above steps, your MySQL installation should now be secure. Thanks for using MySQL!
Always Y. The changes made to the `mysql` system database (removing anonymous users, removing remote root) don't take effect for currently-connected clients until the privilege tables are reloaded. `FLUSH PRIVILEGES` is what this step runs internally. Without it, a connected anonymous session could theoretically remain active until MySQL restarts.

Verifying the Hardening Worked

Don't take the script's "Success" messages at face value — check the results directly. These three queries confirm everything the script promised to do.

$ sudo mysql # 1 — Check that anonymous users are gone (should return 0 rows) mysql> SELECT user, host, plugin FROM mysql.user WHERE user = ''; Empty set (0.00 sec) # 2 — Check that root only exists for localhost (no remote root accounts) mysql> SELECT user, host, plugin FROM mysql.user WHERE user = 'root'; +------+-----------+-------------+ | user | host | plugin | +------+-----------+-------------+ | root | localhost | auth_socket | +------+-----------+-------------+ 1 row in set (0.00 sec) # ✓ Only one row, host is 'localhost', plugin shows socket auth is intact # 3 — Confirm the test database is gone mysql> SHOW DATABASES; +--------------------+ | Database | +--------------------+ | information_schema | | mysql | | performance_schema | | sys | +--------------------+ 4 rows in set (0.00 sec) # ✓ test database is absent — back to the 4 system databases only # 4 — Full user audit: see everything in the user table cleanly mysql> SELECT user, host, plugin, password_expired, account_locked -> FROM mysql.user; +------------------+-----------+-----------------------+------------------+----------------+ | user | host | plugin | password_expired | account_locked | +------------------+-----------+-----------------------+------------------+----------------+ | mysql.infoschema | localhost | caching_sha2_password | N | Y | | mysql.session | localhost | caching_sha2_password | N | Y | | mysql.sys | localhost | caching_sha2_password | N | Y | | root | localhost | auth_socket | N | N | +------------------+-----------+-----------------------+------------------+----------------+ 4 rows in set (0.00 sec) # ✓ mysql.* system accounts are locked — they can't be used to log in # ✓ root is the only active account, localhost only, socket auth **Confirm you can still connect after hardening.** Open a new terminal and run `sudo mysql` — you should land at the MySQL prompt as before. If you chose to switch to password auth in Step 2, test with `mysql -u root -p` using your new password. Only once you've confirmed you can still get in should you consider the hardening complete.

Manual Steps — If mysql_secure_installation Didn't Catch Everything

Occasionally, particularly on Debian or if you interrupted the script, some of the steps may not have completed. These SQL commands do exactly what the script does — run them inside MySQL as root if needed.

$ sudo mysql # Remove any remaining anonymous users manually mysql> DELETE FROM mysql.user WHERE user = ''; Query OK, 2 rows affected (0.01 sec) # Block remote root login manually mysql> DELETE FROM mysql.user WHERE user = 'root' AND host != 'localhost'; Query OK, 0 rows affected (0.00 sec) # Drop the test database manually mysql> DROP DATABASE IF EXISTS test; Query OK, 1 row affected (0.01 sec) # Remove any leftover privileges on the test database mysql> DELETE FROM mysql.db WHERE db = 'test' OR db = 'test_%'; Query OK, 1 row affected (0.00 sec) # Apply all privilege changes immediately mysql> FLUSH PRIVILEGES; Query OK, 0 rows affected (0.00 sec)

Always run `FLUSH PRIVILEGES` after any direct edits to `mysql.user` or `mysql.db`. MySQL caches grant tables in memory. Changes written directly to these tables don't take effect until you reload the cache. `FLUSH PRIVILEGES` is the command that does this. If you use `CREATE USER`, `GRANT`, or `REVOKE` statements (covered in Chapter 3), MySQL updates the cache automatically and you don't need to flush.

What This Chapter Doesn't Cover

`mysql_secure_installation` removes the worst defaults. But there's a broader set of security concerns it deliberately ignores — these are the topics for the rest of the course.

Chapter 3
Application users and privileges
Creating dedicated database accounts per application, granting only the permissions each one needs, revoking them, auditing who has what.
Chapter 4
MySQL configuration
Tuning memory limits, max connections, the slow query log, character sets, and other settings in `mysqld.cnf`.
Chapter 5
Backup and restore
Regular automated dumps, verifying backups, restoring from `mysqldump` output.
Chapter 6
Remote access
Controlling which interfaces MySQL listens on, UFW rules for port 3306, and SSL/TLS encrypted connections.
Chapter 9
phpMyAdmin security
Restricting the phpMyAdmin URL, IP allowlisting, and ensuring it doesn't expose root access to the web.

Troubleshooting

mysql_secure_installation: ERROR 1698 (28000): Access denied for user 'root'@'localhost'
On Ubuntu, `mysql_secure_installation` must be run with `sudo`. Without sudo it tries to connect as the current Linux user rather than root and gets rejected. Run: `sudo mysql_secure_installation`. On Debian, you need the root password: run as root with `su -` first, then run `mysql_secure_installation` (without sudo), or use `sudo mysql_secure_installation` if sudo is configured.
After choosing Y to change root password, sudo mysql no longer works
Choosing Y in Step 2 switches the root plugin from `auth_socket` to `caching_sha2_password`. Socket auth is gone — use `mysql -u root -p` with the password you set. If you want to go back to socket auth: `sudo mysql -u root -p`, then: `ALTER USER 'root'@'localhost' IDENTIFIED WITH auth_socket;`, then `FLUSH PRIVILEGES;`. After this, `sudo mysql` will work again and the root password is no longer used.
Your password does not satisfy the current policy requirements
MEDIUM policy requires: 8+ characters, at least one uppercase letter, one lowercase letter, one digit, and one special character (e.g. `!`, `@`, `#`, `$`, `%`). Example compliant password: `Secure!Db9`. If you're setting a password for an application service account and it's being rejected, also check that you haven't accidentally hit STRONG policy — check with: `SHOW VARIABLES LIKE 'validate_password%';` inside MySQL.
mysql_secure_installation exits immediately with no prompts on Debian
This usually means it connected but encountered an error before the first prompt. Check with: `mysql_secure_installation 2>&1 | head -20` to see the error. Common cause on Debian: the root password set at install time has expired (`password_expired = Y` in `mysql.user`). Fix by connecting via `sudo mysql` and running: `ALTER USER 'root'@'localhost' IDENTIFIED BY 'NewPassword!';` then retry.
mysql.user query shows root still has a host='%' row after hardening
Some MySQL setups or manual configurations add a wildcard root account that the script may not have caught. Remove it manually: `DELETE FROM mysql.user WHERE user='root' AND host='%';` then `FLUSH PRIVILEGES;`. Verify: `SELECT user, host FROM mysql.user WHERE user='root';` — should show only `localhost`.

Quick Reference — Chapter 2

Command / Query	Purpose
sudo mysql_secure_installation	Run the MySQL hardening wizard (always with sudo on Ubuntu)
SELECT user, host, plugin FROM mysql.user;	Full audit of all MySQL accounts — run before and after hardening
SELECT user, host FROM mysql.user WHERE user = '';	Check for anonymous users (expect empty set after hardening)
SELECT user, host FROM mysql.user WHERE user = 'root';	Confirm root is restricted to localhost only

Command / Query	Purpose
SHOW DATABASES;	Confirm the test database is gone (should show only 4 system databases)
DELETE FROM mysql.user WHERE user = '';	Remove anonymous users manually
DELETE FROM mysql.user WHERE user = 'root' AND host != 'localhost';	Block remote root login manually
DROP DATABASE IF EXISTS test;	Remove the test database manually
FLUSH PRIVILEGES;	Apply changes to mysql.user/mysql.db immediately (always needed after direct table edits)
ALTER USER 'root'@'localhost' IDENTIFIED WITH auth_socket;	Switch root back to socket auth if you accidentally changed it
SHOW VARIABLES LIKE 'validate_password%';	See the current password policy settings

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 3

Users and Privileges

Chapter 3 — Users and Privileges

Every database-backed application needs to connect to MySQL. What account it uses and what that account is permitted to do is one of the most consequential security decisions in your stack. This chapter covers creating dedicated application users, understanding MySQL's privilege system, and applying the principle of least privilege — giving each account exactly what it needs, nothing more.

What this chapter covers: Why application code must never connect as root. MySQL's three-level privilege system. The host field — what `localhost`, `127.0.0.1`, and `%` actually mean (and the PHP gotcha that breaks connections). `CREATE USER` and authentication plugins. `GRANT` — the key privilege sets every administrator needs to know. Complete walkthrough: creating a web app user for a database. `SHOW GRANTS` auditing. `REVOKE`, `DROP USER`, and `ALTER USER` for password changes and account control. Seeing who is connected with `SHOW PROCESSLIST`. Troubleshooting.

Why Application Code Must Never Connect as Root

It's tempting to use the root account in your PHP or Python connection string — it always works, it never hits a permissions error, and it removes a whole class of debugging. It's also a critical mistake.

- **SQL injection amplification:** If an attacker finds an injection point in your application, the root account gives them unrestricted access to every database on the server — not just yours. They can read other users' data, drop databases, or create new admin accounts.
- **Credential exposure:** Your application's database credentials live in a config file. If that file is ever exposed (misconfigured web server, Git accident, server compromise), a leaked root password is catastrophic. A leaked app user password with limited grants is contained.
- **No audit trail:** Everything looks like root in the logs. You can't distinguish what your WordPress site did from what your custom app did.
- **Accidental destruction:** A bug in your code that runs `DROP TABLE` as root works. As a constrained app user without DDL privileges, it fails safely.

The rule is simple: one database, one dedicated user, minimal privileges. Root is for administration from the terminal only.

How MySQL's Privilege System Works

MySQL has three tiers of privilege scope. Each `GRANT` statement applies to exactly one tier — and the tier is determined by what you write after `ON`.

Privilege scope — determined by the ON clause in GRANT: GRANT SELECT ON *.* ← GLOBAL — applies to every database on this server GRANT SELECT ON bookshop.* ← DATABASE — applies to every table in bookshop GRANT SELECT ON bookshop.books ← TABLE — applies to one specific table Scope check order: when a connection runs a query, MySQL checks from most specific to most general and applies the highest-matching privilege set. ┌──┐ │ Global level (*.*) │ ALL PRIVILEGES — full server control │ ┌──────────────────────────────────┐ │ │ │ Database level (db.*) │ │ CREATE, DROP, and object privileges on one db │ │ ┌──────────────────────────┐ │ │ │ │ Table level (db.t) │ │ │ SELECT, INSERT, UPDATE, DELETE on one table │ │ └──────────────────────────┘ │ │ │ └──────────────────────────────────────┘ │ │ └──┘

The Host Field — The Most Misunderstood Part of MySQL Users

A MySQL user account is not just a username. It's a **username + host pair**. `'app'@'localhost'` and `'app'@'127.0.0.1'` are completely separate accounts — they can have different passwords and different privileges. Getting this wrong is the single most common cause of "access denied" errors when connecting from PHP.

'localhost'
Connections via the **Unix socket file** (`/var/run/mysqld/mysqld.sock`). Only from the local machine. No network involved.
Use for: local PHP/Python apps connecting via socket
'127.0.0.1'
Connections via **TCP loopback**. Local machine only but goes through the network stack. Different from socket auth.
Use for: tools that force TCP (e.g. some JDBC drivers)
'%'

Wildcard — **any host**. Accepts connections from anywhere on the network. Never use for privileged accounts.
Dangerous — avoid unless explicitly needed
'192.168.1.%'
Subnet wildcard — any host on this subnet. Useful for allowing connections from an internal LAN only.
Use for: LAN-restricted remote access
'192.168.1.10'
A **specific IP address**. Most restrictive remote access pattern — only one known machine can connect.
Use for: locked-down remote admin access
The PHP host=localhost gotcha: In PHP's PDO or mysqli, `host=localhost` in the connection string means "use the Unix socket" — which matches a MySQL account with `@'localhost'`. But `host=127.0.0.1` means "use TCP" — which matches `@'127.0.0.1'` or `@'%'`, but **not** `@'localhost'`. If you create `'app'@'localhost'` but your PHP DSN says `host=127.0.0.1`, you'll get **Access denied** even though the user exists. Either make your DSN and user host consistent, or create both `'app'@'localhost'` and `'app'@'127.0.0.1'` with the same grants.

Creating Users

Basic syntax mysql> CREATE USER 'username'@'host' IDENTIFIED BY 'StrongPassword!9'; # Create a web application user (socket/local connection) mysql> CREATE USER 'bookshop_app'@'localhost' IDENTIFIED BY 'Bk$h0p!2026'; # Create a read-only reporting user mysql> CREATE USER 'bookshop_ro'@'localhost' IDENTIFIED BY 'R3adOnly!55'; # Create a user for a specific remote IP (e.g. a dev workstation) mysql> CREATE USER 'philip'@'192.168.1.50' IDENTIFIED BY 'Dev!Access7'; # Create a user and immediately lock the account (create now, activate later) mysql> CREATE USER 'future_user'@'localhost' IDENTIFIED BY 'Pass!word9' ACCOUNT LOCK; # Check the user was created mysql> SELECT user, host, plugin FROM mysql.user WHERE user = 'bookshop_app'; +--------------+-----------+-----------------------+ | user | host | plugin | +--------------+-----------+-----------------------+ | bookshop_app | localhost | caching_sha2_password | +--------------+-----------+-----------------------+
MySQL 8.0 change: `GRANT` no longer creates users. In MySQL 5.x, you could write `GRANT SELECT ON db.* TO 'user'@'host' IDENTIFIED BY 'pass'` and it would create the user if they didn't exist. MySQL 8.0 removed this — you must `CREATE USER` first, then `GRANT`. If you use an old tutorial that combines both, it will fail with *ERROR 1410: You are not allowed to create a user with GRANT*.

Authentication plugins: caching_sha2_password vs mysql_native_password

caching_sha2_password (MySQL 8.0 default)
Stronger security — uses SHA-256 with salting and caching for performance.

Supported by: PHP 7.4+ with mysqlnd, PHP 8.x, Python mysql-connector 8.0+, MySQL Workbench, modern drivers.

Use this unless you hit a compatibility error.
mysql_native_password (legacy)
Older SHA-1 based authentication. **Required for** older PHP versions with libmysqlclient (not mysqlnd), legacy Laravel setups, some Java JDBC drivers.

Error that tells you to switch: `PHP Warning: The server requested authentication method unknown to the client [caching_sha2_password]`

```
CREATE USER 'app'@'localhost' IDENTIFIED WITH mysql_native_password BY 'Pass!';
```
Switch an existing user to mysql_native_password (for old PHP compatibility) mysql> ALTER USER 'bookshop_app'@'localhost' -> IDENTIFIED WITH mysql_native_password BY 'Bk$h0p!2026'; Query OK, 0 rows affected (0.01 sec) # Or create with the legacy plugin from the start mysql> CREATE USER 'legacy_app'@'localhost' -> IDENTIFIED WITH mysql_native_password BY 'LegacyP@ss7';

Granting Privileges

A freshly created user can connect but can't do anything — they can't even see databases. Privileges are granted separately. The `GRANT` statement specifies what operations the user can perform and on what scope.

Syntax mysql> GRANT privilege_list ON scope TO 'user'@'host'; # Grant SELECT only (read-only user) mysql> GRANT SELECT ON bookshop.* TO 'bookshop_ro'@'localhost'; # Grant typical web app privileges (read + write data, no schema changes) mysql> GRANT SELECT, INSERT, UPDATE, DELETE ON bookshop.* TO 'bookshop_app'@'localhost'; # Grant full control of one database (but not other databases) mysql> GRANT ALL PRIVILEGES ON bookshop.* TO 'bookshop_admin'@'localhost'; # Grant on a single table only mysql> GRANT SELECT ON bookshop.books TO 'catalogue_reader'@'localhost'; # Privileges take effect immediately — no need to FLUSH PRIVILEGES # (FLUSH is only needed for direct mysql.user table edits)

GRANT ALL PRIVILEGES ON bookshop.* vs GRANT ALL PRIVILEGES ON *.* — the difference is enormous. `bookshop.*` gives full control of one database. `*.*` gives full control of the entire server — equivalent to root. Always scope grants to a specific database for application accounts.

The privilege sets you'll actually use

Read-only
SELECT
Reporting tools, analytics dashboards, data exports
Web app (typical)
SELECT, INSERT, UPDATE, DELETE
WordPress, Joomla, custom PHP apps — data manipulation only
Web app (with schema)
SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, INDEX, ALTER
Frameworks that run migrations (Laravel, Django) — add REFERENCES, CREATE TEMPORARY TABLES if needed
Backup account
SELECT, LOCK TABLES, SHOW VIEW, EVENT, TRIGGER
mysqldump requires these — CREATE is not needed for backups
GRANT OPTION — never
WITH GRANT OPTION
Lets user pass their own privileges to others — an instant privilege escalation path. Never grant to application accounts

Walkthrough: Setting Up a Web Application Database and User

Practical Walkthrough · Creating the bookshop Application User
Create a database, a dedicated user, grant appropriate privileges, and verify everything works.
1
Connect as root and create the application database.

$ sudo mysql mysql> CREATE DATABASE IF NOT EXISTS bookshop -> CHARACTER SET utf8mb4 -> COLLATE utf8mb4_unicode_ci; Query OK, 1 row affected (0.01 sec)
Using `utf8mb4` (not `utf8`) — MySQL's `utf8` is a 3-byte encoding that can't handle emoji and some CJK characters. `utf8mb4` is the real 4-byte UTF-8.
2
Create a dedicated application user. Use `localhost` if your PHP app connects via the Unix socket (the most common setup — DSN with `host=localhost`).

mysql> CREATE USER 'bookshop_app'@'localhost' -> IDENTIFIED BY 'Bk$h0p!2026'; Query OK, 0 rows affected (0.01 sec)
3
Grant the privileges the application actually needs. This app reads and writes data but doesn't need to create or drop tables.

mysql> GRANT SELECT, INSERT, UPDATE, DELETE -> ON bookshop.* -> TO 'bookshop_app'@'localhost'; Query OK, 0 rows affected (0.01 sec)
4
Verify the grants are correct.

mysql> SHOW GRANTS FOR 'bookshop_app'@'localhost'; +---+ | Grants for bookshop_app@localhost | +---+ | GRANT USAGE ON *.* TO `bookshop_app`@`localhost` | | GRANT SELECT, INSERT, UPDATE, DELETE ON `bookshop`.* TO `bookshop_app`@`localhost` | +---+ 2 rows in set (0.00 sec)
The first row (`GRANT USAGE ON *.*`) is automatic — it means "user can connect but has no global privileges." The second row shows the database-level grants. This is exactly right.
5
Test the connection as the new user. Open a new terminal (don't close your root session yet).

$ mysql -u bookshop_app -p Enter password: Bk$h0p!2026 Welcome to the MySQL monitor. ... # This user can only see the bookshop database (and information_schema) mysql> SHOW DATABASES; +--------------------+ | Database | +--------------------+ | bookshop | | information_schema | +--------------------+ # Confirm the user can't drop the database (principle of least privilege working) mysql> DROP DATABASE bookshop; ERROR 1044 (42000): Access denied for user 'bookshop_app'@'localhost' to database 'bookshop'
6
Store the credentials in your PHP config outside the web root.

/etc/bookshop/db.php (outside /var/www — not web-accessible) <?php define('DB_HOST', 'localhost'); define('DB_NAME', 'bookshop'); define('DB_USER', 'bookshop_app'); define('DB_PASS', 'Bk$h0p!2026'); # PDO connection using these constants $dsn = 'mysql:host=' . DB_HOST . ';dbname=' . DB_NAME . ';charset=utf8mb4'; $pdo = new PDO($dsn, DB_USER, DB_PASS, [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,]);

Chapter 8 covers securing database credentials in PHP applications in more detail, including keeping config files outside the web root and using environment variables.

Auditing Privileges with SHOW GRANTS

Show grants for a specific user mysql> SHOW GRANTS FOR 'bookshop_app'@'localhost'; # Show grants for the currently connected user mysql> SHOW GRANTS; # Audit ALL users and their privilege level in one query mysql> SELECT -> u.user, -> u.host, -> u.plugin, -> u.account_locked, -> u.password_expired -> FROM mysql.user u -> WHERE u.user NOT LIKE 'mysql.%' -> ORDER BY u.user, u.host; +--------------+-----------+-----------------------+----------------+------------------+ | user | host | plugin | account_locked | password_expired | +--------------+-----------+-----------------------+----------------+------------------+ | bookshop_app | localhost | caching_sha2_password | N | N | | bookshop_ro | localhost | caching_sha2_password | N | N | | root | localhost | auth_socket | N | N | +--------------+-----------+-----------------------+----------------+------------------+ # See database-level privileges across all users mysql> SELECT user, host, db, Select_priv, Insert_priv, Update_priv, Delete_priv -> FROM mysql.db -> ORDER BY db, user; +--------------+-----------+----------+-------------+-------------+-------------+-------------+ | user | host | db | Select_priv | Insert_priv | Update_priv | Delete_priv | +--------------+-----------+----------+-------------+-------------+-------------+-------------+ | bookshop_app | localhost | bookshop | Y | Y | Y | Y | | bookshop_ro | localhost | bookshop | Y | N | N | N | +--------------+-----------+----------+-------------+-------------+-------------+-------------+

Revoking Privileges

Revoke a specific privilege (user keeps all others) mysql> REVOKE DELETE ON bookshop.* FROM 'bookshop_app'@'localhost'; Query OK, 0 rows affected (0.01 sec) # Revoke ALL privileges on a specific database mysql> REVOKE ALL PRIVILEGES ON bookshop.* FROM 'bookshop_app'@'localhost'; Query OK, 0 rows affected (0.01 sec) # Revoke ALL privileges at ALL levels (leaves the user account intact but powerless) mysql> REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'bookshop_app'@'localhost'; Query OK, 0 rows affected (0.01 sec) # Confirm what remains mysql> SHOW GRANTS FOR 'bookshop_app'@'localhost'; +--+ | Grants for bookshop_app@localhost | +--+ | GRANT USAGE ON *.* TO `bookshop_app`@`localhost` | +--+ # Only USAGE remains — user can connect but cannot do anything

REVOKE affects currently-connected sessions. If bookshop_app is already connected when you revoke privileges, MySQL checks grants at query execution time, not at connection time. The running session will immediately lose the revoked permissions on their next query. You don't need to kick them off first.

Modifying and Removing Users

── Changing passwords ── # Change another user's password (as root) mysql> ALTER USER 'bookshop_app'@'localhost' IDENTIFIED BY 'NewP@ssword!8'; Query OK, 0 rows affected (0.01 sec) # Change your own password (as any logged-in user) mysql> ALTER USER USER() IDENTIFIED BY 'MyNewPass!7'; # ── Account locking and unlocking ──────────────────────────────────── # Lock an account temporarily (connection attempts will fail) mysql> ALTER USER 'bookshop_app'@'localhost' ACCOUNT LOCK; Query OK, 0 rows affected (0.00 sec) # Unlock it again mysql> ALTER USER 'bookshop_app'@'localhost' ACCOUNT UNLOCK; # ── Password expiry ─── # Force a user to change password on next login mysql> ALTER USER 'bookshop_app'@'localhost' PASSWORD EXPIRE; # Set a rolling expiry policy (user must change password every 90 days) mysql> ALTER USER 'bookshop_app'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY; # Disable expiry for an account (useful for service accounts) mysql> ALTER USER 'bookshop_app'@'localhost' PASSWORD EXPIRE NEVER; # ── Removing users ── # Drop a user and all their privileges in one step mysql> DROP USER 'bookshop_app'@'localhost'; Query OK, 0 rows affected (0.00 sec) # Safe version — no error if the user doesn't exist mysql> DROP USER IF EXISTS 'bookshop_app'@'localhost';

`DROP USER` removes the user and revokes all their privileges atomically. You don't need to `REVOKE` first. The user's objects (tables, views, stored procedures they created) are NOT dropped — only the account is removed.

Seeing Who Is Connected — SHOW PROCESSLIST

See all active connections and what they're doing mysql> SHOW PROCESSLIST; +----+--------------+-----------+----------+---------+------+-------+------------------+ | Id | User | Host | db | Command | Time | State | Info |

---+------------------+ | 1 | event_scheduler | localhost | | Daemon | 1024 | Waiting | NULL | | 8 | root | localhost | NULL | Query | 0 | starting | SHOW PROCESSLIST | | 11 | bookshop_app | localhost | bookshop | Sleep | 42 | | NULL | +----+--------------+-----------+----------+---------+------+------------------+------------------+ 3 rows in set (0.00 sec) # Terminate a specific connection (use the Id column) mysql> KILL 11; Query OK, 0 rows affected (0.00 sec) # A more detailed view via the sys schema mysql> SELECT user, host, db, command, time, state, info -> FROM sys.processlist -> WHERE command != 'Sleep' -> ORDER BY time DESC;

Use `SHOW PROCESSLIST` when you need to identify runaway queries, check which accounts are currently active, or confirm a connection test worked. The `Time` column shows how many seconds the connection has been in its current state — a very high number on a non-Sleep query indicates a stuck or slow query.

Troubleshooting

ERROR 1045 (28000): Access denied for user 'bookshop_app'@'127.0.0.1'
The user exists as `'bookshop_app'@'localhost'` but your PHP DSN says `host=127.0.0.1`, which makes MySQL see a TCP connection from `127.0.0.1` — a different host string than `localhost`. Either change your PHP DSN to `host=localhost`, or create a second account: `CREATE USER 'bookshop_app'@'127.0.0.1' IDENTIFIED BY 'same_password';` then grant the same privileges to it.
PHP Warning: mysqli_connect(): The server requested authentication method unknown to the client [caching_sha2_password]
Your PHP/mysqli version is too old to speak `caching_sha2_password`. Fix by switching the user to the legacy plugin: `ALTER USER 'bookshop_app'@'localhost' IDENTIFIED WITH mysql_native_password BY 'YourPassword!';`. Long-term fix: upgrade PHP to 7.4+ compiled with mysqlnd (not libmysqlclient). PHP 8.x + mysqlnd handles `caching_sha2_password` natively.
ERROR 1044 (42000): Access denied for user 'bookshop_app'@'localhost' to database 'bookshop'
The user was created but the `GRANT` either didn't run or used a different host string. Check: `SHOW GRANTS FOR 'bookshop_app'@'localhost';` — if it only shows `GRANT USAGE ON *.*` with no database-level grant, run: `GRANT SELECT, INSERT, UPDATE, DELETE ON bookshop.* TO 'bookshop_app'@'localhost';`
ERROR 1410: You are not allowed to create a user with GRANT
You tried to use the old MySQL 5.x syntax: `GRANT ... TO 'user'@'host' IDENTIFIED BY 'pass'`. MySQL 8.0 removed the ability to create a user implicitly in a GRANT statement. Run `CREATE USER` first, then `GRANT` as two separate statements.
User was granted privileges but still gets "Access denied" for specific operations
Check whether the operation requires a privilege you didn't grant. Common missing privileges: **CREATE TABLE** (framework migrations), **LOCK TABLES** (mysqldump backups), **REFERENCES** (foreign keys in some frameworks). Run `SHOW GRANTS FOR 'user'@'host';` and compare against the required privilege. Add missing ones with another `GRANT` statement — grants are additive and safe to run again.

Quick Reference — Chapter 3

Command	Purpose
CREATE USER 'u'@'h' IDENTIFIED BY 'pass';	Create a new MySQL user account
GRANT SELECT, INSERT, UPDATE, DELETE ON db.* TO 'u'@'h';	Grant typical web app privileges on one database
GRANT ALL PRIVILEGES ON db.* TO 'u'@'h';	Grant full control of one database (not global)
SHOW GRANTS FOR 'u'@'h';	Display all privileges assigned to a user
REVOKE DELETE ON db.* FROM 'u'@'h';	Remove a specific privilege
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'u'@'h';	Strip all privileges (user account remains)
ALTER USER 'u'@'h' IDENTIFIED BY 'newpass';	Change a user's password
ALTER USER 'u'@'h' ACCOUNT LOCK;	Temporarily prevent logins without deleting the account
ALTER USER 'u'@'h' ACCOUNT UNLOCK;	Re-enable a locked account
ALTER USER 'u'@'h' IDENTIFIED WITH mysql_native_password BY 'pass';	Switch to legacy auth plugin (for old PHP compatibility)
DROP USER IF EXISTS 'u'@'h';	Delete a user and all their privileges
SHOW PROCESSLIST;	List all active connections and current queries
KILL <Id>;	Terminate a specific connection (use Id from SHOW PROCESSLIST)
SELECT user, host, plugin FROM mysql.user WHERE user NOT LIKE 'mysql.%';	Full audit of all non-system accounts

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 4

MySQL Configuration

Chapter 4 — MySQL Configuration

A freshly installed MySQL runs on conservative defaults designed to work on any hardware. On your actual server, with a known amount of RAM and a specific workload, those defaults leave significant performance on the table — and also omit important diagnostics like the slow query log. This chapter covers finding your config files, the key settings that matter for a shared web server, and how to apply changes safely.

What this chapter covers: How MySQL's configuration file hierarchy works and where to find the right file on Ubuntu/Debian. Three ways to change a setting (file, `SET GLOBAL`, `SET PERSIST`) and when to use each. Validating config before restart. The single most important tuning setting: `innodb_buffer_pool_size`. Connection limits and timeout settings. The slow query log — the most valuable diagnostic tool in MySQL. Default character set. A complete annotated config file for a shared web+database server. Troubleshooting.

The Configuration File Hierarchy

MySQL reads multiple config files at startup. Options in later files override earlier ones. Knowing the order matters — editing the wrong file has no effect.

See the exact files MySQL reads, in priority order $ mysql --help 2>/dev/null | grep -A1 "Default options" Default options are read from the following files in the given order: /etc/my.cnf /etc/mysql/my.cnf ~/.my.cnf # For mysqld specifically (the server process) $ mysqld --help --verbose 2>/dev/null | grep -A1 "Default options" Default options are read from the following files in the given order: /etc/my.cnf /etc/mysql/my.cnf /etc/mysql/mysql.conf.d/*.cnf /etc/mysql/conf.d/*.cnf
/etc/mysql/ ├── my.cnf ← top-level file: just !include directives, don't edit directly ├── mysql.conf.d/ │ └── mysqld.cnf ← SERVER config — THIS is where you make changes on Ubuntu └── conf.d/ └── mysql.cnf ← CLIENT config (mysql client charset, etc.)
On Ubuntu, edit `/etc/mysql/mysql.conf.d/mysqld.cnf`. This is the primary MySQL server configuration file. The top-level `/etc/mysql/my.cnf` just contains `!includedir` lines that pull in the subdirectories — editing it directly achieves nothing for server settings. On Debian with the MySQL APT repo, the same file structure applies.

You can also add your own override files in a `conf.d` or `mysql.conf.d` drop-in directory rather than editing the main file — useful for keeping your customisations separate and easier to manage in version control.

Three Ways to Change a Setting

Edit mysqld.cnf
Add or change a line in **/etc/mysql/mysql.conf.d/mysqld.cnf**, then restart MySQL. Works for all settings — dynamic and static.
Permanent — survives restart
SET GLOBAL
Change a **dynamic** variable immediately without restart: `SET GLOBAL max_connections = 100;`. Good for testing a value. Takes effect instantly for new connections.
Temporary — lost on restart
SET PERSIST
MySQL 8.0+ only. Changes the variable now AND writes it to `/var/lib/mysql/mysqld-auto.cnf`, so it survives a restart. Best of both worlds for dynamic variables.
Permanent — survives restart
RESET PERSIST
Removes a persisted variable from `mysqld-auto.cnf`: `RESET PERSIST max_connections;`. Allows the config file value (or default) to take over again.
Removes the SET PERSIST override
Check the current value of any variable mysql> SHOW VARIABLES LIKE 'max_connections'; +-----------------+-------+ | Variable_name | Value | +-----------------+-------+ | max_connections | 151 | +-----------------+-------+ # Wildcard search — see all InnoDB variables mysql> SHOW VARIABLES LIKE 'innodb%'; # Change a dynamic variable immediately (and persistently, MySQL 8.0+) mysql> SET PERSIST max_connections = 75; Query OK, 0 rows affected (0.00 sec) # Confirm the change took effect right now mysql> SHOW VARIABLES LIKE 'max_connections'; +-----------------+-------+ | Value | 75 | +-----------------+-------+ # See all variables that have been SET PERSIST'd mysql> SELECT variable_name, variable_value, set_time -> FROM performance_schema.persisted_variables;

Validating config before restart

MySQL 8.0+ — validate the config file without starting the server $ sudo mysqld --validate-config 2026-06-15T10:30:01Z 0 [System] Config file is valid. # If there's an error, it tells you exactly what and on which line 2026-06-15T10:30:01Z 0 [ERROR] unknown variable 'innodb_buffer_pool_siz=1G' # Always validate BEFORE doing a systemctl restart on a production server $ sudo mysqld --validate-config && sudo systemctl restart mysql

A restart disconnects everyone. `systemctl restart mysql` closes all active database connections. If your website has a connection pool, connections will fail and reconnect after MySQL comes back up, typically within a few seconds. Static variables require a restart; dynamic ones changed via `SET GLOBAL` or `SET PERSIST` take effect immediately without disconnect.

The Most Important Setting: innodb_buffer_pool_size

InnoDB (MySQL's default storage engine) caches frequently-accessed table data and index pages in a memory pool called the **buffer pool**. If a query's data is in the buffer pool, it's served from RAM — microseconds. If it's not, MySQL reads from disk — milliseconds or worse. The buffer pool size is the single biggest lever for MySQL performance.

The default is **128 MB** — enough to run on a tiny VM, far too small for any real workload. For a shared web server running Apache, PHP, and MySQL, allocate **20–30% of total system RAM** to the buffer pool.

Check how much RAM your server has $ free -h total used free shared buff/cache available Mem: 7.7Gi 2.1Gi 1.2Gi 312Mi 4.4Gi 5.0Gi Swap: 2.0Gi 0B 2.0Gi # Example: 8 GB server. Apache + PHP use ~2 GB. Set buffer pool to 1-2 GB.
innodb_buffer_pool_size guide — shared web + database server
2 GB RAM
256M–512M
Leave headroom for OS + Apache + PHP
4 GB RAM
512M–1G
Comfortable for a personal site
8 GB RAM
1G–2G
Good balance for web+DB server
16 GB RAM
4G–6G
Enough to cache most databases in RAM
Check the current buffer pool size mysql> SHOW VARIABLES LIKE 'innodb_buffer_pool_size'; +-------------------------+-----------+ | Variable_name | Value | +-------------------------+-----------+ | innodb_buffer_pool_size | 134217728 | +-------------------------+-----------+ # 134217728 bytes = 128 MB default — almost certainly too small # Check how well the current buffer pool is performing mysql> SHOW STATUS LIKE 'Innodb_buffer_pool_read%'; +---------------------------------------+----------+ | Variable_name | Value | +---------------------------------------+----------+ | Innodb_buffer_pool_read_requests | 14523891 | | Innodb_buffer_pool_reads | 284 | +---------------------------------------+----------+ # Hit ratio = 1 - (reads / read_requests) = 1 - (284/14523891) = 99.998% ← excellent # If hit ratio is below 95%, the buffer pool is too small for your workload

Buffer pool size is a static variable on most MySQL builds — changing it requires a restart. In MySQL 8.0.14+ on systems with enough RAM, it can be changed online with `SET GLOBAL innodb_buffer_pool_size`, but this is a slow operation that resizes chunk by chunk in the background.

Connection Settings

Each MySQL connection consumes RAM — stack space, buffers, and per-connection memory allocations. Setting `max_connections` too high wastes memory; too low and legitimate connections are refused with "Too many connections."

See current connections and the historical maximum mysql> SHOW STATUS LIKE 'Max_used_connections'; +----------------------+-------+ | Variable_name | Value | +----------------------+-------+ | Max_used_connections | 12 | +----------------------+-------+ # If this peaked at 12, a max_connections of 151 is massively over-provisioned # How many connection attempts were rejected as "too many connections" mysql> SHOW STATUS LIKE 'Connection_errors_max_connections'; +----------------------------------+-------+ | Variable_name | Value | +----------------------------------+-------+ | Connection_errors_max_connections | 0 | +----------------------------------+-------+ # 0 = no one has ever been turned away — limit is fine, can reduce it

Variable	Default	Recommended (shared server)	Notes
max_connections	151	50–100	Each idle connection uses ~1 MB. Match to `Max_used_connections` + 20% headroom
wait_timeout	28800	300	Seconds before an idle non-interactive connection is closed. 28800 = 8 hours — far too long for web apps
interactive_timeout	28800	300	Same but for interactive clients (mysql CLI, phpMyAdmin). Set to same value as wait_timeout
max_allowed_packet	64M (8.0)	64M–256M	Max size of a single query/result packet. Increase if you store large blobs or get "packet too large" errors
connect_timeout	10	leave default	Seconds to wait for a client to authenticate. 10 seconds is fine

Per-connection memory buffers compound with max_connections. Variables like `sort_buffer_size`, `join_buffer_size`, and `read_buffer_size` are allocated per connection when needed. If you have 100 connections each using a 4 MB sort buffer simultaneously, that's 400 MB — just for sort operations. Leave these at their defaults unless you have a specific query pattern that needs them.

The Slow Query Log — Your Most Valuable Diagnostic Tool

MySQL can log every query that takes longer than a threshold you define. This log is the starting point for almost every performance investigation — it shows you exactly which queries are slow, how long they took, and how many rows they examined. Enable it from day one; the overhead on a small server is negligible.

Check if the slow query log is currently enabled mysql> SHOW VARIABLES LIKE 'slow_query%'; +---------------------+-------------------------------+ | Variable_name | Value | +---------------------+-------------------------------+ | slow_query_log | OFF | | slow_query_log_file | /var/log/mysql/mysql-slow.log | +---------------------+-------------------------------+

Enable it permanently by adding these lines to `mysqld.cnf`, then restart:

Slow query log settings (add under [mysqld]) slow_query_log = 1 slow_query_log_file = /var/log/mysql/slow.log long_query_time = 2 # log queries taking > 2 seconds log_queries_not_using_indexes = 1 # also log full table scans log_throttle_queries_not_using_indexes = 10 # max 10 no-index logs per minute

Or enable it right now without restarting — the slow query log is a dynamic variable:

mysql> SET PERSIST slow_query_log = 1; mysql> SET PERSIST long_query_time = 2; mysql> SET PERSIST log_queries_not_using_indexes = 1;

Reading the slow log

Each entry in the slow log shows the query's cost: how long it took, how many rows MySQL looked at, and how many it returned.

Time: 2026-06-15T14:22:33.182671Z # User@Host: bookshop_app[bookshop_app] @ localhost [] Id: 11 # Query_time: 4.823156 Lock_time: 0.000042 Rows_sent: 1 Rows_examined: 128430 SET timestamp=1749999753; SELECT * FROM books WHERE title LIKE '%adventure%'; # ↑ This query took 4.8 seconds, examined 128,430 rows, returned 1 row. # Rows_examined >> Rows_sent = full table scan — needs an index on title, # or better, a FULLTEXT index for substring searches.

Summarising the slow log with mysqldumpslow

Show the top 10 slowest queries (grouped by pattern) $ sudo mysqldumpslow -s t -t 10 /var/log/mysql/slow.log Reading mysql slow query log from /var/log/mysql/slow.log Count: 42 Time=4.82s (202s) Lock=0.00s (0s) Rows=1.0 (42), bookshop_app[bookshop_app]@localhost SELECT * FROM books WHERE title LIKE '%S%' Count: 7 Time=1.94s (13s) Lock=0.00s (0s) Rows=523.0 (3661), bookshop_app[bookshop_app]@localhost SELECT * FROM orders WHERE created_at > 'S' # -s t = sort by total time | -s c = sort by count | -t 10 = top 10 results # The first result ran 42 times, total 202 seconds — the one to fix first

Default Character Set and Collation

By default, MySQL 8.0 uses `utf8mb4` and `utf8mb4_0900_ai_ci` — but earlier versions and some custom installs use the 3-byte `utf8` which can't store emoji or certain CJK characters. Set these explicitly so every new database inherits the right encoding automatically.

Check current defaults mysql> SHOW VARIABLES LIKE 'character_set_server'; +----------------------+---------+ | Variable_name | Value | +----------------------+---------+ | character_set_server | utf8mb4 | +----------------------+---------+ mysql> SHOW VARIABLES LIKE 'collation_server'; +------------------+--------------------+ | Variable_name | Value | +------------------+--------------------+ | collation_server | utf8mb4_0900_ai_ci | +---

---------------+--------------------+ # utf8mb4_0900_ai_ci = accent insensitive, case insensitive (good default) # utf8mb4_unicode_ci = slightly slower but better multilingual sorting

Which collation to choose: `utf8mb4_0900_ai_ci` is the MySQL 8.0 default — faster and uses the Unicode 9.0 algorithm. `utf8mb4_unicode_ci` is the MySQL 5.x default and may be more appropriate if you need consistent sorting across old and new MySQL versions, or if you're following documentation written for MySQL 5.x. Either is fine for a single-server setup.

A Tuned Config for a Shared Web + Database Server

Here is a complete, annotated `mysqld.cnf` for a server that runs Apache, PHP, and MySQL together. Adjust the memory values to match your server's RAM using the guide above.

[mysqld] # # === Networking == pid-file = /var/run/mysqld/mysqld.pid socket = /var/run/mysqld/mysqld.sock datadir = /var/lib/mysql # Only listen on localhost — remote access controlled separately (Chapter 6) bind-address = 127.0.0.1 # # === Logging === log_error = /var/log/mysql/error.log # Slow query log — enable immediately slow_query_log = 1 slow_query_log_file = /var/log/mysql/slow.log long_query_time = 2 log_queries_not_using_indexes = 1 log_throttle_queries_not_using_indexes = 10 # # === Character set == character_set_server = utf8mb4 collation_server = utf8mb4_unicode_ci # # === Memory — ADJUST THESE for your server's RAM ================== # Formula for shared web+DB server: # innodb_buffer_pool_size = total_RAM * 0.25 innodb_buffer_pool_size = 1G # example: 4 GB server # For buffer pools > 8 GB, set instances to number of GB (max 64) innodb_buffer_pool_instances = 1 # 1 is fine for pools <= 8 GB # In-memory temp table size (per query — don't set too high) tmp_table_size = 64M max_heap_table_size = 64M # must match tmp_table_size # # === Connections == max_connections = 75 # check Max_used_connections status first wait_timeout = 300 # close idle connections after 5 minutes interactive_timeout = 300 # same for interactive clients max_allowed_packet = 64M # # === InnoDB reliability === # 1 = safest (flush to disk on every commit) — always use this # 2 = faster, but last ~1 second of transactions can be lost on crash innodb_flush_log_at_trx_commit = 1 # Each table in its own file — makes storage management easier innodb_file_per_table = ON # default in MySQL 8.0, just making it explicit
Apply the changes $ sudo mysqld --validate-config Config file is valid. $ sudo systemctl restart mysql $ sudo systemctl status mysql ● mysql.service - MySQL Community Server Active: active (running)

Binary Logging — Brief Overview

The binary log (`binlog`) records every data change on the server. It's required for replication and enables point-in-time recovery (PITR) — restoring a backup from Monday, then replaying binlog events up to the moment before a disaster on Wednesday afternoon.

mysql> SHOW VARIABLES LIKE 'log_bin'; +---------------+-------+ | Variable_name | Value | +---------------+-------+ | log_bin | ON | +---------------+-------+ # On Ubuntu 22.04+, binary logging is ON by default. Check binlog expiry: mysql> SHOW VARIABLES LIKE 'binlog_expire_logs_seconds'; +----------------------------+---------+ | Variable_name | Value | +----------------------------+---------+ | binlog_expire_logs_seconds | 2592000 | +----------------------------+---------+ # 2592000 seconds = 30 days. Fine for most setups. # Binary logs can accumulate — check disk usage: $ sudo du -sh /var/lib/mysql/binlog.*

Binary logging is on by default in MySQL 8.0 on Ubuntu. If disk space is tight and you don't need PITR or replication, you can disable it by adding `disable_log_bin` to `mysqld.cnf` — but think carefully before doing so. PITR is what lets you recover from "someone accidentally ran DELETE without WHERE" — a situation that happens sooner or later to everyone.

Verifying Your Config Changes

Check the buffer pool is now the new size mysql> SHOW VARIABLES LIKE 'innodb_buffer_pool_size'; +-------------------------+------------+ | Variable_name | Value | +-------------------------+------------+ | innodb_buffer_pool_size | 1073741824 | +-------------------------+------------+ # 1073741824 bytes = 1 GB ✓ # Confirm the slow log is enabled and writing to the right file mysql> SHOW VARIABLES LIKE 'slow_query%'; +---------------------+------------------------+ | Variable_name | Value | +---------------------+------------------------+ | slow_query_log | ON | | slow_query_log_file | /var/log/mysql/slow.log | +---------------------+------------------------+ # Generate a slow query to test the log is working mysql> SELECT SLEEP(3); +----------+ | SLEEP(3) | +----------+ | 0 | +----------+ # Check the slow log captured it $ sudo tail -20 /var/log/mysql/slow.log # Time: 2026-06-15T12:00:03.001234Z # Query_time: 3.001234 Lock_time: 0.000000 Rows_sent: 1 Rows_examined: 1 SELECT SLEEP(3); # ✓ Slow log is working # Confirm timeouts and max_connections mysql> SHOW VARIABLES WHERE variable_name IN -> ('max_connections', 'wait_timeout', 'max_allowed_packet', 'character_set_server');

Troubleshooting

MySQL fails to start after editing mysqld.cnf

Run `sudo mysqld --validate-config` first to see the exact error line — it will name the unknown or invalid variable. If MySQL is already not starting, check the error log: `sudo tail -30 /var/log/mysql/error.log`. Common causes: typo in variable name, wrong value format (wrote `1G` where the variable expects bytes), editing the wrong config file (changes in `/etc/mysql/my.cnf` itself have no effect — only files included from it do). To recover, revert your last change and restart.

innodb_buffer_pool_size change has no effect after restart

You may have a `SET PERSIST` override in `/var/lib/mysql/mysqld-auto.cnf` that takes precedence over `mysqld.cnf`. Check: `SELECT variable_name, variable_value FROM performance_schema.persisted_variables;`. If the old value appears there, clear it: `RESET PERSIST innodb_buffer_pool_size;` then restart. After resetting, the config file value will be used.

Slow query log file is not being created

MySQL can't write to the log file — usually a permissions issue. The `mysql` OS user must own the log directory. Check: `ls -la /var/log/mysql/` — should be `mysql:adm` or `mysql:mysql`. Fix: `sudo chown mysql:mysql /var/log/mysql/slow.log` or `sudo touch /var/log/mysql/slow.log && sudo chown mysql:mysql /var/log/mysql/slow.log`. Then `SET GLOBAL slow_query_log = OFF; SET GLOBAL slow_query_log = ON;` to make MySQL re-open the file.

ERROR 1040: Too many connections

`max_connections` has been reached. Immediate fix: `SET GLOBAL max_connections = 200;` (dynamic, takes effect immediately). Root cause: check `SHOW PROCESSLIST;` for Sleep connections piling up — if many show `Sleep` for a long time, reduce `wait_timeout` to close them sooner (e.g., `SET PERSIST wait_timeout = 60;`). The underlying problem is often a connection pool in your application not returning connections properly.

Out of memory / server becomes unresponsive after changing buffer pool size

Set `innodb_buffer_pool_size` too large — MySQL is competing with Apache and the OS for RAM, causing swapping. A swapping server is slower than one with a small buffer pool. Roll back: set the value to 25% of RAM instead of a higher percentage. Remember that each `max_connections` connection also allocates buffers — total MySQL memory ≈ `innodb_buffer_pool_size` + (`max_connections` × per-connection buffers). On a 2 GB server, keep the buffer pool at 256–512 MB.

Quick Reference — Chapter 4

Variable / Command	Purpose / Notes
/etc/mysql/mysql.conf.d/mysqld.cnf	Primary server config file on Ubuntu/Debian — this is where you make changes
sudo mysqld --validate-config	Check the config file for errors before restarting
SHOW VARIABLES LIKE 'pattern';	See current value of any MySQL variable (supports % wildcard)
SET PERSIST variable = value;	Change a dynamic variable immediately AND persist it across restarts (MySQL 8.0+)
RESET PERSIST variable;	Remove a persisted override so the config file value takes effect again
innodb_buffer_pool_size	Most important setting — set to 20–30% of RAM on shared servers. Requires restart.
max_connections	Cap total connections. Match to Max_used_connections status + 20% headroom
wait_timeout / interactive_timeout	Seconds before idle connections are closed. Set both to 300 on web servers
slow_query_log = 1	Enable the slow query log (dynamic — no restart needed)
long_query_time = 2	Log queries taking longer than 2 seconds
log_queries_not_using_indexes = 1	Also log full table scans — essential for finding missing indexes
sudo mysqldumpslow -s t -t 10 /var/log/mysql/slow.log	Summarise slow log — top 10 worst queries by total execution time
SHOW STATUS LIKE 'Innodb_buffer_pool_read%';	Buffer pool hit ratio — if below 95%, pool is too small
SHOW STATUS LIKE 'Max_used_connections';	Peak concurrent connections since last restart — use to size max_connections

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 5

Backup and Restore

Chapter 5 — Backup and Restore

Hardware fails. People run DELETE without a WHERE clause. Ransomware encrypts /var/lib/mysql. Migrations go wrong. All of these happen, and all of them feel impossible until the moment they happen to you. A working backup and restore process is the single most important operational habit you can build around a database — and it takes less than an hour to set up properly.

What this chapter covers: What mysqldump produces and how it works. The key flags that matter (and why `--single-transaction` is critical for InnoDB). Creating a dedicated backup user with minimal privileges. Credentials files — keeping passwords out of your shell history. Gzip-compressed timestamped dumps. Inspecting and verifying a dump file. Restoring from a dump — single database, all databases, and renaming during restore. A complete automated backup script with rotation. Cron scheduling. Offsite copies with rclone. Testing restores. Binary log and point-in-time recovery basics. Troubleshooting.

Before We Start: The 3-2-1 Rule

The **3-2-1 rule** is the minimum viable backup strategy: **3 copies** of your data, on **2 different media types**, with **1 copy offsite**. For a home server this translates to: a daily dump on the server itself, a second copy on an external drive or NAS, and a third copy in cloud storage.

A backup that lives on the same server as the database is not a real backup. Disk failure, ransomware, accidental `rm -rf`, or a corrupted RAID volume takes both the database and the backup simultaneously. The backup script in this chapter saves locally *and* syncs offsite.

Equally important: **an untested backup is not a backup**. A dump file that can't be restored is worthless. Once your automated backup is running, schedule a monthly test restore to a temporary database. The instructions for this are at the end of the chapter.

What mysqldump Produces

`mysqldump` connects to MySQL and generates a SQL script that, when executed, recreates the database from scratch — schema and data. The output is a plain text file of SQL statements: `DROP TABLE IF EXISTS`, `CREATE TABLE`, then batched `INSERT` statements for the data.

-- MySQL dump 10.13 Distrib 8.0.36, for Linux (x86_64) -- Host: localhost Database: bookshop -- Server version 8.0.36 -- -- Table structure for table `books` DROP TABLE IF EXISTS `books`; CREATE TABLE `books` (`id` int NOT NULL AUTO_INCREMENT, `title` varchar(255) NOT NULL, `author_id` int NOT NULL, `price` decimal(6,2) DEFAULT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4; -- -- Dumping data for table `books` LOCK TABLES `books` WRITE; INSERT INTO `books` VALUES (1,'The Great Gatsby',1,8.99), (2,'Dune',2,12.50), (3,'Neuromancer',3,9.99); UNLOCK TABLES; -- [... more tables ...] -- Dump completed on 2026-06-15 14:30:01 ↑ This final line is how you verify the dump completed successfully

Check for "Dump completed on" at the end. If mysqldump is interrupted — disk full, network drop, killed process — the dump file will be truncated and lack this line. Always verify this line exists before trusting a backup.

Creating a Dedicated Backup User

Don't run backups as root. Create a minimal backup account with only the privileges mysqldump actually needs.

$ sudo mysql # Create a backup-only user mysql> CREATE USER 'backup_user'@'localhost' IDENTIFIED BY 'BackupP@ss!9'; # Grant only what mysqldump requires mysql> GRANT SELECT, LOCK TABLES, SHOW VIEW, EVENT, TRIGGER -> ON *.* TO 'backup_user'@'localhost'; # For backing up all databases you also need RELOAD and PROCESS mysql> GRANT RELOAD, PROCESS ON *.* TO 'backup_user'@'localhost'; mysql> FLUSH PRIVILEGES; mysql> EXIT;

Why not just use root? If your backup script's credentials are ever exposed (read-world config file, leaked cron output in logs), a backup user with SELECT/LOCK can only read data. Root can drop every database, create new admin accounts, and read system tables containing password hashes.

Credentials File — Never Put Passwords in the Command Line

Running `mysqldump -u backup_user -pYourPassword bookshop > dump.sql` exposes the password in your shell history, in `ps aux` output, and potentially in cron logs. The correct approach is a credentials file.

Create the credentials file (as root, in a safe location) $ sudo nano /etc/mysql/backup.cnf
[client] user = backup_user password = BackupP@ss!9
Lock it down — only root can read this file $ sudo chmod 600 /etc/mysql/backup.cnf $ sudo chown root:root /etc/mysql/backup.cnf # Test it — no password prompt, no password in the command $ sudo mysqldump --defaults-file=/etc/mysql/backup.cnf bookshop | tail -3 UNLOCK TABLES; -- Dump completed on 2026-06-15 14:30:01

The Flags That Matter

Flag	What it does	Use?
--single-transaction	Takes a consistent snapshot of InnoDB tables using a transaction, avoiding table locks. Allows reads and writes to continue during the dump.	Always for InnoDB
--routines (-R)	Include stored procedures and functions in the dump. Not included by default.	Yes, if you use them
--events (-E)	Include scheduled events. Not included by default.	Yes, if you use them
--triggers	Include triggers. *Included by default* — shown here for clarity.	Default on — no action needed
--databases db1 db2	Dump multiple named databases. Includes CREATE DATABASE statements, so restore creates the database automatically.	For multi-db dumps
--all-databases (-A)	Dump every database, including mysql (user accounts). Good for full server backups.	For full server backup
--no-tablespaces	Omits tablespace CREATE statements. Required if your backup user doesn't have PROCESS privilege and you encounter INFORMATION_SCHEMA errors.	Add if you see errors
--set-gtid-purged=OFF	Suppress GTID comments in the dump. Needed when restoring between servers that use different GTID configurations.	Add if using replication
--lock-tables	Locks tables one by one before dumping. For MyISAM tables only. Blocks all writes during the dump.	Avoid — use --single-transaction instead

Running mysqldump

── Single database, uncompressed ──────────────────────────────── $ sudo mysqldump \ --defaults-file=/etc/mysql/backup.cnf \ --single-transaction \ --routines \ --events \ bookshop > /var/backups/mysql/bookshop.sql # ── Single database, gzip compressed (much smaller) ──────────────────── $ sudo mysqldump \ --defaults-file=/etc/mysql/backup.cnf \ --single-transaction \ --routines \ --events \ bookshop | gzip > /var/backups/mysql/bookshop.sql.gz # ── With a datestamp in the filename ──────────────────────────────────── $ STAMP=$(date +%Y-%m-%d_%H%M) $ sudo mysqldump \ --defaults-file=/etc/mysql/backup.cnf \ --single-transaction \ --routines --events \ bookshop | gzip > /var/backups/mysql/bookshop_${STAMP}.sql.gz # Result: bookshop_2026-06-15_1430.sql.gz # ── All databases (full server backup) ──────────────────────────── $ sudo mysqldump \ --defaults-file=/etc/mysql/backup.cnf \ --single-transaction \ --routines --events \ --all-databases | gzip > /var/backups/mysql/all_databases_${STAMP}.sql.gz # ── Specific tables only ── $ sudo mysqldump \ --defaults-file=/etc/mysql/backup.cnf \ --single-transaction \ bookshop books authors > /var/backups/mysql/bookshop_tables.sql # ── Verify the dump completed (check for the final line) ──────────── $ zcat /var/backups/mysql/bookshop_${STAMP}.sql.gz | tail -3 UNLOCK TABLES; -- Dump completed on 2026-06-15 14:30:01 # ── Check compressed file size ──────────────────────────────────── $ ls -lh /var/backups/mysql/ -rw-r--r-- 1 root root 284K Jun 15 14:30 bookshop_2026-06-15_1430.sql.gz # Suspiciously tiny? 0 bytes? — something went wrong

Restoring from a Dump

Restoring is the reverse operation — you pipe the SQL file back into the `mysql` client. The database must exist before you restore a single-database dump (one not made with `--databases`); it's created automatically for dumps made with `--databases` or `--all-databases`.

── Restore a single-database dump (database must exist first) ──── $ sudo mysql -e "CREATE DATABASE IF NOT EXISTS bookshop CHARACTER SET utf8mb4;" $ sudo mysql bookshop < /var/backups/mysql/bookshop.sql # ── Restore a compressed dump ── $ sudo mysql -e "CREATE DATABASE IF NOT EXISTS bookshop CHARACTER SET utf8mb4;" $ zcat /var/backups/mysql/bookshop_2026-06-15_1430.sql.gz | sudo mysql bookshop # ── Restore a --databases dump (creates the database automatically) $ zcat /var/backups/mysql/bookshop_2026-06-15_1430.sql.gz | sudo mysql # ── Restore all databases from a full server backup ──────────────── $ zcat /var/backups/mysql/all_databases_2026-06-15_1430.sql.gz | sudo mysql # ── Restore to a DIFFERENT

database name ——————————————————— $ sudo mysql -e "CREATE DATABASE bookshop_restored CHARACTER SET utf8mb4;" $ zcat /var/backups/mysql/bookshop_2026-06-15_1430.sql.gz \ | sudo mysql bookshop_restored # Useful for: testing a restore without touching the live database # ── Monitor restore progress on large databases ———————————————— # (pv = pipe viewer, shows throughput and ETA) $ sudo apt install pv $ pv /var/backups/mysql/bookshop.sql | sudo mysql bookshop

Restoring into an existing database appends to or overwrites existing data. Because a single-table dump includes `DROP TABLE IF EXISTS` before each `CREATE TABLE`, restoring will DROP and recreate every table in the dump — losing any data added after the backup was taken. If you need to keep current data, restore to a temporary database name first, then selectively move rows.

The Automated Backup Script

Setup Walkthrough · Automated Daily Backups

Create a backup directory, write the script, test it, then schedule it with cron.

1

Create the backup directory with secure permissions.

$ sudo mkdir -p /var/backups/mysql $ sudo chown root:root /var/backups/mysql $ sudo chmod 700 /var/backups/mysql # Only root can read/write this directory

2

Write the backup script.

$ sudo nano /usr/local/sbin/mysql_backup.sh

3

Script contents — paste this and adjust the variables at the top.

#!/bin/bash # ── mysql_backup.sh — daily MySQL backup with rotation ——————————— # Adjust the variables below for your setup. # ── Configuration ———————————————————————————————— DEFAULTS_FILE="/etc/mysql/backup.cnf" # credentials file from earlier BACKUP_DIR="/var/backups/mysql" DATABASES=("bookshop") # add more: ("bookshop" "blog" "shop") KEEP_DAYS=7 # keep 7 days of local backups LOG_FILE="/var/log/mysql_backup.log" # ── Setup —— STAMP=$(date +%Y-%m-%d_%H%M) ERRORS=0 log() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"; } log "=== Backup started ===" # ── Dump each database ——————————————————————————— for DB in "${DATABASES[@]}"; do OUTFILE="${BACKUP_DIR}/${DB}_${STAMP}.sql.gz" log "Dumping database: $DB → $OUTFILE" mysqldump \ --defaults-file="$DEFAULTS_FILE" \ --single-transaction \ --routines \ --events \ "$DB" | gzip > "$OUTFILE" # Verify the dump completed (check for "Dump completed" line) if zcat "$OUTFILE" 2>/dev/null | tail -2 | grep -q "Dump completed"; then SIZE=$(du -sh "$OUTFILE" | cut -f1) log " ✓ $DB: OK ($SIZE)" else log " ✗ $DB: FAILED or incomplete — dump may be corrupted" ERRORS=$((ERRORS + 1)) fi done # ── Rotate old backups (delete files older than KEEP_DAYS) ———————— log "Rotating backups older than ${KEEP_DAYS} days..." find "$BACKUP_DIR" -name "*.sql.gz" -mtime +${KEEP_DAYS} -delete log "Rotation complete." # ── Summary ————————————————————————————————————— if ["$ERRORS" -eq 0]; then log "=== Backup completed successfully ===" else log "=== Backup completed with $ERRORS ERROR(S) — check log ===" exit 1 # non-zero exit lets cron/monitoring detect failure fi

4

Make it executable and test it manually before scheduling.

$ sudo chmod 700 /usr/local/sbin/mysql_backup.sh $ sudo /usr/local/sbin/mysql_backup.sh [2026-06-15 14:30:00] === Backup started === [2026-06-15 14:30:00] Dumping database: bookshop → /var/backups/mysql/bookshop_2026-06-15_1430.sql.gz [2026-06-15 14:30:01] ✓ bookshop: OK (46K) [2026-06-15 14:30:01] Rotating backups older than 7 days... [2026-06-15 14:30:01] Rotation complete. [2026-06-15 14:30:01] === Backup completed successfully === $ ls -lh /var/backups/mysql/ -rw-r--r-- 1 root root 46K Jun 15 14:30 bookshop_2026-06-15_1430.sql.gz

5

Schedule with cron — run daily at 2 AM.

$ sudo crontab -e # Add this line: 0 2 * * * /usr/local/sbin/mysql_backup.sh

Format: `minute hour day month weekday command`. `0 2 * * *` = minute 0, hour 2 (2 AM), every day.

6

Verify cron is running the job.

$ sudo tail -f /var/log/mysql_backup.log # After 2 AM the next day, you should see a new backup entry here # Also check cron's own log for errors $ grep "mysql_backup" /var/log/syslog | tail -5 Jun 16 02:00:01 webserver CRON[8422]: (root) CMD (/usr/local/sbin/mysql_backup.sh)

The script exits with code 1 on failure. If you set up logwatch (Chapter 8 of the Web Security course), it will catch cron job failures in its daily report.

Offsite Copies with rclone

Local backups protect against accidental deletion and software errors. An offsite copy protects against hardware failure, fire, theft, and ransomware. `rclone` can sync your backup directory to any S3-compatible storage, Backblaze B2, Google Drive, Dropbox, or a remote SSH server.

Install rclone $ sudo apt install rclone # Configure a remote (interactive wizard — run as your own user, not root) $ rclone config # Follow the wizard to set up your storage provider (Backblaze B2 shown below): # n → new remote # name: b2 # storage type: 5 (Backblaze B2) # account: your_account_id # key: your_application_key # → leave other settings as default # Test the connection and list your buckets $ rclone lsd b2: # Copy local backups to the remote (dry run first) $ rclone copy --dry-run /var/backups/mysql b2:your-bucket-name/mysql-backups/ # If dry run looks right, run it for real $ rclone copy /var/backups/mysql b2:your-bucket-name/mysql-backups/

Add the rclone sync to your backup script, after the local dumps complete:

Add to mysql_backup.sh, after the rotation block: log "Syncing backups to offsite storage..." if rclone copy "$BACKUP_DIR" b2:your-bucket-name/mysql-backups/ \ --log-file="$LOG_FILE" --log-level=INFO; then log "Offsite sync: OK" else log "Offsite sync: FAILED — check rclone config and connectivity" ERRORS=$((ERRORS + 1)) fi

Testing Your Backups — The Restore Test

Once a month, restore your most recent backup to a test database and verify the data looks correct. This confirms the dump is valid and the restore procedure works — before you actually need it in an emergency.

Find the most recent backup $ ls -lt /var/backups/mysql/*.sql.gz | head -3 -rw-r--r-- 1 root root 46K Jun 15 02:00 bookshop_2026-06-15_0200.sql.gz # Restore to a test database (leave the real database untouched) $ sudo mysql -e "DROP DATABASE IF EXISTS bookshop_test; \ CREATE DATABASE bookshop_test CHARACTER SET utf8mb4;" $ zcat /var/backups/mysql/bookshop_2026-06-15_0200.sql.gz \ | sudo mysql bookshop_test # Verify data looks right $ sudo mysql -e "USE bookshop_test; SHOW TABLES; SELECT COUNT(*) FROM books;" +------------------------+ | Tables_in_bookshop_test| +------------------------+ | authors | | books | | customers | | orders | +------------------------+ +----------+ | COUNT(*) | +----------+ | 127 | +----------+ # Clean up the test database $ sudo mysql -e "DROP DATABASE bookshop_test;"

Binary Log and Point-in-Time Recovery

A mysqldump captures the database as it was at the moment of the dump. Binary logs record every change made after that point. Together they enable **point-in-time recovery** (PITR): restore a Monday backup, then replay Wednesday morning's binlog events up to the exact moment before someone ran `DELETE FROM orders;` without a WHERE clause.

Check if binary logging is on (it should be by default on Ubuntu 22.04+) mysql> SHOW VARIABLES LIKE 'log_bin'; +---------------+-------+ | Variable_name | Value | +---------------+-------+ | log_bin | ON | +---------------+-------+ # List current binary log files mysql> SHOW BINARY LOGS; +------------------+-----------+ | Log_name | File_size | +------------------+-----------+ | binlog.000001 | 1048576 | | binlog.000002 | 209712 | +------------------+-----------+ # For a PITR scenario: restore the mysqldump first, then replay binlog up to # a point in time, skipping the bad query # # 1. Restore from dump: $ zcat bookshop_2026-06-15_0200.sql.gz | sudo mysql bookshop # # 2. Find the position of the bad query in the binlog: $ sudo mysqlbinlog /var/lib/mysql/binlog.000002 | grep -A5 "DELETE FROM orders" # # 3. Replay up to just BEFORE the bad query (by position): $ sudo mysqlbinlog --stop-position=28774 /var/lib/mysql/binlog.000002 \ | sudo mysql bookshop # # 4. Then replay AFTER the bad query (skip it): $ sudo mysqlbinlog --start-position=28920 /var/lib/mysql/binlog.000002 \ | sudo mysql bookshop
Include the binlog position in your dumps. Add `--master-data=2` (MySQL 5.7) or `--source-data=2` (MySQL 8.0) to your mysqldump command — it adds a comment near the top of the dump file showing exactly which binlog file and position the dump corresponds to. This is the starting point for PITR.

Backup Tool Comparison

mysqldump
Single-threaded SQL export. Human-readable output, easy to inspect, universally compatible. Slightly slower on large databases. Ships with MySQL.
Recommended for most setups
mysqlpump
Parallel version of mysqldump. Faster for large databases with multiple tables. Being deprecated in MySQL 8.4+ — Oracle recommends MySQL Shell instead.
Avoid for new setups
MySQL Shell dump

Parallel, chunked, fastest option. Uses `util.dumpInstance()` or `util.dumpSchemas()`. Requires MySQL Shell to be installed. Not a SQL file — uses a proprietary format (load with `util.loadDump()`).
For very large databases

Recommended Backup Strategy for a Home Server

Daily — 2 AM
Full mysqldump
All application databases, gzip compressed, timestamped. Local retention: 7 days.
Daily — after dump
Offsite sync
rclone copy to B2/S3/SFTP. Offsite retention: 30 days.
Weekly
Review log
Check /var/log/mysql_backup.log for errors. Verify recent backup file sizes look reasonable.
Monthly
Test restore
Restore latest dump to bookshop_test, verify row counts, drop test db. Takes under 5 minutes.

Troubleshooting

mysqldump: Couldn't execute 'SELECT COLUMN_STATISTICS...' — Unknown table 'COLUMN_STATISTICS'
This happens when a dump made with MySQL 8.0 is being fed to an older MySQL 5.7 client or when the mysql client and server versions mismatch. Quick fix: add `--column-statistics=0` to the mysqldump command. Or ensure the mysql client matches the server version: `mysql --version` should match `mysqld --version`.
mysqldump hangs / never finishes
mysqldump is waiting for a table lock that another query holds. Check: `SHOW PROCESSLIST;` — look for rows with a long-running query in the State column. If another process is holding a lock, `KILL <id>;` it. Long-term fix: always use `--single-transaction` for InnoDB tables — it uses a snapshot and doesn't need table locks. If the hang is on a MyISAM table, schedule the backup when traffic is low.
Restore fails: ERROR 1007 — Can't create database, database exists
The dump was made with `--databases` or `--all-databases`, so it includes a `CREATE DATABASE` statement — and that database already exists. Either drop it first (`DROP DATABASE bookshop;` — careful, data loss!) or add `--force` to the mysql restore command to skip errors. If you want to restore to the same database without dropping it first, use a single-database dump (without `--databases`) and specify the database name in the restore command.
Dump file is 0 bytes or ends abruptly without "Dump completed"
The dump failed silently. Common causes: disk full during dump (`df -h /var/backups`), mysqldump lost the connection partway through, or the backup user lacked privileges on a table. Check: `sudo mysqldump --defaults-file=/etc/mysql/backup.cnf bookshop 2>&1 | tail -20` — run without gzip and pipe to tail to see any error messages. Fix the root cause, then rerun. Add `set -e` to your backup script to make it exit on any error.
Cron backup runs but produces no file (script works fine when run manually)
Cron runs with a minimal environment — it may not find `mysqldump` if it's not on the default PATH. Fix by using the full path in the cron job or script: `/usr/bin/mysqldump` instead of just `mysqldump`. Also confirm the cron job is running as root (use `sudo crontab -e` to edit root's crontab, not your user's). Check the cron log: `grep mysql_backup /var/log/syslog`.

Quick Reference — Chapter 5

Command	Purpose
mysqldump --defaults-file=/etc/mysql/backup.cnf --single-transaction --routines --events db | gzip > db.sql.gz	Full production-quality dump of one database, compressed
mysqldump ... --all-databases | gzip > all.sql.gz	Full server backup including all databases and user accounts
zcat db.sql.gz | tail -3	Verify the dump completed — look for "Dump completed on" in the last lines
zcat db.sql.gz | sudo mysql dbname	Restore a compressed single-database dump (database must already exist)

Command	Purpose
sudo mysql -e "CREATE DATABASE dbname CHARACTER SET utf8mb4;"	Create the target database before restoring a single-database dump
find /var/backups/mysql -name "*.sql.gz" -mtime +7 -delete	Delete backups older than 7 days
rclone copy /var/backups/mysql b2:bucket/mysql/	Sync local backups to Backblaze B2 (or any rclone remote)
sudo crontab -e	Edit root's crontab to schedule the backup script
SHOW BINARY LOGS;	List binary log files on the server (needed for PITR)
mysqlbinlog --stop-position=N binlog.000002 | sudo mysql db	Replay binlog events up to a specific position (point-in-time recovery)
tail -f /var/log/mysql_backup.log	Monitor backup job output

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 6

Securing Remote Access

Chapter 6 — Securing Remote Access

The fastest way to get your MySQL server compromised is to expose port 3306 to the internet with `bind-address = 0.0.0.0` and no firewall. Bots scan for it continuously — if they find it, they'll attempt to brute-force every account within minutes. This chapter covers the right way to think about remote access: restricting what MySQL listens on, layering firewall rules, enforcing SSL, and — for home servers — the much better alternative of an SSH tunnel that opens nothing to the internet at all.

What this chapter covers: Checking what MySQL is actually listening on. The three-layer security model (bind-address + UFW + MySQL user@host). The `bind-address` options and which to use. UFW rules scoped to specific IPs. SSL/TLS status in MySQL 8.0 (on by default), verifying encryption is in use, and enforcing it per-user. The SSH tunnel — why it's the right choice for home servers wanting GUI tool access, with a complete walkthrough. Three practical scenarios covering local-only, LAN access, and remote access. Troubleshooting.

First Question: Do You Actually Need Remote Access?

Before configuring anything, decide whether remote access to MySQL is genuinely needed. Most web servers run the application and the database on the same machine — PHP connects to MySQL via the Unix socket, and `bind-address = 127.0.0.1` is the correct and secure configuration. Remote access is only needed when:

- **Your application runs on a separate server** — a dedicated app tier connecting to a dedicated database server
- **You want to use a GUI tool** like MySQL Workbench from your development machine
- **You're running replication** — a replica server reading from the primary
- **A backup script on a different machine** needs to dump the database

If none of these apply, leave `bind-address = 127.0.0.1` and skip the rest of this chapter — you're already in the most secure configuration possible.

For single-server home setups: if you just want to use MySQL Workbench from your Windows/Mac machine, the SSH tunnel (covered later in this chapter) gives you that without opening any port to the internet. It's the recommended approach.

Checking What MySQL Is Currently Listening On

See which interfaces and ports MySQL is listening on $ sudo ss -tlnp | grep mysql State Recv-Q Send-Q Local Address:Port Peer Address:Port Process LISTEN 0 70 127.0.0.1:3306 0.0.0.0:* users:(("mysqld",pid=1234)) LISTEN 0 70 127.0.0.1:33060 0.0.0.0:* users:(("mysqld",pid=1234)) # 127.0.0.1 = localhost only — not accessible from outside. Good. # 33060 = MySQL X protocol port (used by MySQL Shell) # Compare with what SHOULDN'T be there (bound to all interfaces): LISTEN 0 70 0.0.0.0:3306 0.0.0.0:* users:(("mysqld",pid=1234)) # 0.0.0.0 = listening on ALL interfaces — reachable from the internet # Also check via MySQL itself $ sudo mysql -e "SHOW VARIABLES LIKE 'bind_address';" +--------------+-----------+ | Variable_name| Value | +--------------+-----------+ | bind_address | 127.0.0.1 | +--------------+-----------+

The Three-Layer Security Model

Remote MySQL access is only possible when all three layers permit it. Each layer is independently controlled. Missing any one of them blocks the connection.

1
bind-address — which interfaces MySQL listens on
Set in **mysqld.cnf**. `127.0.0.1` = local only, `0.0.0.0` = all interfaces (internet-accessible). This is the first gate — if MySQL isn't listening on an interface, no connection from that network is even possible.
↓
2
UFW — which source IPs can reach port 3306
Set with **ufw allow**. Even if MySQL is listening on all interfaces, UFW drops packets from addresses not explicitly permitted. Always scope rules to specific source IPs — never `ufw allow 3306` without a source restriction.
↓
3

MySQL user@host — which accounts authenticate from which hosts
Set with **CREATE USER / GRANT**. A connection that passes the firewall still needs a MySQL account whose `@host` matches the source IP. `'user'@'192.168.1.50'` will only authenticate from that exact IP — all others get access denied.
Remote machine (192.168.1.50) trying to connect to MySQL: [192.168.1.50] ──► port 3306 ──► Layer 1: bind-address? │ 0.0.0.0 → pass │ 127.0.0.1 → DROP (never reaches UFW) ▼ Layer 2: UFW rule? │ allow from 192.168.1.50 → pass │ no rule → DROP ▼ Layer 3: MySQL user@host? │ 'user'@'192.168.1.50' → AUTHENTICATED │ no match → ACCESS DENIED All three must say YES for a connection to succeed.

Configuring bind-address

127.0.0.1
Listen on loopback only. Only connections from the same machine (via Unix socket or TCP loopback) are accepted.
Default — use for single-server setups
0.0.0.0
Listen on all IPv4 interfaces. Reachable from the internet — must be combined with tight UFW rules and MySQL user@host restrictions.
Only if remote access is needed
192.168.1.10
Listen on a specific interface (your LAN IP). Unreachable from the internet even without UFW — only accessible from the local network.
For LAN-only access (most secure)
::
Listen on all IPv4 and IPv6 interfaces. Use with care — IPv6 may bypass firewall rules you wrote thinking only about IPv4.
Only if you need IPv6 remote access

Change it in `/etc/mysql/mysql.conf.d/mysqld.cnf`:

[mysqld] # Local only (default, most secure) bind-address = 127.0.0.1 # LAN access only (bind to your server's LAN IP — find it with: ip addr show) # bind-address = 192.168.1.100 # All interfaces (needed for public remote access — must use with UFW + user@host) # bind-address = 0.0.0.0
After editing the config, validate and restart $ sudo mysqld --validate-config && sudo systemctl restart mysql # Confirm the new bind address is active $ sudo ss -tlnp | grep 3306
Find your server's LAN IP first. Before setting a specific LAN IP as the bind address, confirm it: `ip addr show | grep 'inet '`. Use the address on your LAN interface (typically `eth0`, `ens3`, or similar — not `lo`). If that IP changes (DHCP), MySQL won't start next time. Set a static LAN IP on your server or router before using this approach.

UFW Rules for MySQL

Never run `ufw allow 3306` without a source restriction. This opens MySQL to the entire internet. Bots scan for port 3306 continuously. Always scope the rule to specific IPs or subnets.
Allow MySQL from a specific LAN machine only $ sudo ufw allow from 192.168.1.50 to any port 3306 comment 'MySQL - dev workstation' # Allow from an entire LAN subnet $ sudo ufw allow from 192.168.1.0/24 to any port 3306 comment 'MySQL - LAN only' # Allow from a specific remote IP (e.g. a known VPS or office IP) $ sudo ufw allow from 203.0.113.42 to any port 3306 comment 'MySQL - office IP' # Verify rules were added $ sudo ufw status numbered | grep 3306 To Action From -- ------ ---- [5] 3306 ALLOW IN 192.168.1.0/24 # Remove a rule if you no longer need it (use the number from ufw status numbered) $ sudo ufw delete 5

SSL/TLS — Encrypting the Connection

Even on a private LAN, credentials and query results travel in plain text if the connection isn't encrypted. MySQL 8.0 solves this by generating self-signed SSL certificates automatically at first startup and enabling SSL by default — no configuration required. This section shows how to verify encryption is working and how to enforce it.

Checking SSL status

Check that SSL is available on the server mysql> SHOW VARIABLES LIKE 'have_ssl'; +---------------+-------+ | Variable_name | Value | +---------------+-------+ | have_ssl | YES | +---------------+-------+ # Check where the auto-generated certificates live mysql> SHOW VARIABLES LIKE '%ssl_cert%'; +-----------------+-----------------------------+ | Variable_name | Value | +-----------------+-----------------------------+ | ssl_cert | /var/lib/mysql/server-cert.pem| +-----------------+-----------------------------+ # Check if YOUR CURRENT connection is encrypted mysql> SHOW STATUS LIKE 'Ssl_cipher'; +---------------+----------------------------+ | Variable_name | Value | +---------------+----------------------------+ | Ssl_cipher | TLS_AES_256_GCM_SHA384 | +---------------+----------------------------+ # A non-empty cipher = encrypted connection ✓ # Empty Ssl_cipher = unencrypted connection

Socket connection (sudo mysql)
Local Unix socket connections skip SSL — there's nothing to encrypt on a socket. `Ssl_cipher` will be empty. **This is fine** — socket connections never leave the machine.
Remote TCP connection without SSL
`Ssl_cipher` empty on a TCP connection from another machine = credentials and data are in plain text on the wire. Add `--ssl-mode=REQUIRED` to enforce SSL for remote clients.

Connecting with SSL from a remote client

Connect from a remote machine with SSL required $ mysql -u bookshop_app -p -h 192.168.1.100 --ssl-mode=REQUIRED # --ssl-mode options: # DISABLED = never use SSL # PREFERRED = use SSL if available, plain text otherwise (default) # REQUIRED = always use SSL, fail if unavailable # VERIFY_CA = SSL + verify the server certificate against a CA # VERIFY_IDENTITY = SSL + verify CA + verify hostname matches cert # With self-signed certs (MySQL's auto-generated default): # REQUIRED works — encrypts the connection # VERIFY_CA and VERIFY_IDENTITY will FAIL unless you distribute the CA cert # Confirm SSL is active on the remote connection mysql> SHOW STATUS LIKE 'Ssl_cipher'; +---------------+------------------------+ | Ssl_cipher | TLS_AES_256_GCM_SHA384 | +---------------+------------------------+

Enforcing SSL for specific users

Require SSL for any connection from this user account # If a client connects without SSL, the connection is rejected mysql> ALTER USER 'bookshop_app'@'192.168.1.50' REQUIRE SSL; Query OK, 0 rows affected (0.00 sec) # Create a new remote user with SSL required from the start mysql> CREATE USER 'remote_admin'@'192.168.1.50' -> IDENTIFIED BY 'RemoteP@ss!5' -> REQUIRE SSL; # Check what a user REQUIRE setting is mysql> SHOW CREATE USER 'bookshop_app'@'192.168.1.50'\G CREATE USER 'bookshop_app'@'192.168.1.50' IDENTIFIED WITH caching_sha2_password AS '...' REQUIRE SSL PASSWORD EXPIRE DEFAULT ... # Remove the SSL requirement (back to optional) mysql> ALTER USER 'bookshop_app'@'192.168.1.50' REQUIRE NONE;
In PHP/PDO, enable SSL for MySQL connections with the `PDO::MYSQL_ATTR_SSL_CA` option or by setting `ssl_mode=REQUIRED` in the DSN. Without this, even if MySQL requires SSL for the account, PHP may send a plain-text connection and get rejected.

The SSH Tunnel — The Right Answer for Home Servers

Opening port 3306 to the internet, even with UFW restrictions and SSL, adds a persistent attack surface. If your MySQL is only accessed remotely by you — from your dev machine, for administration or using MySQL Workbench — an SSH tunnel gives you full access with **no port opened**, no firewall rules for MySQL, and no credentials exposed. The tunnel forwards a local port on your machine through your SSH connection to MySQL on the server.

Without SSH tunnel (port 3306 open to internet): [Your machine] ──► internet ──► port 3306 ──► MySQL ↑ exposed to bots/scanners With SSH tunnel (nothing open except port 22): [Your machine:3307] ──► SSH (port 22) ──► [Server: localhost:3306] ↑ fully encrypted, protected by SSH keys MySQL sees the connection as coming from 127.0.0.1 (localhost)
Setup Walkthrough · SSH Tunnel to MySQL
Access MySQL remotely via SSH — no port 3306 opening required.
1
Prerequisites: SSH access to the server is working (port 22 open, your SSH key is installed). MySQL's `bind-address` stays at `127.0.0.1` — no changes needed to MySQL config.
2
Open the tunnel from your dev machine (Linux, macOS, or WSL on Windows).

Basic tunnel: forward local port 3307 → server's localhost:3306 via SSH $ ssh -L 3307:127.0.0.1:3306 philip@your-server.example.com -N # Flags: # -L 3307:127.0.0.1:3306 = local port 3307 → remote 127.0.0.1:3306 # -N = don't run a command (tunnel only, no shell) # Leave this terminal open while you need the tunnel # Background version (runs without occupying a terminal) $ ssh -L 3307:127.0.0.1:3306 philip@your-server.example.com -fN # -f = fork to background after authentication
3
Connect MySQL client to the tunnel (while the tunnel is running).

Connect as if MySQL is running locally on port 3307 $ mysql -u bookshop_app -p -h 127.0.0.1 -P 3307 # MySQL sees this connection as coming from 127.0.0.1 (localhost) # so the user account must be bookshop_app@'localhost' or bookshop_app@'127.0.0.1'
4
Connect MySQL Workbench via SSH tunnel (no terminal needed — Workbench handles it).

In MySQL Workbench, create a new connection: # Connection Method: Standard TCP/IP over SSH # # SSH Hostname: your-server.example.com:22 # SSH Username: philip # SSH Key File: C:\Users\philip\.ssh\id_ed25519 (your private key) # # MySQL Hostname: 127.0.0.1

MySQL Port: 3306 # Username: bookshop_app # Password: (stored in vault or entered at connect time) # # Workbench opens the tunnel automatically when you connect
5
Add a shortcut to your SSH config so you don't type the full command each time.

Add to ~/.ssh/config on your dev machine: Host mysql-tunnel HostName your-server.example.com User philip IdentityFile ~/.ssh/id_ed25519 LocalForward 3307 127.0.0.1:3306 # Then open the tunnel with just: $ ssh mysql-tunnel -N
No changes to MySQL config, no UFW rules for port 3306, no new attack surface. The tunnel is authenticated entirely by your SSH key.

Three Practical Scenarios

Scenario A — Recommended for most single-server setups
Local access only — web app and MySQL on the same server
bind-address stays at `127.0.0.1` (default).
UFW: no rule for port 3306 — not needed.
User accounts: `'app_user'@'localhost'` — already correct.
SSL: not needed for socket connections.

This is the most secure configuration. PHP connects via Unix socket (`host=localhost` in DSN). Nothing listens on a network interface for MySQL. Zero attack surface.
Scenario B — LAN access from a dev workstation
Open MySQL to your home/office network only
bind-address: set to your server's static LAN IP (e.g. `192.168.1.100`). This only exposes MySQL on the LAN interface — not to the internet even without UFW.

UFW: `sudo ufw allow from 192.168.1.0/24 to any port 3306 comment 'MySQL LAN'`

User account: `CREATE USER 'dev_user'@'192.168.1.%' IDENTIFIED BY '...' REQUIRE SSL;`

SSL: enforce with `REQUIRE SSL` on the account and `--ssl-mode=REQUIRED` on the client.

Only reachable from inside your LAN. If your server is behind a home router, it's not directly internet-accessible anyway — but binding to the LAN IP and using UFW provides defence in depth.
Scenario C — Recommended for remote admin access
Remote access via SSH tunnel — nothing opened to the internet
bind-address: stays at `127.0.0.1` (no changes).
UFW: no rule for port 3306 — not needed.
User accounts: `'admin'@'127.0.0.1'` — the tunnel makes remote connections appear local.
SSL: not needed — the SSH channel is already encrypted.

To connect: `ssh -L 3307:127.0.0.1:3306 philip@server.example.com -N`, then connect MySQL to `127.0.0.1:3307`. MySQL Workbench supports this natively via "Standard TCP/IP over SSH".

Zero attack surface for MySQL. The only thing exposed is SSH (port 22), which is already protected by key-based auth and SSH hardening from the Security course.

Troubleshooting

Can't connect remotely even though bind-address is 0.0.0.0
Check all three layers in order. **Layer 1:** `sudo ss -tlnp | grep 3306` — does it show `0.0.0.0:3306`? If not, restart MySQL after saving the config change. **Layer 2:** `sudo ufw status` — is there a rule allowing the source IP on port 3306? **Layer 3:** `SELECT user, host FROM mysql.user WHERE user='youruser';` — does the `host` column match the connecting IP (exactly or via wildcard)? All three must pass.
MySQL won't start after changing bind-address
The IP you specified in `bind-address` doesn't exist on any network interface on the server. Check available IPs: `ip addr show`. If you set a static LAN IP that hasn't been assigned yet, or your DHCP IP changed, MySQL can't bind to it. Either fix the network config to assign that IP statically, or change `bind-address` back to `127.0.0.1`. Check the error log: `sudo tail -20 /var/log/mysql/error.log`.
SSH tunnel opens but MySQL connection through it says "Access denied"

The tunnel makes the connection appear to come from `127.0.0.1` (loopback), not from your actual dev machine IP. The MySQL user account must have `@'localhost'` or `@'127.0.0.1'` as its host — not `@'192.168.1.50'`. Check: `SELECT user, host FROM mysql.user WHERE user='youruser';`. If necessary: `CREATE USER 'youruser'@'127.0.0.1' IDENTIFIED BY 'pass';` and grant the same privileges.

SSL required but getting "SSL connection error: unknown error number"

Most common cause: TLS version mismatch. MySQL 8.0 defaults to TLS 1.2 and 1.3; very old MySQL clients or PHP builds may only support TLS 1.0. Check the MySQL server's TLS configuration: `SHOW VARIABLES LIKE 'tls_version';`. Also check the client side: for PHP, ensure OpenSSL is recent. A quick diagnostic: `mysql -u user -p -h host --ssl-mode=REQUIRED --tls-version=TLSv1.2` — if this works, it's a TLS 1.3 compatibility issue on the client.

UFW rule added but connection from that IP is still refused

Check that UFW is actually enabled: `sudo ufw status` — should say "Status: active". Also verify the rule direction: `ufw allow from` creates an ALLOW IN rule (for incoming connections), which is what you want. Check the rule is listed: `sudo ufw status numbered | grep 3306`. If the server is behind a NAT router (e.g. at home), the connecting machine's IP visible to the server is the LAN IP, not the internet IP — make sure your UFW rule matches the LAN IP.

Quick Reference — Chapter 6

Command / Setting	Purpose
sudo ss -tlnp | grep mysql	Check which interfaces and ports MySQL is listening on
bind-address = 127.0.0.1	Listen on localhost only — safest for single-server setups (in mysqld.cnf)
bind-address = 0.0.0.0	Listen on all interfaces — only use if remote access is needed, must combine with UFW
sudo ufw allow from 192.168.1.50 to any port 3306	Open MySQL to a specific IP only — always scope to source IP, never bare "allow 3306"
SHOW STATUS LIKE 'Ssl_cipher';	Check if the current connection is SSL encrypted (empty = not encrypted)
SHOW VARIABLES LIKE 'have_ssl';	Confirm SSL is available on the server (YES = auto-generated certs in /var/lib/mysql)
ALTER USER 'u'@'h' REQUIRE SSL;	Force SSL for a specific user — connections without SSL are rejected
mysql --ssl-mode=REQUIRED -h host -u user -p	Connect with SSL enforced (client-side flag)
ssh -L 3307:127.0.0.1:3306 user@server -N	Open SSH tunnel: connect locally on port 3307 to reach remote MySQL on 3306
mysql -h 127.0.0.1 -P 3307 -u user -p	Connect to MySQL through the SSH tunnel (run after the tunnel is open)
SHOW CREATE USER 'u'@'h'\G	View a user's SSL requirements and other account settings

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 7

Monitoring and Performance Basics

Chapter 7 — Monitoring and Performance Basics

A slow website often traces back to a slow database query. A crashed application often traces back to MySQL running out of connections. These problems are invisible until they become incidents — unless you have monitoring in place. MySQL ships with extensive built-in observability tools. This chapter covers the queries you'll run regularly to understand what MySQL is doing, where time is being spent, and where to look when things go wrong.

What this chapter covers: `SHOW STATUS` — the key variables and calculated health metrics (buffer pool hit ratio, temp table ratio, slow query rate). `SHOW PROCESSLIST` in depth — states explained, identifying blocked queries, KILL. Slow query log analysis and acting on findings. `performance_schema` — finding your top queries by digest, wait events, file I/O, memory. The `sys` schema — the human-readable layer over performance_schema. `SHOW ENGINE INNODB STATUS` for deep-dive diagnostics. Database and table size queries. `mysqladmin` for shell monitoring. A practical daily monitoring checklist.

SHOW STATUS — The Server Health Dashboard

MySQL exposes hundreds of internal counters via `SHOW STATUS`. The counters are cumulative since the last server restart — they tell you the total work done, not a rate. To get a rate, divide by `Uptime`, or compare two snapshots taken seconds apart.

Global status (since last restart) — most useful for baseline checks mysql> SHOW GLOBAL STATUS LIKE 'pattern'; # Session status (since this connection opened) mysql> SHOW SESSION STATUS LIKE 'pattern'; # Quick server summary mysql> SHOW GLOBAL STATUS WHERE variable_name IN (-> 'Uptime', 'Questions', 'Threads_connected', -> 'Threads_running', 'Max_used_connections', 'Slow_queries' ->); +---------- ---------+---------+ | Variable_name | Value | +----------------------+---------+ | Max_used_connections | 12 | | Questions | 4193820 | | Slow_queries | 47 | | Threads_connected | 4 | | Threads_running | 1 | | Uptime | 604800 | +----------------------+---------+ # Uptime 604800 = 7 days. 47 slow queries in 7 days — low enough. # Threads_running=1 = only one query executing right now (the SHOW STATUS itself)

The variables that matter most

Variable	What it tells you	Concern if...
Uptime	Seconds since last restart. Divide other counters by this to get per-second rates.	Unexpected recent restart
Threads_connected	Current open connections.	> 80% of max_connections
Threads_running	Connections actively executing a query (not sleeping). Normal is 1–5.	> 20 sustained
Max_used_connections	Peak simultaneous connections since restart. Use to right-size max_connections.	Close to max_connections
Slow_queries	Total queries exceeding long_query_time.	Growing rapidly
Questions	Total statements executed (includes stored procedure statements).	—
Com_select / Com_insert / Com_update / Com_delete	Count of each DML type — shows your read/write ratio.	—
Aborted_connects	Connections that failed to authenticate. Spikes indicate attacks or misconfigured apps.	Rapid increase
Aborted_clients	Connections closed without a clean disconnect (app crashed, network timeout).	High sustained count
Handler_read_rnd_next	Number of full row-by-row table scans. High = many queries missing indexes.	Very high vs Questions
Created_tmp_disk_tables	Temp tables that spilled to disk (too big for RAM). Expensive operation.	> 25% of Created_tmp_tables
Table_locks_waited	Times a query had to wait for a table lock. Should be near zero for InnoDB.	Any significant value
Innodb_buffer_pool_reads	Pages read from disk (not in buffer pool). Compare to read_requests for hit ratio.	Hit ratio below 95%

Calculated health metrics

── Buffer pool hit ratio (should be > 99%) ──────────────────────────────── mysql> SELECT -> FORMAT(-> (1 - (Innodb_reads / Innodb_read_requests)) * 100, 2 ->) AS buffer_pool_hit_ratio_pct -> FROM (-> SELECT -> VARIABLE_VALUE AS Innodb_reads -> FROM performance_schema.global_status -> WHERE VARIABLE_NAME = 'Innodb_buffer_pool_reads' ->) r, -> (-> SELECT -> VARIABLE_VALUE AS Innodb_read_requests -> FROM performance_schema.global_status -> WHERE VARIABLE_NAME = 'Innodb_buffer_pool_read_requests' ->) rr; +---------------------------+ | buffer_pool_hit_ratio_pct | +---------------------------+ | 99.97 | +---------------------------+ # 99.97% — excellent. If < 95%, innodb_buffer_pool_size needs increasing. # ── Temp table disk spill ratio (should be < 25%) ──────────────────────── mysql> SHOW GLOBAL STATUS WHERE variable_name -> IN ('Created_tmp_tables', 'Created_tmp_disk_tables'); +-------------------------+-------+ | Variable_name | Value | +-------------------------+-------+ | Created_tmp_disk_tables | 8 | | Created_tmp_tables | 1042 | +-------------------------+-------+ # 8/1042 = 0.8% spill to disk — fine. If > 25%, increase tmp_table_size. # ── Connection headroom ── mysql> SELECT -> @@max_connections AS max_connections, -> Max_used_connections, -> ROUND(Max_used_connections / @@max_connections * 100) AS pct_used -> FROM (-> SELECT VARIABLE_VALUE AS Max_used_connections -> FROM performance_schema.global_status -> WHERE VARIABLE_NAME = 'Max_used_connections' ->) m; +-----------------+---------------------+----------+ | max_connections | Max_used_connections| pct_used | +-----------------+---------------------+----------+ | 75 | 12 | 16 | +-----------------+---------------------+----------+ # 16% peak usage — plenty of headroom. Alert at > 80%.

Buffer pool hit ratio

Target: > 99%

1 - (reads / read_requests). Below 95% means the buffer pool is too small and MySQL is reading from disk constantly.

Temp table disk spill

Target: < 25%

Created_tmp_disk_tables / Created_tmp_tables. High ratio means complex queries need more RAM for temp tables — increase tmp_table_size.

Connection usage

Alert at: > 80%

Max_used_connections / max_connections. At 80%, start planning to increase max_connections before the server begins refusing new connections.

Slow query rate

Target: < 0.1%

Slow_queries / Questions. For most sites a handful of slow queries per hour is acceptable; a rising rate suggests a new code deployment added a bad query.

SHOW PROCESSLIST — Seeing What MySQL Is Doing Right Now

Standard view (truncates queries at 100 chars) mysql> SHOW PROCESSLIST; +----+--------------+-----------+----------+---------+------+------------------------+------------------+ | Id | User | Host | db | Command | Time | State | Info | +----+--------------+-----------+----------+---------+------+------------------------+------------------+ | 1 | event_sched | localhost | NULL | Daemon | 3600 | Waiting on empty queue | NULL | | 8 | root | localhost | NULL | Query | 0 | starting | SHOW PROCESSLIST | | 11 | bookshop_app | localhost | bookshop | Sleep | 42 | | NULL | | 14 | bookshop_app | localhost | bookshop | Query | 18 | Sending data | SELECT * FROM ... | +----+--------------+-----------+----------+---------+------+------------------------+------------------+ # FULL version — shows complete query text (no truncation) mysql> SHOW FULL PROCESSLIST; # Kill a specific connection (use the Id from SHOW PROCESSLIST) mysql> KILL 14; Query OK, 0 rows affected (0.00 sec) # Kill only the current query (not the connection) — KILL QUERY mysql> KILL QUERY 14;

Understanding the State column

Sleep

Idle connection waiting for the next query. Normal. If many connections sit in Sleep for a long time, reduce wait_timeout.

starting

Just started executing a query — initializing. Seen briefly on every query, not a concern.

Sending data

Reading rows and sending results to client. Large result sets or slow full scans show up here. The most common active state.

Sorting result

ORDER BY is being applied. If sustained, the query may be sorting a large result set — check whether an index covers the sort column.

Creating sort index

Building a temporary index for sorting. Common with filesort — indicates the query can't use an existing index for the ORDER BY.

Waiting for lock

Blocked waiting for a row lock or table lock held by another transaction. Multiple queries in this state indicate a lock contention problem.

Locked

Waiting for a table lock (MyISAM). Should not appear with InnoDB tables — if it does, check storage engine with SHOW CREATE TABLE.
Copying to tmp table
Building a temporary table — a GROUP BY, DISTINCT, or complex JOIN that MySQL can't resolve with indexes. Look for missing indexes.

Finding blocked queries — the lock detection query

Show all queries waiting for a lock, and what's blocking them mysql> SELECT -> r.trx_id AS waiting_trx, -> r.trx_mysql_thread_id AS waiting_thread, -> r.trx_query AS waiting_query, -> b.trx_id AS blocking_trx, -> b.trx_mysql_thread_id AS blocking_thread, -> b.trx_query AS blocking_query -> FROM information_schema.innodb_lock_waits w -> JOIN information_schema.innodb_trx b ON b.trx_id = w.blocking_trx_id -> JOIN information_schema.innodb_trx r ON r.trx_id = w.requesting_trx_id\G *************************** 1. row *************************** waiting_trx: 421938 waiting_thread: 14 waiting_query: UPDATE books SET price = 9.99 WHERE id = 42 blocking_trx: 421930 blocking_thread: 11 blocking_query: NULL (transaction is idle / doing something else) # Thread 11 holds a lock on a row that thread 14 needs. # Thread 11 is idle (NULL query) — it may have forgotten to COMMIT. # Fix: KILL 11; or ask the app to commit its transaction.

Slow Query Log — Analysis and Action

The slow query log (enabled in Chapter 4) captures queries that exceeded `long_query_time`. Reading individual entries is useful for one-off investigation, but the real power comes from aggregating the log to find the worst offenders across thousands of queries.

Summarise the slow log — top 10 by total time spent $ sudo mysqldumpslow -s t -t 10 /var/log/mysql/slow.log Count: 1823 Time=4.21s (7675s) Lock=0.00s Rows=1.0 (1823) SELECT * FROM books WHERE title LIKE '%S%' Count: 340 Time=2.84s (965s) Lock=0.00s Rows=87.2 (29641) SELECT o.*, b.title FROM orders o JOIN books b ON o.book_id = b.id WHERE o.customer_id = N Count: 12 Time=11.7s (140s) Lock=0.00s Rows=128430.0 (1541160) SELECT * FROM orders WHERE created_at BETWEEN 'S' AND 'S' # First result: ran 1823 times, total 7675 seconds — this is the #1 priority to fix # The leading % in LIKE means no index can be used — consider FULLTEXT index # Third result: 128,430 rows examined each time — needs index on created_at # mysqldumpslow sort options: # -s t = sort by total time (usually most useful) # -s at = sort by average time per query # -s c = sort by count (most frequent) # -s r = sort by rows examined # -t N = top N results
The action after finding a slow query: take the query to Chapter 7 of the Advanced SQL course — `EXPLAIN` and `EXPLAIN ANALYZE` will tell you exactly why it's slow. The fix is almost always one of: add an index on the filtered/sorted columns, rewrite the query to avoid a leading wildcard, or break up a massive result set with pagination (LIMIT/OFFSET).

performance_schema — The Query-Level Microscope

The slow query log shows individual bad queries. `performance_schema` goes further — it aggregates statistics across all queries, normalised by pattern (a "digest"), so you can find the query type that's costing the most time even if each individual execution is fast.

Top queries by total execution time

The most useful single query in performance_schema # Find top 10 queries by total time spent (across all executions) mysql> SELECT -> SCHEMA_NAME AS db, -> DIGEST_TEXT AS query_pattern, -> COUNT_STAR AS executions, -> ROUND(SUM_TIMER_WAIT / 1e12, 2) AS total_sec, -> ROUND(AVG_TIMER_WAIT / 1e12, 4) AS avg_sec, -> ROUND(MAX_TIMER_WAIT / 1e12, 2) AS max_sec, -> SUM_ROWS_EXAMINED AS rows_examined, -> SUM_ROWS_SENT AS rows_sent -> FROM performance_schema.events_statements_summary_by_digest -> WHERE SCHEMA_NAME IS NOT NULL -> ORDER BY SUM_TIMER_WAIT DESC -> LIMIT 10\G *************************** 1. row *************************** db: bookshop query_pattern: SELECT * FROM `books` WHERE `title` LIKE ? executions: 1823 total_sec: 7674.82 avg_sec: 4.2100 max_sec: 9.34 rows_examined: 234563590 rows_sent: 1823 # rows_examined / rows_sent = 128,680 : 1 — classic full table scan # This one query has burned 7,674 seconds of server time

What MySQL is waiting on

Top 10 wait events — what is MySQL spending time waiting for? mysql> SELECT -> EVENT_NAME, -> COUNT_STAR AS count, -> ROUND(SUM_TIMER_WAIT / 1e12, 2) AS total_wait_sec -> FROM performance_schema.events_waits_summary_global_by_event_name -> WHERE COUNT_STAR > 0 -> ORDER BY SUM_TIMER_WAIT DESC -> LIMIT 10; +---------------------------------------+----------+----------------+ | EVENT_NAME | count | total_wait_sec | +---------------------------------------+----------+----------------+ | wait/io/file/innodb/innodb_data_file | 24891034 | 1842.34 | | wait/io/file/sql/FRM | 1200443 | 12.44 | | wait/synch/mutex/innodb/... | 4923122 | 3.21 | +---------------------------------------+----------+----------------+ # innodb_data_file at the top means most waits are disk I/O # → buffer pool is too small, frequently reading from disk # Solution: increase innodb_buffer_pool_size (Chapter 4)

Memory usage breakdown

How much memory is MySQL allocating, broken down by component mysql> SELECT -> EVENT_NAME, -> CURRENT_NUMBER_OF_BYTES_USED / 1024 / 1024 AS mb_used -> FROM performance_schema.memory_summary_global_by_event_name -> WHERE CURRENT_NUMBER_OF_BYTES_USED > 0 -> ORDER BY CURRENT_NUMBER_OF_BYTES_USED DESC -> LIMIT 10; +----------+---------+ | EVENT_NAME | mb_used | +----------+---------+ | memory/innodb/buf_buf_pool | 1024.00 | | memory/performance_schema/... | 87.32 | | memory/sql/Filesort_buffer::buffer | 14.11 | +----------+---------+ # buf_buf_pool = InnoDB buffer pool (expected — should be your biggest user) # Large Filesort_buffer = many in-progress sort operations # Reset all performance_schema statistics (counters restart from zero) mysql> TRUNCATE TABLE performance_schema.events_statements_summary_by_digest;

The sys Schema — Human-Readable Diagnostics

The sys schema wraps performance_schema in views with human-readable formatting — times as "2.3s" instead of picoseconds, sizes as "1.2 MiB" instead of raw bytes. It's the first place to look for diagnostic information.

sys.statement_analysis
Top queries by total latency, with count, average latency, rows examined, full scans, and tmp tables. **The most useful diagnostic view in MySQL.**
sys.processlist
Active connections with human-readable durations, current statement, and wait info. Filters out Sleep connections by default.
sys.schema_table_statistics
I/O statistics per table — reads vs writes, total wait times. Shows which tables are the busiest.
sys.schema_unused_indexes
Indexes that have never been used since the server started. Candidates for removal — unused indexes waste write performance and storage.
sys.schema_redundant_indexes
Indexes that duplicate or overlap another index on the same table. MySQL uses only one — the duplicate is pure overhead.
sys.io_global_by_file_by_bytes
Which files MySQL reads and writes most, with total bytes. Shows if a specific table's data file is causing I/O pressure.
sys.memory_by_user_by_current_bytes
RAM allocated per MySQL user. Useful for finding which application is using the most memory.
Top 5 queries by total time — clean, human-readable output mysql> SELECT query, exec_count, total_latency, rows_examined_avg, full_scans -> FROM sys.statement_analysis -> ORDER BY total_latency DESC -> LIMIT 5\G *************************** 1. row *************************** query: SELECT * FROM `books` WHERE `title` LIKE ? exec_count: 1823 total_latency: 2.12 h rows_examined_avg: 128430 full_scans: 1823 # Indexes that have never been used since last restart mysql> SELECT table_schema, table_name, index_name -> FROM sys.schema_unused_indexes -> WHERE table_schema NOT IN ('performance_schema', 'sys') -> ORDER BY table_schema, table_name; +--------------+------------+-----------------+ | table_schema | table_name | index_name | +--------------+------------+-----------------+ | bookshop | books | idx_isbn | | bookshop | orders | idx_shipped_at | +--------------+------------+-----------------+ # These indexes are never queried — they slow every INSERT/UPDATE on those tables # Consider dropping them: ALTER TABLE books DROP INDEX idx_isbn; # Redundant indexes (duplicates) mysql> SELECT table_schema, table_name, redundant_index_name, dominant_index_name -> FROM sys.schema_redundant_indexes -> WHERE table_schema NOT IN ('sys', 'mysql')\G # Active connections right now (excluding Sleep) mysql> SELECT user, db, command, time, state, current_statement -> FROM sys.processlist -> WHERE command != 'Sleep' -> ORDER BY time DESC;

SHOW ENGINE INNODB STATUS — The Deep Diagnostic

When something's wrong and other tools haven't pinpointed it, `SHOW ENGINE INNODB STATUS` dumps a detailed internal snapshot of InnoDB's state — active transactions, lock waits, buffer pool internals, I/O threads, and the all-important deadlock history.

mysql> SHOW ENGINE INNODB STATUS\G *************************** 1. row *************************** Type: InnoDB Name: Status: ===================================== 2026-06-15 14:30:01 0x7f3abc INNODB MONITOR OUTPUT ===================================== Per second averages calculated from the last 38 seconds ---------- SEMAPHORES ---------- OS WAIT ARRAY INFO: reservation count 142 Mutex spin waits 0, rounds 0, OS waits 0 # Semaphores: internal locking overhead. High numbers = CPU contention. ------------ TRANSACTIONS ------------ Trx id counter 421942 Purge done for trx's n:o < 421930 undo n:o < 0 History list length 0 LIST OF TRANSACTIONS FOR EACH SESSION: ---TRANSACTION 421938, ACTIVE 18 sec starting index read 2 lock struct(s), heap size 1136, 1 row lock(s) LOCK WAIT 1 lock struct(s) MySQL thread id 14, OS thread handle ..., query id 9834 localhost bookshop_app UPDATE books SET price = 9.99 WHERE id = 42 # A transaction has been waiting 18 seconds for a row lock — indicates contention ---------- BUFFER POOL AND MEMORY ---------- Buffer pool size 65536 ← pages (65536 × 16 KB = 1 GB) Free buffers 12481 Database pages 52847 ← pages currently holding data Modified db pages 234 ← dirty pages (unflushed to disk) Pages read ahead 0.00/s, evicted without access 0.00/s Buffer pool hit rate 9997 / 10000 ← 99.97% hit rate -------------- ROW OPERATIONS -------------- Number of rows inserted 28442, updated 9123, deleted 1204, read 14523891

The LATEST DETECTED DEADLOCK section in `SHOW ENGINE INNODB STATUS` shows the most recent deadlock — the two transactions, the rows they were fighting over, and which one was rolled back. If you're seeing unexpected rollbacks in your application, this is where to look.

Database and Table Sizes

Size of each database in MB mysql> SELECT -> table_schema AS database_name, -> ROUND(SUM(data_length + index_length) / 1024 / 1024, 1) AS size_mb -> FROM information_schema.tables -> GROUP BY table_schema -> ORDER BY size_mb DESC; +--------------------+---------+ | database_name | size_mb | +--------------------+---------+ | bookshop | 284.5 | | mysql | 2.4 | | performance_schema | 0.0 | | sys | 0.0 | +--------------------+---------+ # Largest tables in a specific database mysql> SELECT -> table_name, -> ROUND(data_length / 1024 / 1024, 1) AS data_mb, -> ROUND(index_length / 1024 / 1024, 1) AS index_mb, -> ROUND((data_length + index_length) / 1024 / 1024, 1) AS total_mb, -> table_rows AS approx_rows -> FROM information_schema.tables -> WHERE table_schema = 'bookshop' -> ORDER BY (data_length + index_length) DESC; +------------+---------+----------+----------+-------------+ | table_name | data_mb | index_mb | total_mb | approx_rows | +------------+---------+----------+----------+-------------+ | orders | 198.3 | 42.1 | 240.4 | 1284300 | | books | 32.4 | 9.2 | 41.6 | 128430 | | customers | 2.1 | 0.8 | 2.9 | 28442 | +------------+---------+----------+----------+-------------+ # Actual disk usage of the MySQL data directory $ sudo du -sh /var/lib/mysql/ 312M /var/lib/mysql/ # Per-database disk usage $ sudo du -sh /var/lib/mysql/*/ 284M /var/lib/mysql/bookshop/ 2.4M /var/lib/mysql/mysql/

mysqladmin — Quick Shell Monitoring

Quick one-line server summary (no SQL required) $ sudo mysqladmin status Uptime: 604800 Threads: 4 Questions: 4193820 Slow queries: 47 Opens: 1284 Flush tables: 3 Open tables: 1282 Queries per second avg: 6.938 # See all status variables (equivalent to SHOW GLOBAL STATUS) $ sudo mysqladmin extended-status | grep -E "Threads|Slow|Question|Conn" | Connections | 8432 | | Max_used_connections | 12 | | Questions | 4193820 | | Slow_queries | 47 | | Threads_connected | 4 | | Threads_running | 1 | # Poll status every 5 seconds — like vmstat for MySQL $ sudo mysqladmin extended-status -i 5 -r | grep -E "Threads_running|Questions|Slow" # -i 5 = interval 5 seconds | -r = show differences (deltas) from last poll # Show processlist from the shell (no MySQL client needed) $ sudo mysqladmin processlist # Ping MySQL (exit code 0 = alive, non-zero = down) $ sudo mysqladmin ping mysqld is alive

Practical Monitoring Checklist

📋
Daily — check backup log: `tail -5 /var/log/mysql_backup.log` — confirm last night's backup completed successfully and the file size looks right.

📋
Daily — check slow query count: `mysqladmin status | grep -i slow` — if Slow_queries is rising faster than usual, something changed (new code, more traffic, missing index on a new table).

📋
Daily — check error log: `sudo tail -30 /var/log/mysql/error.log` — look for [ERROR] lines. InnoDB recovery messages, disk full warnings, or authentication failures all appear here first.

🗓️
Weekly — top slow queries: `sudo mysqldumpslow -s t -t 5 /var/log/mysql/slow.log` — identify whether the same queries keep appearing or if new ones have emerged. Investigate any query with total_time > 1 hour.

🗓️
Weekly — disk space: `df -h /var/lib/mysql` and `sudo du -sh /var/lib/mysql/*/` — databases that grow without bound eventually fill the disk and crash MySQL. Alert at 75% disk usage.

🗓️
Weekly — connection headroom: `SELECT VARIABLE_VALUE FROM performance_schema.global_status WHERE VARIABLE_NAME='Max_used_connections';` compared to `@@max_connections` — alert at 80%.

🚨
Alert immediately — Aborted_connects rising: repeated authentication failures may indicate a brute-force attempt against MySQL accounts. Check `SHOW GLOBAL STATUS LIKE 'Aborted_connects'` compared to yesterday's value.

🚨
Alert immediately — Threads_running > 20: many simultaneously executing queries usually means a slow query is holding locks and others are piling up behind it. Run `SHOW FULL PROCESSLIST` to identify the blocker.

Troubleshooting

performance_schema tables return empty results

The performance_schema may be disabled. Check: `SHOW VARIABLES LIKE 'performance_schema';` — should be ON. If OFF, enable it by adding `performance_schema = ON` to `mysqld.cnf` and restarting. Note: performance_schema uses memory (typically 80–100 MB) — on very small servers (under 512 MB RAM total) it may have been disabled to save RAM. Also check that the relevant instruments are enabled: `SELECT NAME, ENABLED FROM performance_schema.setup_instruments WHERE NAME LIKE 'statement/%' LIMIT 5;`

High CPU but SHOW PROCESSLIST shows mostly Sleep connections

If CPU is high but queries look idle, check for InnoDB background activity: `SHOW ENGINE INNODB STATUS\G` — look at the I/O section for pending reads/writes and the BUFFER POOL section for modified (dirty) pages being flushed. High dirty page count means InnoDB is aggressively flushing to disk. Also run `SELECT * FROM sys.statement_analysis ORDER BY total_latency DESC LIMIT 5` — a fast but very frequent query can consume CPU without appearing in SHOW PROCESSLIST long enough to be seen.

Many threads in "Waiting for lock" state

A transaction is holding a row lock and not releasing it. Run the lock detection query from this chapter to identify the blocking thread ID, then `SHOW FULL PROCESSLIST` to see what it's doing (or if it's idle/stuck). If the blocking thread is an idle Sleep connection, it opened a transaction without committing. `KILL <thread_id>` forces a rollback and releases the locks. Long-term fix: ensure application code always commits or rolls back transactions, and reduce `wait_timeout` to evict stale connections.

sys schema views return "Table 'sys.xxx' doesn't exist"

The sys schema wasn't installed or was accidentally dropped. Reinstall it: locate the `mysql_sys_schema.sql` file (on Ubuntu: `find /usr/share/mysql -name 'mysql_sys_schema.sql'`; for MySQL 8.0 it may be in the shell package). Apply it: `sudo mysql < /path/to/mysql_sys_schema.sql`. Alternatively, on Ubuntu/Debian: `sudo apt install --reinstall mysql-server` (the sys schema is bundled with the server package).

Quick Reference — Chapter 7

Command / Query	Purpose
SHOW GLOBAL STATUS WHERE variable_name IN (...);	Check specific health counters — Threads_connected, Slow_queries, Uptime
SHOW FULL PROCESSLIST;	All active connections with full query text — first stop when the site is slow
KILL <Id>;	Terminate a stuck connection and roll back its open transaction
SELECT ... FROM performance_schema.events_statements_summary_by_digest ORDER BY SUM_TIMER_WAIT DESC LIMIT 10;	Top queries by total time — finds the queries costing the most cumulatively
SELECT ... FROM sys.statement_analysis ORDER BY total_latency DESC LIMIT 10;	Same as above but with human-readable time values
SELECT ... FROM sys.schema_unused_indexes;	Indexes never used — candidates for removal to improve write performance
SELECT ... FROM sys.schema_redundant_indexes;	Duplicate indexes — pick the dominant one, drop the redundant
SHOW ENGINE INNODB STATUS\G	Deep InnoDB diagnostics — transactions, lock waits, deadlocks, buffer pool stats
sudo mysqladmin status	One-line server summary from the shell — no MySQL session needed
sudo mysqladmin extended-status -i 5 -r	Poll all status counters every 5 seconds, showing deltas
sudo mysqldumpslow -s t -t 10 /var/log/mysql/slow.log	Top 10 slow queries by total time spent
SELECT table_schema, ROUND(SUM(data_length+index_length)/1024/1024,1) FROM information_schema.tables GROUP BY table_schema;	Database sizes in MB
TRUNCATE TABLE performance_schema.events_statements_summary_by_digest;	Reset query statistics so you get a clean baseline after fixing a bad query

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 8

MySQL with Apache and PHP

Chapter 8 — MySQL with Apache and PHP

Installing the PHP MySQL extension is straightforward. Using it correctly — with prepared statements, proper error handling, and credentials stored outside the web root — is where most PHP applications fall short. This chapter builds a solid, secure foundation for PHP-MySQL development: the connection setup, the credential storage pattern, and the query techniques you'll use on every project.

What this chapter covers: the three PHP MySQL APIs and why PDO is the right choice. Installing `php-mysql` on Ubuntu/Debian. Keeping database credentials out of the web root — the most important security rule. The PDO connection with all three essential options. Prepared statements for SELECT, INSERT, UPDATE, and DELETE. Error handling that logs without leaking. Transaction handling. A complete practical example.

The Three PHP MySQL APIs

PHP has three ways to talk to MySQL. Two of them are wrong choices for new code.

mysql_*
Removed — PHP 7+

- Removed from PHP 7. Dead.
- No prepared statements → SQL injection by default
- Old tutorials still use it. Don't copy them.

MySQLi
OK — MySQL only

- MySQL-specific. Won't work with PostgreSQL or SQLite.
- Prepared statements supported
- OOP and procedural interfaces — inconsistent
- Reasonable choice if you're sure you'll never switch databases

PDO
Recommended

- Works with MySQL, PostgreSQL, SQLite, and more
- Clean, consistent OOP interface
- Named placeholders (`:name`) — much more readable than `?`
- Exception-based error handling
- This chapter uses PDO exclusively

Installing the PHP MySQL Extension

Install php-mysql — this installs BOTH MySQLi and PDO_MySQL drivers $ sudo apt update && sudo apt install php-mysql # If you're running a specific PHP version (e.g. 8.2), the package is version-pinned $ sudo apt install php8.2-mysql # ALWAYS restart Apache after installing PHP extensions $ sudo systemctl restart apache2 # Verify both drivers loaded $ php -m | grep -i mysql mysqli mysqlnd PDO pdo_mysql # Check which PDO drivers are available $ php -r "print_r(PDO::getAvailableDrivers());" Array ([0] => mysql [1] => sqlite)
Restart is not optional: PHP extensions are loaded at Apache startup. Installing `php-mysql` without restarting Apache leaves the old PHP process running without the driver. The result is a cryptic "could not find driver" PDOException even though the package is installed. `sudo systemctl restart apache2` every time you install or remove a PHP extension.

Credentials Outside the Web Root

This is the single most important security rule in PHP-MySQL development. Your database password must never be inside a directory that Apache can serve.

Why the web root is dangerous for credentials

If your db.php is inside /var/www/html/, any of these can expose the password: A missing or misconfigured `.htaccess` / Apache rule could cause PHP files to be served as plain text instead of executed. A misconfigured PHP install. A malicious include path. Apache directory listing with the wrong permissions. A future accidental `git push` to a public repository. A LFI (local file inclusion) vulnerability in another file on the same server.

The directory layout that protects credentials

/var/www/ ├── config/ ← ABOVE the web root. Apache cannot serve this. │ └── db.php ← database credentials live here │ └── html/ ← Apache DocumentRoot (/var/www/html) ├── db_WRONG.php ← NEVER put credentials here ├── index.php ← your application ├── books.php └── api/ └── search.php

Apache's DocumentRoot is `/var/www/html` — it serves everything inside that directory. Anything in `/var/www/` but *above* `/var/www/html/` is unreachable from the web, but still accessible to your PHP scripts with `require`.

Create the config directory above the web root $ sudo mkdir -p /var/www/config # Restrict permissions: www-data can read, nobody else can $ sudo chown root:www-data /var/www/config $ sudo chmod 750 /var/www/config

The credentials file — db.php

# /var/www/config/db.php $ sudo nano /var/www/config/db.php
<?php // Abort if this file is somehow reached via HTTP // (belt-and-suspenders: the file should not be web-accessible at all) if (php_sapi_name() === 'cli' || defined('STDIN')) { // running from command line — OK } elseif (!defined('APP_ROOT')) { // not called from within the app http_response_code(403); exit('Forbidden'); } return ['host' => 'localhost', // socket connection — fastest 'dbname' => 'bookshop', 'charset' => 'utf8mb4', // always utf8mb4 — full Unicode + emoji 'username' => 'bookshop_app', 'password' => 'StrongPassw0rd!',];
# Set file permissions: readable only by root and www-data $ sudo chown root:www-data /var/www/config/db.php $ sudo chmod 640 /var/www/config/db.php # Verify Apache (www-data) can read it but others cannot $ sudo -u www-data cat /var/www/config/db.php # should succeed $ cat /var/www/config/db.php # your user — should succeed as it's in www-data group
Multiple sites on the same server? Give each site its own config directory: `/var/www/site1/config/`, `/var/www/site2/config/`. Each PHP process only needs to read its own site's credentials, not all sites' credentials.

The PDO Connection

DSN — the data source name

The DSN is a string that describes the database connection. For MySQL it looks like this:

DSN format mysql:host=localhost;dbname=bookshop;charset=utf8mb4 # host=localhost → Unix socket connection (faster, same machine) # host=127.0.0.1 → TCP connection (use only if you have a specific reason) # charset=utf8mb4 → send charset in DSN, not as a SET NAMES query (safer)

The three essential PDO options

ATTR_ERRMODE
→ ERRMODE_EXCEPTION
Always set this. Without it, PDO silently returns `false` on errors — and most PHP code doesn't check for `false`. With this option, any database error throws a `PDOException` which your `try/catch` can handle. The default (ERRMODE_SILENT) is dangerous.
ATTR_DEFAULT_FETCH_MODE
→ FETCH_ASSOC
Set this to avoid surprises. By default PDO returns both numeric and associative keys (`$row[0]` and `$row['title']`). `FETCH_ASSOC` returns named keys only — cleaner, smaller arrays, no accidental use of column position.
ATTR_EMULATE_PREPARES
→ false
Use real prepared statements. When true (the default!), PDO emulates prepared statements by doing string substitution in PHP, then sending the final SQL to MySQL. When false, MySQL handles the prepared statement — you get proper type safety and the protection is enforced at the database level, not in PHP. Always set to `false`.

The connection factory — db_connect.php

<?php // /var/www/html/includes/db_connect.php // This file is inside the web root but contains NO credentials. // It reads them from the config file above the web root. define('APP_ROOT', true); // allows db.php to verify it's called from within the app function get_db(): PDO { static $pdo = null; if ($pdo !== null) { return $pdo; // reuse existing connection within the same request } $cfg = require '/var/www/config/db.php'; // above the web root $dsn = sprintf('mysql:host=%s;dbname=%s;charset=%s', $cfg['host'], $cfg['dbname'], $cfg['charset']); $pdo = new PDO($dsn, $cfg['username'], $cfg['password'], [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC, PDO::ATTR_EMULATE_PREPARES => false,]); return $pdo; }

Why the `static` variable: PHP creates a new process (or reuses a worker) per HTTP request. Within a single request, `static $pdo` ensures `get_db()` connects once and returns the same connection object on every subsequent call — no reconnection overhead. Between requests the connection is closed naturally. You don't need a connection pool in most PHP setups because PHP-FPM handles this at the process level.

Prepared Statements — Not Optional

A prepared statement separates the SQL structure from the data. MySQL receives the query template first, compiles it, then receives the values separately. The values are never interpreted as SQL — SQL injection is structurally impossible.

Never build SQL by string concatenation with user input: `$sql = "SELECT * FROM books WHERE title = '" . $_GET['q'] . "'";` — this is SQL injection. If `$_GET['q']` is `' OR '1'='1`, the WHERE clause becomes always-true and returns every row. If it's `'; DROP TABLE books; --`, it drops the table. Always use prepared statements for any value that comes from outside your code: user input, URL parameters, form data, API responses, file contents.

SELECT — single row

<?php // Get one book by ID require_once __DIR__ . '/includes/db_connect.php'; $book_id = (int) $_GET['id']; // cast to int before using (extra safety layer) $pdo = get_db(); $stmt = $pdo->prepare('SELECT id, title, author, price FROM books WHERE id = :id'); $stmt->execute([':id' => $book_id]); $book = $stmt->fetch(); // fetch() returns one row as array, or false if no match if ($book === false) { // No row found http_response_code(404); echo 'Book not found'; exit; } // $book is now ['id' => 42, 'title' => 'Dune', 'author' => 'Herbert', 'price' => '9.99'] echo htmlspecialchars($book['title']); // always escape output for HTML

SELECT — multiple rows

<?php // Search books by author — user input from a search form $search = trim($_GET['author'] ?? ''); $stmt = $pdo->prepare('SELECT id, title, author, price FROM books WHERE author LIKE :author ORDER BY title LIMIT 50'); $stmt->execute([':author' => '%' . $search . '%']); $books = $stmt->fetchAll(); // returns array of all matching rows (empty array if none) foreach ($books as $book) { echo htmlspecialchars($book['title']) . ' by ' . htmlspecialchars($book['author']); echo ' — £' . number_format($book['price'], 2) . "\n"; } // Note: the LIKE wildcard % is safe to concatenate here because it's not user input // — the user's actual text goes through the prepared statement placeholder :author

INSERT

<?php // Add a new book $stmt = $pdo->prepare('INSERT INTO books (title, author, price, isbn) VALUES (:title, :author, :price, :isbn)'); $stmt->execute([':title' => $_POST['title'], ':author' => $_POST['author'], ':price' => (float) $_POST['price'], ':isbn' => $_POST['isbn'],]); $new_id = (int) $pdo->lastInsertId(); // the AUTO_INCREMENT id of the new row echo "Created book #$new_id";

UPDATE and DELETE

<?php // Update a book's price $stmt = $pdo->prepare('UPDATE books SET price = :price WHERE id = :id'); $stmt->execute([':price' => (float) $_POST['price'], ':id' => (int) $_POST['id']]); $rows_changed = $stmt->rowCount(); // number of rows actually modified if ($rows_changed === 0) { echo 'No book found with that ID, or price was already the same'; } // Delete a book $stmt = $pdo->prepare('DELETE FROM books WHERE id = :id'); $stmt->execute([':id' => (int) $_POST['id']]); echo $stmt->rowCount() . ' book(s) deleted';

Reusing a prepared statement in a loop

<?php // Inserting many rows efficiently — prepare ONCE, execute MANY TIMES $books_to_import = [['title' => 'Foundation', 'author' => 'Asimov', 'price' => 8.99], ['title' => 'Neuromancer', 'author' => 'Gibson', 'price' => 7.99], ['title' => 'Hyperion', 'author' => 'Simmons', 'price' => 9.99],]; $stmt = $pdo->prepare('INSERT INTO books (title, author, price) VALUES (:title, :author, :price)'); foreach ($books_to_import as $book) { $stmt->execute($book); // PHP matches array keys to named placeholders automatically }

Error Handling — Log, Don't Leak

With `ERRMODE_EXCEPTION` set, any database error throws a `PDOException`. The exception's `getMessage()` contains the SQL error — useful for debugging, but dangerous to display to users.

Never display PDOException messages to users: a PDOException message can contain the table name, column name, failing query, and sometimes partial data. Attackers look for these. A caught exception that echoes `$e->getMessage()` directly to the page is a reconnaissance gift.
<?php // ── The correct pattern: log the real error, show a generic message ── function get_book(int $id): ?array { try { $pdo = get_db(); $stmt = $pdo->prepare('SELECT * FROM books WHERE id = :id'); $stmt->execute([':id' => $id]); $row = $stmt->fetch(); return $row === false ? null : $row; } catch (PDOException $e) { // Log the full error — visible only in the server log, never to the user error_log('DB error in get_book(): ' . $e->getMessage()); // Re-throw a generic exception — your error handler shows a friendly page throw new RuntimeException('A database error occurred. Please try again later.'); } } // At the top level (e.g. index.php) — catch the generic exception for the user try { $book = get_book(42); } catch (RuntimeException $e) { // Show a safe, generic message — no database details echo ' ' . htmlspecialchars($e->getMessage()) . ' '; }
# PHP logs database errors to the Apache error log by default # Watch it in real time during development $ sudo tail -f /var/log/apache2/error.log [Mon Jun 15 14:30:01.123456 2026] [php:error] [pid 12345] [client 127.0.0.1:42841] PHP Fatal error: Uncaught PDOException: SQLSTATE[HY000] [1045] Access denied for user 'bookshop_app'@'localhost' DB error in get_book(): SQLSTATE[42S02]: Base table or view not found: 1146 Table 'bookshop.bookz' doesn't exist # Full error in the log — no user ever sees this

Transaction Handling

When an operation involves multiple writes that must all succeed or all fail together — creating an order and deducting stock, for example — wrap them in a transaction.

<?php // Place an order: insert order row + decrement stock — both must succeed function place_order(int $book_id, int $customer_id, int $qty): int { $pdo = get_db(); try { $pdo->beginTransaction(); // Step 1: Check stock (lock the row so nobody else can race us) $stmt = $pdo->prepare('SELECT stock FROM books WHERE id = :id FOR UPDATE'); $stmt->execute([':id' => $book_id]); $row = $stmt->fetch(); if (!$row || $row['stock'] < $qty) { $pdo->rollBack(); throw new RuntimeException('Insufficient stock'); } // Step 2: Insert order row $stmt = $pdo->prepare('INSERT INTO orders (book_id, customer_id, quantity, created_at) VALUES (:book_id, :customer_id, :qty, NOW())'); $stmt->execute([':book_id' => $book_id, ':customer_id' => $customer_id, ':qty' => $qty,]); $order_id = (int) $pdo->lastInsertId(); // Step 3: Decrement stock $stmt = $pdo->prepare('UPDATE books SET stock = stock - :qty WHERE id = :id'); $stmt->execute([':qty' => $qty, ':id' => $book_id]); $pdo->commit(); // all steps succeeded — write to disk return $order_id; } catch (PDOException $e) { $pdo->rollBack(); // any DB error → undo all steps in the transaction error_log('place_order() failed: ' . $e->getMessage()); throw new RuntimeException('Could not place order. Please try again.'); } }
Always call rollBack() in the catch block. If a `PDOException` is thrown mid-transaction and you don't roll back, InnoDB holds the locks until the connection closes (end of the PHP request) — potentially seconds. During that time every other query touching those rows is blocked. The `rollBack()` releases the locks immediately.

Practical Example — User Login

The classic PHP-MySQL pattern: check a submitted username/password against the database. This example shows the correct approach: no SQL injection, no timing attacks, no credential leaks.

<?php // login.php — verify username and password require_once __DIR__ . '/includes/db_connect.php'; function check_login(string $username, string $password): ?array { try { $pdo = get_db(); $stmt = $pdo->prepare('SELECT id, username, password_hash, role FROM users WHERE username = :username LIMIT 1'); $stmt->execute([':username' => $username]); $user = $stmt->fetch(); if ($user === false) { // Username not found — but don't say that. // Run the hash anyway to avoid timing attacks that reveal valid usernames password_verify($password, '$2y10fakehashhere000000000000000000000000000000000000000'); return null; } if (!password_verify($password, $user['password_hash'])) { return null; // wrong password } // Upgrade hash if bcrypt cost factor has been increased since account creation if (password_needs_rehash($user['password_hash'], PASSWORD_DEFAULT)) { $new_hash = password_hash($password, PASSWORD_DEFAULT); $upd = $pdo->prepare('UPDATE users SET password_hash = :hash WHERE id = :id'); $upd->execute([':hash' => $new_hash, ':id' => $user['id']]); } // Don't return the password hash to the caller unset($user['password_hash']); return $user; } catch (PDOException $e) { error_log('Login DB error: ' . $e->getMessage()); throw new RuntimeException('Login temporarily unavailable.'); } } // In your form handler if ($_SERVER['REQUEST_METHOD'] === 'POST') { $user = check_login(trim($_POST['username'] ?? ''), $_POST['password'] ?? ''); if ($user) { session_regenerate_id(true); // prevent session fixation $_SESSION['user_id'] = $user['id']; $_SESSION['username'] = $user['username']; header('Location: /dashboard.php'); } else { $error = 'Invalid username or password'; // same message for both cases } }

utf8mb4 — The Only Correct Character Set

MySQL's "utf8" is not real UTF-8. MySQL's `utf8` character set only stores 3-byte Unicode characters — it cannot store emoji, some Chinese characters, or other 4-byte code points. The correct character set is `utf8mb4` (utf-8 with 4-byte support). Always use `utf8mb4` in your DSN, on

your database, and on your tables. Setting it in the PDO DSN ensures every string sent over the connection is interpreted correctly — no `SET NAMES` query needed.

Verify your database and tables use utf8mb4 mysql> SELECT schema_name, default_character_set_name -> FROM information_schema.schemata -> WHERE schema_name = 'bookshop'; +-------------+----------------------------+ | schema_name | default_character_set_name | +-------------+----------------------------+ | bookshop | utf8mb4 | +-------------+----------------------------+ # Fix an existing database (does not convert existing table data) mysql> ALTER DATABASE bookshop CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci; # Fix an existing table (converts data in place) mysql> ALTER TABLE books CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

Troubleshooting

PDOException: "could not find driver"

The `pdo_mysql` extension isn't loaded by the running Apache PHP. Steps: **1)** Install: `sudo apt install php-mysql`. **2)** Restart Apache: `sudo systemctl restart apache2` — this is the most commonly skipped step. **3)** Verify: run `php -m | grep pdo_mysql` from the shell, then also check via a phpinfo() page in the browser (the CLI and Apache PHP may have different configs). **4)** If running PHP-FPM: restart `php-fpm` as well as Apache.

PDOException: "Access denied for user 'app'@'localhost'"

Either the wrong password, or a host mismatch. MySQL treats `localhost` (Unix socket) and `127.0.0.1` (TCP) as different hosts. If you created the user as `'app'@'localhost'` but your DSN uses `host=127.0.0.1`, MySQL sees `'app'@'127.0.0.1'` — a different user that doesn't exist. Fix: keep `host=localhost` in the DSN (uses the socket, faster) and create the user as `'app'@'localhost'`. Or change the DSN to `127.0.0.1` and create the user as `'app'@'127.0.0.1'`. They must match.

PDOException: "No such file or directory" (SQLSTATE[HY000] [2002])

MySQL's Unix socket file can't be found. `host=localhost` in PDO means "use the socket", but the socket path in `php.ini` may differ from MySQL's actual socket path. Check MySQL's socket: `mysql -e "SHOW VARIABLES LIKE 'socket';"` (typically `/var/run/mysqld/mysqld.sock`). Check PHP's configured path: `php -i | grep pdo_mysql.default_socket`. If they differ, either add `unix_socket=/var/run/mysqld/mysqld.sock` to your DSN, or set `host=127.0.0.1` to switch to TCP.

Blank white page with no error

PHP is catching or swallowing the exception silently, or `display_errors` is off (which is correct for production, but makes debugging hard). During development: check the Apache error log immediately — `sudo tail -f /var/log/apache2/error.log`. All PHP errors and uncaught exceptions appear there. Temporarily you can also set `ini_set('display_errors', '1');` at the top of the script, but remove it before deploying.

Credentials visible in process list (ps aux) or shell history

Never pass database passwords on the command line. If you ran `php -r "new PDO(..., 'password')"` during testing, that password is visible in `ps aux` while running and stored in shell history. For scripts that need the password: use the credentials file pattern, not environment variables set on the command line. Check shell history: `history | grep -i pass` — delete any lines containing passwords with `history -d <line_number>`, or clear the whole history with `history -c`.

Quick Reference — Chapter 8

Pattern	The correct approach
Install extension	`sudo apt install php-mysql && sudo systemctl restart apache2`
Verify drivers	`php -m | grep -i mysql` — expect mysqli, pdo_mysql
Credentials location	Above the web root: `/var/www/config/db.php` (not inside `/var/www/html/`)
Credentials file permissions	`chown root:www-data` + `chmod 640`
DSN for local MySQL	`mysql:host=localhost;dbname=mydb;charset=utf8mb4`
Required PDO options	ERRMODE_EXCEPTION + FETCH_ASSOC + EMULATE_PREPARES=false
User input in queries	Always via prepared statement placeholders — never string concatenation
Fetch one row	`$stmt->fetch()` — returns array or false
Fetch all rows	`$stmt->fetchAll()` — returns array of arrays (empty if none)
Last inserted ID	`$pdo->lastInsertId()` after INSERT
Rows affected	`$stmt->rowCount()` after UPDATE / DELETE
Error handling	catch PDOException → `error_log($e->getMessage())` → throw generic RuntimeException
Transaction	beginTransaction() → execute steps → commit() or rollBack() in catch

Pattern	The correct approach
Character set	Always `utf8mb4` — never MySQL's `utf8`
Password hashing	`password_hash($pass, PASSWORD_DEFAULT)` to store, `password_verify()` to check

MYSQL INSTALLATION & ADMINISTRATION — CHAPTER 9

phpMyAdmin

Chapter 9 — phpMyAdmin

phpMyAdmin is a browser-based MySQL administration interface that lets you browse tables, run SQL queries, export backups, and manage users without touching the command line. It's widely used and genuinely useful — but its default installation leaves a well-known URL open to the internet with no extra protection. This chapter covers the full installation, the four security hardening steps you must complete before using it, navigating the interface, and the common tasks you'll do through the UI.

What this chapter covers: `apt install phpmyadmin` walkthrough with the interactive prompts explained. What the package creates. Why the defaults are insecure and the four steps to fix them: change the URL alias, IP restriction, HTTP Basic Auth as a second credential layer, `config.inc.php` hardening. Navigating the interface. Creating databases and tables. Running SQL queries. Export and Import. Managing users via the Privileges tab. Cloudflare Access as an alternative auth layer. Keeping phpMyAdmin updated.

Installation

$ sudo apt update && sudo apt install phpmyadmin

The installer is interactive. It asks three questions:

Please choose the web server that should be automatically configured to run phpMyAdmin:
→ Select: apache2 (press Space to tick, then Enter)
This writes `/etc/apache2/conf-enabled/phpmyadmin.conf` automatically — you don't need to configure Apache manually. If you miss this step, phpMyAdmin still installs but the Apache config isn't created and the URL won't work.
Configure database for phpmyadmin with dbconfig-common?
→ Select: Yes
Creates the `phpmyadmin` MySQL database used internally by phpMyAdmin to store bookmarks and settings, and creates a `phpmyadmin` MySQL user with privileges on that database only.
MySQL application password for phpmyadmin:
→ Enter a strong password (or leave blank to auto-generate)
This is the password for the internal `phpmyadmin` MySQL user — not your login password. Generate a strong random one: `openssl rand -base64 24`. You don't need to remember it; it's stored in `/etc/phpmyadmin/config-db.php`.

What the package creates

/usr/share/phpmyadmin/
The application itself — PHP files, CSS, JS. Do not edit files here; they'll be overwritten on upgrade.
/etc/phpmyadmin/
Configuration directory. `config.inc.php` is the main config you'll edit. `config-db.php` has the internal database credentials.
/etc/apache2/conf-enabled/
phpmyadmin.conf
Apache config that defines the `/phpmyadmin` URL alias. This is the file you'll edit to change the URL.
MySQL: phpmyadmin user
MySQL: phpmyadmin DB
The `phpmyadmin` MySQL user has privileges only on the `phpmyadmin` internal database — not on your application databases.
After installation, the default URL is accessible immediately # http://yourserver/phpmyadmin ← this needs to change before you use it # Verify PHP sees all required extensions $ php -m | grep -E "mbstring|zip|gd|json|curl|mysqli" curl gd json mbstring mysqli # If mbstring is missing (needed by phpMyAdmin): $ sudo apt install php-mbstring && sudo systemctl restart apache2

Why the Default Installation Needs Hardening

The /phpmyadmin URL is one of the first things bots scan for. Within minutes of a server becoming internet-accessible, automated scanners will probe `/phpmyadmin`, `/pma`, `/mysql`, and similar paths looking for phpMyAdmin login pages. A fresh install with the default URL is hammered with brute-force login attempts around the clock. The four steps below eliminate most of that risk.
1
Change the URL alias — make the path unpredictable

Edit the Apache config to change `/phpmyadmin` to something that isn't in any bot's wordlist.
2
IP restriction — allow only your IP or LAN
If you only access the server from home or a known IP, restrict the phpMyAdmin URL to that address. Nobody else can reach the login page at all.
3
HTTP Basic Auth — second credential layer
Add an Apache password prompt in front of phpMyAdmin. Even if someone finds the URL, they need a second set of credentials before they see the MySQL login screen.
4
config.inc.php hardening — disable root login and blank passwords
Configure phpMyAdmin itself to reject attempts to log in as root or with no password, regardless of what MySQL allows.

Step 1 — Change the URL Alias

The file that controls the URL $ sudo nano /etc/apache2/conf-enabled/phpmyadmin.conf # Find the Alias line (near the top) Alias /phpmyadmin /usr/share/phpmyadmin # Change it to something unpredictable — use letters+numbers, no dictionary words Alias /dbadmin-x7k2 /usr/share/phpmyadmin # Also update the <Directory> block path to match (it's directly below the Alias line) # Change: <Directory /usr/share/phpmyadmin> → stays the same # The Directory block doesn't change — only the Alias line # Reload Apache to apply $ sudo systemctl reload apache2 # Your phpMyAdmin is now at: http://yourserver/dbadmin-x7k2 # The old /phpmyadmin URL returns 404
Choose a good alias: avoid common abbreviations (pma, dba, myadmin, phpadmin, db). Use something like `/manage-x9q4r` or `/panel-kzw2` — a short memorable path mixed with random characters. Write it somewhere safe; if you forget it, just re-read the config file.

Step 2 — IP Restriction

If you only need to access phpMyAdmin from home, you can restrict access to your IP address or local network entirely. Anyone from any other IP gets a 403 Forbidden before they even see the login page.

$ sudo nano /etc/apache2/conf-enabled/phpmyadmin.conf
# Find the <Directory /usr/share/phpmyadmin> block # Add access controls inside it — below the existing Options/AllowOverride lines <Directory /usr/share/phpmyadmin> Options SymLinksIfOwnerMatch DirectoryIndex index.php AllowOverride None # ── IP restriction — choose ONE of these approaches ── # Option A: Allow only your home IP (replace with your actual public IP) Require ip 203.0.113.42 # Option B: Allow your entire LAN (if accessing from local network only) Require ip 192.168.1.0/24 # Option C: Allow multiple specific IPs (home + work VPN, etc.) Require ip 203.0.113.42 198.51.100.15 # Option D: Allow loopback only (access via SSH tunnel — most restrictive) Require ip 127.0.0.1 ::1 </Directory> $ sudo systemctl reload apache2
Cloudflare Tunnel users — use Option D (localhost only): if your server's internet access goes through Cloudflare Tunnel, incoming requests arrive from localhost or Cloudflare's own IPs — not your home IP. The cleanest approach is to restrict phpMyAdmin to localhost (`Require ip 127.0.0.1`) and access it via an SSH tunnel from your dev machine: `ssh -L 8080:127.0.0.1:80 user@server -N`, then browse to `http://127.0.0.1:8080/dbadmin-x7k2`. phpMyAdmin is never exposed to the internet at all. Alternatively, use Cloudflare Access (covered at the end of this chapter).
Find your current public IP (run this on your home machine, not the server) $ curl -s https://ifconfig.me 203.0.113.42 # Test the restriction is working (403 = access denied from the wrong IP) $ curl -I http://yourserver/dbadmin-x7k2 HTTP/1.1 403 Forbidden

Step 3 — HTTP Basic Auth (Second Credential Layer)

HTTP Basic Auth adds an Apache-level password prompt that appears before the phpMyAdmin login form. Even if the URL is found and your IP restriction isn't in place, an attacker needs a second set of credentials.

Create the htpasswd file with a username (e.g. "dbadmin") # -c creates the file — only use -c the first time, it overwrites $ sudo htpasswd -c /etc/phpmyadmin/.htpasswd dbadmin New password: [enter a strong password — different from your MySQL passwords] Re-type new password: Adding password for user dbadmin # Add a second user later (no -c flag) $ sudo htpasswd /etc/phpmyadmin/.htpasswd anotheruser # Restrict the htpasswd file so only root and www-data can read it $ sudo chown root:www-data /etc/phpmyadmin/.htpasswd $ sudo chmod 640 /etc/phpmyadmin/.htpasswd
Add Basic Auth directives to the phpmyadmin.conf Directory block $ sudo nano /etc/apache2/conf-enabled/phpmyadmin.conf <Directory /usr/share/phpmyadmin> Options SymLinksIfOwnerMatch DirectoryIndex index.php AllowOverride None Require ip 127.0.0.1 # keep your IP restriction too # ── HTTP Basic Auth ── AuthType Basic AuthName "Database Administration" AuthUserFile /etc/phpmyadmin/.htpasswd # Require BOTH a valid IP AND a valid htpasswd credential <RequireAll>

Require ip 127.0.0.1 Require valid-user </RequireAll> </Directory> $ sudo systemctl reload apache2 # Visiting /dbadmin-x7k2 now shows a browser password prompt first, # then phpMyAdmin's own login page after that prompt is passed

Step 4 — config.inc.php Hardening

$ sudo nano /etc/phpmyadmin/config.inc.php
<?php /* Authentication type and info */ $cfg['Servers'][$i]['auth_type'] = 'cookie'; // browser cookie — the right choice /* ── blowfish_secret: MUST be set to a 32-character random string ── */ /* Used to encrypt the cookie phpMyAdmin stores login credentials in. */ /* An empty or short secret makes cookies forgeable. */ $cfg['blowfish_secret'] = 'Rk7vX2mQp9wYnLjHd4sBtZeAcNuF8gK3'; /* Generate your own: php -r "echo bin2hex(random_bytes(16)) . PHP_EOL;" */ /* ── Disable root login ── */ $cfg['Servers'][$i]['AllowRoot'] = false; /* ── Disable no-password logins ── */ $cfg['Servers'][$i]['AllowNoPassword'] = false; /* ── Session timeout (seconds) — log out idle sessions ──────────────── */ $cfg['LoginCookieValidity'] = 1440; // 24 minutes /* ── Hide phpMyAdmin version in page title (minor obscurity) ────────── */ $cfg['ShowPhpInfo'] = false; $cfg['ShowChgPassword'] = true;
# Generate a random 32-character blowfish_secret $ php -r "echo bin2hex(random_bytes(16)) . PHP_EOL;"
Rk7vX2mQp9wYnLjHd4sBtZeAcNuF8gK3 # Test config — phpMyAdmin should show no warnings at the bottom of the page # Look for a green bar or no "Configuration of phpMyAdmin" warning section

Logging In

After passing HTTP Basic Auth (if configured), you'll see phpMyAdmin's own login form. Log in with a MySQL user account — the same credentials you'd use in a `mysql -u username -p` command.

Don't log in as root: phpMyAdmin running as root can drop any database, modify any user, or delete system tables with a single mis-click. Log in as an application user (like `bookshop_app`) for day-to-day browsing, or create a dedicated admin user with only the privileges you need for admin tasks. If you've followed Step 4, root login is blocked by `AllowRoot = false` anyway.
Create a phpMyAdmin admin user in MySQL — with all privileges except GRANT OPTION mysql> CREATE USER 'pma_admin'@'localhost' IDENTIFIED BY 'StrongAdminPass!'; mysql> GRANT ALL PRIVILEGES ON *.* TO 'pma_admin'@'localhost'; mysql> FLUSH PRIVILEGES; # This user can do anything except create other users with GRANT OPTION. # Use this account in phpMyAdmin for admin tasks. # Use bookshop_app (from Chapter 3) when browsing application data only.

Navigating the Interface

Left panel → Server
The top level when you first log in. Shows all databases your user can see. Click a database name to expand it.
Left panel → Database
Expanding a database shows all its tables. Click a table name to open it and see its data.
Left panel → Table
Click a table to go directly to its Browse view (data rows) in the main panel.
Tab: Structure
Columns and indexes. Shows every column with type, null, default, and index status. Add/edit/drop columns here. Click an index entry to view or delete it.
Tab: SQL
Run arbitrary SQL. A multi-line editor pre-scoped to the current database. Run SELECT, INSERT, UPDATE, CREATE TABLE — anything. Results appear below the editor with column headers you can click to sort.
Tab: Browse
All rows in the table with pagination. Click any row's edit icon (pencil) to update it inline. Tick checkboxes to delete multiple rows.
Tab: Insert
Form to insert a new row — one field per column. Useful for adding test data without writing SQL.
Tab: Search
Per-column filter form — builds a WHERE clause for you. Useful for finding rows without writing SQL.
Tab: Export
Download a SQL dump of the database or selected tables. Equivalent to `mysqldump` via the browser.
Tab: Import
Upload and run a SQL file. Use for restoring from a mysqldump file or running a large schema migration. Size limit set by PHP's `upload_max_filesize`.
Top nav: Privileges

Visible at the Server level. Shows all MySQL users, their hosts, and granted privileges. Add/edit/drop users here — the UI equivalent of CREATE USER, GRANT, and REVOKE.

Common Tasks Through the UI

Create a new database
Server level → Databases tab → "Create database"
Enter the database name, choose **utf8mb4_unicode_ci** as the collation (not utf8 — see Chapter 8), click Create. The database appears in the left panel immediately.
Create a table
Click database in left panel → "Create table" panel on the right
Enter table name and number of columns, click Go. You then fill in each column: name, type (INT, VARCHAR, TEXT, DATETIME…), length, null, default, index. Set your primary key column's Index to PRIMARY. Click Save to generate and run the CREATE TABLE statement — phpMyAdmin shows you the SQL it executed.
Browse and edit table data
Click table name → Browse tab (default view)
Click the pencil icon on any row to edit it inline. Click the red X to delete a row (you'll get a confirmation). Use the checkbox column to tick multiple rows, then use the "With selected" dropdown at the bottom to delete them in bulk.
Run a SQL query
Click database → SQL tab (or click a table → SQL tab)
Type your SQL in the editor. Click Go (or press Ctrl+Enter). Results appear as a sortable table. Click any column header to sort. You can export the result set directly from the results view. The query history at the bottom of the page shows all previous queries in this session — click one to re-run it.
Export (backup) a database
Click database → Export tab → Quick → Go
Quick export produces a standard mysqldump-compatible SQL file. Use Custom export if you need to select specific tables, exclude data (schema only), or add INSERT IGNORE instead of INSERT. The output downloads as a .sql file to your browser's downloads folder.
Import a SQL file
Click database → Import tab → Choose file → Go
Select the .sql or .sql.gz file. phpMyAdmin runs the statements and reports success or the first error. For large files the PHP `upload_max_filesize` limit applies — check it in phpinfo() if imports fail silently. For very large databases, use `mysql bookshop < dump.sql` from the command line instead.
Create / manage MySQL users
Server level → User accounts tab
Click "Add user account" to get the full CREATE USER form: username, host, password strength indicator, and a privilege matrix where you check the privileges to grant. Click "Go" to create the user — phpMyAdmin runs CREATE USER and GRANT automatically. To edit an existing user, click their "Edit privileges" link.

The SQL Tab — Tips for Power Use

── Bookmarked / favourite queries ──────────────────────────── # After running any query, tick "Bookmark this SQL query" before clicking Go # Saved bookmarks appear in the dropdown below the SQL editor # Useful for monitoring queries from Chapter 7 you run regularly # ── Multi-statement execution ──────────────────────────────── # phpMyAdmin runs multiple SQL statements separated by semicolons CREATE TABLE IF NOT EXISTS authors (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(200) NOT NULL); INSERT INTO authors (name) VALUES ('Frank Herbert'), ('Isaac Asimov'); SELECT * FROM authors; # All three statements run in sequence — results for SELECT appear below # ── EXPLAIN from phpMyAdmin ────────────────────────────────── # Write any SELECT, then tick "Show query execution plan" before clicking Go # phpMyAdmin runs EXPLAIN automatically and shows the execution plan inline

Cloudflare Access — A Better Auth Layer for Tunnel Users

If your server is behind a Cloudflare Tunnel, Cloudflare Access provides a more elegant authentication layer than HTTP Basic Auth. Instead of an htpasswd prompt, users see a Cloudflare-branded one-time passcode or Google/GitHub OAuth login before reaching phpMyAdmin.

Cloudflare Access setup (free tier, Zero Trust dashboard): in the Cloudflare dashboard, go to Zero Trust → Access → Applications → Add an application. Choose "Self-hosted". Set the domain to your phpMyAdmin subdomain (e.g. `dbadmin.yoursite.com`). Create an Access Policy: allow your email address, or a Google Workspace domain, or one-time PIN. The tunnel proxies the request, Cloudflare validates the identity, and

then (and only then) the request reaches Apache. Your Apache IP restriction can then be set to Cloudflare's gateway IP ranges — or kept as localhost if the tunnel connects locally.

The advantage: Cloudflare Access logs every access attempt, supports MFA, and blocks bots entirely before they reach your server. The free tier allows unlimited users for up to 50 applications. If you're already using Cloudflare Tunnel for the rest of your site, this is the lowest-friction way to protect phpMyAdmin.

Keeping phpMyAdmin Updated

phpMyAdmin has had several serious security vulnerabilities over the years, including XSS and SQL injection flaws in the admin interface itself. Keeping it updated is not optional.

Update phpMyAdmin with the rest of the system $ sudo apt update && sudo apt upgrade phpmyadmin # Check installed version $ dpkg -l phpmyadmin | grep phpmyadmin ii phpmyadmin 4:5.2.1+dfsg-1 all MySQL web administration tool # Check the version in the UI: phpMyAdmin → About in the top navigation bar # The Debian/Ubuntu packaged version often lags behind upstream releases. # For the latest phpMyAdmin on Ubuntu, use their APT repo or download directly: # https://www.phpmyadmin.net/downloads/ # (The apt package is fine for most home/self-hosted use cases.) # Add unattended-upgrades for security patches (if not already done) $ sudo apt install unattended-upgrades $ sudo dpkg-reconfigure -plow unattended-upgrades # Select "Yes" to enable automatic security updates

Troubleshooting

http://yourserver/phpmyadmin returns 404 after install

The Apache config wasn't linked during installation. Fix: **1)** Check if the config exists: `ls /etc/apache2/conf-available/phpmyadmin.conf`. **2)** Enable it manually: `sudo a2enconf phpmyadmin && sudo systemctl reload apache2`. If the file doesn't exist in conf-available, the installer didn't create it — re-run: `sudo dpkg-reconfigure phpmyadmin` and select apache2 when prompted.

Warning: "The configuration file now needs a secret passphrase (blowfish_secret)"

The `blowfish_secret` in `config.inc.php` is empty or too short. Generate one: `php -r "echo bin2hex(random_bytes(16)) . PHP_EOL;"` — copy the 32-character output and paste it into `$cfg['blowfish_secret'] = 'your32chars';` in `/etc/phpmyadmin/config.inc.php`. The warning disappears on the next page load (no Apache restart needed).

Cannot log in — "Access denied for user"

The MySQL user exists but can't connect via socket. Check: **1)** The user must be `'username'@'localhost'` (not `'%'`) since phpMyAdmin on the same machine connects via the Unix socket. **2)** Run: `sudo mysql -e "SELECT user, host, plugin FROM mysql.user WHERE user='youruser';"` — confirm the host is `localhost`. **3)** If the plugin is `auth_socket`, phpMyAdmin can't authenticate with a password — change it: `ALTER USER 'youruser'@'localhost' IDENTIFIED WITH mysql_native_password BY 'pass';`

403 Forbidden after adding IP restriction

Your current IP isn't in the `Require ip` list. Your public IP may have changed (common with home broadband). Check your current IP: `curl -s https://ifconfig.me` (run this on your home machine, not the server). Update the `Require ip` line in `/etc/apache2/conf-enabled/phpmyadmin.conf` and reload Apache. Consider using a CIDR range for your ISP's block instead of a single IP.

Import fails for large SQL files — no error, just stops

PHP's upload and execution limits are too low for large files. Check the limits phpMyAdmin shows on its Import page (it lists the current values). Increase them in `/etc/php/8.X/apache2/php.ini`: set `upload_max_filesize = 128M`, `post_max_size = 128M`, and `max_execution_time = 300`. Restart Apache. For files over 100 MB, skip phpMyAdmin entirely and use the command line: `sudo mysql bookshop < /path/to/dump.sql` — no size limits there.

Quick Reference — Chapter 9

Task	How
Install	`sudo apt install phpmyadmin` — select apache2, yes to dbconfig, set a password
Enable Apache config	`sudo a2enconf phpmyadmin && sudo systemctl reload apache2`
Change URL alias	Edit `/etc/apache2/conf-enabled/phpmyadmin.conf` — change Alias line, reload Apache
IP restriction	Add `Require ip <your-ip>` inside the Directory block in phpmyadmin.conf
HTTP Basic Auth	`sudo htpasswd -c /etc/phpmyadmin/.htpasswd user`, add Auth* directives to Directory block
blowfish_secret	`php -r "echo bin2hex(random_bytes(16));"` → paste into `/etc/phpmyadmin/config.inc.php`
Disable root login	`$cfg['Servers'][$i]['AllowRoot'] = false;` in config.inc.php
Create database	Server → Databases tab → Create database — choose utf8mb4_unicode_ci
Run SQL	Click database → SQL tab → type query → Go (or Ctrl+Enter)

Task	How
Export backup	Database → Export → Quick → Go → downloads as .sql
Import SQL file	Database → Import → Choose file → Go (size limited by PHP upload_max_filesize)
Manage users	Server level → User accounts tab → Add user account
Update phpMyAdmin	`sudo apt update && sudo apt upgrade phpmyadmin`
Cloudflare Tunnel users	Restrict to localhost + use Cloudflare Access, or access via SSH tunnel (`ssh -L 8080:127.0.0.1:80 user@server -N`)