Todo App

Your First Full-Stack Web App

A Beginner's Step-by-Step Tutorial

React 19 + Node.js + Express + MySQL

JavaScript syntax explained from the ground up

What you will build and learn

A complete todo app: add, edit, complete, delete, and filter tasks.
Real database storage - tasks survive page refreshes and restarts.
Learn React on the frontend (components, state, fetch).
Learn Node.js + Express on the backend (REST API, middleware).
Learn MySQL queries (SELECT, INSERT, UPDATE, DELETE).
Learn JavaScript syntax: arrow functions, async/await, destructuring.
Learn how the three layers talk to each other.
Deploy your app online for free.

egotechworld.com

Free coding tutorials, project source code, and AI tools

Beginner-Friendly - Step by Step - 2026 Edition

Table of Contents

1. What You Will Build and Learn

2. How the Three Pieces Fit Together

3. JavaScript Crash Course - The Syntax You Need

4. Tools You Need - Node, MySQL, VS Code

5. Setting Up the MySQL Database

6. Building the Backend - Node.js + Express Setup

7. The First Endpoint - GET /api/todos

8. POST, PUT, DELETE - Full CRUD API

9. Testing the API with Thunder Client

10. Setting Up the React Frontend

11. Fetching Todos from the API

12. Adding New Todos with a Form

13. Toggling, Editing, and Deleting

14. Filters and Polish

15. Deployment and Where to Go Next

--- Quick Reference Card ---

Chapter 1 - What You Will Build and Learn

Welcome! This tutorial will take you from "I have heard of React and Node.js but never used them" to "I built a real full-stack web app from scratch". The project: a **todo app**. Boring? Maybe. But every full-stack pattern you need is in there - frontend forms, REST APIs, database storage, error handling, deployment. Master a todo app and you can build anything.

What is full-stack?

A web app has two halves:

- **Frontend** - what runs in the user's browser. Buttons, forms, lists, animations. Built with React in our case.
- **Backend** - what runs on a server somewhere. The database, the business logic, anything that has to be private or persistent. Built with Node.js and MySQL in our case.

Full-stack means writing both halves. A full-stack developer is someone who is comfortable on both sides. That is the job we are preparing you for in this tutorial.

Why React + Node + MySQL?

- **React** is the most popular frontend library in 2026. Massive job market, huge ecosystem, used by Facebook, Netflix, Airbnb.
- **Node.js + Express** is the most popular JavaScript backend. You write the same language on both sides - no context switching.
- **MySQL** is the most widely-deployed open-source database in the world. Free, fast, and the SQL knowledge you gain transfers to PostgreSQL, SQLite, MariaDB - all the relational databases.
- **The job market loves this stack**. Search any job board for "React Node MySQL" - thousands of postings.

Who this tutorial is for

- You know basic HTML, CSS, and a tiny bit of JavaScript.
- You have heard of frontend / backend but never built one.
- You want to understand WHY each line of code is there - not just copy.
- You learn best from a real project, not toy snippets.

> **TIP**
>
> If you are completely new to JavaScript, do not worry. Chapter 3 is a JavaScript crash course covering all the syntax we will use - arrow functions, async/await, destructuring, the spread operator. Even if you only know HTML and CSS, you will be ready after chapter 3.

What the finished app looks like

Before we touch any code, here is a preview of every screen you will build. Keep these pictures in mind as you work through each chapter.

1. Empty state - the starting point

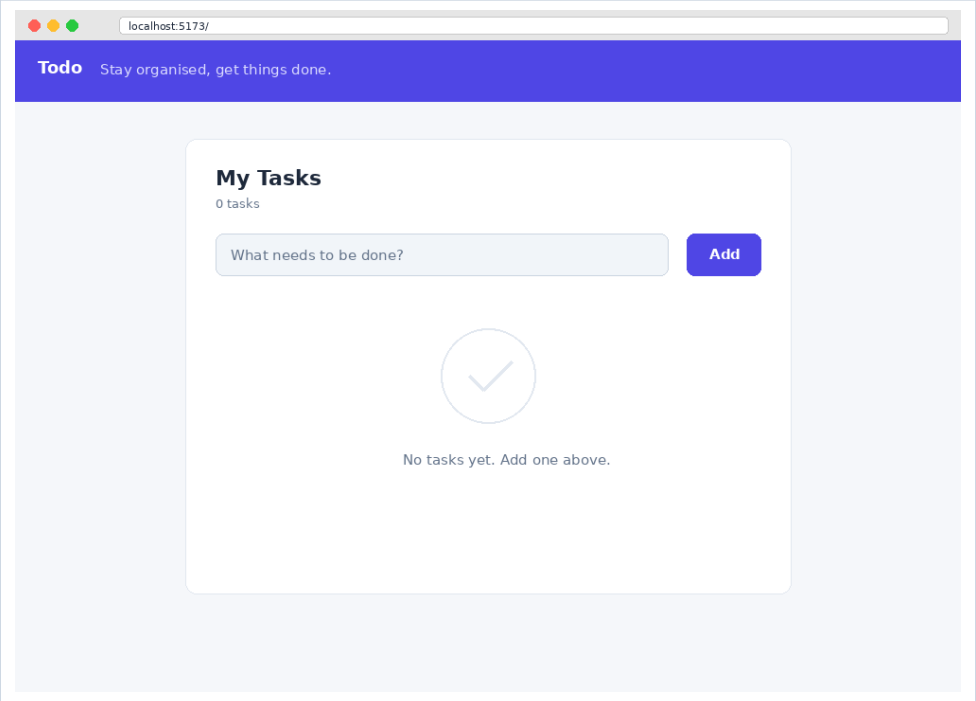

Figure 1 - The clean welcome screen when the user has no tasks yet.

2. Active list with mixed states

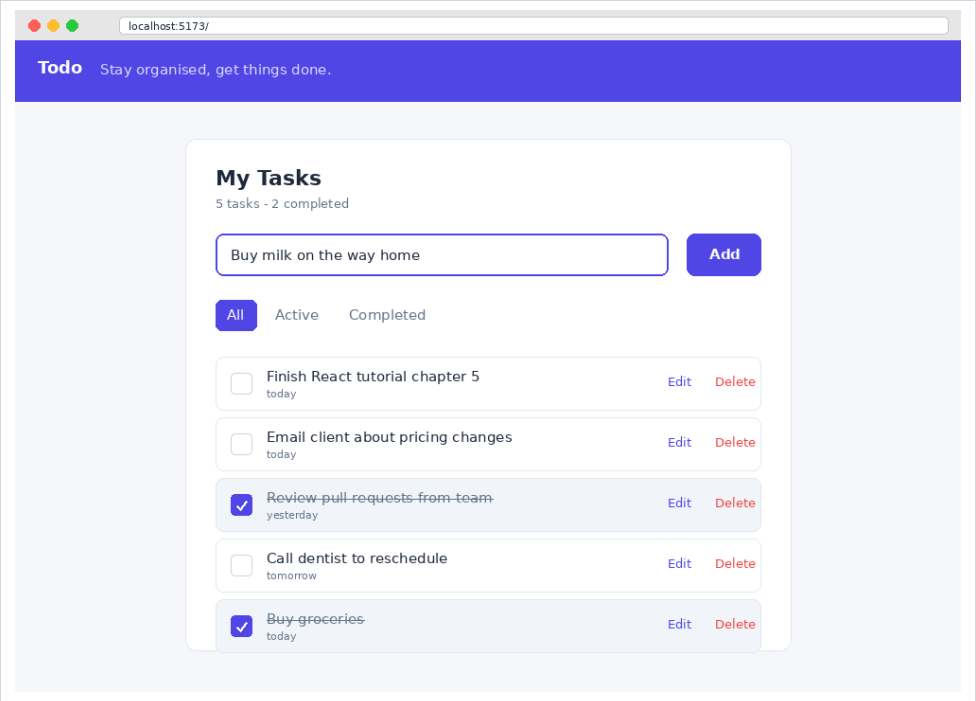

Figure 2 - Tasks added and partially completed. Notice the checkbox state, the strikethrough on done items, and the filter tabs.

3. Inline edit mode

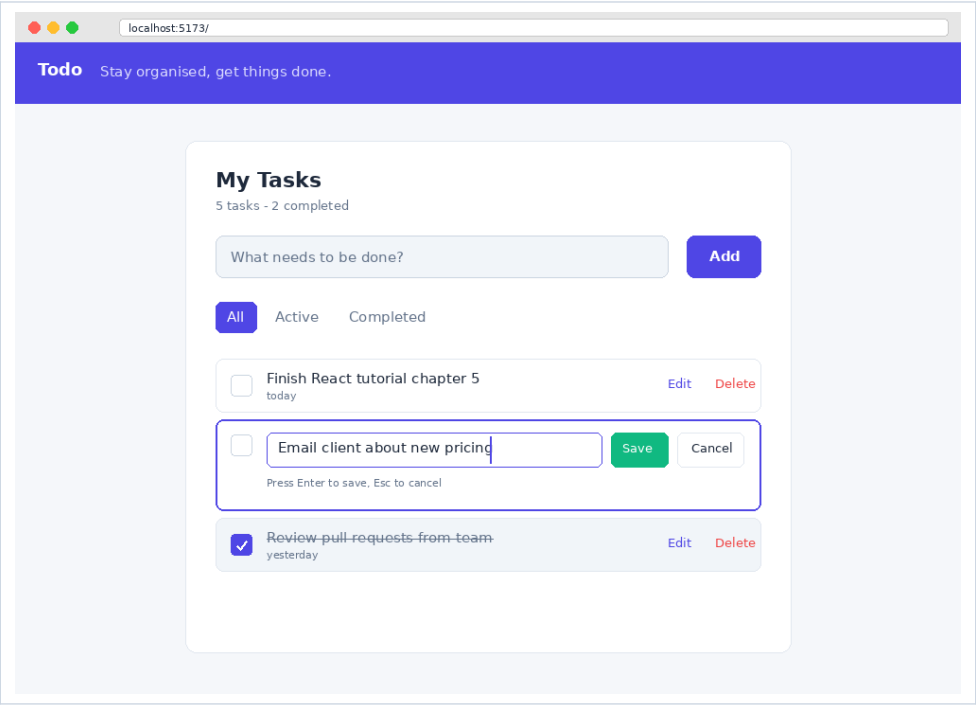

Figure 3 - Click Edit on any task to modify its text in place. Save with the green button or by pressing Enter; cancel with Esc.

4. Filtered view - completed tasks only

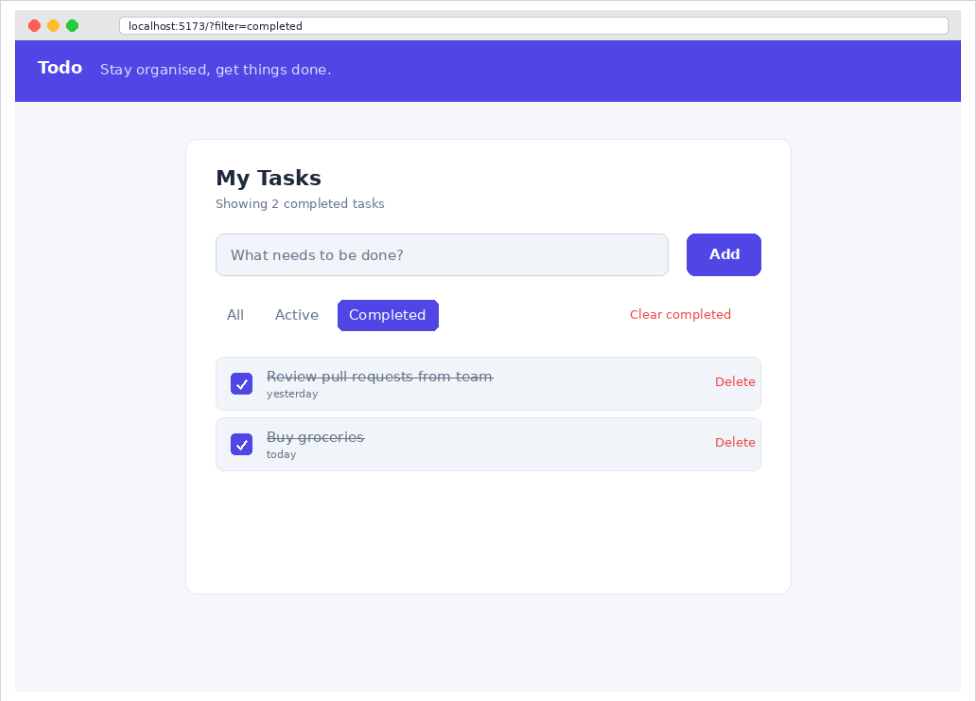

Figure 4 - The Completed tab shows only finished tasks, with a Clear completed link to wipe them all at once.

> **NOTE**
>
> All source code from this tutorial will be available at **egotechworld.com** in the projects section. Stuck? Compare your code to the reference there.

Chapter 2 - How the Three Pieces Fit Together

Before we write code, let us understand what we are building. A full-stack app like ours has three separate moving parts that talk to each other over the network. Once you see the big picture, every step will make sense.

The three layers

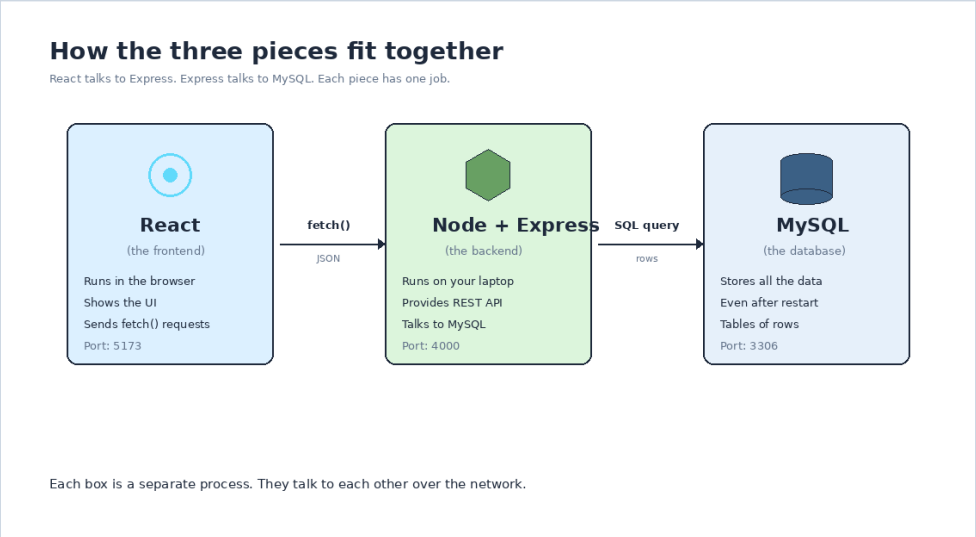

Figure 5 - The three-tier architecture. React is the face. Express is the brain. MySQL is the memory.

What each layer does

Layer	Where it runs	Its job
React (frontend)	User's browser	Show the UI. Capture clicks. Send fetch requests to the API.
Node + Express (backend)	Your laptop or a server	Receive requests. Validate them. Talk to the database. Send JSON back.
MySQL (database)	Your laptop or a server	Store data permanently in tables of rows. Survive restarts.

Why three layers and not one?

- **Security** - the database password lives only on the backend. The browser never sees it. If someone steals your frontend code, they cannot reach the database.
- **Multi-device** - the backend is shared. The same Express server can serve data to a web browser, a mobile app, a desktop app, all at once.
- **Scalability** - if your app gets popular, you can run multiple Express servers behind a load balancer, while keeping one MySQL database. Each layer scales separately.
- **Different languages** - you can rewrite the backend in Python or Go or Java tomorrow, and React does not care. As long as the API stays the same, the frontend keeps working.

What is a REST API?

API stands for Application Programming Interface - a fancy term for "a list of URLs the frontend can call". **REST** is a popular style of designing APIs that uses HTTP verbs (GET, POST, PUT, DELETE) to indicate what you want.

HTTP method	What it means	Example URL
GET	Read data	GET /api/todos
POST	Create something	POST /api/todos
PUT	Update something	PUT /api/todos/5
DELETE	Delete something	DELETE /api/todos/5

Notice that **/api/todos** appears in three of those - same URL, different verb. The verb tells the backend what to do. This is the elegance of REST: a small number of verbs, applied consistently, covers most needs.

The journey of one click

Let us trace what happens when a user types a new task and clicks Add. Following this 10-step path is the single best way to internalise how the three layers cooperate:

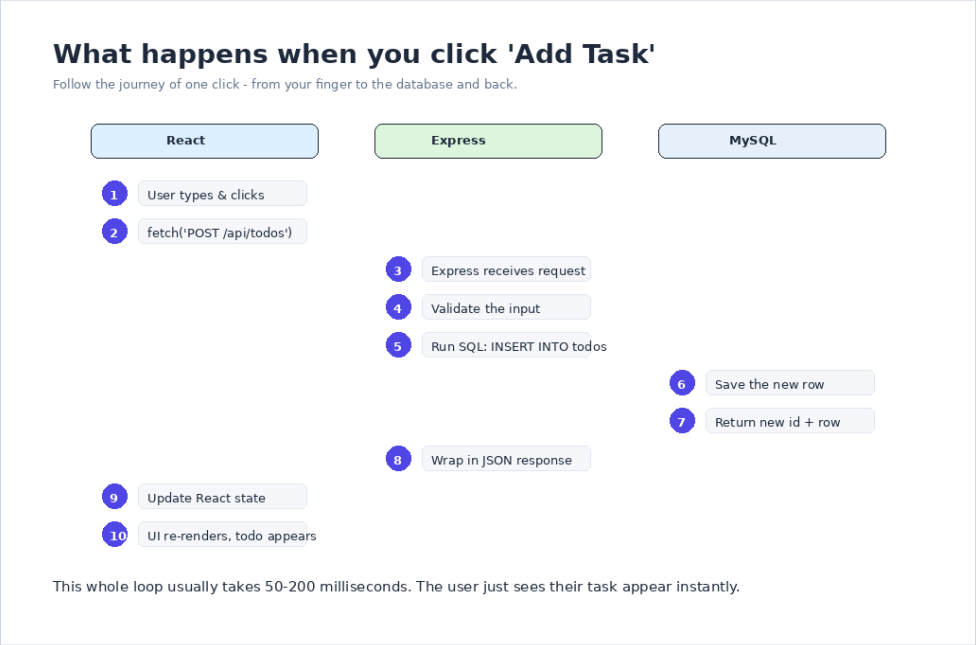

Figure 6 - The 10-step round trip. Reading from top to bottom, each row shows where work is happening at that moment.

> **TIP**
>
> Read the diagram twice. Trace one step at a time with your finger. This pattern - frontend sends a request, backend processes, database persists, response flows back, frontend updates - repeats for every user action in every full-stack app you will ever build.

Chapter 3 - JavaScript Crash Course - The Syntax You Need

We will use modern JavaScript syntax throughout this tutorial. This chapter covers everything you will see - skim it now, refer back whenever a piece of syntax looks confusing later.

Variables - let and const

Modern JavaScript has two ways to declare a variable. We almost never use the old *var* keyword anymore.

```
const name = 'Saman'        // can never be reassigned
let count = 0                // can be changed later
count = count + 1            // OK

name = 'Ayesha'              // ERROR - const can't change
```

- **const** - the value can NEVER be reassigned. Use this 90 percent of the time.
- **let** - the value can change. Use it for counters, loop variables, things that genuinely vary.
- Both are **block-scoped** - they only exist inside the { } where they were declared.

Arrow functions

Arrow functions are a shorter way to write functions. We will use them constantly. Compare the two ways:

```
// Old way
function add(a, b) {
    return a + b
}

// Arrow function (same thing)
const add = (a, b) => {
    return a + b
}

// Even shorter - one-line arrow function (implicit return)
const add = (a, b) => a + b

// Single argument can drop the parens
const double = n => n * 2
```

All four versions of *add* do the same thing. The arrow syntax is shorter and is the standard for callbacks and short functions. The traditional *function* keyword is still fine for top-level named functions.

Template literals - strings with variables

Backticks let you embed variables in strings without messy concatenation:

```
const name = 'Saman'
const age = 25

// Old way - clunky
const greeting1 = 'Hello, ' + name + '! You are ' + age + ' years old.'

// New way - much cleaner
const greeting2 = `Hello, ${name}! You are ${age} years old.`

// Multi-line strings work too
const html = `
    <div>
        <h1>${name}</h1>
    </div>
`
```

> **TIP**
>
> The **${ ... }** placeholder can hold any JavaScript expression - not just variables. *${a + b}* works. *${user.name.toUpperCase()}* works. Anything that produces a value.

Objects and destructuring

An object is a bag of key-value pairs:

```
const user = {
    name: 'Saman',
    age: 25,
    isAdmin: false,
}

// Read a value
console.log(user.name)        // 'Saman'
console.log(user['age'])      // 25 - same thing
```

Destructuring pulls values out of an object into separate variables in one line. We will use this constantly - it makes code much more readable.

```
const user = { name: 'Saman', age: 25, isAdmin: false }

// Without destructuring
const name1 = user.name
const age1 = user.age

// With destructuring (much shorter)
const { name, age } = user

// Rename while destructuring
const { name: userName } = user   // userName === 'Saman'

// Provide a default if the key is missing
const { role = 'user' } = user    // role === 'user'
```

Arrays and the spread operator

```
const fruits = ['apple', 'banana', 'cherry']

// Add to the end
fruits.push('date')              // mutates the original

// Better - create a new array with the spread operator (...)
const moreFruits = [...fruits, 'elderberry']

// Spread also copies arrays
const copy = [...fruits]

// And merges arrays
const all = [...fruits, ...moreFruits]
```

The **...** in front of an array (or object) is called the *spread operator*. It "unpacks" the contents. Spread is the modern way to create new arrays without mutating originals - exactly what React wants for state updates.

Array methods - map, filter, find

These three methods are the bread and butter of modern JS:

```
const numbers = [1, 2, 3, 4, 5]

// map - transform each item, return a new array
const doubled = numbers.map(n => n * 2)
// doubled === [2, 4, 6, 8, 10]

// filter - keep only items matching a test
const even = numbers.filter(n => n % 2 === 0)
// even === [2, 4]

// find - return the FIRST item matching a test
const firstBig = numbers.find(n => n > 3)
// firstBig === 4
```

> **TIP**
>
> All three return a NEW array (or value) and do NOT mutate the original. This is crucial for React, which detects changes by comparing array references.

Promises and async/await

When your code does something that takes time - fetching from a server, reading a file, querying a database - JavaScript does not wait around. It returns a **Promise**: an IOU for a future value. The two ways to handle a promise:

```
// Old way - .then() chains
fetch('/api/todos')
    .then(response => response.json())
    .then(data => {
        console.log(data)
    })
    .catch(error => {
        console.error(error)
    })

// New way - async/await (much cleaner)
async function loadTodos() {
    try {
        const response = await fetch('/api/todos')
        const data = await response.json()
        console.log(data)
    } catch (error) {
        console.error(error)
    }
}
```

- **async** - put it before *function* to mark a function as asynchronous. The function automatically returns a promise.
- **await** - put it before any function call that returns a promise. The runtime pauses on that line until the promise resolves, then continues with the value.
- **try / catch** - catches errors. Equivalent to *.catch()* in promise chains.
- You can ONLY use **await** inside an **async** function.

Modules - import and export

Modern JavaScript splits code across files. Each file is called a **module**. Use *export* to make things available, *import* to bring them in.

```
// math.js
export function add(a, b) {
    return a + b
}

export const PI = 3.14159

// Default export - one per file
export default function multiply(a, b) {
    return a * b
}

// app.js
import multiply, { add, PI } from './math.js'

// 'multiply' is the default - no curly braces
// 'add' and 'PI' are named exports - inside curly braces
```

Quick syntax cheatsheet

Pattern	What it does
const x = 5	declare a constant
let x = 5	declare a mutable variable
(a, b) => a + b	arrow function
`Hello, ${name}!`	template literal
{ a, b } = obj	destructure object
[a, b] = arr	destructure array
[...arr, 'new']	spread - copy array, add item
{...obj, key: 'new'}	spread - copy object, override key
arr.map(fn)	transform each item
arr.filter(fn)	keep items where fn returns true
arr.find(fn)	return first item where fn returns true
await fetch(url)	wait for the request to finish
import x from './y'	import default from y
import { x } from './y'	import a named export

> **TIP**
>
> Bookmark this chapter. When something later in the tutorial feels confusing, the syntax explanation is probably right here.

Chapter 4 - Tools You Need to Install

Three tools, all free, all open source. Install them now before chapter 5.

Tool	Why we need it	Where to get it
Node.js 22+	Runs the backend AND React's build tool.	nodejs.org
MySQL 8	The database. Stores our todos.	Comes with XAMPP / dev.mysql.com
VS Code	Code editor with great JavaScript and SQL support.	code.visualstudio.com

Installing Node.js

Node.js is a JavaScript runtime - it runs JavaScript outside a browser. We need it for two reasons:

• Our backend (Express server) is JavaScript running on Node.

• React's build tool (Vite) also runs on Node, even though the React code itself eventually runs in the browser.

1. Visit **nodejs.org**.

2. Click the big green button labelled **LTS**. Currently this is version 22 or 24.

3. Run the installer with default options. Tick "Add to PATH" if asked.

4. Open a NEW terminal and run **node --version**. It should print v22 or higher.

5. Run **npm --version**. It should print 10.x or higher. *npm* is Node's package manager - it ships with Node.

Installing MySQL

The easiest path: install **XAMPP**, which bundles MySQL plus the lovely **phpMyAdmin** web tool for browsing your database. Even if you never use the PHP that XAMPP also installs, the MySQL + phpMyAdmin combo is the friendliest setup for beginners.

1. Visit **apachefriends.org** and download XAMPP.

2. Run the installer; you only need MySQL + phpMyAdmin from the options (uncheck the rest if you like).

3. Open the XAMPP Control Panel. Click **Start** next to MySQL.

4. Visit **http://localhost/phpmyadmin** in your browser. You should see the phpMyAdmin login page or be logged in already.

> **NOTE**
>
> If you prefer the official MySQL installer (without XAMPP), grab MySQL Community Server from dev.mysql.com. It works the same; you will just install phpMyAdmin separately or use the MySQL Workbench app.

Installing VS Code

1. Visit **code.visualstudio.com**.
2. Click **Download**. The site auto-detects your OS.
3. Run the installer with default options.

VS Code extensions worth installing

• **ES7+ React/Redux/React-Native snippets** - type *rfc* + Tab to scaffold a React component.

• **Prettier - Code formatter** - auto-format on save.

• **SQLTools** with the SQLTools MySQL/MariaDB driver - browse and query MySQL right inside VS Code.

• **Thunder Client** - a lightweight Postman alternative for testing APIs. We will use it in chapter 9.

• **GitLens** - line history.

Quick check - all tools at once

Open a terminal and run each of these. They should all print version numbers:

```
node --version       # v22.x.x or higher
npm --version        # 10.x.x or higher

mysql --version      # 8.x or higher (Windows: from C:\xampp\mysql\bin)
```

> **NOTE**
>
> If **mysql --version** says "not found" on Windows, MySQL is not on your PATH. You can either add *C:\xampp\mysql\bin* to your PATH, or just use the MySQL Console from XAMPP for command-line access.

Chapter 5 - Setting Up the MySQL Database

Before we write any code, let us create the database and table where todos will live. This will take five minutes.

Step 1 - Open phpMyAdmin

Make sure MySQL is running in the XAMPP Control Panel (the MySQL row should show green "Running"). Then visit **http://localhost/phpmyadmin** in your browser.

Step 2 - Create the database

1. Click the **Databases** tab at the top.
2. In the "Create database" form, type **todo_app**.
3. Choose **utf8mb4_unicode_ci** as the collation.
4. Click **Create**.

You will see **todo_app** appear in the left sidebar. Click it - the database is empty. Time to add a table.

Step 3 - Create the todos table

We could click through phpMyAdmin's UI, but the cleaner approach is to run the SQL directly. Click the **SQL** tab at the top and paste this query:

```
CREATE TABLE todos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    completed BOOLEAN NOT NULL DEFAULT FALSE,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
                            ON UPDATE CURRENT_TIMESTAMP
);
```

Click **Go**. phpMyAdmin runs the query and shows "Query executed successfully". You now have a **todos** table inside the **todo_app** database.

Reading the SQL line by line

Line	What it does
CREATE TABLE todos	Create a new table called todos.
id INT AUTO_INCREMENT PRIMARY KEY	Auto-numbered ID column. Database fills it in (1, 2, 3...).
title VARCHAR(255) NOT NULL	The task text. Up to 255 characters, must not be empty.
completed BOOLEAN NOT NULL DEFAULT FALSE	Whether it is done. Stored as 0 or 1. New todos start unfinished.
created_at DATETIME ... DEFAULT CURRENT_TIMESTAMP	When this row was inserted. Set automatically.
updated_at DATETIME ... ON UPDATE CURRENT_TIMESTAMP	When this row was last changed. MySQL updates this automatically every time the row is modified.

Step 4 - Add a couple of test rows

Run another SQL query (still in the SQL tab):

```
INSERT INTO todos (title) VALUES
    ('Buy groceries'),
    ('Finish React tutorial'),
    ('Call the dentist');
```

Click the **todos** table in the left sidebar, then the **Browse** tab. You should see three rows. Each has an id (1, 2, 3), a title, completed=0, and timestamps. **Excellent.** We have a working database.

Step 5 - Create a database user (optional but recommended)

By default XAMPP's MySQL has a **root** user with no password. That is fine for local development but bad for production. For this tutorial we will keep using *root*, but you should know how to make a proper user later:

```
CREATE USER 'todoapp'@'localhost' IDENTIFIED BY 'a-good-password';

GRANT ALL ON todo_app.* TO 'todoapp'@'localhost';

FLUSH PRIVILEGES;
```

> **WARNING**
>
> Never use **root** or empty passwords in production. Each app should have its own database user with permissions to ONLY its own database. The principle of least privilege.

What you will see in phpMyAdmin

Throughout this tutorial, when we modify data via the API, you can refresh the **Browse** tab in phpMyAdmin to see the rows change in real time. It is a great way to confirm your backend is doing what you expect.

Chapter 6 - Building the Backend - Node.js + Express Setup

Time to write our backend. We will use **Express** - the most popular Node.js framework, in use since 2010. Tens of thousands of production apps run on it.

Step 1 - Create the project folder

We will create two folders side by side: one for the backend, one for the frontend. Open a terminal:

```
cd ~/code            # or wherever you keep projects
mkdir todo-app
cd todo-app
mkdir backend frontend
cd backend
```

Step 2 - Initialise the npm project

```
npm init -y
```

npm init -y creates a **package.json** file with default values. *package.json* is your project's identity card - it lists the project name, version, dependencies, and scripts.

Step 3 - Install the packages we need

```
npm install express cors mysql2 dotenv
npm install --save-dev nodemon
```

What each package does

Package	Purpose
express	The web framework. Routes URLs to handler functions.
cors	Allows the React frontend (different port) to talk to the API.
mysql2	Driver to talk to MySQL. The /promise version supports async/await.
dotenv	Loads secrets (DB password) from a .env file.
nodemon	Dev tool. Restarts the server automatically when you save a file.

Step 4 - Create the .env file

In the **backend/** folder, create a file called **.env** with this content:

```
PORT=4000
DB_HOST=127.0.0.1
DB_PORT=3306
DB_USER=root
DB_PASSWORD=
DB_NAME=todo_app
```

> **WARNING**
>
> **.env** contains secrets. Never commit it to Git. We will add it to .gitignore in chapter 15.

Step 5 - Create db.js (the database connection)

Create **backend/db.js**:

```
// backend/db.js
import mysql from 'mysql2/promise'
import 'dotenv/config'

export const pool = mysql.createPool({
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
    waitForConnections: true,
    connectionLimit: 10,
})
```

Reading db.js line by line

- **import mysql from 'mysql2/promise'** - load the MySQL driver. The */promise* version supports async/await.
- **import 'dotenv/config'** - reads the .env file and puts each variable into *process.env*. The bare-import syntax runs the side effects without binding a name.
- **mysql.createPool({...})** - a connection pool keeps a few MySQL connections open so we don't pay the connect cost on every query. Massively faster than creating a new connection each time.
- **connectionLimit: 10** - up to 10 simultaneous queries.
- **export const pool** - so other files can import the pool.

Step 6 - Tell Node we are using ES modules

By default Node.js uses an older module system called CommonJS (*require/module.exports*). The modern style is ES modules (*import/export*) - same as React. To enable it, open **backend/package.json** and add one line:

```
{
    "name": "backend",
    "version": "1.0.0",
    "type": "module",
    "main": "index.js",
    ...
}
```

The **"type": "module"** line tells Node to treat .js files in this project as ES modules. Now *import* and *export* work without any compilation step.

Step 7 - Create index.js (the server)

Create **backend/index.js**:

```
// backend/index.js
import express from 'express'
import cors from 'cors'
import 'dotenv/config'

const app = express()
const PORT = process.env.PORT || 4000

// Middleware
app.use(cors())            // allow requests from React (port 5173)
app.use(express.json())    // parse JSON request bodies

// A test route - delete later
app.get('/api/health', (req, res) => {
    res.json({ status: 'ok' })
})

app.listen(PORT, () => {
    console.log(`Server running on http://localhost:${PORT}`)
})
```

What is happening here?

- **const app = express()** - create an Express application.
- **app.use(cors())** - add CORS middleware. Without this, the browser blocks fetches from React (port 5173) to Express (port 4000) for security.
- **app.use(express.json())** - middleware that parses JSON request bodies. Without this, *req.body* would be undefined.
- **app.get('/api/health', ...)** - register a GET route. The callback receives *req* (request) and *res* (response).
- **res.json(...)** - send a JSON response.
- **app.listen(PORT, ...)** - actually start the HTTP server.

Step 8 - Add a dev script

Open **package.json** and replace the "scripts" section:

```
"scripts": {
    "dev": "nodemon index.js",
    "start": "node index.js"
},
```

npm run dev uses nodemon - it restarts the server when you save a file. **npm start** uses plain node - for production.

Step 9 - Run it!

```
npm run dev
```

You should see *Server running on http://localhost:4000*. Open **http://localhost:4000/api/health** in your browser - you should see *{"status":"ok"}*. **Excellent.** Your first Node.js server is running.

> **TIP**
>
> Keep this terminal open. nodemon will keep watching - whenever you save a file in this folder, it restarts the server automatically. Open a SECOND terminal for running other commands.

Chapter 7 - The First Endpoint - GET /api/todos

Now we wire Express to MySQL. The first endpoint will return all todos as JSON. Once this works, the rest of the API is just variations on the same theme.

Step 1 - Add the GET route

Open **backend/index.js** and add this above the test /api/health route. Also import the pool from db.js:

```
// at the top of index.js
import { pool } from './db.js'

// ... existing code ...

// GET /api/todos - return all todos, newest first
app.get('/api/todos', async (req, res) => {
    try {
        const [rows] = await pool.query(
            'SELECT * FROM todos ORDER BY id DESC'
        )
        res.json(rows)
    } catch (error) {
        console.error(error)
        res.status(500).json({ error: 'Failed to fetch todos' })
    }
})
```

Reading the route line by line

- **async (req, res) => {...}** - the handler is async because we use await inside.
- **const [rows] = await pool.query(...)** - MySQL2 returns *[rows, fields]*. We destructure to grab just the rows.
- **SELECT * FROM todos ORDER BY id DESC** - SQL: "give me every column from todos, newest id first".
- **res.json(rows)** - send the rows as JSON.
- **try / catch** - if anything fails (DB down, bad query), we log it and return HTTP 500 instead of crashing the server.
- **res.status(500).json(...)** - 500 means Internal Server Error.

Step 2 - Try it

nodemon should have restarted automatically. Open **http://localhost:4000/api/todos** in your browser. You should see your three test todos as a JSON array:

```
[
    {
        "id": 3,
        "title": "Call the dentist",
        "completed": 0,
        "created_at": "2026-04-27T...",
        "updated_at": "2026-04-27T..."
    },
    {
        "id": 2,
        "title": "Finish React tutorial",
        ...
    },
    ...
]
```

> **NOTE**
>
> Notice **completed** comes back as 0 or 1, not true/false. MySQL stores BOOLEAN as a TINYINT under the hood. We will convert to real booleans on the React side.

What just happened?

Trace the path of the request through what we built:

1. Browser made a GET request to /api/todos.
2. Express matched the URL to our handler function.
3. Our function called *pool.query("SELECT ...")*.
4. *mysql2* sent the SQL to MySQL on port 3306.
5. MySQL returned the rows.
6. *mysql2* resolved the await with *[rows, fields]*.
7. We destructured to *rows*, then *res.json(rows)*.
8. Express serialised the array to JSON and sent it back.
9. The browser displayed the JSON.

> **TIP**
>
> This is the entire backend pattern in nine steps. Every API endpoint we write follows the same shape - just with different SQL and validation in the middle.

Chapter 8 - POST, PUT, DELETE - Full CRUD API

GET reads. We also need to **C**reate, **U**pdate, and **D**elete - the four together make **CRUD**. Three more endpoints, and our backend is complete.

Step 1 - POST /api/todos (create a todo)

Add this route to **backend/index.js**:

```
// POST /api/todos - create a new todo
app.post('/api/todos', async (req, res) => {
    try {
        const { title } = req.body

        // Validate
        if (!title || title.trim() === '') {
            return res.status(400).json({ error: 'Title is required' })
        }

        const [result] = await pool.query(
            'INSERT INTO todos (title) VALUES (?)',
            [title.trim()]
        )

        // Return the created todo with its new id
        const [rows] = await pool.query(
            'SELECT * FROM todos WHERE id = ?',
            [result.insertId]
        )
        res.status(201).json(rows[0])
    } catch (error) {
        console.error(error)
        res.status(500).json({ error: 'Failed to create todo' })
    }
})
```

What is new here

- **const { title } = req.body** - destructure the title from the JSON body. *express.json()* middleware (added in chapter 6) parsed the body for us.
- **title.trim() === ''** - *trim()* removes leading/trailing whitespace. Stops users from creating todos full of spaces.
- **VALUES (?)** - the question mark is a **parameter placeholder**. mysql2 substitutes it safely. NEVER use string concatenation in SQL - that's how SQL injection attacks happen.
- **result.insertId** - the id MySQL just generated for the new row.
- **res.status(201)** - 201 means "Created". The convention for successful POST.

> **WARNING**
>
> **Always use parameterised queries (the ? placeholder).** Never build SQL with string concatenation - that opens your database to injection attacks. mysql2's ? syntax is safe.

Step 2 - PUT /api/todos/:id (update)

```
// PUT /api/todos/:id - update a todo
app.put('/api/todos/:id', async (req, res) => {
    try {
        const id = Number(req.params.id)
        const { title, completed } = req.body

        // Build the update dynamically - only update fields we received
        const updates = []
        const values = []

        if (title !== undefined) {
            updates.push('title = ?')
            values.push(title.trim())
        }
        if (completed !== undefined) {
            updates.push('completed = ?')
            values.push(completed ? 1 : 0)
        }

        if (updates.length === 0) {
            return res.status(400).json({ error: 'No fields to update' })
        }

        values.push(id)  // for the WHERE clause
        await pool.query(
            `UPDATE todos SET ${updates.join(', ')} WHERE id = ?`,
            values
        )

        // Return the updated row
        const [rows] = await pool.query(
            'SELECT * FROM todos WHERE id = ?',
            [id]
        )
        if (rows.length === 0) {
            return res.status(404).json({ error: 'Todo not found' })
        }
        res.json(rows[0])
    } catch (error) {
        console.error(error)
        res.status(500).json({ error: 'Failed to update todo' })
    }
})
```

Reading the dynamic update

• **req.params.id** - the *:id* in the URL becomes *req.params.id*. URL params are always strings, so we wrap with *Number()*.

• We build the SET clause dynamically because the frontend might send only *completed* (toggling) or only *title* (editing) - not always both.

• **updates.push('title = ?')** - add a SET clause. **values.push(...)** - add the matching value.

• **updates.join(', ')** - join with commas: "title = ?, completed = ?".

• **completed ? 1 : 0** - convert true/false to 1/0 for MySQL.

• **404 if rows.length === 0** - the id did not exist.

Step 3 - DELETE /api/todos/:id

```
// DELETE /api/todos/:id
app.delete('/api/todos/:id', async (req, res) => {
    try {
        const id = Number(req.params.id)
        const [result] = await pool.query(
            'DELETE FROM todos WHERE id = ?',
            [id]
        )
        if (result.affectedRows === 0) {
            return res.status(404).json({ error: 'Todo not found' })
        }
        res.status(204).end()
    } catch (error) {
        console.error(error)
        res.status(500).json({ error: 'Failed to delete todo' })
    }
})
```

- **result.affectedRows** - tells us how many rows the query actually changed. If zero, the id did not exist - 404.
- **res.status(204).end()** - 204 means "No Content". The delete worked but we don't have anything to send back.

Step 4 - The complete index.js

Your **backend/index.js** should now look like this. We can also delete the test /api/health route since we have real endpoints:

```
// backend/index.js (final)
import express from 'express'
import cors from 'cors'
import 'dotenv/config'
import { pool } from './db.js'

const app = express()
const PORT = process.env.PORT || 4000

app.use(cors())
app.use(express.json())

// GET /api/todos - list all
app.get('/api/todos', async (req, res) => {
    try {
        const [rows] = await pool.query(
            'SELECT * FROM todos ORDER BY id DESC'
        )
        res.json(rows)
    } catch (error) {
        console.error(error)
        res.status(500).json({ error: 'Failed to fetch todos' })
    }
})

// POST /api/todos - create
app.post('/api/todos', async (req, res) => {
    // ...as above
})

// PUT /api/todos/:id - update
app.put('/api/todos/:id', async (req, res) => {
```

```
    // ...as above
})

// DELETE /api/todos/:id - remove
app.delete('/api/todos/:id', async (req, res) => {
    // ...as above
})

app.listen(PORT, () => {
    console.log(`Server running on http://localhost:${PORT}`)
})
```

> **TIP**
>
> We have a full CRUD API in about 80 lines of code. That is the Express + mysql2 productivity gain. In chapter 9 we will test every endpoint before touching React.

Chapter 9 - Testing the API with Thunder Client

Always test the backend BEFORE building the frontend. If your API is broken, no amount of React magic will save you. We will use **Thunder Client**, a free VS Code extension that lets you send HTTP requests right from your editor.

Step 1 - Install Thunder Client

If you did not install it in chapter 4: open VS Code, go to Extensions (Ctrl+Shift+X), search "Thunder Client", click Install. A lightning-bolt icon appears in the left sidebar.

Step 2 - Make sure your server is running

In your backend terminal, you should still have **npm run dev** running. If not, run it now.

Step 3 - Test GET /api/todos

1. Click the lightning-bolt icon in VS Code.
2. Click **New Request**.
3. Set the method to **GET**.
4. URL: **http://localhost:4000/api/todos**.
5. Click **Send**.

On the right side you should see the response - your three test todos. Status code should be **200 OK** in the top right.

Step 4 - Test POST (create)

1. New Request. Method: **POST**.
2. URL: **http://localhost:4000/api/todos**.
3. Click the **Body** tab below the URL.
4. Choose **JSON** as the body type.
5. Paste: *{ "title": "Learn full-stack development" }*
6. Click Send.

You should get back HTTP 201 with the new todo (including its newly-assigned id and timestamps). Switch to phpMyAdmin and browse the todos table - your new row is there!

Step 5 - Test PUT (update)

1. New Request. Method: **PUT**.
2. URL: **http://localhost:4000/api/todos/1** (use a real id from your DB).
3. Body tab -> JSON: *{ "completed": true }*
4. Click Send.

Status 200, response body shows the updated row with *completed: 1*. phpMyAdmin should show the change too. Try another PUT with *{ "title": "New title" }* - it should update only the title.

Step 6 - Test DELETE

1. New Request. Method: **DELETE**.
2. URL: **http://localhost:4000/api/todos/1**.
3. No body needed.
4. Click Send.

Status 204 (No Content). phpMyAdmin: the row is gone. Try the same DELETE again - now you should get 404, because that id no longer exists.

Step 7 - Test the error paths

A real API gets garbage requests every day. Make sure yours handles them gracefully:

- **POST with empty body**: *{}* - should return 400 "Title is required".
- **POST with whitespace only**: *{ "title": " " }* - should return 400.
- **PUT with bogus id**: PUT /api/todos/99999 - should return 404.
- **DELETE with bogus id**: DELETE /api/todos/99999 - should return 404.

> **TIP**
>
> Good APIs **fail loudly and consistently**. A 400 for bad input, 404 for missing resources, 500 for server errors. The frontend will rely on these status codes to display the right error messages.

What you have built

Your backend is now a real REST API. It can be consumed by ANY frontend - a React app (next chapter), a mobile app, a desktop app, a curl script, another backend service. That portability is the value of separating frontend and backend.

Chapter 10 - Setting Up the React Frontend

The backend is done and tested. Now the fun part - the user interface. We will use the same tooling as our React tutorial: **Vite** as the build tool, **Bootstrap 5** for styling.

Step 1 - Create the React project

Go back to the **todo-app** root folder (one level up from backend), then create the frontend:

```
cd ..                  # back to todo-app/
cd frontend            # we made this folder in chapter 6
npm create vite@latest . -- --template react

# When asked, choose React (not React TypeScript) and JavaScript.

npm install
```

The **.** after *vite@latest* means "use the current folder" instead of creating a new sub-folder.

Step 2 - Add Bootstrap to index.html

Open **frontend/index.html** and add Bootstrap before the closing </head> tag:

```
<link
    href="https://cdn.jsdelivr.net/npm/[email protected]/dist/css/bootstrap.min.css"
    rel="stylesheet"
>
```

Step 3 - Clean out the default app

Replace the contents of **frontend/src/App.jsx**:

```
// frontend/src/App.jsx
import './App.css'

function App() {
    return (
        <div>
            <header className="bg-primary text-white py-3 px-4">
                <h2 className="mb-0">Todo</h2>
                <small>Stay organised, get things done.</small>
            </header>
            <main className="container py-4">
                <p>Coming soon - the todo list.</p>
            </main>
        </div>
    )
}

export default App
```

Empty out **App.css** and **index.css**. We will rely on Bootstrap for now.

Step 4 - Run the React dev server

```
npm run dev
```

You should see Vite print a URL like **http://localhost:5173**. Open it in the browser - blue header at the top, "Coming soon" below. Beautiful start.

> **TIP**
>
> You now have THREE things running: MySQL (port 3306), Express (port 4000), and React/Vite (port 5173). Three terminals open. Welcome to full-stack development.

Step 5 - Set up the API base URL

We will be calling fetch('http://localhost:4000/...') from many places. Let us put the base URL in one place so we can change it later (e.g. for production). Create **frontend/src/api.js**:

```
// frontend/src/api.js
const API_BASE = 'http://localhost:4000/api'

export async function fetchTodos() {
    const response = await fetch(`${API_BASE}/todos`)
    if (!response.ok) {
        throw new Error('Failed to fetch todos')
    }
    return response.json()
}

export async function createTodo(title) {
    const response = await fetch(`${API_BASE}/todos`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ title }),
    })
    if (!response.ok) {
        throw new Error('Failed to create todo')
    }
    return response.json()
}

export async function updateTodo(id, changes) {
    const response = await fetch(`${API_BASE}/todos/${id}`, {
        method: 'PUT',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(changes),
    })
    if (!response.ok) {
        throw new Error('Failed to update todo')
    }
    return response.json()
}

export async function deleteTodo(id) {
    const response = await fetch(`${API_BASE}/todos/${id}`, {
        method: 'DELETE',
    })
    if (!response.ok) {
        throw new Error('Failed to delete todo')
    }
}
```

Reading api.js

• Each function wraps a single fetch call.

• **response.ok** - true for status 200-299, false for 400+.

• **throw new Error(...)** - if the request failed, throw so the caller can catch.

• **response.json()** - parse the response body as JSON. Returns a promise.

• **JSON.stringify(...)** - convert a JavaScript object to a JSON string. The opposite of JSON.parse.

• **headers: { 'Content-Type': 'application/json' }** - tells the server we are sending JSON. Without this, Express's json() middleware would not parse the body.

> **TIP**
>
> Centralising fetch calls in one file is a useful habit. When the API URL changes, when you add authentication tokens, when you add error logging - you change one file, not 12.

Chapter 11 - Fetching Todos from the API

Time to make React talk to our backend. We will load todos from the API when the page loads, and display them. By the end of this chapter, your React app will be reading real data from MySQL through Express.

Step 1 - useEffect for loading data

When a component mounts (appears on screen), we want to fetch todos from the API. The right hook for this is **useEffect**:

```
import { useState, useEffect } from 'react'
import { fetchTodos } from './api'

function App() {
    const [todos, setTodos] = useState([])
    const [loading, setLoading] = useState(true)
    const [error, setError] = useState(null)

    useEffect(() => {
        async function load() {
            try {
                const data = await fetchTodos()
                setTodos(data)
            } catch (err) {
                setError(err.message)
            } finally {
                setLoading(false)
            }
        }
        load()
    }, [])  // empty array = run once on mount

    // ...rest of component
}
```

Reading useEffect

- **useEffect(() => {...}, [])** - run the function once after the component first renders. The empty dependency array is the magic.
- **async function load()** - we cannot make the useEffect callback itself async, so we define an async helper inside and call it.
- **try / catch / finally** - on success, store the data. On error, store the message. Either way, mark loading as done.
- Three states: **loading** (waiting for API), **error** (API failed), **todos** (success). The UI shows different things for each.

Step 2 - Display the todos

Update **App.jsx** with the full component:

```
// frontend/src/App.jsx
import { useState, useEffect } from 'react'
import { fetchTodos } from './api'
import './App.css'

function App() {
    const [todos, setTodos] = useState([])
    const [loading, setLoading] = useState(true)
    const [error, setError] = useState(null)

    useEffect(() => {
        async function load() {
            try {
                const data = await fetchTodos()
                setTodos(data)
            } catch (err) {
                setError(err.message)
            } finally {
                setLoading(false)
            }
        }
        load()
    }, [])

    if (loading) {
        return (
            <div className="container py-5 text-center">
                <p>Loading todos...</p>
            </div>
        )
    }

    if (error) {
        return (
            <div className="container py-5">
                <div className="alert alert-danger">{error}</div>
            </div>
        )
    }

    return (
        <div>
            <header className="bg-primary text-white py-3 px-4">
                <h2 className="mb-0">Todo</h2>
                <small>Stay organised, get things done.</small>
            </header>
            <main className="container py-4">
                <div className="card mx-auto" style={{ maxWidth: 640 }}>
                    <div className="card-body">
                        <h3>My Tasks</h3>
                        <p className="text-muted small">
                            {todos.length} task
                            {todos.length !== 1 ? 's' : ''}
                        </p>

                        {todos.length === 0 ? (
                            <p className="text-muted">
                                No tasks yet. Add one above.
                            </p>
                        ) : (
```

```
                            <ul className="list-unstyled">
                                {todos.map((todo) => (
                                    <li
                                        key={todo.id}
                                        className="border rounded p-3 mb-2"
                                    >
                                        {todo.title}
                                    </li>
                                ))}
                            </ul>
                        )}
                    </div>
                </div>
            </main>
        </div>
    )
}

export default App
```

Try it!

Save. Make sure the backend is still running on port 4000. Refresh **http://localhost:5173**. You should see your test todos appearing as cards. **That** is your React frontend talking to your Node backend talking to your MySQL database. Three layers, one app.

> **TIP**
>
> Open the browser DevTools, Network tab. Refresh the page. You will see one request to **localhost:4000/api/todos** with the JSON response. This is the contract between frontend and backend made visible.

What about CORS?

If you see an error like *"Access-Control-Allow-Origin"* in the console, your browser blocked the cross-origin request. We added **app.use(cors())** in chapter 6 to allow this. If you see this error, double-check that line is in your backend/index.js.

Chapter 12 - Adding New Todos with a Form

Reading is half the job. Now we add an input field that creates new todos via POST.

Step 1 - Track the input value with state

Add a new state variable for the input field. In React, form inputs that read from state and write to state are called **controlled components**:

```
// inside App, add a new state
const [newTitle, setNewTitle] = useState('')

// The function that submits the form
async function handleAdd(e) {
    e.preventDefault()
    if (newTitle.trim() === '') return

    try {
        const newTodo = await createTodo(newTitle)
        setTodos([newTodo, ...todos])  // newest at top
        setNewTitle('')                // clear the input
    } catch (err) {
        alert(err.message)
    }
}
```

Reading handleAdd line by line

- **e.preventDefault()** - stop the form's default behaviour (reload the page).
- **if (newTitle.trim() === '') return** - guard against empty input before even calling the API.
- **const newTodo = await createTodo(newTitle)** - call the API helper from chapter 10. The server returns the new todo.
- **setTodos([newTodo, ...todos])** - put the new one at the front of the array. The spread copies the existing todos. NEVER use *todos.unshift(newTodo)* - that mutates and React would not re-render.
- **setNewTitle('')** - clear the input.

Step 2 - Add the form to the JSX

Inside the card-body, before the H3, add the form:

```
<form onSubmit={handleAdd} className="mb-4">
    <div className="input-group">
        <input
            type="text"
            className="form-control"
            placeholder="What needs to be done?"
            value={newTitle}
            onChange={(e) => setNewTitle(e.target.value)}
        />
        <button type="submit" className="btn btn-primary">
            Add
        </button>
    </div>
</form>
```

Save. Type a task in the input. Press Enter or click Add. The task appears at the top of the list. Refresh the page - it is still there because it was saved to MySQL.

Why controlled inputs?

Notice the pattern - the input's **value** comes from state, and **onChange** writes back to state. That makes React the single source of truth. We can read or modify the input's value from anywhere in our component, validate as the user types, transform input (uppercase, format), or clear it programmatically. Uncontrolled inputs (where the DOM owns the value) are harder to work with.

Don't import yet - import what we use

Update the imports at the top of App.jsx:

```
import { useState, useEffect } from 'react'
import { fetchTodos, createTodo } from './api'
import './App.css'
```

> **TIP**
>
> Open phpMyAdmin and watch the todos table while you add tasks. Each Add click inserts a new row. This is the satisfying moment when full-stack "clicks" - data flowing from your React click all the way down to a row in your MySQL database.

Chapter 13 - Toggling, Editing, and Deleting

Three more user actions to wire up: marking a todo as complete, editing its title, and deleting it. Each one calls a different API method. Same pattern as Add.

Step 1 - Toggle complete

Add this function inside App:

```
async function handleToggle(todo) {
    try {
        const updated = await updateTodo(todo.id, {
            completed: !todo.completed,
        })
        setTodos(todos.map((t) => (t.id === todo.id ? updated : t)))
    } catch (err) {
        alert(err.message)
    }
}
```

How this works

- **!todo.completed** - flip true to false (or 1 to 0).
- **todos.map(t => t.id === todo.id ? updated : t)** - create a new array where the matching todo is replaced with the updated one. All other todos pass through unchanged.
- Once again - we never mutate. Always create a new array via map.

Step 2 - Delete a todo

```
async function handleDelete(id) {
    if (!confirm('Delete this task?')) return

    try {
        await deleteTodo(id)
        setTodos(todos.filter((t) => t.id !== id))
    } catch (err) {
        alert(err.message)
    }
}
```

Reading handleDelete

- **confirm('...')** - browser native confirmation dialog. Returns true/false.
- **todos.filter(t => t.id !== id)** - new array without the deleted todo.
- Notice we update local state AFTER the API call succeeded. If the API fails, the alert shows and the local list stays unchanged.

Step 3 - Edit a todo (the trickiest)

Editing is more involved because we need to track which todo is being edited and what the new text is. Let us track that with two more state variables:

```
const [editingId, setEditingId] = useState(null)
const [editingTitle, setEditingTitle] = useState('')

function startEdit(todo) {
    setEditingId(todo.id)
    setEditingTitle(todo.title)
}

function cancelEdit() {
    setEditingId(null)
    setEditingTitle('')
}

async function saveEdit() {
    if (editingTitle.trim() === '') {
        cancelEdit()
        return
    }
    try {
        const updated = await updateTodo(editingId, {
            title: editingTitle,
        })
        setTodos(todos.map((t) => (t.id === editingId ? updated : t)))
        cancelEdit()
    } catch (err) {
        alert(err.message)
    }
}
```

How edit mode works

- **editingId === null** - no row is being edited.
- **editingId === 5** - row with id 5 is being edited; show an input instead of the text.
- **editingTitle** - the current value in the input. Updates as the user types.
- **cancelEdit()** - exit edit mode without saving.
- **saveEdit()** - PUT the new title, then exit edit mode.

Step 4 - Update the JSX

Replace the todo <li> rendering with this richer version that handles all three actions:

```
<ul className="list-unstyled">
    {todos.map((todo) => (
        <li key={todo.id} className="border rounded p-3 mb-2 d-flex">
            <input
                type="checkbox"
                className="form-check-input me-3 mt-1"
                checked={todo.completed === 1}
                onChange={() => handleToggle(todo)}
            />

            <div className="flex-grow-1">
                {editingId === todo.id ? (
                    <input
                        type="text"
                        className="form-control"
                        value={editingTitle}
                        autoFocus
                        onChange={(e) =>
                            setEditingTitle(e.target.value)
                        }
                        onKeyDown={(e) => {
                            if (e.key === 'Enter') saveEdit()
                            if (e.key === 'Escape') cancelEdit()
                        }}
                    />
                ) : (
                    <span
                        style={{
                            textDecoration:
                                todo.completed === 1
                                    ? 'line-through'
                                    : 'none',
                            color:
                                todo.completed === 1
                                    ? '#888'
                                    : 'inherit',
                        }}
                    >
                        {todo.title}
                    </span>
                )}
            </div>

            <div className="ms-3">
                {editingId === todo.id ? (
                    <>
                        <button
                            className="btn btn-success btn-sm me-1"
                            onClick={saveEdit}
                        >
                            Save
                        </button>
                        <button
                            className="btn btn-light btn-sm"
                            onClick={cancelEdit}
                        >
                            Cancel
                        </button>
                    </>
                ) : (
```

```
                    <>
                        <button
                            className="btn btn-link btn-sm text-primary"
                            onClick={() => startEdit(todo)}
                        >
                            Edit
                        </button>
                        <button
                            className="btn btn-link btn-sm text-danger"
                            onClick={() => handleDelete(todo.id)}
                        >
                            Delete
                        </button>
                    </>
                )}
            </div>
        </li>
    ))}
</ul>
```

What the <> ... </> means

<>...</> is a *fragment* - a way to return multiple elements from JSX without adding an extra wrapper div. We need it here because each branch of the ternary returns two buttons.

Don't forget the imports

```
import { useState, useEffect } from 'react'
import {
    fetchTodos,
    createTodo,
    updateTodo,
    deleteTodo,
} from './api'
```

> **TIP**
>
> Save and try the full flow: add, check, edit, delete. Refresh the page to confirm everything persisted to MySQL. You now have a working full-stack CRUD app.

Chapter 14 - Filters and Polish

Time for the finishing touches. We will add filter tabs (All / Active / Completed) and a Clear Completed button. Pure frontend work this chapter - no API changes needed.

Step 1 - Track the active filter

```
const [filter, setFilter] = useState('all')   // 'all' | 'active' | 'completed'

const visibleTodos = todos.filter((todo) => {
    if (filter === 'active') return todo.completed === 0
    if (filter === 'completed') return todo.completed === 1
    return true   // 'all'
})

const completedCount = todos.filter((t) => t.completed === 1).length
```

Reading the filter logic

- **filter** state holds one of three strings: 'all', 'active', 'completed'.
- **visibleTodos** is a derived value - we don't store it as state because it can be calculated from *todos* and *filter* on every render. **Never duplicate state** - if it can be derived, derive it.
- **completedCount** - same idea, derived value.

Step 2 - The filter tabs

Add this between the form and the list:

```
<div className="d-flex justify-content-between align-items-center mb-3">
    <div className="btn-group">
        {['all', 'active', 'completed'].map((f) => (
            <button
                key={f}
                className={
                    'btn btn-sm ' +
                    (filter === f
                        ? 'btn-primary'
                        : 'btn-outline-primary')
                }
                onClick={() => setFilter(f)}
            >
                {f.charAt(0).toUpperCase() + f.slice(1)}
            </button>
        ))}
    </div>

    {completedCount > 0 && (
        <button
            className="btn btn-link btn-sm text-danger"
            onClick={handleClearCompleted}
        >
            Clear completed ({completedCount})
        </button>
    )}
</div>
```

What's new here

- **['all', 'active', 'completed'].map(f => ...)** - render three buttons from one array. Saves repeating ourselves.
- **f.charAt(0).toUpperCase() + f.slice(1)** - capitalise the first letter for display: 'all' -> 'All'.
- **{completedCount > 0 && (...)}** - conditionally render the Clear button only when there is something to clear. The *&&* trick is a common React pattern.

Step 3 - Clear completed

```
async function handleClearCompleted() {
    const completedTodos = todos.filter((t) => t.completed === 1)
    if (!confirm(`Delete ${completedTodos.length} completed task(s)?`)) {
        return
    }
    try {
        // Delete each one in parallel
        await Promise.all(
            completedTodos.map((t) => deleteTodo(t.id))
        )
        setTodos(todos.filter((t) => t.completed !== 1))
    } catch (err) {
        alert(err.message)
    }
}
```

Promise.all - parallel requests

If we had 10 completed todos and used a regular for-loop with *await*, the deletes would happen one after another - 10 round trips. **Promise.all** fires them off in parallel and waits for all to finish. Much faster.

Step 4 - Use visibleTodos in the list

Change the .map() to use **visibleTodos** instead of **todos**:

```
{visibleTodos.length === 0 ? (
    <p className="text-muted">
        {filter === 'all'
            ? 'No tasks yet. Add one above.'
            : `No ${filter} tasks.`}
    </p>
) : (
    <ul className="list-unstyled">
        {visibleTodos.map((todo) => (
            // ...same li as before
        ))}
    </ul>
)}
```

Save. Try the filter buttons. Add some todos, check a few, switch between All / Active / Completed - the list updates instantly. Click Clear Completed - poof, all the checked ones are gone.

Step 5 - A counter at the top

Final polish. Replace the simple count below the H3 with something more useful:

```
<p className="text-muted small">
    {todos.length} task{todos.length !== 1 ? 's' : ''}
    {completedCount > 0 && ` - ${completedCount} completed`}
</p>
```

> **TIP**
>
> Notice how the user interface feels alive. Every state change is reflected instantly across the page - the count, the filter view, the button states. We never wrote any DOM update code. React did all of it because we kept state as the source of truth.

Chapter 15 - Deployment and Where to Go Next

Your app works locally. Deploying a full-stack app is more involved than a static site - you have three pieces to ship. We will walk through the most beginner-friendly path.

Three things to deploy

- **The MySQL database** - hosted somewhere reachable.
- **The Express backend** - a Node.js process running 24/7.
- **The React frontend** - static HTML/CSS/JS files on a CDN.

The easy path - Railway (recommended for beginners)

Railway (railway.app) is a hosting platform built for full-stack apps. They give you a free starter tier with $5 of credit per month - enough for a small project. They handle the server provisioning so you only worry about code.

1. Push your project to GitHub (both backend and frontend folders).

2. Sign up at **railway.app** using your GitHub account.

3. Click **New Project** -> **Deploy from GitHub repo**.

4. Choose your repo. Railway scans it and offers to deploy.

5. Click **+ New** -> **Database** -> **Add MySQL**. Railway provisions a MySQL instance and gives you connection details.

6. Add a **Service** for the backend folder. Set environment variables (DB_HOST, DB_USER, DB_PASSWORD, DB_NAME) using Railway's MySQL connection details.

7. Run the schema (CREATE TABLE) once via Railway's MySQL Console.

8. Backend deploys, gets a URL like **todo-backend.up.railway.app**.

9. For the frontend - update **API_BASE** in api.js to your Railway backend URL.

10. Run **npm run build** in frontend/. Drop the **dist/** folder onto Netlify or Vercel for free hosting.

> **WARNING**
>
> Production URLs are different from localhost. Make sure you update **API_BASE** in api.js (or use an environment variable) before building the frontend for production.

Other options

Host	Cost	Notes
Railway	$5/mo credit free	Easiest. All three pieces in one platform.
Render	Free tier exists	Similar to Railway. Free tier has cold starts.
DigitalOcean	$4/mo droplet	Full control. You install Node, MySQL, Nginx yourself.
Hostinger / shared	Cheap	Most shared hosts don't run Node.js. MySQL only.
AWS / GCP / Azure	Free tier	Most powerful. Steepest learning curve.

Where to go next

You have built a real full-stack app. Here are 10 ideas to extend it - each one teaches another important concept:

1. **User accounts** - register and login. Each user sees only their own todos. Use bcrypt for password hashing and JWT for auth tokens.
2. **Categories or tags** - group todos by project or context.
3. **Due dates** - add a date column and a date picker.
4. **Drag-and-drop reordering** - install react-beautiful-dnd or @dnd-kit/core.
5. **Sub-tasks** - todos that can have child todos. Models a tree in the database.
6. **Search** - input field that filters todos by text.
7. **Pagination** - load 20 at a time as the user scrolls.
8. **Real-time updates** - use Server-Sent Events or websockets so multiple browsers stay in sync.
9. **Convert to TypeScript** - add types to catch bugs at compile time.
10. **Add tests** - Jest for the backend, React Testing Library for the frontend.

Final thoughts

You started this tutorial knowing little about React or Node.js. You finished by building a working full-stack app with a real database, a REST API, and a polished UI. You wrote SQL, you wrote middleware, you wrote async JavaScript, you wrote React components. That is more than what most developers do in their first year on the job.

Now build something else. Pick anything you actually want - a bookmark manager, an expense tracker, a habit logger, a weather app. Use the same stack. Iterate. Get stuck. Search Stack Overflow. Read the React docs. That cycle - frustration and breakthrough - is how real skill grows.

Share your project on LinkedIn. Add the GitHub link to your CV. Write a blog post about what you learned. Send it to **egotechworld.com** - we love featuring guest posts from learners.

Welcome to full-stack development. Happy coding!

Quick Reference Card

REST API endpoints we built

Method	URL	Body	Response
GET	/api/todos	-	Array of todos
POST	/api/todos	{ title }	201 Created todo
PUT	/api/todos/:id	{ title?, completed? }	Updated todo
DELETE	/api/todos/:id	-	204 No Content

HTTP status codes you will use most

Code	Meaning
200 OK	Success.
201 Created	POST succeeded; new resource exists.
204 No Content	Success but nothing to return (typical for DELETE).
400 Bad Request	Invalid input from the client.
401 Unauthorized	Need to log in.
403 Forbidden	Logged in but not allowed.
404 Not Found	Resource does not exist.
500 Internal Server Error	Something blew up on the server.

JavaScript syntax we used

Pattern	Where we used it
arrow function () => {}	Every event handler and array callback
destructuring const { x } = y	Reading req.body, useState, props
spread [...arr, x]	Adding to state arrays
template literal `${var}`	Building API URLs
async/await	Every API call
array.map(fn)	Rendering todo lists
array.filter(fn)	Filter tabs, deletion
array.find(fn)	Locating a specific todo
import/export	Every file

React hooks we used

Hook	What it does
useState	Declare state. Returns [value, setter].
useEffect	Run code after render. Use for fetching data on mount.

Final project structure

```
todo-app/
+-- backend/
|   +-- node_modules/         # auto, never edit
|   +-- .env                  # secrets, never commit
|   +-- db.js                 # MySQL connection pool
|   +-- index.js              # Express server with all routes
|   +-- package.json
|   `-- package-lock.json
+-- frontend/
|   +-- node_modules/         # auto, never edit
|   +-- public/
|   +-- src/
|   |   +-- api.js            # fetch wrappers for the backend
|   |   +-- App.jsx           # the whole app (one component)
|   |   +-- App.css
|   |   +-- index.css
|   |   `-- main.jsx          # mounts App into the DOM
|   +-- index.html            # the only HTML file
|   +-- package.json
|   `-- vite.config.js
`-- README.md
```

MySQL commands you will use most

Command	What it does
SELECT * FROM table	Read every row from a table.
SELECT col1, col2 FROM table	Read just specific columns.
WHERE id = ?	Filter to matching rows.
ORDER BY col DESC	Sort - newest first.
INSERT INTO table (cols) VALUES (?)	Add a new row.
UPDATE table SET col = ? WHERE id = ?	Change a row.
DELETE FROM table WHERE id = ?	Remove a row.

Common mistakes and fixes

Mistake	Fix
CORS error in browser console	Make sure **app.use(cors())** is in your Express app.
Body is undefined in POST	Make sure **app.use(express.json())** is in Express.
State not updating	You are mutating an array/object. Always create new ones with spread.
Missing key prop warning	Add **key={item.id}** to looped JSX elements.
fetch returns the response, not the data	Call **await response.json()** to get the actual body.
completed shows 0/1 not true/false	MySQL stores boolean as TINYINT. Compare with === 1.
Page reloads on form submit	Add **e.preventDefault()** in your onSubmit handler.

Where to learn more

- **react.dev** - the official React docs.
- **expressjs.com** - Express documentation.
- **dev.mysql.com/doc** - MySQL reference manual.
- **developer.mozilla.org** - the JavaScript bible.
- **egotechworld.com** - more Sinhala and English coding tutorials, free project source code, and AI tools.

Closing note from egotechworld.com

If this tutorial helped you, share it with a friend who's learning to code. Bookmark **egotechworld.com** for more React, Node, Laravel, PHP, and Python tutorials. We post new content regularly and welcome guest articles from learners like you.

Happy coding!