MySQL Course

Flavio Copes

Updated August 3, 2026

Contents

Learn MySQL fundamentals: install and navigate the server, manage users and privileges, design tables, inspect queries, connect applications, and protect your data.

What you'll learn: Run a MySQL database for a small application with a restricted user, safe queries, useful indexes, migrations, and a tested backup.

Start using MySQL

Install the server, connect with the client, and navigate databases and tables.

What MySQL is

Understand the client-server database model and where MySQL sits between your application and its data files.

MySQL is a **relational database management system**. Relational means the data lives in tables: rows of records, columns with fixed names and types, and relationships between tables expressed through shared values. You work with all of it through one language, SQL.

The part that shapes everything else in this course is the **client-server model**. A server process owns the data and accepts connections from command-line clients, graphical tools, and applications. Your application does not edit database files directly. It authenticates, sends SQL over a connection, and receives rows or an error from the server.

That indirection is the point. Because every read and write flows through the server, the server can enforce rules that no single application could guarantee on its own: many clients writing at once without corrupting each other, accounts that can read but not delete, and transactions that either fully happen or fully do not. When we create users and grants later in the course, we are configuring exactly this layer.

A typical exchange looks like this. The client sends:

```
SELECT title FROM notes WHERE id = 1;
```

The server checks the account's permissions, finds the row, and answers:

```
+---------------+
| title         |
+---------------+
| Plan the week |
+---------------+
1 row in set (0.00 sec)
```

Every tool you will meet — the `mysql` client, a Node.js driver, a backup program — is just another client having this same conversation.

MySQL earned its place by running an enormous share of the web. WordPress sits on it, and most hosting platforms offer it as a managed service, so the skills here transfer directly to real projects. PostgreSQL is the other major open-source relational database; the SQL fundamentals overlap heavily, so nothing you learn here is wasted if you switch.

This course uses MySQL 8.4 LTS, the long-term support release. The basic SQL also works on newer releases, but installation, upgrades, and defaults can differ between release lines, so the lessons pin one version to keep every command reproducible.

Install MySQL 8.4 on macOS

Install the MySQL 8.4 LTS server and client with Homebrew, then verify the version before creating any data.

Installing MySQL gives you two things: the server, a background process that owns the data and listens for connections, and the `mysql` client you use to talk to it. One install, both pieces.

This course uses the versioned MySQL 8.4 LTS Homebrew formula. Pinning the version matters: **8.4 is a long-term support release**, so its behavior stays stable while you learn, and the versioned formula will not surprise you with a major upgrade during a routine `brew upgrade`.

```
brew install mysql@8.4
brew services start mysql@8.4
```

The first command installs the software. The second registers the server with `launchd` and starts it, so it also comes back after a reboot.

The formula is **keg-only**, which means Homebrew installs it without linking its binaries into your `PATH`. Typing `mysql` right after installation can therefore fail with `command not found` even though the install succeeded — nothing is broken. You can run its client with the full Homebrew path:

```
$(brew --prefix mysql@8.4)/bin/mysql --version
```

The output should start with `mysql Ver 8.4`. That single line verifies both the install and which release line you are on. If you want plain `mysql` to work, add the directory to your `PATH` using the line Homebrew printed at the end of the install; run `brew info mysql@8.4` to see it again.

Confirm the server side too:

```
brew services list
```

The `mysql@8.4` row should say `started`. If it says `error`, check the log file the output points at before reinstalling anything.

If you installed MySQL another way — the official installer, Docker, Linux packages — use that installation's service and client instructions. The SQL in this course is identical everywhere.

Authentication defaults depend on the package and platform. A fresh Homebrew installation may allow a local administrative connection without a password until you run `mysql_secure_installation`. Do not assume another installation behaves the same way.

Start and stop the MySQL server

Run the versioned MySQL 8.4 Homebrew service and verify which server release is active before troubleshooting.

The MySQL server is a long-running background process. Clients, including your future application, only work while it runs, so knowing how to start it, stop it, and check it is the foundation for every other lesson.

Start and stop the versioned Homebrew service with:

```
brew services start mysql@8.4
brew services stop mysql@8.4
```

`start` does two things: it launches the server now and registers it with `launchd`, so it starts again automatically after a reboot. `stop` is the clean way to shut it down — the server flushes its data to disk before exiting. Use it before uninstalling, before restoring data files, or when you want the machine quiet. Prefer it over killing the process.

Check its state before changing passwords or reinstalling anything:

```
brew services list
```

```
Name       Status  User    File
```

```
mysql@8.4  started flavio  ~/Library/LaunchAgents/homebrew.mxcl.mysql@8.4.plist
```

`started` in green means the server is up. `none` means it is not registered, and `error` means it tried to start and failed — in that case read the MySQL error log before retrying, because starting it again usually reproduces the same failure. The service name also tells you which MySQL release Homebrew is running.

What a stopped server looks like

A client cannot connect while the server is stopped. The error is specific:

```
ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)
```

Learn to read this one. It does not mean a wrong password or a missing account — it means nothing is listening at all. When you see it, check `brew services list` first. The fix is almost always `brew services start mysql@8.4`, not touching credentials or configuration.

If you need the server only once, without registering it for automatic startup, `brew services run mysql@8.4` starts it for the current session and it stays down after a reboot.

Connect with the mysql client

Open an administrative command-line session, keep passwords out of shell history, and verify the server and matched account.

The `mysql` command-line client is the tool this course uses for everything administrative. Graphical tools are fine for browsing, but the client shows you exactly what the server said, and every command in it is copy-pasteable into scripts and documentation.

If the administrative account has a password, ask the client to prompt for it:

```
$(brew --prefix mysql@8.4)/bin/mysql -u root -p
```

Do not put the password after `-p`. That can expose it in shell history and process lists. With a bare `-p`, the client prompts for the password and nothing sensitive touches the command line.

A fresh local Homebrew installation may initially use `mysql -u root` without `-p`. Follow the installation output, then secure the account before allowing network access.

When the connection succeeds you land in an interactive session:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 8.4.8 Homebrew


mysql>
```

Statements end with a semicolon. Press Enter without one and the client shows a continuation prompt, waiting for the rest of the statement.

Verify the server after connecting:

```
SELECT VERSION();
SELECT CURRENT_USER();
```

`CURRENT_USER()` shows the full MySQL account that matched the connection, including the host part. That matters because the account you asked for and the account that matched can differ, and grants attach to the matched account. Leave the client with `QUIT;`.

When the login fails

The most common failure looks like this:

```
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)
```

Read it carefully before retyping anything. It tells you which account MySQL tried (`'root'@'localhost'`) and whether a password was sent (`using password: YES`). A wrong password, a missing account, and an account created for a different host all produce this same error, so the account name in the message is your first clue.

Navigate databases and tables

Create the notes database with deliberate Unicode defaults, select it for the session, and inspect its tables and schema.

MySQL calls a named collection of tables a database. One server hosts many of them, so the first navigation skill is knowing which databases exist and which one your session is pointed at.

See what the server already has:

```
SHOW DATABASES;
```

A fresh installation lists system databases such as `mysql`, `information_schema`, and `performance_schema`. Leave those alone — the server uses them for accounts, metadata, and diagnostics.

Create this course's database with an explicit character set and collation:

```
CREATE DATABASE notes_app
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_0900_ai_ci;

USE notes_app;
SHOW TABLES;
```

`USE` selects the database for the rest of the session, so table names in your statements resolve inside it. `SHOW TABLES` returns an empty set right now, which is correct — the database exists and has no tables yet.

If you skip `USE` and run a table statement anyway, MySQL refuses:

```
ERROR 1046 (3D000): No database selected
```

That error is harmless and instantly fixable, but it teaches the session model: the server does not guess which database you meant.

Once tables exist, inspect one with `DESCRIBE`:

```
DESCRIBE notes;
```

The output lists each column with its type, whether it accepts `NULL`, key information, and defaults. It is the quickest answer to "what does this table actually look like?" without reading application code. For the full definition, including indexes and the character set, use `SHOW CREATE TABLE notes;` instead.

Two commands keep you oriented before anything destructive:

```
SELECT DATABASE(), CURRENT_USER();
```

`DATABASE()` returns the currently selected database, or `NULL` when none is selected. Always confirm the active database before running destructive statements. A `DROP TABLE` aimed at the wrong database is exactly the kind of mistake this two-second check prevents.

Check your understanding: MySQL server basics

Check the client-server model, local installation, service control, client authentication, databases, and table inspection.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Users and privileges

Keep root for administration and give applications only the access they need.

Keep root for administration

Keep the powerful root account away from application traffic and separate row access from schema migration responsibility.

The MySQL root account can change users, permissions, databases, and server configuration. Look at what it holds:

```
SHOW GRANTS FOR 'root'@'localhost';
```

The answer is essentially everything, on every database, with the right to grant those powers to others. A bug running through that account can damage far more than one application. One malformed query, one SQL injection hole, one leaked `.env` file — and the blast radius is the entire server, not one database.

That is why this module builds a different setup. Root stays with you, the human administrator, for the handful of tasks that genuinely need it: creating databases, creating accounts, granting privileges. Everything an application does runs through accounts that hold only what that application needs.

Use one account for the running application and another for schema migrations. The application account changes rows. The migration account changes table structure. The split is not bureaucracy — the two jobs have different risk profiles. The application executes untrusted user input all day; the migration tool runs reviewed SQL during a deploy. Give the always-exposed account the smaller set of powers.

You can check at any time which identity a session is using:

```
SELECT CURRENT_USER();
```

```
+----------------+
| CURRENT_USER() |
+----------------+
| root@localhost |
+----------------+
```

If your application's connection ever prints `root@localhost` there, stop and fix the configuration before adding features.

The failure mode you are preventing is easy to picture. An application connected as root plus one injection vulnerability equals an attacker who can read every database on the server, create their own account, and drop whatever they like. The same vulnerability through a restricted account is still an incident, but a bounded one: it touches one database, and only through the privileges you deliberately granted.

The next lessons create those restricted accounts and grant them exactly what they need.

Create MySQL application accounts

Create separate local accounts for application queries and schema migrations without giving either one global access.

MySQL accounts include both a name and a connection host. We need two of them for the notes project: one identity for the running application and one for schema migrations. Separate accounts mean separate passwords, separate privileges, and a clear answer to "what can this credential do if it leaks?"

Connect as the administrator and create both local accounts:

```
CREATE USER 'notes_app'@'localhost'
  IDENTIFIED BY 'replace-with-a-generated-secret';

CREATE USER 'notes_migrator'@'localhost'
  IDENTIFIED BY 'replace-with-another-generated-secret';
```

Use generated secrets in real deployments. Do not copy the example text into production.

Verify that both accounts exist:

```
SELECT user, host FROM mysql.user WHERE user LIKE 'notes%';
```

You should see two rows, both with `localhost` as the host. Then prove the login works from a new terminal:

```
mysql -u notes_app -p
```

The connection succeeds, and that is all it does. Creating an account does not grant access to `notes_app` or any other database. Run `SHOW DATABASES;` in that session and you see almost nothing, because the account has no privileges yet. We will add only the privileges each responsibility needs.

When the statement fails

Running `CREATE USER` twice for the same account produces:

```
ERROR 1396 (HY000): Operation CREATE USER failed for 'notes_app'@'localhost'
```

The account already exists. In a script you want to run repeatedly, use `CREATE USER IF NOT EXISTS ...` so the statement succeeds whether or not the account is there. If you instead need to start over — for example, you lost the generated password — drop the account with `DROP USER 'notes_app'@'localhost';` and create it again. There is no way to read an existing password back out of MySQL, and that is by design.

A MySQL account includes its host

Match an account to the real connection source and use specific remote hosts with TLS instead of normalizing the percent wildcard.

MySQL treats `'notes_app'@'localhost'` and `'notes_app'@'10.0.0.24'` as different accounts. An **account** is the pair of user name and host, never the name alone. Always write the full account name in `CREATE USER`, `GRANT`, `SHOW GRANTS`, and `REVOKE`.

The host part answers one question: where may this connection come from? When a client connects, the server looks for an account whose host matches the connection's origin. No matching host means no matching account, and the login fails even with the correct password.

Use `localhost` when the application and MySQL share a machine. For a remote application, use its specific private host or network pattern and require TLS:

```
CREATE USER 'notes_app'@'10.0.0.24'
  IDENTIFIED BY 'replace-with-a-generated-secret'
  REQUIRE SSL;
```

`REQUIRE SSL` rejects unencrypted connections for this account. Credentials that cross a network should never travel in plain text.

See which accounts exist

List every account for a user name:

```
SELECT user, host FROM mysql.user WHERE user = 'notes_app';
```

The output makes duplicates visible. If you see both `localhost` and `%` rows for the same user, they are separate accounts with separate passwords and separate grants.

The wrong fix

Here is the classic trap. The application moves to a new server and logins start failing:

```
ERROR 1045 (28000): Access denied for user 'notes_app'@'10.0.0.31' (using password: YES)
```

The password is correct. The problem is that the account was created for `10.0.0.24`, and the connection now arrives from `10.0.0.31`. The error message shows you the real origin, which is exactly the evidence you need.

The `%` host wildcard accepts every source host. Do not use it as the normal fix for a failed connection. It converts a precise access rule into "anyone who knows the password, from anywhere." Check DNS, network rules, TLS, and the real application source first, then create the account for the host the error message reported.

Grant MySQL privileges

Give the notes application row access while keeping table creation and schema changes in a separate migration account.

A new account can log in and do nothing else. `GRANT` adds capabilities, and each grant names three things: which privileges, on what scope, to which account. The scope `notes_app.*` means every table in the `notes_app` database and nothing outside it.

The running application needs to read and change rows. It does not need to create or drop tables:

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON notes_app.*
TO 'notes_app'@'localhost';
```

The migration account changes the schema. It also needs row access for backfills:

```
GRANT SELECT, INSERT, UPDATE, DELETE,
  CREATE, ALTER, INDEX, DROP, REFERENCES
ON notes_app.*
TO 'notes_migrator'@'localhost';
```

Neither account receives privileges on `*.*`, the global scope. A leaked application credential is now limited to one database and cannot change its tables.

`GRANT` takes effect immediately for new connections. You do not need `FLUSH PRIVILEGES` after it — that command is only for the rare case where someone edited the grant tables directly with `INSERT` or `UPDATE`.

Verify the boundary

Check what an account holds:

```
SHOW GRANTS FOR 'notes_app'@'localhost';
```

The output lists one line per grant. For `notes_app` you should see exactly two: a `GRANT USAGE ON *.*` line, which means "may connect, nothing more" and appears for every account, and the `SELECT, INSERT, UPDATE, DELETE` grant on `notes_app.*` you just issued. Anything beyond those two lines is access this account should not have.

Reading grants is necessary but not sufficient. The convincing check is behavioral: test the boundary by connecting as `notes_app` and trying something the account must not be able to do:

```
DROP TABLE notes;
```

MySQL should return an access-denied error:

```
ERROR 1142 (42000): DROP command denied to user 'notes_app'@'localhost' for table 'notes'
```

A failed dangerous action proves the restriction works. My advice is to run this negative test every time you set up accounts. It takes ten seconds and catches the day someone "temporarily" granted too much.

Inspect and revoke privileges

Verify the full account and privilege scope, then remove access with MySQL's REVOKE FROM syntax and test the result.

Privileges drift. Someone grants extra access during an incident, the incident ends, and the grant stays. Reviewing what each account holds needs to be as routine as reviewing code.

Inspect an account with:

```
SHOW GRANTS FOR 'notes_app'@'localhost';
```

```
+-----------------------------------------------------------------------------------+
| Grants for notes_app@localhost                                                    |
+-----------------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO `notes_app`@`localhost`                                     |
| GRANT SELECT, INSERT, UPDATE, DELETE ON `notes_app`.* TO `notes_app`@`localhost`  |
+-----------------------------------------------------------------------------------+
```

Read the output as a complete inventory. Anything not listed, the account cannot do. Note that

you must name the full account including its host — `SHOW GRANTS FOR 'notes_app'@'%'` would describe a different account, if it exists at all.

Suppose a review decides the running application should not delete rows anymore. Remove a privilege with `REVOKE ... FROM` and the same scope used by `GRANT`:

```
REVOKE DELETE
ON notes_app.*
FROM 'notes_app'@'localhost';
```

The scope has to match. Privileges granted on `notes_app.*` are revoked on `notes_app.*`; you cannot subtract a single table from a database-level grant.

Prove the change

Run `SHOW GRANTS` again and confirm `DELETE` disappeared from the list. Then test both directions from a session connected as `notes_app`:

```
SELECT COUNT(*) FROM notes;   -- still works
DELETE FROM notes WHERE id = 1;
```

The `SELECT` succeeds and the `DELETE` fails with `ERROR 1142 (42000): DELETE command denied to user 'notes_app'@'localhost' for table 'notes'`. That pair of results is the verification: you removed exactly the capability you intended and nothing else.

One catch: an existing connection may keep working with the privileges it had, because some privilege changes only apply to new sessions or the next `USE` statement. Restart the application or its pool after a revoke so every connection picks up the reduced grants, and re-run the behavioral test through the application itself.

Grant `DELETE` back only if the running application genuinely needs it. The default answer to "does this account need that privilege?" should be no, backed by the `SHOW GRANTS` output that proves it.

Check your understanding: users and privileges

Check root usage, account hosts, least privilege, GRANT scopes, SHOW GRANTS, and REVOKE.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Schema and performance

Choose types, keys, character sets, indexes, and transaction behavior.

Choose MySQL column types

Choose exact numeric, text, and time types while separating stored instants from local wall-clock dates and times.

Every column type is a promise about what the column stores and how MySQL compares it. Pick each one deliberately, because changing a large table later can be expensive.

Numbers

Use integer types for whole numbers and `DECIMAL` for exact decimal values. For example, `DECIMAL(5,2)` stores up to five digits with two after the point, so an estimate such as `12.50` comes back exactly as you stored it.

Avoid `FLOAT` and `DOUBLE` for money and quantities you sum. They store binary approximations, and small rounding errors accumulate across additions. `DECIMAL` exists precisely to avoid that.

Text

Use `VARCHAR` for bounded text and `TEXT` for longer content:

```
title VARCHAR(200) NOT NULL,
body TEXT
```

The `VARCHAR` length counts characters, not bytes, and it is a real constraint. In the default strict SQL mode, inserting a 300-character title into `VARCHAR(200)` fails:

```
ERROR 1406 (22001): Data too long for column 'title' at row 1
```

That error is a feature. It tells you at write time that the data does not match the model, instead of silently truncating it.

Dates and times

`DATETIME` stores the date and time you give it without time-zone conversion. It fits a local wall-clock value such as "2026-08-10 at 09:00 in the user's chosen zone." Store the zone separately.

`TIMESTAMP` converts between the session time zone and UTC. It fits created and updated instants when every connection uses a deliberate time-zone setting. In MySQL 8.4 it also has a smaller supported range than `DATETIME`: it ends in January 2038, while `DATETIME` reaches the year 9999.

Here is the pattern in one table:

```
CREATE TABLE notes (
  estimated_hours DECIMAL(5,2),
  remind_at DATETIME,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

`created_at` records an instant, so `TIMESTAMP` with a default fits. `remind_at` is a wall-clock appointment, so `DATETIME` fits. Check a column's actual definition anytime with `SHOW CREATE TABLE notes;`.

Primary keys and AUTO_INCREMENT

Give each row a stable identifier and let MySQL allocate ordinary numeric identifiers when that fits the model.

A **primary key** is the column that identifies exactly one row. It must be unique and it can never be `NULL`. Every table you design in this course gets one, because without it there is no reliable way to update or delete a specific row, and no way for other tables to reference it.

A common primary key looks like this:

```
id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
```

`AUTO_INCREMENT` tells MySQL to allocate the value itself: each insert that omits `id` receives the next number. You never generate identifiers in application code, and two concurrent inserts never collide.

Insert a row and read back the identifier the server chose:

```
INSERT INTO notes (title) VALUES ('Plan the week');
SELECT LAST_INSERT_ID();
```

`LAST_INSERT_ID()` returns the value generated for your connection, unaffected by inserts from other sessions. Application drivers expose the same number — in mysql2 it arrives as `result.insertId` — so you can insert a note and immediately use its id for related rows.

The database enforces uniqueness for you. Insert an explicit `id` that already exists and MySQL refuses:

```
ERROR 1062 (23000): Duplicate entry '1' for key 'notes.PRIMARY'
```

That rejection is the primary key doing its job. Fix the insert, not the constraint.

Identifiers, not information

Two habits keep auto-increment values trouble-free.

First, expect gaps. A rolled-back transaction or a deleted row leaves holes in the sequence, and MySQL does not reuse them. Gaps are normal and harmless. Code that assumes `MAX(id)` equals the row count, or that ids form an unbroken series, is wrong.

Second, the value identifies a row inside the database and means nothing more. Do not expose assumptions about consecutive identifiers to application behavior or authorization. "The user can see note 42 because they asked for note 42" is a security hole — check ownership in the query, and treat the id as an opaque handle.

Use utf8mb4 for text

Store complete Unicode text and test how the chosen collation changes equality, uniqueness, and sorting in the application.

A **character set** defines which characters a column can store. Use `utf8mb4` for new MySQL applications: it stores every Unicode character, using up to four bytes each.

The trap is historical. MySQL 8.4 keeps the old `utf8` name as a deprecated alias for the three-byte `utf8mb3` character set. Despite the name, `utf8mb3` cannot store characters that need four bytes — which includes every emoji and many CJK characters. With a `utf8mb3` column, saving a title like `Ship it 🚀` fails in the default strict mode:

```
ERROR 1366 (HY000): Incorrect string value: '\xF0\x9F\x9A\x80' for column 'title' at row 1
```

In older non-strict setups the same insert silently truncated the text at the emoji, which is worse: the data loss surfaced weeks later as user complaints. So spell out `utf8mb4` and never rely on the bare `utf8` name.

Inspect what the current database uses:

```
SELECT @@character_set_database, @@collation_database;
```

For our `notes_app` database this returns `utf8mb4` and `utf8mb4_0900_ai_ci`, because we created it with explicit settings.

Collations decide comparisons

A **collation** controls equality and sorting. `utf8mb4_0900_ai_ci` is accent-insensitive and case-insensitive, so a unique tag named `Résumé` conflicts with `resume`. You can watch it happen:

```
INSERT INTO tags (name) VALUES ('Résumé');
INSERT INTO tags (name) VALUES ('resume');
-- ERROR 1062 (23000): Duplicate entry 'resume' for key 'tags.name'
```

The two strings differ byte for byte, yet the collation calls them equal, and the unique index enforces that definition of equality.

That behavior is often useful for names and tags — users expect `Flavio` and `flavio` to be the same person. If case or accents carry meaning in your product, choose and test a different collation before storing data. `utf8mb4_0900_as_cs` compares accent- and case-sensitively. Changing a collation after millions of rows exist means rebuilding indexes and re-checking uniqueness, so this is a decision to make early.

Inspect queries with EXPLAIN

Read the query plan before adding an index and verify that a new index helps the actual access pattern.

Prefix a query with `EXPLAIN` to see table order, access type, candidate keys, selected key, and estimated rows.

Start with the query the application actually runs:

```
EXPLAIN
SELECT id, created_at
FROM orders
WHERE customer_id = 42
ORDER BY created_at DESC
LIMIT 20;
```

Read the plan as a prediction of how MySQL will find the rows. Useful fields include `type`, the possible and selected keys, estimated `rows`, filtered percentage, and `Extra`. A full scan is not automatically wrong—a small table may be cheaper to scan—but a large estimate for a selective lookup deserves attention.

This query suggests a compound index:

```
CREATE INDEX orders_customer_created_idx
ON orders(customer_id, created_at);
```

Column order follows the access pattern. MySQL can locate one customer's index range and read it in date order. The same index is less helpful for a query that only filters by `created_at`, because `customer_id` is its leading column.

Run `EXPLAIN` again after adding the index. Check that MySQL selected it and that the estimated work improved. Then measure the real query under representative data; optimizer estimates can be wrong, and a plan change is not the same as a user-visible speedup. `EXPLAIN ANALYZE` can execute the statement and report actual timing and row counts, so use it carefully on statements with production cost.

Every index consumes storage and adds work to inserts, deletes, and updates of indexed columns.

Overlapping indexes can be redundant. Add one for a measured read path, confirm the benefit, and keep the smallest index that supports it.

Try it: compare an index on `(customer_id)` with one on `(customer_id, created_at)` for the query above. Inspect both the plan and execution time before deciding which to keep.

Use InnoDB transactions

Keep related changes atomic, release locks quickly, and handle deadlocks and lock-wait timeouts with bounded transaction retries.

InnoDB is the normal storage engine for application tables. It supports transactions, row-level locking, crash recovery, and foreign keys.

A **transaction** groups statements so they succeed or fail together. Creating a note and attaching its tag is one logical change; you never want the note without the tag row:

```
START TRANSACTION;

INSERT INTO notes (title) VALUES ('Plan the week');
INSERT INTO note_tags (note_id, tag_id) VALUES (LAST_INSERT_ID(), 1);

COMMIT;
```

Until `COMMIT`, other sessions do not see either row. If anything goes wrong in between, `ROLLBACK` undoes both inserts and the database returns to its previous state. Verify this yourself: run the inserts, issue `ROLLBACK` instead of `COMMIT`, and `SELECT` shows the note never existed.

By default MySQL runs with `autocommit` enabled, so each standalone statement is already its own small transaction. `START TRANSACTION` is how you make a bigger unit.

Keep the transaction focused. It holds row locks and one connection from `START TRANSACTION` to `COMMIT`, and everyone else who needs those rows waits. Do not call an external API, send an email, or wait for user input inside a transaction.

When transactions collide

Concurrent transactions can fail in two documented ways:

```
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

A **deadlock** means two transactions each hold a lock the other needs, so InnoDB kills one to unblock the other. A lock-wait timeout means your transaction waited too long for a lock and gave up.

Both errors are transient, and the server itself tells you the fix: roll back the whole transaction, then retry it a small number of times with a short delay. Retry only these lock errors, never validation or syntax errors — retrying a genuinely broken statement just fails again.

Access rows in a consistent order when possible, for example always parent table before join table. That reduces deadlocks, but your application must still handle them. Under enough concurrency they are a normal event, not a bug.

Check your understanding: schema and performance

Check MySQL types, AUTO_INCREMENT keys, Unicode, indexes, EXPLAIN, InnoDB, and transaction behavior.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Applications and operations

Connect through a pool, parameterize queries, migrate, back up, and troubleshoot.

Connect to MySQL from Node.js

Optionally connect from Node.js with mysql2, environment-based credentials, a bounded pool, and explicit connection timeouts.

The application lessons use Node.js as an optional track. You can follow the database and operations lessons without it.

mysql2 is the actively maintained MySQL driver for Node.js. It speaks the MySQL protocol directly, supports prepared statements, and ships a promise API, so you can use `await` instead of callbacks.

Install it:

```
npm install mysql2
```

Create a bounded pool from environment variables:

```
import mysql from 'mysql2/promise'

export const pool = mysql.createPool({
  host: process.env.DB_HOST,
  port: Number(process.env.DB_PORT ?? 3306),
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
  waitForConnections: true,
  connectionLimit: 5,
  queueLimit: 20,
  connectTimeout: 5000,
  enableKeepAlive: true,
})
```

A pool is the right default even for small applications. It reuses connections instead of paying the TCP and authentication handshake on every query, and `connectionLimit` caps how many connections this process can open. Five connections is an example budget, not a universal value. Count every application replica before choosing the limit.

Verify the connection before building anything on top of it:

```
const [rows] = await pool.query('SELECT VERSION() AS version')
```

```
console.log(rows[0].version) // 8.4.x
```

If the credentials are wrong, or the account host does not match where your code runs from, the driver rejects with `ER_ACCESS_DENIED_ERROR`, the same error 1045 you saw in the client lessons. Test the identical host, port, and account with the `mysql` command-line client to decide whether the problem is the driver configuration or the account itself.

Call `await pool.end()` when a script or worker shuts down. It waits for queries in flight, then closes every connection. A long-running web process keeps the pool open until its shutdown hook runs. Without `pool.end()`, a finished script keeps the event loop alive and the process never exits — that hanging script is usually the first pool bug people meet.

Use a bounded connection pool

Budget connections across every process and keep all transaction statements on one checked-out connection with reliable cleanup.

Opening a MySQL connection costs a TCP handshake, a TLS negotiation, and authentication. Paying that on every query would dominate the query time itself. A pool avoids repeated connection setup and queues work when all its connections are busy — that queue is what makes the pool **bounded**, and the bound is what protects the server.

The server enforces its own limit: `max_connections`, 151 by default. When the combined demand of every client exceeds it, new connections fail with `ERROR 1040 (HY000): Too many connections` — including yours, when you try to connect and investigate.

So count the pool size across every application process, not per file. Ten replicas with a pool of ten can open one hundred connections, before workers, migrations, and administration. Check real usage on the server anytime:

```
SHOW STATUS LIKE 'Threads_connected';
```

Compare that number with `max_connections` and keep headroom for an emergency administrative session.

Transactions need one connection

A transaction must stay on one checked-out connection, because MySQL scopes the transaction to the connection that started it:

```
const connection = await pool.getConnection()

try {
  await connection.beginTransaction()

  const [note] = await connection.execute(
    'INSERT INTO notes (title) VALUES (?)',
    ['Plan the week']
  )

  await connection.execute(
    'INSERT INTO note_tags (note_id, tag_id) VALUES (?, ?)',
    [note.insertId, 1]
  )
```

```
  await connection.commit()
} catch (error) {
  await connection.rollback()
  throw error
} finally {
  connection.release()
}
```

Do not call `pool.execute()` inside that transaction. The pool may choose another connection, and that statement would run outside the transaction — committed immediately, invisible to your rollback.

The `finally` block is not decoration. Always roll back after an error and release in `finally`. A code path that returns or throws without `connection.release()` leaks that connection: the pool believes it is still busy forever. Leak a few and every request starts waiting on an empty pool, which looks exactly like a database outage. If the application slows down while `Threads_connected` sits at your pool limit and the database itself is idle, look for a missing `release()` before anything else.

Parameterize MySQL queries

Use placeholders for values so input cannot change the structure of a SQL statement.

Never build SQL by gluing user input into the string. This looks harmless:

```
// vulnerable - do not do this
const [rows] = await pool.query(
  `SELECT id, title FROM notes WHERE title = '${searchTerm}'`
)
```

Now imagine `searchTerm` arrives from a form as `' OR '1'='1`. The final statement becomes `WHERE title = '' OR '1'='1'`, which is true for every row. The input stopped being data and became part of the query structure. That is **SQL injection**, and variants of it read other users' rows, bypass logins, and delete tables.

With mysql2, pass values separately:

```
const [rows] = await pool.execute(
  'SELECT id, title FROM notes WHERE id = ?',
  [noteId]
)
```

Each `?` is a **placeholder**. The statement text and the values travel to MySQL separately, and the server treats every value as data, never as SQL. The injection attempt above would just search for a note literally titled `' OR '1'='1` and find nothing.

Multiple values work the same way — the array fills the placeholders in order:

```
const [rows] = await pool.execute(
  'SELECT id, title FROM notes WHERE title = ? AND estimated_hours < ?',
  ['Plan the week', 3]
)
```

`pool.execute()` uses real prepared statements, so the query structure is fixed on the server before any value arrives.

Verify the protection the same way you verify a grant: attack yourself. Send `' OR '1'='1` through your own search feature. A parameterized query returns an empty result; a vulnerable one returns everything. Two minutes, definitive answer.

One boundary to know: placeholders represent values, not table or column names. `ORDER BY ?` does not do what you hope — the driver would quote the column name as a string. When identifiers must vary, choose them from a fixed allowlist in your code:

```
const sortable = { date: 'created_at', title: 'title' }
const column = sortable[req.query.sort] ?? 'created_at'
```

Input selects an option you wrote; it never becomes SQL text itself.

Keep connection configuration out of code

Keep credentials in deployment secrets, bound connection waits, and verify remote MySQL connections with the provider's TLS certificate.

Connection details change more often than code. The database moves to a new host, a password rotates after someone leaves, staging points at a different server than production. If those values live in your source files, every change means a code change, a commit, and a redeploy — and the secrets sit in git history forever.

Keep the host, port, full account name, password, and database in deployment secrets. Do not commit them, print them in logs, or put them in browser code.

Locally that usually means an env file that stays out of version control:

```
# .env - listed in .gitignore
DB_HOST=127.0.0.1
DB_PORT=3306
DB_USER=notes_app
DB_PASSWORD=replace-with-a-generated-secret
DB_NAME=notes_app
```

In production, use your platform's secret store instead of a file. The code reads `process.env` either way, so nothing changes between environments except the values.

Verify the wiring at startup by logging the non-secret parts:

```
console.log(`db: ${process.env.DB_USER}@${process.env.DB_HOST}/${process.env.DB_NAME}`)
```

Never log the password. This one line catches the classic mistake where production quietly loaded staging values.

Use a short connection timeout and a bounded pool queue. They turn a network or capacity problem into a controlled failure instead of an unlimited wait.

TLS for remote connections

Remote database connections need TLS. Use the certificate authority supplied by the hosting provider:

```
ssl: {
  ca: process.env.DB_CA,
  minVersion: 'TLSv1.2',
}
```

When TLS is misconfigured, the driver fails with a certificate error such as `self-signed certificate in certificate chain` or a hostname mismatch. Do not fix certificate errors with `rejectUnauthorized: false`. That setting keeps the encryption but stops verifying who you are encrypting to, which allows a machine-in-the-middle to read your credentials and data. Fix the CA, hostname, or provider configuration instead.

Run MySQL migrations safely

Use compatible expand-and-contract changes while accounting for MySQL DDL implicit commits and interrupted migrations.

A **migration** is a versioned schema change. Apply migrations in order and record successful versions in the database, so every environment can answer "which changes have already run here?"

The risky part is timing. A deployment is not instantaneous: for a while, old and new application code run against the same database. A migration that instantly renames a column breaks every old instance still querying the old name — and breaks the rollback too, because the old code cannot find its column anymore.

The safe pattern is **expand and contract**. For a risky rename, add the new column first, deploy code that supports both versions, backfill data, switch reads, then remove the old column in a later release:

```
-- expand: old code keeps working, new code can start
ALTER TABLE notes ADD COLUMN headline VARCHAR(200);
```

Backfill existing rows in bounded, restartable steps:

```
UPDATE notes
SET headline = title
WHERE headline IS NULL
LIMIT 1000;
```

Run it repeatedly until it reports `0 rows affected`. The `WHERE headline IS NULL` condition makes the backfill **idempotent**: if it stops halfway, running it again continues instead of redoing finished rows. Only after every reader uses the new column does a later migration drop the old one — the contract step.

MySQL-specific rules

Two behaviors matter more in MySQL than elsewhere.

First, MySQL data-definition statements such as `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` cause an **implicit commit**. You cannot make a multi-statement schema migration atomic by wrapping it in `START TRANSACTION` — MySQL commits at each DDL statement anyway. Plan every migration assuming it can stop between statements, and test what happens when it stops halfway through.

Second, the cost of `ALTER TABLE` depends on the table and the operation. Some changes are instant metadata updates; others rebuild the whole table and block writes for the duration. Check how MySQL executes each `ALTER TABLE` on the table size and version you actually run — rehearse against a restored copy of production data, and you get the real duration for free.

Back up first, make each step restartable, and never let a migration be the first time a statement touches production.

Back up and restore MySQL

Create a consistent InnoDB logical dump, restore it into a clean database, and verify both schema and application data.

A successful backup job does not prove you can recover. The file might be incomplete, or the restore might fail against a real server. You only have a usable backup after you restore and test it. This lesson does both.

`mysqldump` produces a **logical dump**: a plain SQL text file containing the `CREATE TABLE` and `INSERT` statements that rebuild the database. Create one for the InnoDB database:

```
mysqldump -u root -p \
  --single-transaction \
  --routines --events --triggers \
  notes_app > notes_app.sql
```

`--single-transaction` gives a consistent snapshot for transactional InnoDB tables: the dump reads the database as it existed at one instant, even while the application keeps writing. It does not protect nontransactional tables, and schema changes must not run during the dump. A database dump also does not include MySQL accounts and their grants — after restoring on a fresh server, you recreate accounts with the statements from the privileges module.

Because the dump is plain text, you can sanity-check it immediately:

```
head -n 20 notes_app.sql
```

You should see a MySQL dump header followed by SQL. An empty or truncated file means the dump failed, and better to learn that now than during an incident.

Restore and verify

Restore into a clean database, never over the original:

```
mysql -u root -p -e "CREATE DATABASE notes_restore CHARACTER SET utf8mb4 COLLATE utf8mb4_0900
mysql -u root -p notes_restore < notes_app.sql
```

Restoring over `notes_app` itself would destroy your only good copy while testing whether the backup works. The separate database makes the drill safe to repeat.

Verify the restored schema and data:

```
mysql -u root -p notes_restore -e "SHOW TABLES; SELECT COUNT(*) AS notes FROM notes; CHECK TA
```

The table list must match the original, the count must be plausible, and `CHECK TABLE` must report `OK` for each table. Then start a non-production application against `notes_restore` too. A successful dump command alone does not prove recovery works — only a restored database that your application can actually read does.

Troubleshoot a MySQL connection

Separate server availability, network reachability, authentication, permissions, and database selection when a connection fails.

Start with the exact error. Verify that MySQL is running, the host and port are reachable, and the account matches the connection origin.

Treat the connection as layers and stop at the first layer that fails.

First, verify the inputs. Print the resolved host, port, database name, and username, but never the password. `localhost` may use a Unix socket while `127.0.0.1` uses TCP, so test the same transport as the application.

Next, prove that the server is reachable:

```
mysqladmin -h db.example.com -P 3306 ping
```

A timeout points toward DNS, routing, firewall rules, a provider allowlist, or a server that is not listening. An immediate authentication error proves that the network path worked and moves the investigation to credentials or grants.

Connect with the command-line client using the same host, port, account, database, and TLS expectations:

```
mysql -h db.example.com -P 3306 -u app_user -p app_database
```

If this works while the application fails, compare driver settings, environment loading, connection-string escaping, and certificate configuration. If it fails, inspect the server-side account and grants with an administrative connection:

```
SHOW GRANTS FOR 'app_user'@'application-host';
```

MySQL accounts include both a user and a host. `'app_user'@'localhost'` and `'app_user'@'%'` are different accounts, so a correct password can still produce "access denied" from a different origin. Once connected, confirm the selected database with `SELECT DATABASE();` and test the smallest failing query. A successful login does not imply permission to read every table.

TLS failures are another distinct layer. Check whether the server requires encryption, whether the client trusts the certificate authority, and whether the hostname matches the certificate. Do not disable certificate verification as the permanent fix.

Changing every setting at once destroys the evidence that tells you which layer failed.

Try it: take one real connection error and write down which layer it proves succeeded. For example, "access denied" means DNS, TCP reachability, and a responding MySQL server already worked.

Build a MySQL notes application

Build and verify a complete MySQL notes application with restricted accounts, InnoDB relations, safe queries, and a restored backup.

Let's finish the course with one runnable project. Every lesson so far contributed one piece: accounts, grants, schema, transactions, safe queries, and backups. This lab connects them, and each step includes a check that proves it worked.

As the administrator, create the database and accounts using the commands from the privileges module. You need `notes_app` for row access and `notes_migrator` for schema changes. Then connect as `notes_migrator` and create the schema:

```
USE notes_app;

CREATE TABLE notes (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(200) NOT NULL,
  body TEXT,
```

```
  estimated_hours DECIMAL(5,2),
  remind_at DATETIME,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  INDEX notes_title_idx (title)
) ENGINE=InnoDB;

CREATE TABLE tags (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL UNIQUE
) ENGINE=InnoDB;

CREATE TABLE note_tags (
  note_id BIGINT UNSIGNED NOT NULL,
  tag_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (note_id, tag_id),
  CONSTRAINT note_tags_note_fk
    FOREIGN KEY (note_id) REFERENCES notes(id) ON DELETE CASCADE,
  CONSTRAINT note_tags_tag_fk
    FOREIGN KEY (tag_id) REFERENCES tags(id) ON DELETE CASCADE
) ENGINE=InnoDB;
```

If `CREATE TABLE` returns an access-denied error here, you are connected as the wrong account. Run `SELECT CURRENT_USER();` and fix the session before continuing.

Now connect as `notes_app`. Confirm that inserts work and schema changes fail:

```
INSERT INTO notes (title, estimated_hours)
VALUES ('Plan the week', 1.50);

SELECT id, title, estimated_hours, created_at
FROM notes;

ALTER TABLE notes ADD COLUMN should_fail INT;
```

The `SELECT` must return the row you just inserted. The `ALTER TABLE` must return an access-denied error like `ERROR 1142 (42000): ALTER command denied to user 'notes_app'@'localhost'`. That failure is the point: a leaked application credential cannot rewrite your schema.

Try one more deliberate failure. Insert into `note_tags` with a `tag_id` that does not exist. InnoDB rejects it with `ERROR 1452`, a foreign key constraint failure. The schema, not application discipline, protects the relationship.

Run the optional Node.js track with the bounded pool and a parameterized lookup:

```
const [rows] = await pool.execute(
  'SELECT id, title FROM notes WHERE title = ?',
  ['Plan the week']
)

console.log(rows)
await pool.end()
```

Inspect the lookup with `EXPLAIN`, then perform one expand-and-contract migration as

`notes_migrator`.

Finally, dump `notes_app`, restore it into `notes_restore`, run `CHECK TABLE`, and query `Plan the week` from the restored database. The lab is complete only when the restricted account, application query, migration, and restore all work.

Check your understanding: applications and operations

Check pools, placeholders, connection secrets, migrations, logical backups, restore tests, and connection debugging.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/mysql/.