Databases for Web Technology

John P. Baugh, Ph.D.

Version	Date	Description
v. 2025.11.1.1	11/1/2025	Initial release

Table of Contents

Introduction and Motivation

Web applications are built on two main parts: what the user sees and what the application knows. The HTML, CSS, and JavaScript define the layout and appearance that users interact with in their browsers. The data that drives the content, such as products, users, blog posts, recipes, or comments, is usually stored and managed by a database system.

In early web development, small websites often kept their information in plain text files or simple arrays within scripts. As soon as multiple users needed to access or modify data at the same time, these approaches became unreliable. Data had to be organized, validated, and shared safely. This is where database systems became essential.

A database provides a consistent, centralized way to store, retrieve, and update information. It also enforces data integrity, ensuring that stored values follow defined rules and relationships. For example, if a recipe is deleted, its ingredients should not remain behind as disconnected records. Databases handle these details automatically when properly designed.

In web development, the database often serves as the backbone of the application. The code acts as a bridge, sending queries and commands to the database and then using the results to render dynamic pages. When a visitor views a product, leaves a comment, or logs in, those actions involve database reads or writes behind the scenes.

The examples in this chapter will focus on two major categories of databases:

1. **Relational Databases** – Systems such as MySQL and SQLite that organize data into tables with rows and columns, using keys to link related records. These are the backbone of many traditional web applications and form the focus of our first two examples.
2. **NoSQL Databases** – Systems such as MongoDB that store data in a more flexible format, often as documents (JSON-like structures). These are common in applications that need to handle large volumes of unstructured or rapidly changing data.

The first part of this guide uses **XAMPP**, which includes PHP, MySQL, and the Apache web server. This environment allows you to run and test database-driven web applications locally, without the need for an external host. Later sections will demonstrate a NoSQL example using Node.js and **MongoDB**.

Throughout the guide, the emphasis is on understanding why each step matters, not simply executing code that works. You will design a simple schema, run queries manually in phpMyAdmin, connect to the database from PHP using PDO, and later compare how a NoSQL model stores similar information.

The goal is to understand the database as an integral part of the web application rather than a separate tool. Once this connection is clear, it becomes easier to see how real-world systems such as blogs, online stores, and streaming platforms manage their data efficiently and securely.

Example 1:
Recipe Share with MySQL, PHP, and PDO

Outcome

- List of recipes at `/recipes.php`.
- Recipe detail at `/recipe.php?id=...` showing joined ingredients.
- Author stats at `/stats_authors.php` using GROUP BY.
- All lookups use prepared statements.

Step 0. Prerequisites and VS Code setup

Software

- XAMPP (Apache, PHP, MySQL).
- VS Code.

VS Code extensions

- PHP Intelephense (better PHP IntelliSense).
- PHP Debug (optional Xdebug integration if you enable it in XAMPP later).
- EditorConfig or Prettier is optional.

Folder to edit

- XAMPP's web root is typically:
  - Windows: C:\xampp\htdocs
  - macOS: /Applications/XAMPP/htdocs
- **Create** a project folder inside **htdocs**, for example: **C:\xampp\htdocs\recipe-share**

Open the folder in VS Code

- File → Open Folder
- Select **recipe-share**.
- You will edit files here so Apache can serve them directly.

Step 1. Start XAMPP and verify PHP

1. Open **XAMPP Control Panel**.
2. Start **Apache** and **MySQL**.
3. Visit **http://localhost/** in a browser. You should see the XAMPP start page.
4. Create a quick test file info.php in recipe-share:

```
<?php phpinfo();
```

Note: Unless you have non-PHP content, it is recommended to omit the closing question-angle-bracket, ?>

5. Visit **http://localhost/recipe-share/info.php**.
    a. If PHP renders, you are good.

Step 2. Create the database in phpMyAdmin

What is an index?

Before going further, here's a note about **indices** (**indexes)**:

An **index** is a data structure that allows the database to locate rows faster without scanning the entire table. You can think of it like the index in the back of a textbook: instead of reading every page to find "Tomato Sauce," you go directly to the entry that tells you which page it's on.

In MySQL, indexes are usually implemented as **balanced trees** (B-trees). The tree keeps the column values sorted internally, so lookups, inserts, and updates can be performed efficiently.

Follow the steps below

- Go to **http://localhost/phpmyadmin/**
- Click **Databases**
    - Create database: **recipes_db**
    - Collation: **utf8mb4_general_ci**
    - Create.
- With recipes_db selected, open the **SQL** tab and run the following DDL:

```
-- Tables Creation fun
CREATE TABLE recipes (
  recipe_id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(200) NOT NULL,
  author VARCHAR(120) NOT NULL,
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE ingredients (
  ingredient_id INT AUTO_INCREMENT PRIMARY KEY,
  recipe_id INT NOT NULL,
  name VARCHAR(200) NOT NULL,
  amount VARCHAR(80) NOT NULL,
  FOREIGN KEY (recipe_id) REFERENCES recipes(recipe_id)
    ON DELETE CASCADE
```

```
    ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- Helpful index for joins
CREATE INDEX idx_ingredients_recipe_id ON ingredients(recipe_id);

-- Seed the data
INSERT INTO recipes (title, author) VALUES
  ('Classic Tomato Pasta', 'Alice Brown'),
  ('Lemon Herb Chicken', 'John Doe'),
  ('Veggie Chili', 'Alice Brown');

INSERT INTO ingredients (recipe_id, name, amount) VALUES
  (1, 'Spaghetti', '200 g'),
  (1, 'Tomato Sauce', '1 cup'),
  (1, 'Garlic', '2 cloves'),
  (2, 'Chicken Breast', '2 pieces'),
  (2, 'Lemon', '1'),
  (2, 'Rosemary', '1 tsp'),
  (3, 'Kidney Beans', '1 can'),
  (3, 'Tomato', '2'),
  (3, 'Chili Powder', '1 tbsp');
```

- Click **Go**.

You now have the schema and initial data.

Step 3. Store DB connection settings

Create **protected/config.inc.php** inside **recipe-share** and place the following code:

```
<?php
define('DBHOST', 'localhost');
define('DBNAME', 'recipes_db');
define('DBUSER', 'root');
define('DBPASS', ''); // default XAMPP MySQL root has empty password on
Windows
define('DBCONNSTRING', 'mysql:host=' . DBHOST . ';dbname=' . DBNAME .
';charset=utf8mb4');
```

Security note for production:

- do not keep secrets in web-served paths.
- For class demos on localhost this is acceptable.

Step 4. Create a small PDO helper

Create **lib/Database.php**:

```
<?php
class Database {
  public static function connect() {
    require_once __DIR__ . '/../protected/config.inc.php';
    $pdo = new PDO(DBCONNSTRING, DBUSER, DBPASS, [
      PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
      PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);
    return $pdo;
  }
}
```

This keeps your scripts clean and consistent.

Step 5. Recipe list page

Create **recipes.php**:

```
<?php
require_once __DIR__ . '/lib/Database.php';
$pdo = Database::connect();

$sql = "SELECT recipe_id, title, author, created_at
        FROM recipes
        ORDER BY created_at DESC";
$stmt = $pdo->query($sql);
$recipes = $stmt->fetchAll();
?>
<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>Recipes</title></head>
<body>
  <h1>Recipes</h1>
  <ul>
    <?php foreach ($recipes as $r): ?>
      <li>
        <a href="recipe.php?id=<?= htmlspecialchars($r['recipe_id']) ?>">
          <?= htmlspecialchars($r['title']) ?>
        </a>
        by <?= htmlspecialchars($r['author']) ?>
      </li>
    <?php endforeach; ?>
  </ul>
</body>
```

```
</html>
```

Note since we're mixing HTML in with PHP, we do use the closing ?>.

Visit **http://localhost/recipe-share/recipes.php** and confirm you see the list.

Step 6. Recipe detail page with a join

Note the name of this PHP file is **singular**, not plural like the previous *recipes.php*.

Create **recipe.php**:

```
<?php
require_once __DIR__ . '/lib/Database.php';
$pdo = Database::connect();

// Validate id
$id = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);
if (!$id) {
  http_response_code(400);
  exit('Invalid recipe id');
}

// Get recipe
$sqlRecipe = "SELECT recipe_id, title, author, created_at
              FROM recipes
              WHERE recipe_id = :id";
$s1 = $pdo->prepare($sqlRecipe);
$s1->bindValue(':id', $id, PDO::PARAM_INT);
$s1->execute();
$recipe = $s1->fetch();

if (!$recipe) {
  http_response_code(404);
  exit('Recipe not found');
}

// Get ingredients via join
$sqlIng = "SELECT i.name, i.amount
           FROM ingredients i
           WHERE i.recipe_id = :id
           ORDER BY i.ingredient_id ASC";


$s2 = $pdo->prepare($sqlIng);
$s2->bindValue(':id', $id, PDO::PARAM_INT);
$s2->execute();
$ingredients = $s2->fetchAll();
?>
<!doctype html>
<html lang="en">
```

```
<head><meta charset="utf-8"><title><?= htmlspecialchars($recipe['title'])
?></title></head>
<body>
  <a href="recipes.php">Back to recipes</a>
  <h1><?= htmlspecialchars($recipe['title']) ?></h1>
  <p>By <?= htmlspecialchars($recipe['author']) ?>  Posted <?=
htmlspecialchars($recipe['created_at']) ?></p>

  <h2>Ingredients</h2>
  <ul>
    <?php foreach ($ingredients as $ing): ?>
      <li><?= htmlspecialchars($ing['amount']) ?>  <?=
htmlspecialchars($ing['name']) ?></li>
    <?php endforeach; ?>
  </ul>
</body>
</html>
```

This page validates input, uses prepared statements, and prints escaped HTML.

Step 7. Author stats with GROUP BY

Create **stats_authors.php**:

```
<?php
require_once __DIR__ . '/lib/Database.php';
$pdo = Database::connect();

$sql = "SELECT author, COUNT(*) AS recipe_count
        FROM recipes
        GROUP BY author
        ORDER BY recipe_count DESC, author ASC";
$stmt = $pdo->query($sql);
$rows = $stmt->fetchAll();
?>
<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>Author Stats</title></head>
<body>
  <a href="recipes.php">Back to recipes</a>
  <h1>Recipes by Author</h1>
  <table border="1" cellpadding="6">
    <tr><th>Author</th><th>Recipe Count</th></tr>
    <?php foreach ($rows as $row): ?>
      <tr>
        <td><?= htmlspecialchars($row['author']) ?></td>
        <td><?= htmlspecialchars($row['recipe_count']) ?></td>
      </tr>
```

```
    <?php endforeach; ?>
  </table>
</body>
</html>
```

This page validates input, uses prepared statements, and prints escaped HTML.

Step 8. Quick index to tie it together

Create **index.php**:

```
<?php header('Location: recipes.php'); exit;
```

Now visiting **http://localhost/recipe-share/** jumps to the list.

Step 9. Testing

1. In a browser, open **http://localhost/recipe-share/recipes.php**.
    - Verify seeded recipes appear newest first.
2. Click a recipe.
    - Verify ingredients display and the URL contains a numeric id.
3. Open **/stats_authors.php**.
    - Verify counts for each author.
4. Try **recipe.php?id=abc** and **recipe.php?id=9999** to confirm 400 and 404 responses.
5. In phpMyAdmin, add another recipe and ingredients.
    - Note that you must either use SQL or add manually
    - If adding manually, you Insert for both the recipe and the ingredients
    - The ingredients must refer to the recipe_id of the recipe you just added
    - E.g., German Tomato Soup with ingredients
        - Tomato (2)
        - Salt (1 pinch)
        - Soup-things (3)
        - Or maybe just a can of German tomato soup (1) if you want to be funny
6. Refresh pages and confirm changes.

Common pitfalls and fixes

- **Blank page or PHP showing as text**
    - Apache not running or file not in htdocs.
    - Start Apache and move files to htdocs\recipe-share.
- **Cannot connect to DB**
    - Check config.inc.php values.
    - In XAMPP, default root password is empty. Ensure MySQL is running.
- **Headers already sent**
    - Avoid stray whitespace before <?php or after ?>.

  - Use UTF-8 without BOM in VS Code.
- **XSS risks**
  - Always wrap dynamic output with htmlspecialchars.

Example 2:
Understanding Transactions for Data Integrity

Outcome

- By the end of this example you will have a PHP script that simulates a small checkout process.
- The script performs several related database updates (creating an order, adding order items, and adjusting inventory) inside a single **transaction**.
- If any step fails, the database automatically rolls back so that no partial or inconsistent data remains.

Step 0. Environment continuity

1. In XAMPP's **htdocs** directory, **create** a new folder such as **store-demo**.

The setup and database connection code from Example 1 will still work. You only need to create a new database and tables for this project.

2. Copy over the **protected** and **lib** folders from the recipe-share project.

Step 1. Create the database and tables in phpMyAdmin

1. Open **phpMyAdmin** at http://localhost/phpmyadmin/.
2. Create a new database named **store_db**.
3. In the SQL tab, paste and run the following script:

```
CREATE TABLE products (
  product_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  price DECIMAL(8,2) NOT NULL,
  stock INT NOT NULL
) ENGINE=InnoDB;

CREATE TABLE orders (
  order_id INT AUTO_INCREMENT PRIMARY KEY,
  placed_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  total DECIMAL(8,2) NOT NULL
) ENGINE=InnoDB;

CREATE TABLE order_items (
  item_id INT AUTO_INCREMENT PRIMARY KEY,
  order_id INT NOT NULL,
```

```
  product_id INT NOT NULL,
  qty INT NOT NULL,
  line_total DECIMAL(8,2) NOT NULL,
  FOREIGN KEY (order_id) REFERENCES orders(order_id)
    ON DELETE CASCADE,
  FOREIGN KEY (product_id) REFERENCES products(product_id)
    ON DELETE RESTRICT
) ENGINE=InnoDB;

INSERT INTO products (name, price, stock) VALUES
  ('Notebook', 3.50, 20),
  ('Pen', 1.00, 50),
  ('Backpack', 35.00, 10);
```

Transactions

The **InnoDB** engine supports ***transactions***. Other engines, such as MyISAM, do not.

A **transaction** is a sequence of one or more SQL operations that are treated by the database as a single, indivisible unit of work.

In simpler terms, it is a *package* of database actions that must either **all succeed together** or **all fail together**. If any part of the transaction encounters an error, the database will undo everything that was done within that transaction so that the data remains consistent and correct.

Transactions are often described by the **ACID** principles, which define their reliability guarantees.

Property	**Meaning**
Atomicity	The transaction's operations act as a single unit. Either all of them succeed or none do.
Consistency	The transaction brings the database from one valid state to another, preserving rules such as foreign key relationships or unique constraints.
Isolation	Transactions can occur concurrently without interfering with each other. One transaction should not see another's partial results.
Durability	Once a transaction is committed, the changes are permanent, even if the system crashes immediately afterward.

Step 2. Update your connection configuration

If you kept the previous config file, make a copy named **protected/config.inc.php** and change the database name:

```
<?php
define('DBHOST', 'localhost');
define('DBNAME', 'store_db');
define('DBUSER', 'root');
define('DBPASS', '');
define('DBCONNSTRING', 'mysql:host=' . DBHOST . ';dbname=' . DBNAME .
';charset=utf8mb4');
```

Step 3. Create a test checkout script

Create **checkout.php** in your project root:

```
<?php
require_once __DIR__ . '/lib/Database.php';
$pdo = Database::connect();

// Example cart: product_id => quantity
$cart = [
  1 => 2,   // 2 notebooks
  2 => 5,   // 5 pens
  3 => 1    // 1 backpack
];

try {
  // Begin the transaction
  $pdo->beginTransaction();

  // Calculate total
  $total = 0;
  foreach ($cart as $id => $qty) {
    $stmt = $pdo->prepare('SELECT price, stock FROM products WHERE
product_id = :id');
    $stmt->bindValue(':id', $id, PDO::PARAM_INT);
    $stmt->execute();
    $product = $stmt->fetch();
    if (!$product) {
      throw new Exception("Product $id not found");
    }
    if ($product['stock'] < $qty) {
      throw new Exception("Insufficient stock for product $id");
    }
    $lineTotal = $product['price'] * $qty;
    $total += $lineTotal;
```

```
  }

  // Insert the order
  $stmtOrder = $pdo->prepare('INSERT INTO orders (total) VALUES (:total)');
  $stmtOrder->bindValue(':total', $total);
  $stmtOrder->execute();
  $orderId = $pdo->lastInsertId();

  // Insert order items and update inventory
  foreach ($cart as $id => $qty) {
    $stmtProduct = $pdo->prepare('SELECT price FROM products WHERE
product_id = :id');
    $stmtProduct->bindValue(':id', $id);
    $stmtProduct->execute();
    $price = $stmtProduct->fetchColumn();
    $lineTotal = $price * $qty;

    $stmtItem = $pdo->prepare('INSERT INTO order_items (order_id,
product_id, qty, line_total)
                               VALUES (:order_id, :pid, :qty, :line)');
    $stmtItem->execute([
      ':order_id' => $orderId,
      ':pid' => $id,
      ':qty' => $qty,
      ':line' => $lineTotal
    ]);

    $stmtUpdate = $pdo->prepare('UPDATE products SET stock = stock - :qty
WHERE product_id = :id');
    $stmtUpdate->execute([':qty' => $qty, ':id' => $id]);
  }

  // Commit all changes
  $pdo->commit();
  echo "<p>Order #$orderId completed successfully.  Total: $$total</p>";

} catch (Exception $e) {
  // Roll back if any error occurs
  $pdo->rollBack();
  echo '<p style="color:red;">Transaction failed: ' . htmlspecialchars($e-
>getMessage()) . '</p>';
}
```

Step 4. Test the transaction

1. Visit **http://localhost/store-demo/checkout.php**.
    - You should see a success message and new entries in orders and **order_items**.
2. In phpMyAdmin, check that product stock values decreased appropriately.
3. Now **force an error**: change the $cart array to request more items than are in stock. The script should catch the exception, roll back, and leave all data unchanged.

Step 5. Verify rollback behavior

In phpMyAdmin, run:

```
SELECT * FROM products;
SELECT * FROM orders;
SELECT * FROM order_items;
```

If you caused a failure, no new orders or order_items should appear, and product stock should remain the same.

This demonstrates **atomicity** - either every statement succeeds or the entire group is undone.

Discussion

Why transactions matter

- Web applications often involve several dependent database actions.
- Without transactions, an error halfway through could leave partial data, such as an order without items or deducted stock without a recorded sale.
- Transactions guarantee all-or-nothing consistency.

How this applies in real systems

- E-commerce checkout
- Bank account transfers
- Reservation systems
- Batch updates where many tables must remain synchronized

Troubleshooting

"Cannot start transaction"

Make sure your tables use ENGINE=InnoDB. Transactions are not supported on MyISAM.

"Lock wait timeout exceeded"

Two scripts might be updating the same rows simultaneously. Restart MySQL and test again.

Blank page or fatal error

Enable error reporting for debugging:

Summary

- A transaction groups several SQL statements into one logical unit.
- Use beginTransaction(), commit(), and rollBack() in PDO.
- If an error occurs inside the block, always roll back to maintain integrity.
- InnoDB supports transactions and foreign keys.
- The example demonstrates how MySQL enforces the ACID property of atomicity for reliability.

Example 3:
Comments and Replies in MongoDB (Node + Atlas)

Outcome

- **GET /api/recipes/:id/comments** returns all comments for a recipe as nested JSON.
- **POST /api/recipes/:id/comments** adds a top-level comment.
- **POST /api/comments/:commentId/replies** adds a reply under an existing comment.
- An index on **recipeId** speeds up lookups.
- All endpoints are exercised and verified in **Postman**.

Step 0. Prerequisites and VS Code setup

Software

- Node.js 18 or newer.
- VS Code.
- Postman desktop app.

VS Code extensions

- ESLint (optional).

Create the project

1. Create a **new** folder in VS Code, for example **recipe-comments-api**.
2. Open VS Code Terminal and run:

```
npm init -y
npm install express mongoose dotenv cors
npm install --save-dev nodemon
```

3. Update **package.json** scripts:

```
{
  "name": "recipe-comments-api",
  "version": "1.0.0",
  "main": "src/server.js",
  "type": "module",
  "scripts": {
    "dev": "nodemon src/server.js",
    "start": "node src/server.js"
```

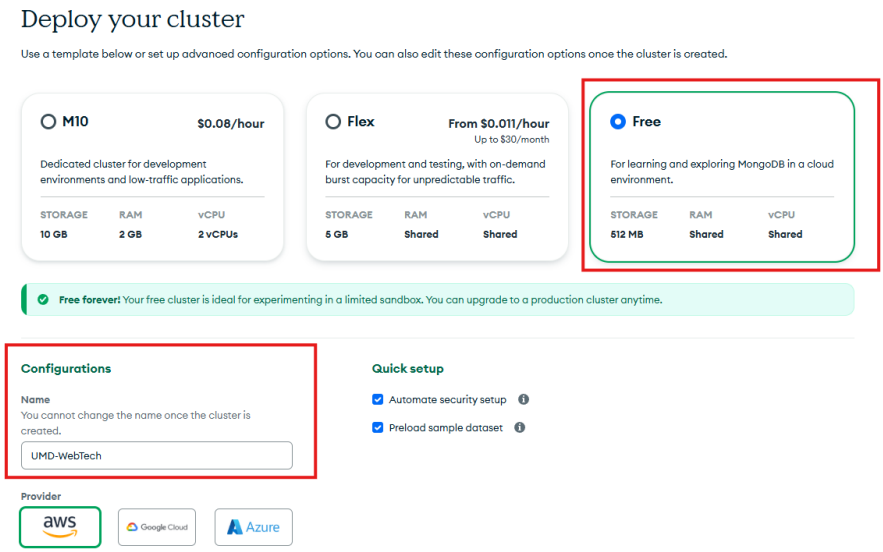
```
    }
  }
```

Project structure

```
recipe-comments-api/
  .env                    # to be created
  package.json
  src/
    server.js
    db.js
    models/
      Comment.js
    routes/
      comments.js
```

Step 1. Create a MongoDB Atlas cluster and credentials

1. Sign in to MongoDB Atlas.
2. Create a Project, then create a free Cluster (M0).



   - This may appear as **"Create Deployment"** or **"Build a Database."**
3. Click **Create Deployment**
4. In **Network Access**, add your current IP (or 0.0.0.0/0 for class demos)
   - Add your current IP, or 0.0.0.0/0 for class demos to allow all.
5. In **Database Access**, create a user with Read and write to any database
   - Create a username and password.
   - Choose **Read and write to any database.**
   - Save those credentials (you will need them in your .env file).
6. Once the cluster is ready, click "Connect."

- Choose **Drivers**.
- Choose **Node.js** as the driver and select the latest version.
- Copy the provided connection string.

7. **Create your .env file** in VS Code and paste the connection string (Example:)

```
MONGO_URL="mongodb+srv://your_user:your_pass@cluster0.xxxxxx.mongodb.net
/?retryWrites=true&w=majority"
MONGO_DB="recipes_app"
PORT=3000
```

- Save the file.
- Restart your Node server with **`npm run dev`**.

Step 2. Database connector

Create **src/db.js**:

```
import mongoose from 'mongoose';

export async function connectToMongo(url, dbName) {
  if (!url) throw new Error('Missing MONGO_URL');
  if (!dbName) throw new Error('Missing MONGO_DB');

  await mongoose.connect(url, { dbName });
  mongoose.connection.on('connected', () => {
    console.log(`Mongo connected to ${dbName}`);
  });
  mongoose.connection.on('error', (err) => {
    console.error('Mongo connection error:', err.message);
  });
}
```

Step 3. Comment model with nested replies

Create **src/models/Comment.js**:

```
import mongoose from 'mongoose';

const replySchema = new mongoose.Schema(
  {
    user: { type: String, required: true, trim: true, maxlength: 100 },
    text: { type: String, required: true, trim: true, maxlength: 2000 },
    createdAt: { type: Date, default: Date.now }
  },
  { _id: true }
);

const commentSchema = new mongoose.Schema(
  {
    recipeId: { type: Number, required: true, index: true },
```

```
    user: { type: String, required: true, trim: true, maxlength: 100 },
    comment: { type: String, required: true, trim: true, maxlength: 4000 },
    rating: { type: Number, min: 1, max: 5 },
    createdAt: { type: Date, default: Date.now },
    replies: { type: [replySchema], default: [] }
  },
  { versionKey: false }
);

commentSchema.index({ recipeId: 1, createdAt: -1 });

export const Comment = mongoose.model('Comment', commentSchema);
```

Step 4. Express routes

Create **src/routes/comments.js**:

```
import { Router } from 'express';
import { Comment } from '../models/Comment.js';

const router = Router();

// GET /api/recipes/:id/comments
router.get('/recipes/:id/comments', async (req, res) => {
  const recipeId = Number(req.params.id);
  if (!Number.isInteger(recipeId) || recipeId <= 0) {
    return res.status(400).json({ error: 'Invalid recipe id' });
  }
  try {
    const docs = await Comment.find({ recipeId })
      .sort({ createdAt: -1 })
      .lean()
      .exec();
    return res.json(docs);
  } catch {
    return res.status(500).json({ error: 'Failed to fetch comments' });
  }
});

// POST /api/recipes/:id/comments
// Body: { "user": "abrown", "comment": "Loved this!", "rating": 5 }
router.post('/recipes/:id/comments', async (req, res) => {
  const recipeId = Number(req.params.id);
  const { user, comment, rating } = req.body || {};

  if (!Number.isInteger(recipeId) || recipeId <= 0) {
    return res.status(400).json({ error: 'Invalid recipe id' });
  }
  if (!user || !comment) {
```

```
    return res.status(400).json({ error: 'Missing user or comment' });
  }

  try {
    const doc = await Comment.create({ recipeId, user, comment, rating });
    return res.status(201).json(doc);
  } catch (err) {
    return res.status(422).json({ error: 'Validation failed', details:
err.message });
  }
});

// POST /api/comments/:commentId/replies
// Body: { "user": "jdoe", "text": "Same here!" }
router.post('/comments/:commentId/replies', async (req, res) => {
  const { commentId } = req.params;
  const { user, text } = req.body || {};

  if (!user || !text) {
    return res.status(400).json({ error: 'Missing user or text' });
  }

  try {
    const result = await Comment.findByIdAndUpdate(
      commentId,
      { $push: { replies: { user, text } } },
      { new: true, runValidators: true }
    ).lean();

    if (!result) {
      return res.status(404).json({ error: 'Comment not found' });
    }
    return res.status(201).json(result);
  } catch (err) {
    return res.status(422).json({ error: 'Validation failed', details:
err.message });
  }
});

export default router;
```

Step 5. Server entry point

Create **src/server.js**:

```
import 'dotenv/config';
import express from 'express';
import cors from 'cors';
```

```
import { connectToMongo } from './db.js';
import commentsRouter from './routes/comments.js';

const app = express();
app.use(cors());
app.use(express.json());

app.get('/health', (req, res) => res.json({ ok: true }));

app.use('/api', commentsRouter);

const port = process.env.PORT || 3000;
const url = process.env.MONGO_URL;
const dbName = process.env.MONGO_DB;

connectToMongo(url, dbName)
  .then(() => {
    app.listen(port, () => {
      console.log(`API listening on http://localhost:${port}`);
    });
  })
  .catch((err) => {
    console.error('Failed to connect to Mongo:', err.message);
    process.exit(1);
  });
```

Run the server:

```
npm run dev
```

You should see a message that Mongo connected and the API is listening.

Step 6. Test with Postman

A. Create a Postman collection and environment

1. Open Postman.
2. Click **Environments +** to create a new environment.
    - Name: **Recipe API Local**
    - Add variable **baseUrl** with initial and current value **http://localhost:3000**
    - Save the environment. Select it in the top right environment picker.
3. Click **Collections + New Collection** name it *Recipe Comments API*.
4. Then, Save.

B. Health check request

1. In your collection, click **Add a request**.

2. Name it Health. Method GET.
3. URL: {{baseUrl}}/health
4. Click **Send**. You should get { "ok": true }. Save the request to the collection.

C. Create a top-level comment (in Postman)

1. In your Postman collection, click **Add a new request**.
2. Name it **Create Comment**.
3. Set **Method: POST**.
4. SetURL:

```
{{baseUrl}}/api/recipes/7/comments
```

5. Go to **Body** select **raw** choose **JSON**.
6. Paste:

```
{ "user": "abrown", "comment": "Loved this!", "rating": 5 }
```

7. Click **Send**.
    - You should receive **201 Created** and a JSON response containing fields such as _id, recipeId, and createdAt.Highlight the _id value in the response and copy it.
8. In the response, highlight the value of **`_id`**.
9. Copy it.
    - This is the comment's unique **ObjectId**, which you will use when posting replies.
10. In Postman, open your **environment** (click on the Recipe API Local in the Environments tab to the left).
11. Add a new variable:
    - **Name:** commentId
    - **Initial Value / Current Value:** paste the _id you just copied.
12. Save the environment.

D. List comments for a recipe

1. Add a new request named List Comments.
2. Method **GET**.
3. URL: **{{baseUrl}}/api/recipes/7/comments**
4. Click **Send**.
    - You should see an array with your newly created comment first.
5. Save the request.

E. Add a reply to a comment

1. Add a new request named Add Reply. Method POST.
2. URL: {{baseUrl}}/api/comments/{{commentId}}/replies
3. **Body** raw JSON:

```
{ "user": "jdoe", "text": "Same here!" }
```

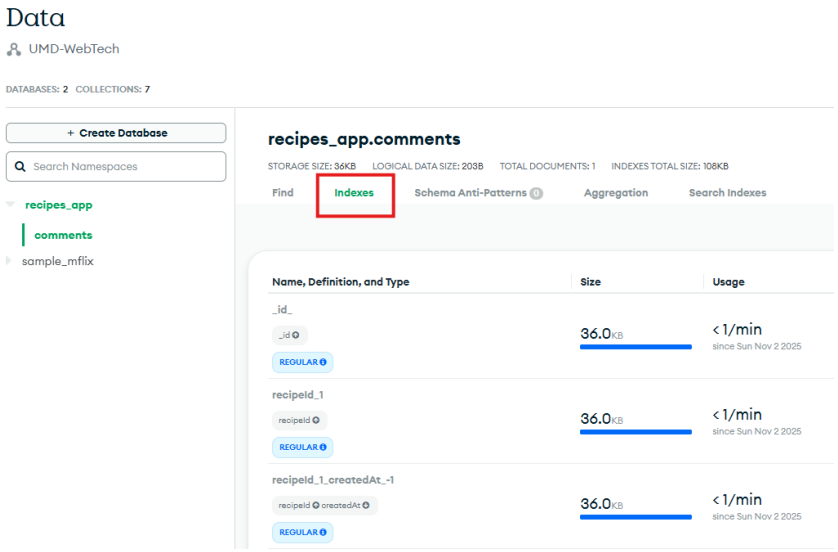
4. Click **Send**. Expect 201 and the updated comment document with your reply included.
5. Save the request.

F. Quick sanity checks

- Change the recipe id to a non-integer such as abc and confirm you get 400 with an error message.
- Post a comment with a missing user or comment and confirm you get 400.
- Post a reply to a fake commentId and confirm you get 404.

Step 7. Verify indexes in Atlas

1. In Atlas, open your Cluster **Collections**.
2. Choose database **recipes_app** and collection comments.
3. Select the **comments** collection
4. You can see the comment objects under **Find**.
5. Now, open **Indexes**.



6. You should see something like:
    - { recipeId: 1 }

- { recipeId: 1, createdAt: -1 }
- If you do not see them yet, they will appear after the first writes and index builds.

Index Name	Definition	Meaning
`_id_`	`{ _id: 1 }`	The default primary index automatically created on every collection.
`recipeId_1`	`{ recipeId: 1 }`	A **single-field index** on `recipeId`, created by Mongoose because you marked that field as indexed in your schema. The 1 means ascending order (not the value 1).
`recipeId_1_createdAt_-1`	`{ recipeId: 1, createdAt: -1 }`	A **compound index** that sorts by `recipeId` ascending and `createdAt` descending. This speeds up queries like "find all comments for recipe 7, with the newest first."

Security and validation checklist

- Validate and sanitize inputs in routes.
- Enforce length limits in schemas.
- Keep .env out of version control.
- If you later render this data in HTML, escape output to prevent cross site scripting.
- Consider rate limits if exposing publicly.

Common pitfalls and fixes

- **Authentication failed** Confirm Atlas database user credentials and IP allowlist.
  - Update .env if the password changed.
- **Timeouts** Some networks block required ports. Try another network or a hotspot for a quick check.
- **CastError:**
  - **Cast to ObjectId failed** Ensure commentId is a valid ObjectId.
  - Use the value captured in Postman from the create response.
- **CORS in browser**
  - Keep app.use(cors()).
  - For production, restrict origins.