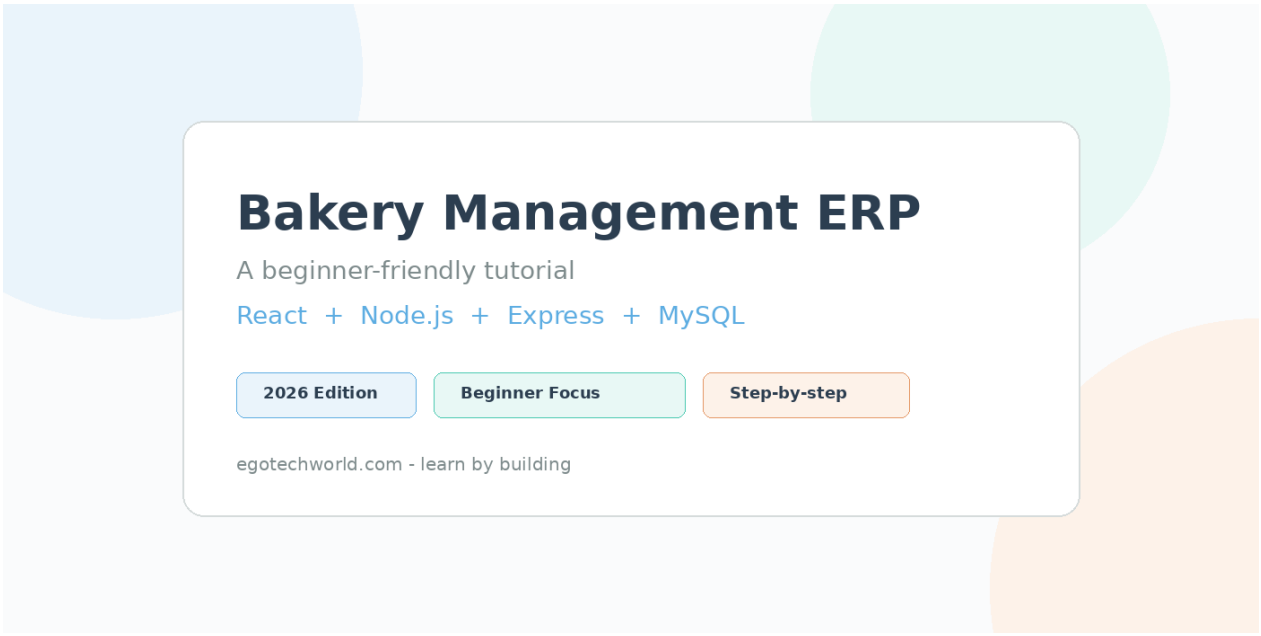

Build a complete bakery management system from scratch.
No prior experience required.

Tech stack	React 18 · Node.js 20 · Express 4 · MySQL 8
Difficulty	Beginner - assumes only basic computer use
Build time	Approximately 4 to 6 evenings of focused work
You will learn	JavaScript basics, REST APIs, databases, React UI

Table of Contents

Each chapter builds on the last. Read in order on your first pass.

GETTING STARTED

Part 1 - JavaScript - The Absolute Basics

Before we build the bakery app, we need to understand the language that powers everything in this tutorial: JavaScript. The same JavaScript runs in the browser (through React) and on the server (through Node.js). Learning it once means you can write both sides of the application.

Figure 1.1 - The four core JavaScript ideas you will use everywhere

1.1 Variables - storing values

A variable is a labelled box that holds a piece of information. In modern JavaScript we use **let** when the value can change later, and **const** when it should stay the same.

variables.js

```
// 'let' allows the value to be changed later
let stock = 50;
stock = stock - 1;       // now stock is 49

// 'const' means the value will not change
const SHOP_NAME = "Sweet Crust Bakery";

// Common types you will use
const price   = 250;            // number
const itemName = "Bread";        // string (text)
const inStock = true;            // boolean (true/false)
let   notes   = null;            // null means "no value yet"
```

Syntax

Each statement ends with a semicolon (**;**). Strings can use either single or double quotes - just be consistent. Names are case sensitive: **price** and **Price** are different variables.

1.2 Functions - reusable instructions

A function is a named block of code that runs when you call it. It can take input (parameters) and return an output. Functions are how we avoid copy-pasting the same code in many places.

functions.js

```
// Traditional function
function calculateTotal(quantity, unitPrice) {
  return quantity * unitPrice;
}

// Calling the function
const total = calculateTotal(3, 250);   // total is 750

// Arrow function (shorter, very common in React)
const addTax = (amount) => amount * 1.05;

const finalPrice = addTax(total);       // finalPrice is 787.5
console.log(finalPrice);                // prints to the terminal
```

1.3 Arrays - lists of things

An array stores many values in order. We use arrays all the time - the list of products, the items in a customer's cart, the recent orders on the dashboard.

arrays.js

```
const products = ["Bread", "Bun", "Cake", "Pastry"];

console.log(products.length);   // 4
console.log(products[0]);       // "Bread"  (first item, index 0)
console.log(products[2]);       // "Cake"

products.push("Cookie");        // add to the end
products.length;                // 5

// Loop through every item
for (const item of products) {
  console.log(item);
}

// .map() builds a new array by transforming each item
const upper = products.map(p => p.toUpperCase());
// upper = ["BREAD", "BUN", "CAKE", "PASTRY", "COOKIE"]
```

1.4 Objects - records with named fields

An object groups related information under one name. One product, one order, one customer - they are all natural fits for objects.

objects.js

```
const product = {
  name: "Chocolate Cake",
  category: "Cake",
  price: 1500,
  inStock: true
};

console.log(product.name);      // "Chocolate Cake"
console.log(product["price"]);  // 1500  (alternative syntax)

// Update a field
product.price = 1600;

// Arrays of objects - extremely common
const orders = [
  { id: 1, customer: "Nimal", total: 750 },
  { id: 2, customer: "Saman", total: 1500 }
];

// Find an order by id
const target = orders.find(o => o.id === 2);
```

> **Tip**
> When you see **{ ... }** in JavaScript code, it is almost always an object. When you see **[...]**, it is an array. JSON - the format used to send data between client and server - is built from these same two shapes.

1.5 Async and await - waiting for data

Talking to a database or an API takes time. We do not want our program to freeze while waiting. JavaScript handles this with **async** functions and the **await** keyword. You will see this pattern dozens of times in our project.

async.js

```
// 'async' marks a function as one that may wait for things
async function loadProducts() {
  // 'await' pauses HERE until the data arrives
  const response = await fetch("http://localhost:5000/api/products");
  const data = await response.json();
  return data;
}

// Calling it
loadProducts().then(list => console.log(list));
```

1.6 Import and export - splitting code into files

Real projects have many files. We share code between them using **export** (make available) and **import** (use elsewhere).

math.js (one file)

```
// Export a function so other files can use it
export function calculateTax(amount) {
  return amount * 0.05;
}

export const TAX_RATE = 0.05;
```

orders.js (another file)

```
// Bring in what we need from math.js
import { calculateTax, TAX_RATE } from "./math.js";

const tax = calculateTax(1000);   // 50
```

PROJECT OVERVIEW

Part 2 - What We Are Building

Our finished application will manage the day-to-day operations of a small bakery. Think of it as the digital backbone behind the counter - what gets made, what gets sold, what runs low, and how much money came in.

2.1 Modules at a glance

Module	What it does	Main page
Dashboard	Shows daily orders, revenue, low stock alerts, sales trend	/dashboard
Products	Add, edit, delete bakery items; track price and stock	/products
Orders	Record customer orders with multiple items and totals	/orders
Inventory	Track raw ingredients (flour, sugar, butter) with min levels	/inventory
Login	Username and password, JWT-based session security	/login

2.2 What the finished app looks like

Here are clean previews of three of the main pages. The styling stays simple - Bootstrap defaults with light colours - so the focus is on functionality, not complicated CSS.

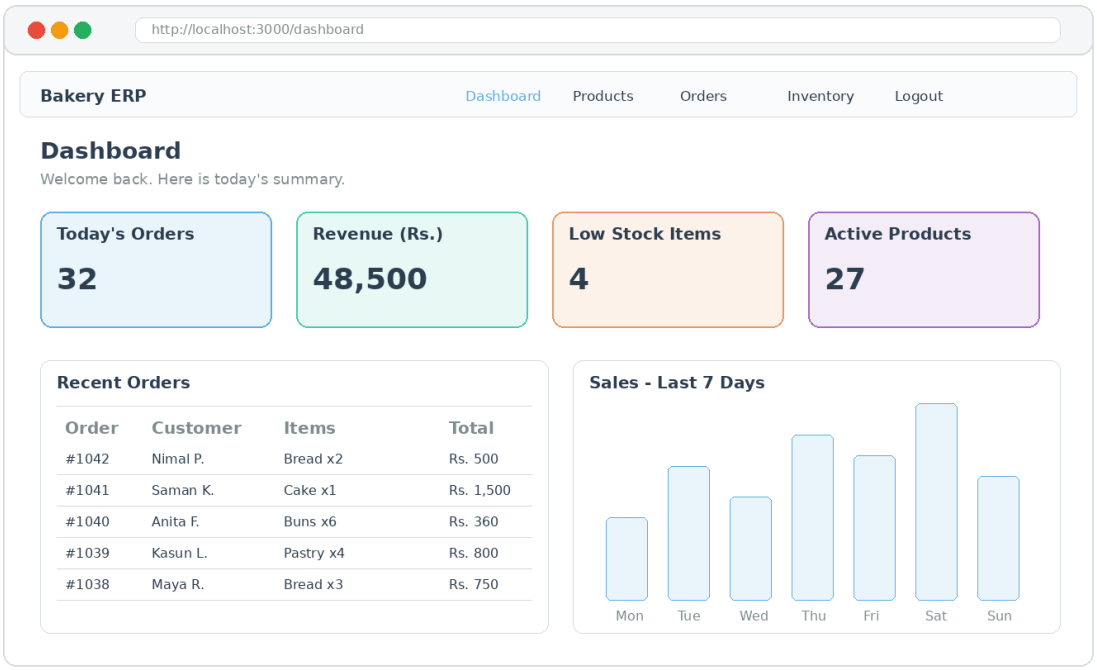

Figure 2.1 - Dashboard page with KPI cards and a 7-day sales bar chart

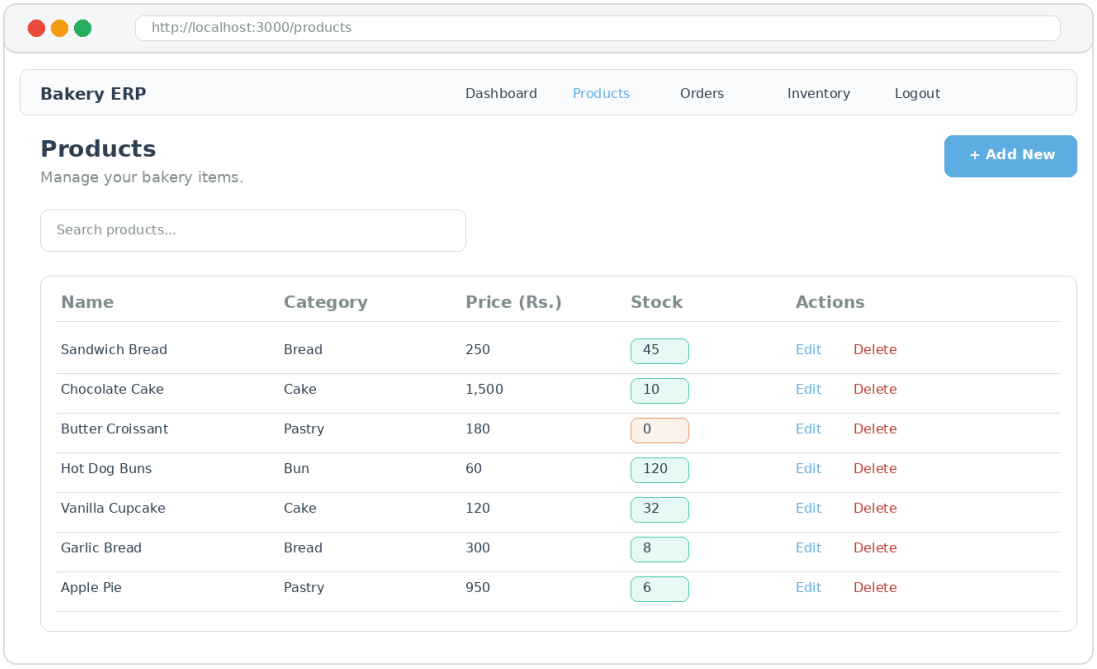

Figure 2.2 - Products page with searchable table and stock badges

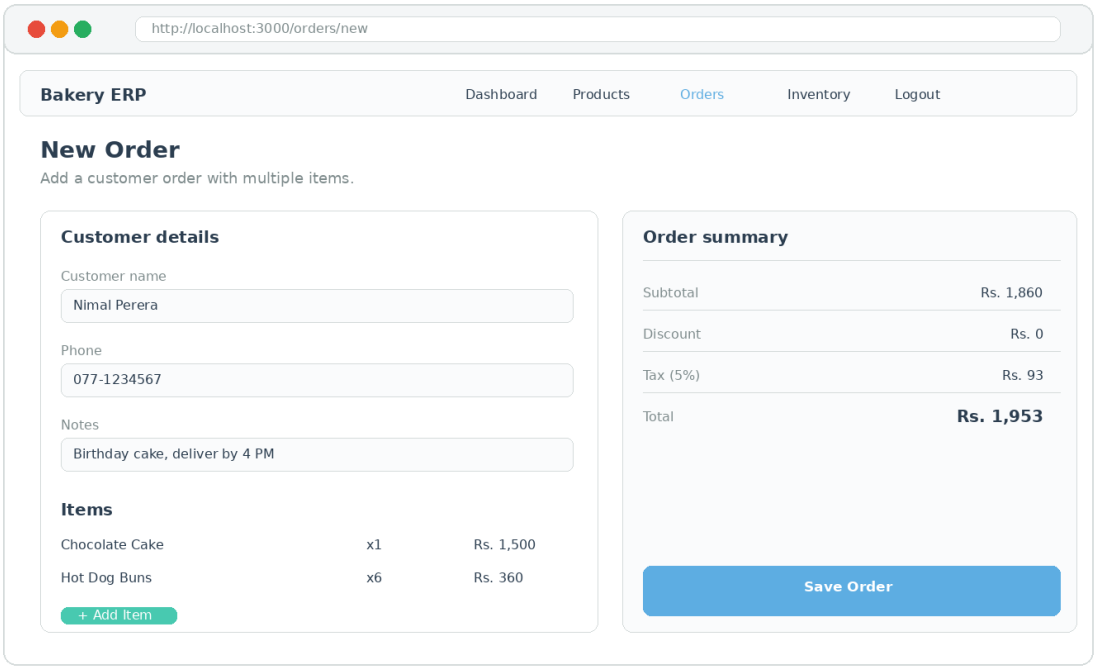

Figure 2.3 - New order form with live total summary on the right

ENVIRONMENT SETUP

Part 3 - Software Installation - Step by Step

Before writing code, we need three tools installed on your computer: **Node.js** (runs JavaScript on the server), **MySQL** (our database), and **VS Code** (the editor where we will write everything).

3.1 Install Node.js

Node.js lets us run JavaScript outside the browser. It also gives us **npm**, the tool that installs all the other packages we need.

1. Open your browser and go to **nodejs.org**.
2. Download the **LTS** (Long Term Support) version - it is the safer choice. In 2026 this is Node 20 or higher.
3. Run the installer. Click Next on every screen, accepting the defaults.
4. When it finishes, open a terminal and verify the install (commands below).

Terminal - check Node and npm

```
node --version
# Expected output:  v20.11.0  (or higher)

npm --version
# Expected output:  10.2.4  (or higher)
```

Important

If the commands above print **command not found**, the installer probably did not add Node to your system PATH. Restart your terminal first. If that fails, re-run the installer and check the box that says 'Add to PATH'.

3.2 Install MySQL

MySQL is where our bakery data will live. We will install it locally so the whole project can run on your machine without any cloud account. The official installer also includes **MySQL Workbench**, a graphical tool for browsing tables and running queries.

1. Go to **dev.mysql.com/downloads/installer** and download the MySQL Installer for your operating system. On macOS or Linux, use the DMG or apt/yum packages instead.
2. Run the installer and pick **Developer Default** - this gives you the server, Workbench, and the command line client all together.
3. When the wizard asks for a **root password**, choose a strong one and write it down. We will need it in a moment.
4. Accept the default port **3306** and the default authentication method (*Use Strong Password Encryption*).
5. Finish the wizard. The MySQL service will start automatically and run in the background each time you boot your computer.

> **Tip**
> On macOS the easiest install is **brew install mysql** followed by **brew services start mysql**. On Ubuntu use **sudo apt install mysql-server** and then **sudo mysql_secure_installation** to set a root password.

Verify MySQL is alive by opening a terminal and connecting:

Terminal - confirm MySQL works

```
mysql -u root -p
# It will ask for the password you set during install.
# After login you will see the prompt:  mysql>

# Try one quick command:
mysql> SHOW DATABASES;

# You should see a small list including:  information_schema, mysql, performa
nce_schema
# Type:  exit;   to leave the prompt.
```

3.3 Install VS Code (the editor)

Visual Studio Code is a free, fast editor with excellent JavaScript support. We will write every line of our project in it.

1. Go to **code.visualstudio.com** and download for your operating system.
2. Install with default options.
3. Open VS Code. On the left sidebar, click the **Extensions** icon.
4. Install these three helpful extensions: **ES7+ React snippets**, **Prettier**, and **SQLTools** (which can connect to MySQL right inside VS Code).

3.4 Final verification

Open VS Code. From the top menu choose **Terminal** → **New Terminal**. Run all four checks below. If each prints a version number, you are ready.

Terminal

```
node --version    # v20.x or higher
npm --version     # 10.x or higher
git --version     # any version is fine (optional but useful)
code --version    # confirms VS Code CLI is on your PATH
```

THE BIG PICTURE

Part 4 - System Architecture and How It All Connects

Modern web apps split work across three layers. Each layer has one job, and they talk to each other over well-defined channels. Understanding this picture now will make every later chapter feel obvious.

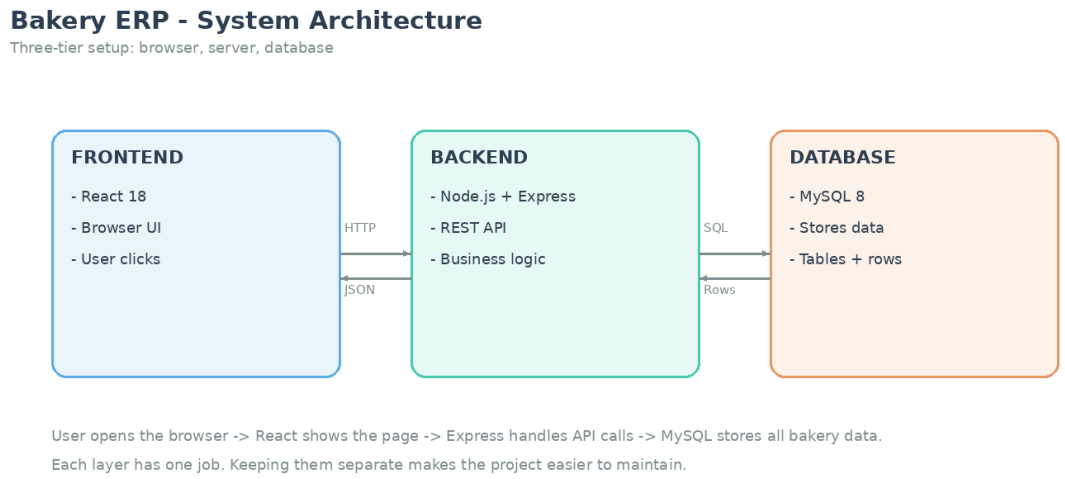

Figure 4.1 - Three layers: React (UI), Express (API), MySQL (data)

4.1 What each layer does

Frontend (React) runs in the user's browser. It draws the buttons, tables and forms. When you click *Save Order*, it does not save anything by itself - it sends a message to the backend.

Backend (Node + Express) runs on a server. It listens for messages from the frontend, runs the business logic (calculate totals, check stock, validate input), and talks to the database.

Database (MySQL) stores everything for the long term: products, orders, users, inventory levels. When the server is restarted the data is still there, sitting safely on disk in well-defined tables.

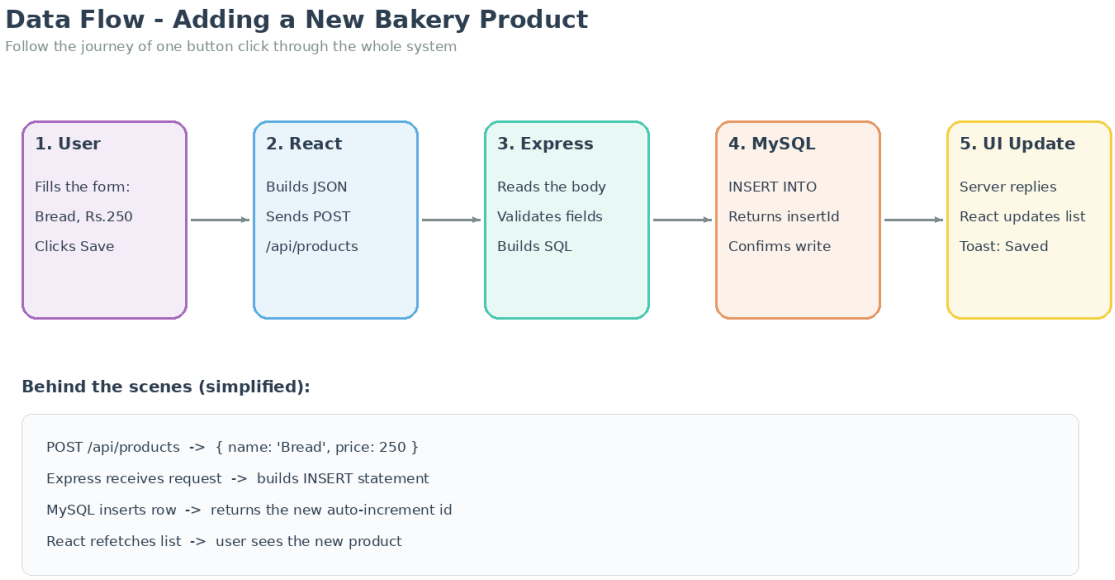

Figure 4.2 - One button click flowing through every layer

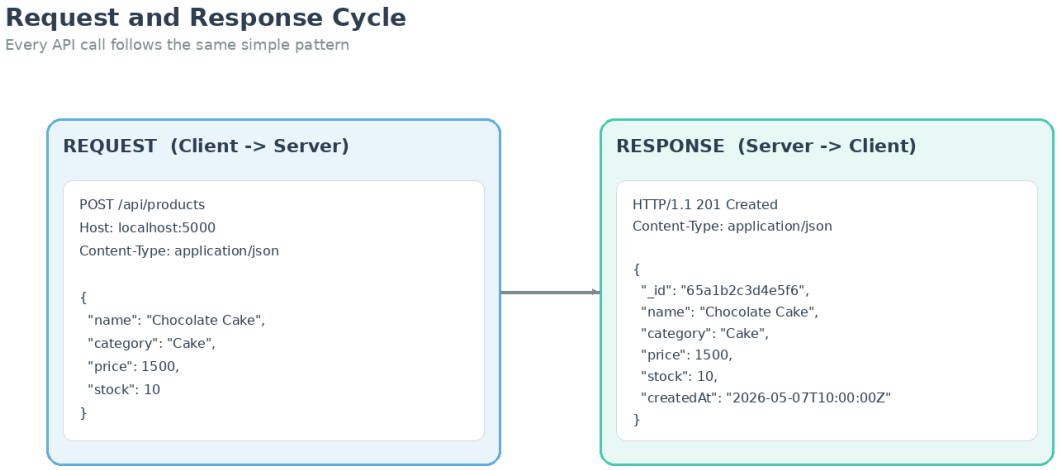

Figure 4.3 - Anatomy of an HTTP request and its response

> **Note**
> The frontend and backend are **two separate programs** running on different ports. React on port 3000, Express on port 5000. They communicate with each other over the network, even when both are running on your laptop.

PROJECT STRUCTURE

Part 5 - Setting Up the Project Folders

Now we create the actual folders on your computer. We will keep one parent folder called **bakery-erp** with two children inside it: **client** for the React app and **server** for the Express API.

Project Folder Structure

Two main folders: client (React) and server (Node + Express)

```
bakery-erp/
|
|-- client/                  -- React frontend (port 3000)
|   |-- public/
|   |   `-- index.html       -- root HTML file
|   |-- src/
|   |   |-- components/      -- small reusable pieces
|   |   |   |-- Navbar.js
|   |   |   `-- ProductCard.js
|   |   |-- pages/           -- one file per screen
|   |   |   |-- Dashboard.js
|   |   |   |-- Products.js
|   |   |   |-- Orders.js
|   |   |   `-- Inventory.js
|   |   |-- App.js           -- main React component
|   |   `-- index.js         -- entry point
|   `-- package.json
|
`-- server/                  -- Node + Express backend (5000)
    |-- db.js                -- MySQL connection pool
    |-- routes/              -- API endpoints
    |   |-- products.js
    |   |-- orders.js
    |   `-- inventory.js
    |-- sql/                 -- SQL files
    |   `-- schema.sql       -- CREATE TABLE statements
    |-- server.js            -- starts the Express app
    |-- .env                 -- secret values (never share)
```

Figure 5.1 - Final folder layout we are aiming for

5.1 Create the parent folder

Terminal

```
# Move to wherever you keep code projects
cd Documents

# Create and enter the parent folder
mkdir bakery-erp
cd bakery-erp
```

5.2 Create the React frontend

We use **Vite** to scaffold the React project. It is faster and simpler than older alternatives.

Terminal - inside bakery-erp/

```
npm create vite@latest client -- --template react

# When it finishes:
cd client
npm install
cd ..
```

> **Syntax**
> The double dash **--** separates arguments meant for npm from arguments meant for the create-vite tool itself. Everything after **--** is passed straight to Vite.

5.3 Create the Node + Express backend

Terminal - inside bakery-erp/

```
mkdir server
cd server

# Initialize a package.json file
npm init -y

# Install the libraries we will need
npm install express mysql2 cors dotenv bcryptjs jsonwebtoken

# Install nodemon - it auto-restarts the server when files change
npm install --save-dev nodemon
```

5.4 What each library does

Package	Purpose
express	Builds the REST API and listens for HTTP requests
mysql2	Talks to MySQL using the modern Promise-based driver
cors	Lets the React frontend (port 3000) talk to Express (port 5000)
dotenv	Loads secrets like the DB password from a .env file
bcryptjs	Hashes passwords before they go into the database
jsonwebtoken	Creates the login token (JWT) we use for sessions
nodemon	Restarts the server automatically when we save a file

DATA MODELING

Part 6 - Database Design with MySQL

Before we write any backend code, we need to decide what data the bakery produces and how to organise it. MySQL stores data in **tables** - rows and columns, just like a spreadsheet - and tables can reference each other through **foreign keys**.

Figure 6.1 - The five tables that hold every piece of data

6.1 Create the database

First we create an empty database and a dedicated user. Open a terminal and connect with the root account, then run the SQL below. We use a non-root user so that our app cannot accidentally drop other databases.

Terminal

```
mysql -u root -p
```

MySQL prompt - run inside mysql>

```
CREATE DATABASE bakery_db
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

CREATE USER 'bakery_user'@'localhost' IDENTIFIED BY 'StrongPassword123!';
GRANT ALL PRIVILEGES ON bakery_db.* TO 'bakery_user'@'localhost';
FLUSH PRIVILEGES;

USE bakery_db;
```

> **Syntax**
> **utf8mb4** is the modern Unicode character set - it stores any language and any emoji correctly. Older *utf8* in MySQL only handled 3-byte characters. Always pick utf8mb4 for new projects.

6.2 The schema file

We will keep all our CREATE TABLE statements in one file. This way anyone can rebuild the database with a single command. Save this as **server/sql/schema.sql**.

server/sql/schema.sql - products and inventory

```
-- Products: bread, cakes and other items the bakery sells
CREATE TABLE products (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  name        VARCHAR(120)   NOT NULL,
  category    VARCHAR(60)    NOT NULL,
  price       DECIMAL(10,2)  NOT NULL CHECK (price >= 0),
  stock       INT            NOT NULL DEFAULT 0,
  image_url   VARCHAR(255)   DEFAULT '',
  created_at  DATETIME       DEFAULT CURRENT_TIMESTAMP,
  updated_at  DATETIME       DEFAULT CURRENT_TIMESTAMP
                              ON UPDATE CURRENT_TIMESTAMP
);

-- Inventory: raw ingredients used in production
CREATE TABLE inventory (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  ingredient  VARCHAR(120)   NOT NULL UNIQUE,
  unit        VARCHAR(20)    NOT NULL DEFAULT 'kg',
  quantity    DECIMAL(10,2)  NOT NULL DEFAULT 0,
  min_level   DECIMAL(10,2)  NOT NULL DEFAULT 5,
  updated_at  DATETIME       DEFAULT CURRENT_TIMESTAMP
                              ON UPDATE CURRENT_TIMESTAMP
);
```

Syntax

AUTO_INCREMENT tells MySQL to pick the next integer for us when we insert a row, so we never repeat ids. **DECIMAL(10,2)** means up to 10 digits with 2 after the decimal point - perfect for money. **ON UPDATE CURRENT_TIMESTAMP** automatically refreshes updated_at every time the row changes.

6.3 Orders and order_items (with foreign keys)

An order has many items, so we use two tables linked by a foreign key. The **order_items** table stores one row per line in the order. **ON DELETE CASCADE** makes MySQL remove the items automatically if we delete the parent order.

server/sql/schema.sql - orders + items

```
CREATE TABLE orders (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  customer    VARCHAR(120)   NOT NULL,
  phone       VARCHAR(30),
  total       DECIMAL(10,2)  NOT NULL DEFAULT 0,
  status      ENUM('pending','completed','cancelled')
                              NOT NULL DEFAULT 'pending',
  created_at  DATETIME       DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE order_items (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  order_id    INT            NOT NULL,
  product_id  INT            NOT NULL,
  quantity    INT            NOT NULL DEFAULT 1,
  unit_price  DECIMAL(10,2)  NOT NULL,
  subtotal    DECIMAL(10,2)  NOT NULL,
  FOREIGN KEY (order_id)   REFERENCES orders(id)   ON DELETE CASCADE,
  FOREIGN KEY (product_id) REFERENCES products(id)
);
```

6.4 Users table

server/sql/schema.sql - users

```
CREATE TABLE users (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  username    VARCHAR(60)    NOT NULL UNIQUE,
  password    CHAR(60)       NOT NULL,
  role        ENUM('admin','staff') NOT NULL DEFAULT 'staff',
  email       VARCHAR(120),
  created_at  DATETIME       DEFAULT CURRENT_TIMESTAMP
);
```

> **Note**
> **CHAR(60)** is exactly the right size for a bcrypt password hash. We never store plain passwords. **UNIQUE** on the username column tells MySQL to refuse duplicate usernames at the database level.

6.5 Run the schema and seed some data

From the project root, load the schema in one command and add a couple of demo rows so the screens are not empty during testing.

Terminal

```
# Loads every CREATE TABLE statement
mysql -u bakery_user -p bakery_db < server/sql/schema.sql
```

MySQL prompt - quick seed

```
USE bakery_db;

INSERT INTO products (name, category, price, stock) VALUES
  ('White Bread',     'Bread',  150.00, 30),
  ('Chocolate Cake',  'Cake',  1500.00,  8),
  ('Hot Dog Bun',     'Bread',   60.00, 80),
  ('Butter Cookies',  'Snack',  220.00, 50);

INSERT INTO inventory (ingredient, unit, quantity, min_level) VALUES
  ('Flour', 'kg', 25, 10),
  ('Sugar', 'kg', 12,  5),
  ('Butter','kg',  8,  3);
```

THE API

Part 7 - Building the Backend (Node + Express)

Time to write the server that the React app will talk to. Express makes this surprisingly small - the entire main file fits on one screen.

7.1 The .env file (secrets)

Never put passwords or connection strings directly in your code. Put them in a **.env** file and load them at runtime.

server/.env

```
PORT=5000
DB_HOST=localhost
DB_PORT=3306
DB_USER=bakery_user
DB_PASSWORD=StrongPassword123!
DB_NAME=bakery_db
JWT_SECRET=change-this-to-a-long-random-string
```

Important
Add a file called **.gitignore** with the line **.env** inside it. This stops your secret keys from ever being uploaded to GitHub.

7.2 The MySQL connection pool

We will create the connection in one place and reuse it from every route. A **pool** keeps a small group of connections open so we never wait for a fresh handshake on each request.

server/db.js

```
import mysql from "mysql2/promise";
import dotenv from "dotenv";

dotenv.config();

// Create the pool once. Routes import this and call pool.query(...).
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,
  queueLimit: 0
});

export default pool;
```

Syntax

mysql2/promise gives us **async/await** support out of the box - no callback nesting required. **connectionLimit: 10** caps the pool at 10 connections, which is plenty for a small bakery app.

7.3 The main server file

server/server.js

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import pool from "./db.js";

import productRoutes   from "./routes/products.js";
import orderRoutes     from "./routes/orders.js";
import inventoryRoutes from "./routes/inventory.js";
import authRoutes      from "./routes/auth.js";

dotenv.config();

const app = express();

// --- Middleware (run for every request) ---
app.use(cors());                    // allow the React app to call us
app.use(express.json());            // parse JSON bodies into req.body

// --- Routes (group endpoints by topic) ---
app.use("/api/products",  productRoutes);
app.use("/api/orders",    orderRoutes);
app.use("/api/inventory", inventoryRoutes);
app.use("/api/auth",      authRoutes);

// --- Verify the DB pool, then start listening ---
const PORT = process.env.PORT || 5000;

pool.query("SELECT 1")
  .then(() => {
    app.listen(PORT, () => {
      console.log(`Server running on http://localhost:${PORT}`);
    });
  })
  .catch(err => console.error("DB connection failed:", err));
```

7.4 Tell Node we are using ES modules

Open **server/package.json** and add the line **"type": "module"** at the top level. This lets us use **import** instead of the older **require()** syntax.

server/package.json (snippet)

```
{
  "name": "server",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "node server.js",
    "dev":   "nodemon server.js"
  },
  ...
}
```

7.5 Building the products route

Each route file groups all endpoints for one resource. Below is the products route - it supports **GET**, **POST**, **PUT**, and **DELETE**. We use **parameterised queries** (the **?** placeholders) so user input can never be turned into malicious SQL.

server/routes/products.js

```
import express from "express";
import pool from "../db.js";

const router = express.Router();

// GET /api/products  -- list all products
router.get("/", async (req, res) => {
  const [rows] = await pool.query(
    "SELECT * FROM products ORDER BY created_at DESC"
  );
  res.json(rows);
});

// GET /api/products/:id  -- one product by id
router.get("/:id", async (req, res) => {
  const [rows] = await pool.query(
    "SELECT * FROM products WHERE id = ?", [req.params.id]
  );
  if (rows.length === 0)
    return res.status(404).json({ error: "Not found" });
  res.json(rows[0]);
});

// POST /api/products  -- create new product
router.post("/", async (req, res) => {
  try {
    const { name, category, price, stock = 0 } = req.body;
    const [result] = await pool.query(
      `INSERT INTO products (name, category, price, stock)
       VALUES (?, ?, ?, ?)`,
      [name, category, price, stock]
    );
    res.status(201).json({ id: result.insertId, name, category, price, stock
});
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```

```
// PUT /api/products/:id  -- update product
router.put("/:id", async (req, res) => {
  const { name, category, price, stock } = req.body;
  await pool.query(
    `UPDATE products
     SET name = ?, category = ?, price = ?, stock = ?
     WHERE id = ?`,
    [name, category, price, stock, req.params.id]
  );
  res.json({ id: Number(req.params.id), name, category, price, stock });
});

// DELETE /api/products/:id
router.delete("/:id", async (req, res) => {
  await pool.query("DELETE FROM products WHERE id = ?", [req.params.id]);
  res.json({ ok: true });
});

export default router;
```

Important

Always pass values through the **?** placeholder array, never string-concatenate them into the SQL. The mysql2 driver escapes each value safely - this is your main defence against SQL injection. Never write **`SELECT ... WHERE id=${id}`**.

7.6 Quick test with the browser

Run **npm run dev** in the server folder. Open **http://localhost:5000/api/products** in your browser. You will see the four seed rows from the previous chapter as JSON. If you do, the whole stack - Express, the pool, MySQL - is wired up correctly.

THE USER INTERFACE

Part 8 - Building the Frontend (React)

Now the fun part - the screens the user actually sees. We will keep CSS to a minimum and lean on Bootstrap defaults so we focus on the JavaScript logic, not styling debates.

8.1 Add Bootstrap and a router

Terminal - inside client/

```
npm install react-router-dom axios bootstrap recharts
```

8.2 Wire Bootstrap into the entry file

client/src/main.jsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import App from "./App.jsx";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

8.3 The App component (routes)

client/src/App.jsx

```
import { Routes, Route } from "react-router-dom";
import Navbar     from "./components/Navbar";
import Dashboard  from "./pages/Dashboard";
import Products   from "./pages/Products";
import Orders     from "./pages/Orders";
import Inventory  from "./pages/Inventory";
import Login      from "./pages/Login";

export default function App() {
  return (
    <>
      <Navbar />
      <div className="container py-4">
        <Routes>
          <Route path="/"          element={<Dashboard />} />
          <Route path="/products"  element={<Products />} />
          <Route path="/orders"    element={<Orders />} />
          <Route path="/inventory" element={<Inventory />} />
          <Route path="/login"     element={<Login />} />
        </Routes>
      </div>
    </>
  );
}
```

Syntax

JSX is the HTML-like syntax inside React. The angle brackets are real - they get compiled to function calls. The fragment **<>...</>** lets you return multiple elements without an extra wrapper div.

8.4 The Navbar component

client/src/components/Navbar.jsx

```
import { Link, NavLink } from "react-router-dom";

export default function Navbar() {
  return (
    <nav className="navbar navbar-expand-lg navbar-light bg-light border-bott
om">
      <div className="container">
        <Link to="/" className="navbar-brand fw-bold">Bakery ERP</Link>
        <div className="d-flex gap-3">
          <NavLink to="/"          className="nav-link">Dashboard</NavLink>
          <NavLink to="/products"  className="nav-link">Products</NavLink>
          <NavLink to="/orders"    className="nav-link">Orders</NavLink>
          <NavLink to="/inventory" className="nav-link">Inventory</NavLink>
        </div>
      </div>
    </nav>
  );
}
```

8.5 A reusable API helper

Rather than typing the API base URL in every page, we centralise it once.

client/src/api.js

```
import axios from "axios";

const api = axios.create({
  baseURL: "http://localhost:5000/api"
});

// Attach JWT token to every request if we have one
api.interceptors.request.use(config => {
  const token = localStorage.getItem("token");
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});

export default api;
```

8.6 The Products page

This page lists products, lets the user add new ones, and deletes existing ones. It demonstrates the three React skills you will use everywhere: **useState** for data, **useEffect** for loading, and event handlers for buttons and forms.

client/src/pages/Products.jsx

```
import { useEffect, useState } from "react";
import api from "../api";

export default function Products() {
  const [items, setItems] = useState([]);
  const [form, setForm]   = useState({
    name: "", category: "", price: "", stock: ""
  });

  // Load products once when the page first opens
  useEffect(() => { loadProducts(); }, []);

  async function loadProducts() {
    const res = await api.get("/products");
    setItems(res.data);
  }

  function handleChange(e) {
    setForm({ ...form, [e.target.name]: e.target.value });
  }

  async function handleAdd(e) {
    e.preventDefault();
    await api.post("/products", form);
    setForm({ name: "", category: "", price: "", stock: "" });
    loadProducts();
  }

  async function handleDelete(id) {
    if (!confirm("Delete this product?")) return;
    await api.delete(`/products/${id}`);
    loadProducts();
  }

  return (
    <div>
      <h2 className="mb-4">Products</h2>
```

```
      <form onSubmit={handleAdd} className="row g-2 mb-4">
        <div className="col">
          <input className="form-control" name="name"
                 placeholder="Name" value={form.name} onChange={handleChange}
 />
        </div>
        <div className="col">
          <input className="form-control" name="category"
                 placeholder="Category" value={form.category} onChange={handl
eChange} />
        </div>
        <div className="col">
          <input className="form-control" name="price" type="number"
                 placeholder="Price" value={form.price} onChange={handleChang
e} />
        </div>
        <div className="col">
          <input className="form-control" name="stock" type="number"
                 placeholder="Stock" value={form.stock} onChange={handleChang
e} />
        </div>
        <div className="col-auto">
          <button className="btn btn-primary">Add</button>
        </div>
      </form>

      <table className="table">
        <thead>
          <tr>
            <th>Name</th><th>Category</th><th>Price</th>
            <th>Stock</th><th></th>
          </tr>
        </thead>
        <tbody>
          {items.map(p => (
            <tr key={p.id}>
              <td>{p.name}</td>
              <td>{p.category}</td>
              <td>Rs. {p.price}</td>
              <td>{p.stock}</td>
              <td>
                <button className="btn btn-sm btn-outline-danger"
```

```
                      onClick={() => handleDelete(p.id)}>
                  Delete
                </button>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}
```

Syntax

useState creates a piece of data the component remembers. **useEffect** with an empty array **[]** runs once on mount - perfect for loading data. **{ ...form, [e.target.name]: e.target.value }** is the spread operator, which copies the existing form and overrides one field.

AUTHENTICATION

Part 9 - Login System with JWT

We do not want strangers to manage our products. The classic solution is a **JSON Web Token** (JWT) - a small signed string the server gives the client at login. The client sends it back with every request as proof of identity.

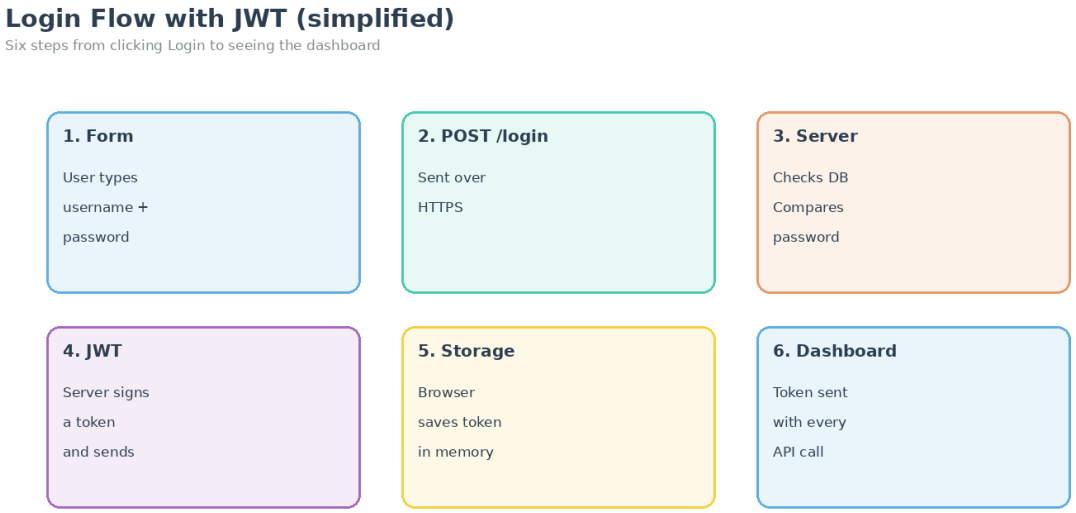

Figure 9.1 - Six-step JWT login flow

9.1 Auth route on the server

server/routes/auth.js

```
import express from "express";
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import pool from "../db.js";

const router = express.Router();

// Register a new user
router.post("/register", async (req, res) => {
  try {
    const { username, password, role = "staff" } = req.body;
    const hash = await bcrypt.hash(password, 10);
    const [result] = await pool.query(
      `INSERT INTO users (username, password, role)
       VALUES (?, ?, ?)`,
      [username, hash, role]
    );
    res.status(201).json({ id: result.insertId, username });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// Login - returns a JWT
router.post("/login", async (req, res) => {
  const { username, password } = req.body;
  const [rows] = await pool.query(
    "SELECT id, username, password, role FROM users WHERE username = ?",
    [username]
  );
  const user = rows[0];
  if (!user) return res.status(401).json({ error: "Invalid credentials" });

  const valid = await bcrypt.compare(password, user.password);
  if (!valid) return res.status(401).json({ error: "Invalid credentials" });
```

```
  const token = jwt.sign(
    { id: user.id, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: "8h" }
  );
  res.json({ token, role: user.role });
});

export default router;
```

9.2 The Login page in React

client/src/pages/Login.jsx

```
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import api from "../api";

export default function Login() {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");
  const navigate = useNavigate();

  async function handleSubmit(e) {
    e.preventDefault();
    try {
      const res = await api.post("/auth/login", { username, password });
      localStorage.setItem("token", res.data.token);
      navigate("/");
    } catch (err) {
      setError("Wrong username or password");
    }
  }

  return (
    <div className="row justify-content-center">
      <div className="col-md-5">
        <h3 className="mb-4">Sign in</h3>
        <form onSubmit={handleSubmit}>
          <div className="mb-3">
            <label className="form-label">Username</label>
            <input className="form-control" value={username}
                   onChange={e => setUsername(e.target.value)} />
          </div>
          <div className="mb-3">
            <label className="form-label">Password</label>
            <input className="form-control" type="password" value={password}
                   onChange={e => setPassword(e.target.value)} />
          </div>
          {error && <div className="alert alert-danger">{error}</div>}
          <button className="btn btn-primary w-100">Login</button>
        </form>
      </div>
    </div>
  );
}
```

Tip

Storing the token in **localStorage** is fine for a learning project. For a production system handling real money, switch to HTTP-only cookies which are safer against cross-site scripting attacks.

DATA VISUALIZATION

Part 10 - Dashboard with Charts

Numbers in tables are useful, but a chart turns numbers into a story. We use **Recharts** - a React chart library that pairs well with Bootstrap and needs almost no configuration.

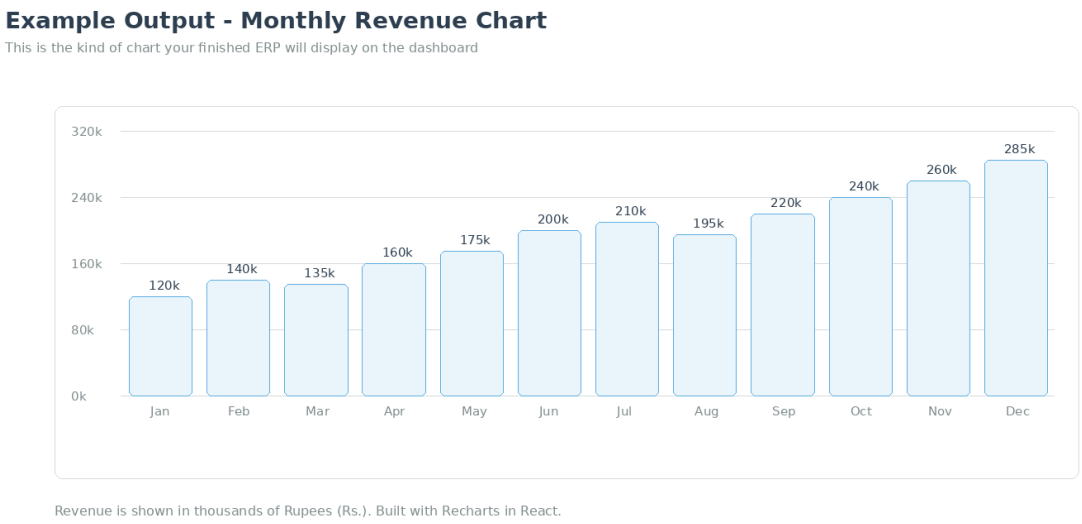

Figure 10.1 - Example monthly revenue chart on the dashboard

10.1 The Dashboard component

client/src/pages/Dashboard.jsx

```
import { useEffect, useState } from "react";
import { BarChart, Bar, XAxis, YAxis, Tooltip,
         ResponsiveContainer } from "recharts";
import api from "../api";

export default function Dashboard() {
  const [stats, setStats] = useState({
    todayOrders: 0, revenue: 0, lowStock: 0, products: 0
  });
  const [sales, setSales] = useState([]);

  useEffect(() => {
    api.get("/orders/stats").then(r => setStats(r.data));
    api.get("/orders/weekly").then(r => setSales(r.data));
  }, []);

  return (
    <div>
      <h2 className="mb-4">Dashboard</h2>

      <div className="row g-3 mb-4">
        <Card title="Today's Orders" value={stats.todayOrders} />
        <Card title="Revenue (Rs.)"  value={stats.revenue.toLocaleString()} /
>
        <Card title="Low Stock"      value={stats.lowStock} />
        <Card title="Products"       value={stats.products} />
      </div>

      <div className="card">
        <div className="card-body">
          <h5 className="card-title mb-3">Sales - Last 7 Days</h5>
          <ResponsiveContainer width="100%" height={280}>
            <BarChart data={sales}>
              <XAxis dataKey="day" />
              <YAxis />
              <Tooltip />
              <Bar dataKey="total" fill="#5DADE2" radius={[6,6,0,0]} />
            </BarChart>
          </ResponsiveContainer>
```

```
        </div>
      </div>
    </div>
  );
}

function Card({ title, value }) {
  return (
    <div className="col-md-3">
      <div className="card h-100">
        <div className="card-body">
          <div className="text-muted small">{title}</div>
          <div className="fs-2 fw-bold">{value}</div>
        </div>
      </div>
    </div>
  );
}
```

10.2 Backend endpoints that feed the chart

server/routes/orders.js (extra endpoints)

```
// GET /api/orders/stats  -- summary numbers for the cards
router.get("/stats", async (req, res) => {
  const [[{ todayOrders }]] = await pool.query(
    `SELECT COUNT(*) AS todayOrders
     FROM orders
     WHERE DATE(created_at) = CURDATE()`
  );
  const [[{ revenue }]] = await pool.query(
    `SELECT COALESCE(SUM(total), 0) AS revenue
     FROM orders
     WHERE DATE(created_at) = CURDATE()`
  );
  const [[{ lowStock }]] = await pool.query(
    `SELECT COUNT(*) AS lowStock FROM products WHERE stock < 5`
  );
  const [[{ products }]] = await pool.query(
    `SELECT COUNT(*) AS products FROM products`
  );

  res.json({ todayOrders, revenue, lowStock, products });
});

// GET /api/orders/weekly  -- one bar per day for the last 7 days
router.get("/weekly", async (req, res) => {
  const [rows] = await pool.query(
    `SELECT
       DATE(created_at)              AS d,
       DAYNAME(created_at)           AS day_name,
       COALESCE(SUM(total), 0)       AS total
     FROM orders
     WHERE created_at >= CURDATE() - INTERVAL 6 DAY
     GROUP BY DATE(created_at), DAYNAME(created_at)
     ORDER BY d ASC`
  );
```

```
  // Fill missing days with zero so the chart always shows 7 bars
  const days = [];
  for (let i = 6; i >= 0; i--) {
    const d = new Date(); d.setDate(d.getDate() - i);
    const iso = d.toISOString().slice(0, 10);
    const hit = rows.find(r => r.d.toISOString().slice(0, 10) === iso);
    days.push({
      day:   d.toLocaleDateString("en", { weekday: "short" }),
      total: hit ? Number(hit.total) : 0
    });
  }
  res.json(days);
});
```

Syntax

CURDATE() is the SQL function for today's date. **COALESCE(x, 0)** returns 0 when the sum is NULL (no rows). The destructuring **[[{ revenue }]]** reads: *first row of the first result set* - the mysql2 driver returns **[rows, fields]**, and pool.query gives one row here.

LAUNCH

Part 11 - Testing and Running the App

Time to see everything work together. You will need **two terminal windows** open at the same time - one for the backend, one for the frontend.

11.1 Terminal 1 - start the backend

Terminal 1

```
cd bakery-erp/server
npm run dev

# Expected output:
# Server running on http://localhost:5000
```

11.2 Terminal 2 - start the frontend

Terminal 2

```
cd bakery-erp/client
npm run dev

# Expected output:
# Local:  http://localhost:3000
```

11.3 Try it in the browser

1. Open **http://localhost:3000** in your browser.
2. You should see the Bakery ERP dashboard with empty stats.
3. Go to **Products** and add a few items - Bread, Cake, Buns.
4. Refresh the page. The items are still there because they live in MySQL.
5. Open the dashboard again - the product count card should update.

11.4 Common problems and fixes

Symptom	Likely cause	Fix
CORS error in console	cors not enabled	Confirm **app.use(cors())** in server.js
DB connection failed	Wrong DB user or password	Edit .env and restart the server
ER_ACCESS_DENIED_ERROR	MySQL user has no rights	Re-run the GRANT statement from chapter 6
Cannot find module ...	npm install was skipped	Run **npm install** in that folder
Page is blank, white screen	Routing typo	Check your route paths match the URLs
Token errors after login	JWT_SECRET missing	Add it to .env and restart

GOING FURTHER

Part 12 - What to Build Next

You now have a working bakery ERP with all the basics: data model, REST API, user interface, login, and a chart. The skills you used here transfer to almost any business application. Here are good next steps to keep growing.

12.1 Features to add

- **Receipt printing** - generate a PDF of each order using a library like jsPDF.
- **Barcode scanning** - read product barcodes through the device camera.
- **Customer accounts** - track repeat buyers and reward loyalty.
- **Supplier management** - record where each ingredient comes from.
- **Daily closing reports** - email a sales summary at end of day.
- **Mobile responsive layout** - already free with Bootstrap, just review on a phone.
- **Image upload** - product photos using multer on the server.

12.2 Skills to deepen

- **Testing** - learn Jest for backend and React Testing Library for frontend.
- **Deployment** - host the backend on Render or Railway, frontend on Netlify or Vercel.
- **TypeScript** - add types to catch bugs at write-time.
- **Real-time** - push live order updates with Socket.IO.
- **Authentication** - add Google OAuth so staff can log in with their work email.
- **Advanced SQL** - learn JOINs, subqueries, indexes, and transactions to make queries fast and reliable.

Congratulations

You have built a complete full-stack web application from scratch. The same architecture - React + Express + MySQL - powers thousands of real-world apps. Keep building, keep breaking things, keep learning.

egotechworld.com