LECTURE 22

Database Connectivity

Connecting a Node/Express app to PostgreSQL and MongoDB — and choosing the right shape for your data.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Compare the relational (PostgreSQL) and document (MongoDB) data models
2. Learn how to choose the right database for a given project
3. Set up a database connection, understand connection strings, and connection pooling
4. Perform CRUD operations from server-side code in both MongoDB and PostgreSQL
5. Design collections/schemas and add basic validation
6. Use environment variables safely and handle database errors properly

What Is a Database?

1. Software dedicated to storing, organizing, and retrieving data reliably
2. Persistence: the data survives — "persists" — beyond the life of the program
3. Two kinds you'll meet constantly: relational (PostgreSQL, MySQL) and document (MongoDB)

PostgreSQL: Tables and Foreign Keys

```
-- users table
| id | name     | email              |
|----|----------|--------------------|
| 1  | Ayesha   | ayesha@email.com   |
| 2  | Bilal    | bilal@email.com    |

-- orders table (references users via user_id)
| id | user_id | total |
|----|---------|-------|
| 1  | 1       | 45.00 |
| 2  | 1       | 12.50 |
```

A JOIN matches rows across tables using the foreign key — here, orders.user_id points back to users.id.

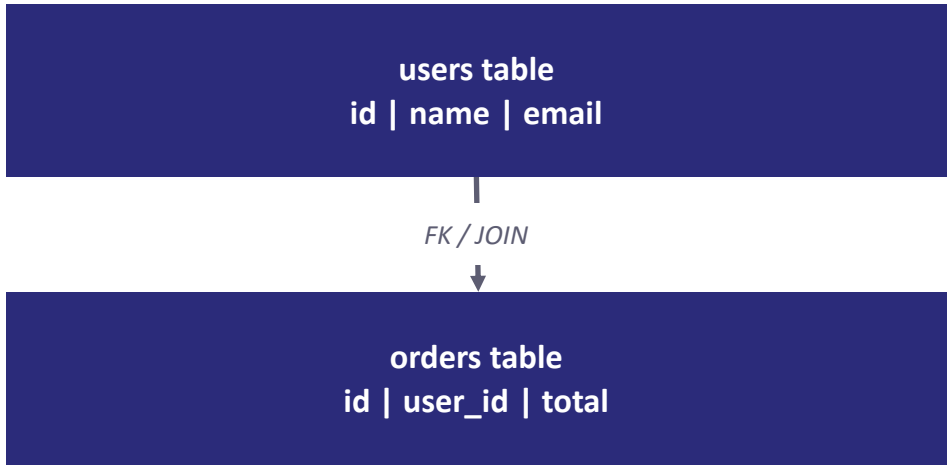
MongoDB: Flexible JSON-Like Documents

```
// A document in the "users" collection
{
  "_id": "64f1a2b3c4d5e6f7a8b9c0d1",
  "name": "Ayesha",
  "email": "ayesha@email.com",
  "orders": [
    { "total": 45.00, "date": "2024-01-15" },
    { "total": 12.50, "date": "2024-02-02" }
  ]
}
```

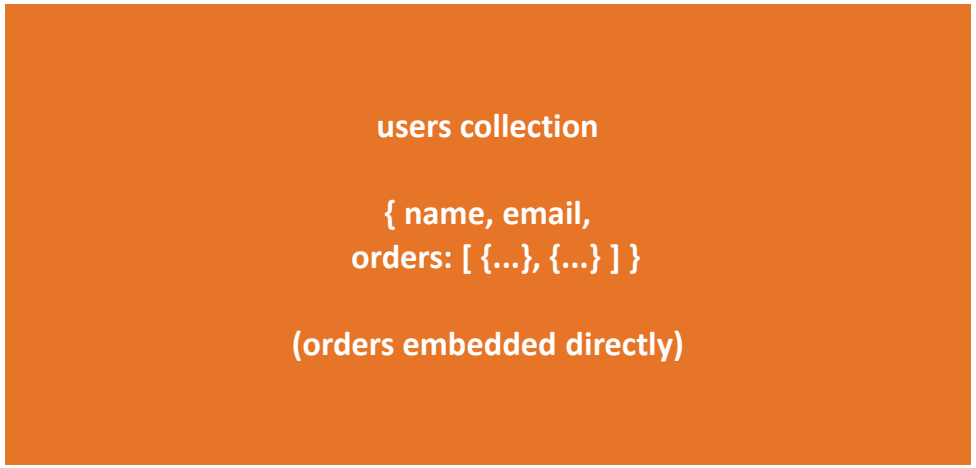
Ayesha's orders live inside her own document — no separate table, no JOIN needed to read them together.

Two Very Different Shapes

RELATIONAL (PostgreSQL)



DOCUMENT (MongoDB)



Relational data lives in separate tables joined by foreign keys; document data is often embedded directly inside the parent document — no join needed.

COMPARING THE TWO

Relational vs. Document, Side by Side

	Relational (PostgreSQL)	Document (MongoDB)
Structure	Tables with fixed columns	Collections of flexible JSON-like documents
Schema	Strict — defined up front, enforced by the database	Flexible — documents in the same collection can differ
Relationships	Foreign keys + JOINs	Embedding (nested data) or manual references
Best for	Clear structure, strong consistency (banking, inventory)	Naturally nested, evolving data (profiles, catalogs)
Query language	SQL	MongoDB Query Language (JS-like methods)
Scaling style	Traditionally scales up; modern versions scale out too	Designed from the start to scale out

COMPARING THE TWO

Terminology: SQL vs. NoSQL

NOTE

PostgreSQL/MySQL are often called SQL databases, after their query language. MongoDB is called a NoSQL database — "not only SQL" — a broad category that includes document, key-value, and other non-relational databases.

There's No Universal "Best"

CHOOSE POSTGRESQL

- Data has a clear, stable structure
- Relationships between entities matter a lot — orders belong to customers, who belong to a store
- You need strong consistency guarantees, e.g. financial transactions

CHOOSE MONGODB

- Data is naturally nested or hierarchical — a blog post with embedded comments
- The schema is likely to evolve quickly during early development
- You mostly read/write whole "documents" at once — a user profile with all its settings

Many Real Systems Use Both

TIP

A relational database for structured, transactional data (like payments), and a document database for flexible content (like product catalogs or activity logs). Choosing a database is a design decision per use case, not a one-time choice for the whole company.

Drivers and Connection Strings

1. A driver is a library that knows how to speak a database's network protocol — mongodb (or mongoose) for MongoDB, pg for PostgreSQL

2. A connection string (URI) packs together protocol, host, port, database name, and credentials

3. Your server passes this string to the driver once, at startup

Connection String Formats

```
mongodb://username:password@localhost:27017/myAppDB

postgresql://username:password@localhost:5432/myAppDB
```

Never Hard-Code Credentials

WARNING

Anyone who sees your source code — including everyone on GitHub, if the repo is public — sees the password too. Always add .env to your .gitignore file; committing real credentials to Git, even a private repo, is one of the most common causes of security breaches in student and professional projects alike.

.env File

```
# .env  (never commit this file to Git)
MONGO_URI=mongodb://localhost:27017/myAppDB
PG_CONNECTION_STRING=postgresql://user:pass@localhost:5432/myAppDB
PORT=3000
```

Reading Environment Variables

```
// at the very top of your entry file
require("dotenv").config();

console.log(process.env.MONGO_URI);
```

npm install dotenv first — it reads .env into process.env at runtime.

MongoClient

```
const { MongoClient } = require("mongodb");
require("dotenv").config();

const client = new MongoClient(process.env.MONGO_URI);

async function main() {
  await client.connect();
  console.log("Connected to MongoDB");
  const db = client.db("myAppDB");
  return db;
}

main().catch(console.error);
```

PostgreSQL: Pool

```
const { Pool } = require("pg");
require("dotenv").config();

const pool = new Pool({
  connectionString: process.env.PG_CONNECTION_STRING,
  max: 10, // max connections kept in the pool
});

async function getUsers() {
  const result = await pool.query("SELECT * FROM users");
  return result.rows;
}
```

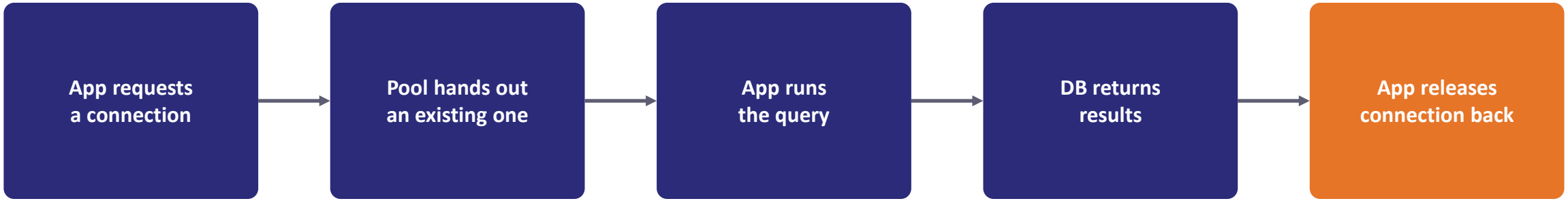
Opening a new connection per query is slow; both drivers reuse a pool of already-open connections instead.

MongoDB Pools Too

NOTE

MongoDB's driver also manages a connection pool internally — by default, up to 100 connections — even though you only call client.connect() once. You rarely need to configure this yourself outside a high-traffic production app.

Borrowing a Connection from the Pool



The pool is a set of already-open connections, reused across many queries instead of opened fresh each time.

Create and Read

```
const users = db.collection("users");

// CREATE
const result = await users.insertOne({
  name: "Ayesha", email: "ayesha@email.com", age: 21
});

// READ
const oneUser = await users.findOne({
  email: "ayesha@email.com"
});
const adults = await users.find({
  age: { $gte: 18 }
}).toArray();
```

insertedId comes back on the result; find() returns a cursor — toArray() collects every match.

Update and Delete

```
// UPDATE
await users.updateOne(
  { email: "ayesha@email.com" },
  { $set: { age: 22 } }
);

// DELETE
await users.deleteOne({
  email: "ayesha@email.com"
});
```

$set only touches the fields you list — every other field on the document is left alone.

Create and Read

```
// CREATE
const result = await pool.query(
  "INSERT INTO users (name, email, age)" +
  " VALUES ($1, $2, $3) RETURNING id",
  ["Ayesha", "ayesha@email.com", 21]
);

// READ
const oneUser = await pool.query(
  "SELECT * FROM users WHERE email = $1",
  ["ayesha@email.com"]
);
```

$1, $2, $3 are placeholders — the driver safely substitutes the array values.

Update and Delete

```
// UPDATE
await pool.query(
  "UPDATE users SET age = $1 WHERE email = $2",
  [22, "ayesha@email.com"]
);

// DELETE
await pool.query(
  "DELETE FROM users WHERE email = $1",
  ["ayesha@email.com"]
);
```

Always Use Parameterized Queries

WARNING

Building queries by concatenating strings opens the door to SQL injection, letting an attacker smuggle their own SQL commands through user input. The MongoDB driver's object-based query syntax is naturally safer the same way, as long as you don't build queries from raw, unvalidated strings yourself.

Plan Your Structure Deliberately

1. MongoDB doesn't force a rigid schema, but you should still decide what fields exist, their types, and whether related data is embedded or referenced

2. MongoDB also supports optional schema validation at the database level

3. PostgreSQL enforces structure automatically, through the table definition itself

MongoDB Schema Validation

```
await db.createCollection("users", {
  validator: { $jsonSchema: {
    bsonType: "object",
    required: ["name", "email"],
    properties: {
      name: { bsonType: "string" },
      email: { bsonType: "string", pattern: "^.+@.+$" },
      age: { bsonType: "int", minimum: 0 }
    }
  } }
});
```

PostgreSQL Table Definition

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(150) UNIQUE NOT NULL,
  age INTEGER CHECK (age >= 0)
);
```

Always Wrap Database Calls

```
app.post("/api/users", async (req, res) => {
  try {
    const result = await usersCollection
      .insertOne(req.body);
    res.status(201).json({ id: result.insertedId });
  } catch (err) {
    console.error("Database error:", err.message);
    res.status(500).json({ error: "Could not create user." });
  }
});
```

Never assume a database call succeeds — the network can drop, a constraint can fail, or the server can be down.

Never Leak Raw Errors to the Client

TIP

Raw database error messages can leak details about your schema or internal setup. Log the full error on the server, and send the client a short, generic message instead.

KEY TAKEAWAYS

Lecture 22 in Six Points

1. Relational databases (PostgreSQL) use tables, fixed schemas, and JOINs; document databases (MongoDB) use flexible, JSON-like documents, often embedding related data.
2. Choose based on your data's shape and your app's needs — structured/relational data favors PostgreSQL, nested/evolving data favors MongoDB.
3. A connection string tells your driver how to reach the database; a connection pool reuses connections instead of opening a new one per query.
4. Keep credentials out of your source code — use environment variables and a .env file, never committed to Git.
5. CRUD operations look different in MongoDB (insertOne, find) versus PostgreSQL (SQL via pool.query), but the underlying goal is the same.
6. Always use parameterized queries to prevent SQL injection, and always wrap database calls in try/catch with sensible error responses.