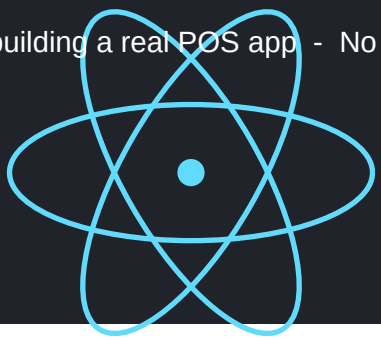
Pizza Shop Billing System

Built with React 19

A Beginner's Step-by-Step Tutorial

Learn React by building a real POS app - No backend needed



What you will build and learn

A working point-of-sale billing system for a pizza shop.
Add pizzas to a live cart with quantity controls.
Calculate subtotal, tax, and total automatically.
Generate printable receipts with all customer details.
Browse sales history, all stored in the browser.
Learn components, props, useState, and lifting state up.
Learn React Router for multi-page apps.
Learn localStorage so your data survives page refresh.

egotechworld.com

Free coding tutorials, project source code, and AI tools

Beginner-Friendly - Step by Step - 2026 Edition

Table of Contents

Chapter 1 - Why React and What You Will Build

Welcome! In this tutorial you will build a complete **Pizza Shop Billing System** using **React 19**, the most popular JavaScript library for building user interfaces in 2026. By the end you will have a working point-of-sale app that runs entirely in the browser - no backend, no database server. Just React, your laptop, and a few files.

What is React, in one sentence?

React is a JavaScript library that lets you build a webpage by splitting it into small, reusable pieces called **components**. Each component is a chunk of HTML and JavaScript glued together. When data changes, React updates only the parts of the page that actually changed - which is what makes React apps feel so fast and responsive.

Why React?

- **Components** - build once, reuse everywhere. Like Lego blocks for HTML.
- **Reactive UI** - change the data, React updates the screen. No manual DOM work.
- **Massive job market** - React is the most-requested frontend skill in 2026.
- **Huge ecosystem** - thousands of free packages for routing, charts, forms, and more.
- **Backed by Meta** - actively developed since 2013, used by Facebook, Instagram, Netflix, Airbnb.
- **Works on web and mobile** - the same skills transfer to React Native for iOS and Android apps.

Who this tutorial is for

- You know basic HTML, CSS, and a little JavaScript.
- You have heard of React but never built anything with it.
- You learn best by building real, useful things - not toy examples.
- You want a portfolio project that looks professional.

> **TIP**
>
> If you only know HTML and CSS so far, take an hour first to read about JavaScript fundamentals - variables, functions, arrays, and arrow functions. We have a free JavaScript primer at egotechworld.com.

What you will build

A real point-of-sale (POS) billing app for a pizza shop. The cashier uses it to take orders and print receipts. Here is exactly what it will do:

- Show a menu of pizzas with prices, in a tappable grid.
- Build up an order in a live cart sidebar.
- Increase, decrease, or remove items from the cart.
- Calculate the subtotal, tax (10 percent), and grand total automatically.
- Capture customer name, phone number, and table number.
- Generate a printable receipt with a unique bill number.
- Save every bill in the browser so the cashier can review sales history.
- Show daily stats: number of bills, revenue, average bill size.

What you will learn along the way

- How to create a React project with **Vite** (the modern, fast way).
- How to write **components** and use **JSX**.
- How to pass data with **props**.
- How to manage data that changes with **useState**.
- The fundamental rule of React: **data flows down, events flow up**.
- How to render lists from arrays.
- How to handle clicks, form input, and keyboard events.
- How to navigate between pages with **React Router**.
- How to save data in the browser with **localStorage**.
- How to deploy your finished app to the internet for free.

What we are NOT covering

To keep this tutorial focused and beginner-friendly, we are leaving a few advanced topics for later. You don't need any of these to ship a useful app:

- Backend APIs (we use no server - all data lives in the browser).
- TypeScript (plain JavaScript only - one less thing to learn).
- Redux, Zustand, or other state libraries (useState is enough).
- Server-side rendering, Next.js, Remix (we build a plain SPA).
- Custom hooks beyond what React gives us out of the box.

> **NOTE**
>
> All source code from this tutorial will be available at **egotechworld.com** in the projects section. Bookmark it - if you get stuck, compare your code to the reference there.

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 - they are your destination.

1. Empty cart - the starting screen

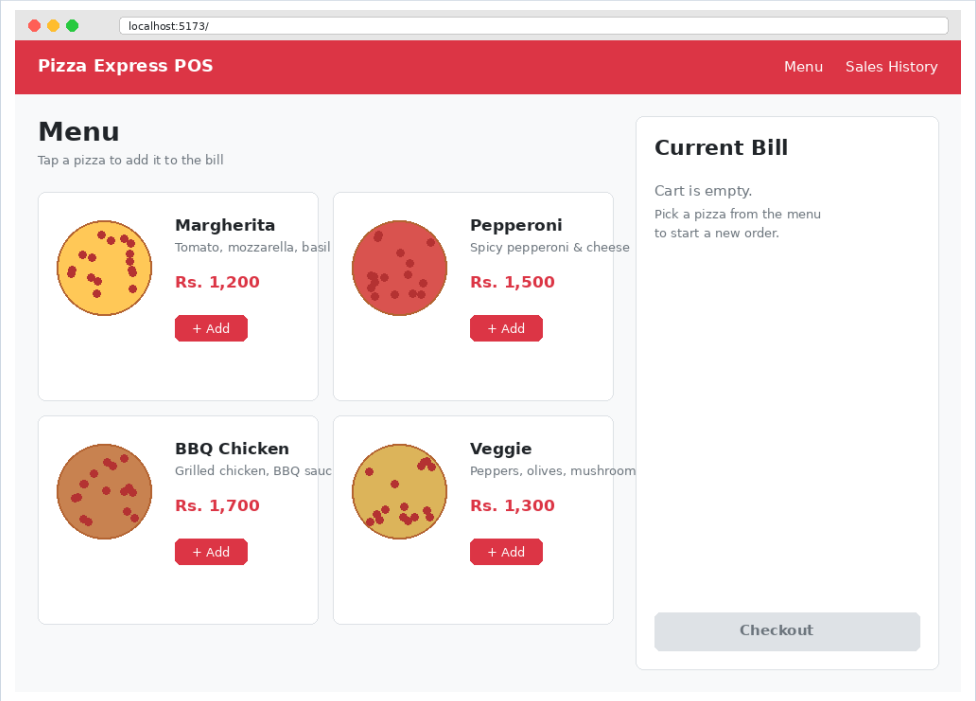


Figure 1 - The home page when no items have been added yet.

2. Menu with items in the cart

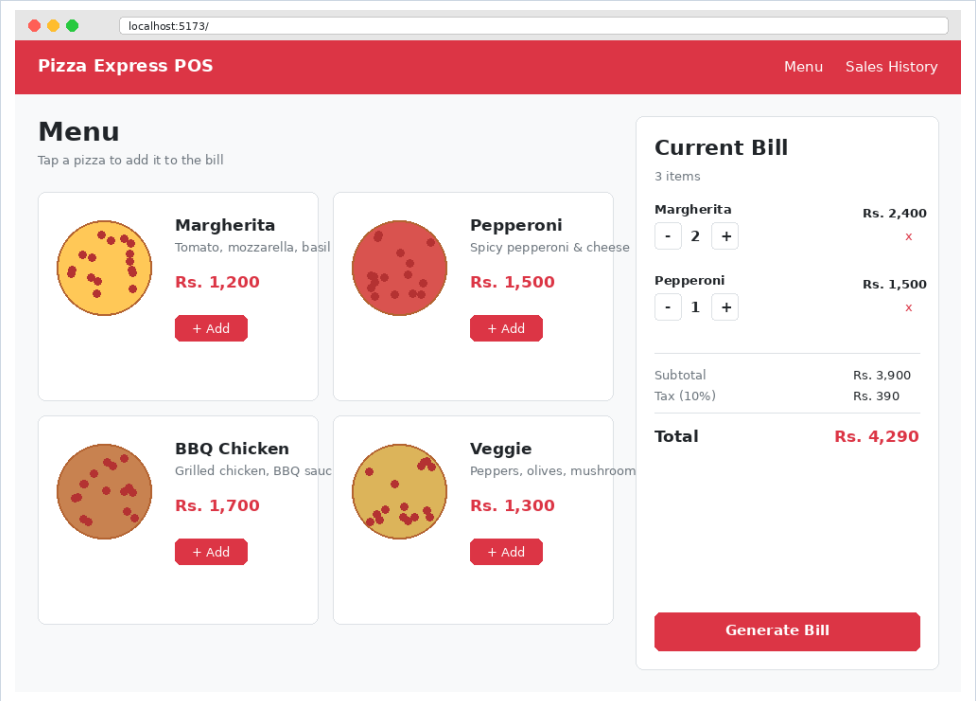


Figure 2 - As pizzas are added, the cart sidebar fills up with quantity controls, running totals, and a Generate Bill button.

3. Checkout - capture customer details

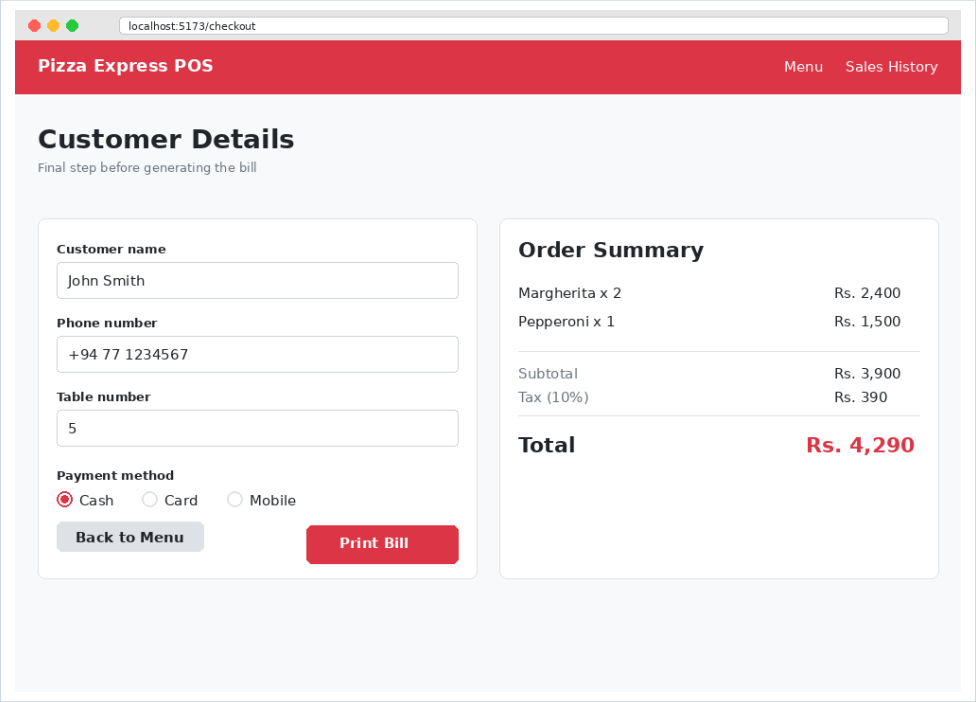


Figure 3 - The customer details form before generating the bill. The right side shows a live order summary.

4. The printable bill / receipt

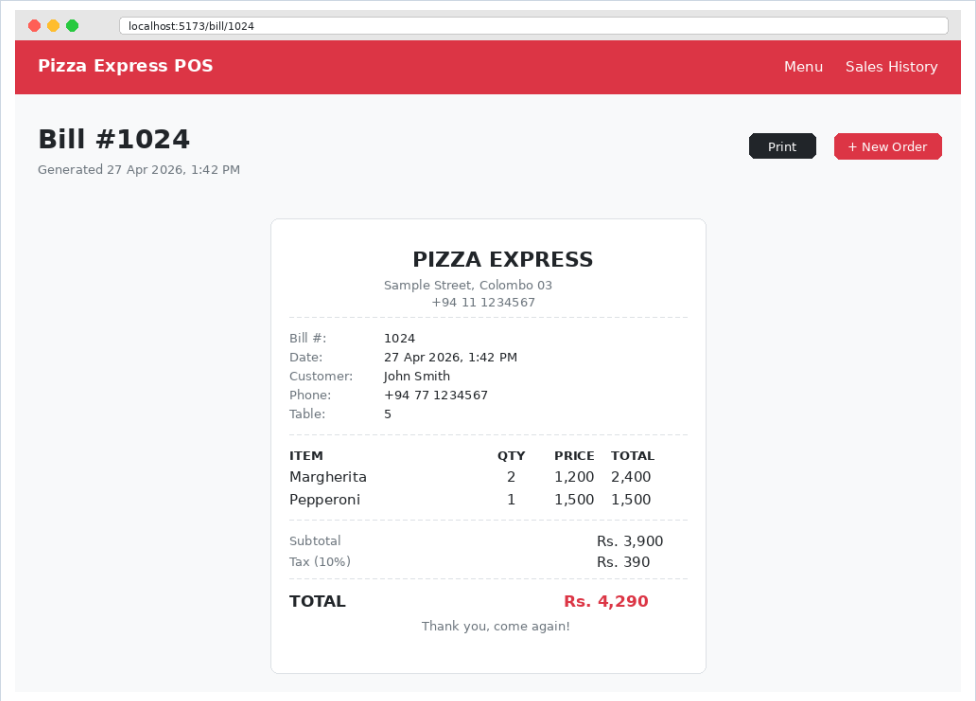


Figure 4 - The generated receipt with all customer info, itemised list, totals, and a Print button. Looks like a real POS receipt.

5. Sales history dashboard

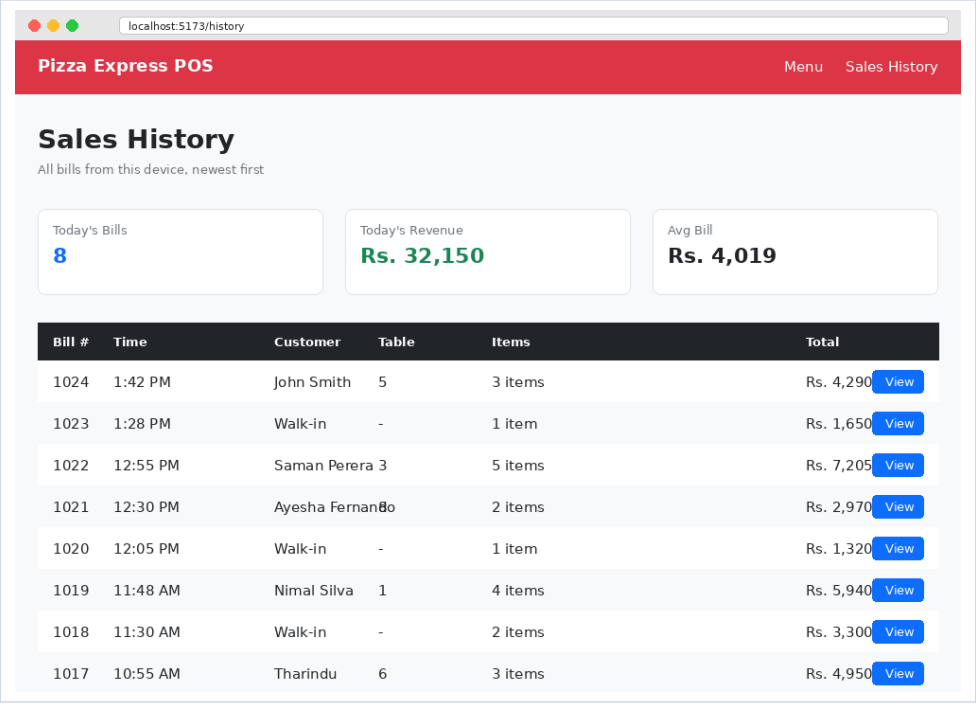


Figure 5 - All past bills stored in the browser, with daily stats at the top and a View button to re-open any bill.

Chapter 2 - How React Thinks - Components, Props, State

Before we install anything, let us learn the three big ideas behind React. Once these click, every React project you ever read will start to make sense.

Idea 1 - Components

A component is a chunk of UI written as a JavaScript function. It returns the HTML that should appear on screen. Here is the smallest possible React component:

```
function Welcome() {
    return <h1>Hello from React!</h1>;
}
```

That is it. A function that returns HTML-like markup. The markup language inside the function is called **JSX** - it looks like HTML but is actually JavaScript. We will see lots more of it.

Why components are amazing

- Build once, reuse everywhere. Need 100 pizza cards? Write the component once, render it 100 times.
- Each component is a small, focused file. Easier to read, debug, and test.
- Components can contain other components. The whole app becomes a tree of nested components - like Russian dolls.

Idea 2 - Props

Props (short for *properties*) are how you pass data INTO a component. They look exactly like HTML attributes:

```
// Define the component to accept a 'name' prop:
function Welcome(props) {
    return <h1>Hello, {props.name}!</h1>;
}

// Use it like an HTML tag, passing the prop:
<Welcome name="Saman" />
<Welcome name="Ayesha" />
<Welcome name="John" />
```

The same Welcome component renders three different greetings, just by passing a different *name* prop each time. Props are the input to a component - exactly like arguments to a function.

Idea 3 - State

State is data that can change over time. When state changes, React automatically re-renders the parts of the page that depend on it - no manual DOM work needed.

We declare state with the **useState** hook. Here is a counter:

```
import { useState } from 'react';

function Counter() {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    );
}
```

Reading the counter line by line

- **useState(0)** - tell React we want a piece of state, starting at zero.
- **const [count, setCount] = ...** - destructure the array. *count* is the current value, *setCount* is the function to update it.
- **{count}** - display the current count inside the JSX. The curly braces let you put any JavaScript expression in JSX.
- **onClick={() => setCount(count + 1)}** - when the button is clicked, call setCount with a new value. React then re-renders.

> **WARNING**
>
> **Never modify state directly.** *count = count + 1* will not work - React won't notice. Always use the setter function (*setCount*) so React knows to re-render.

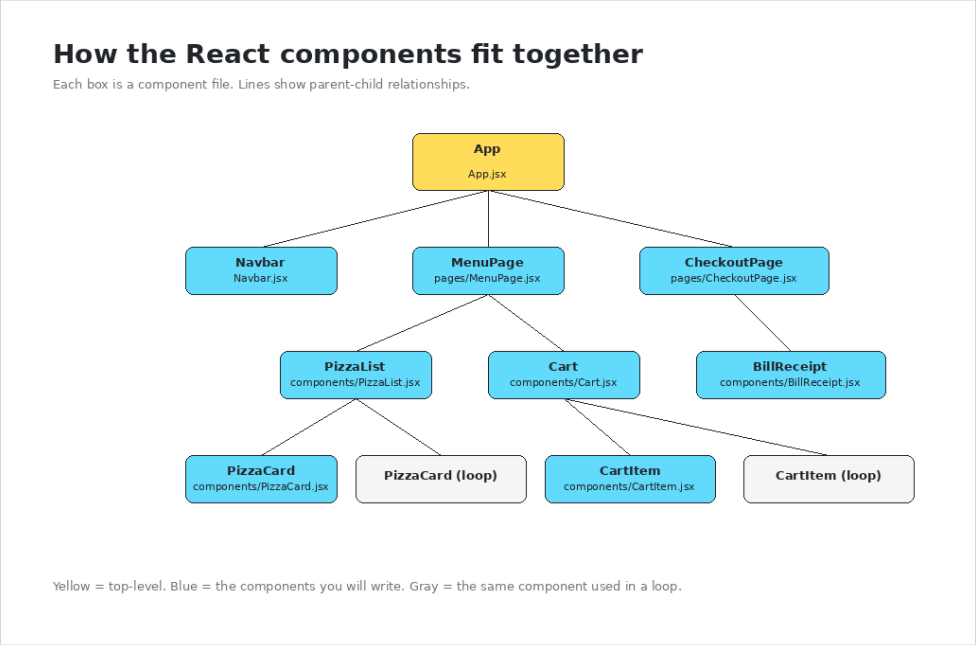
How the three ideas fit together

Components are the building blocks. Props pass data DOWN from parent components to child components. State holds data that changes, owned by one component. When state changes, the component re-renders - and any child components receiving that state via props also re-render.

That is the whole framework, in one paragraph. Everything else is details and patterns built on top of these three ideas.

Our app's component tree

To make this concrete, here is how our pizza billing app will be organised. Each box is a component file you will write:

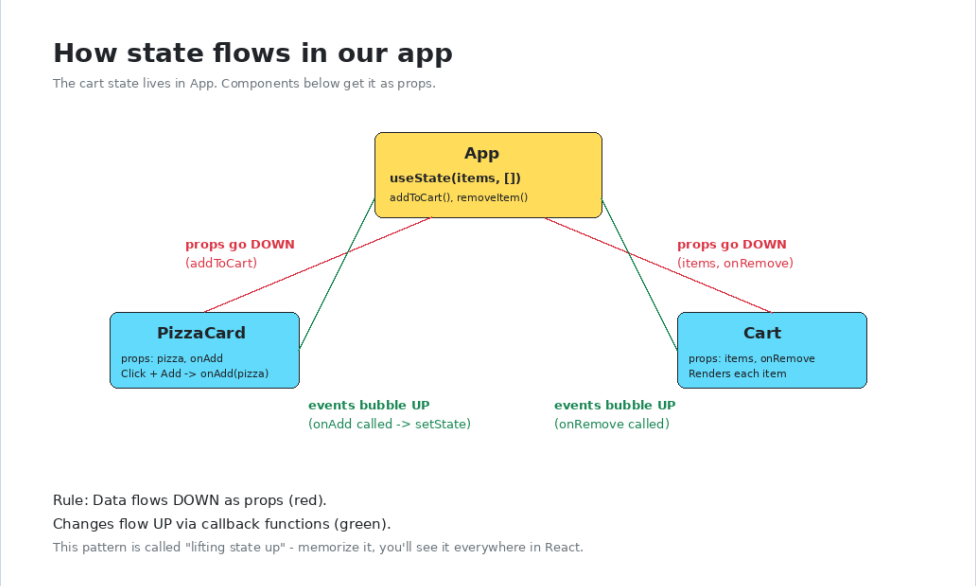


Component tree of the finished app.

Don't worry about understanding the tree right now. We will build each box one chapter at a time. Just notice how everything is made of nested components, with App at the top.

How state will flow in our app

The cart - which pizzas the customer has ordered - is the most important piece of state. It will live in the App component (the top of the tree) and flow down to children as props. When a child needs to change the cart, it calls a function that App provides.



Data flows DOWN as props (red arrows). Changes flow UP via callback functions (green arrows). This pattern is called "lifting state up".

> **TIP**
>
> Lifting state up is the single most important pattern in React. When two components need to share data, that data lives in their common parent and flows down as props. Memorise this - you will use it constantly.

Chapter 3 - Tools You Need to Install

React projects need only three things on your computer. Install everything below before chapter 4 - it is all free, all open source, and works on Windows, Mac, and Linux.

What you need and why

Tool	Why we need it	Where to get it
Node.js 20+	Runs Vite (the React build tool) and npm (package manager).	nodejs.org
VS Code	Code editor with great React/JSX support.	code.visualstudio.com
A modern browser	Chrome, Firefox, or Edge - to run and test your app.	Already on your computer

That is the whole list. No backend server, no database, no Docker, no Apache - just Node.js and a code editor. React lives entirely in the browser at runtime; Node only helps us during development.

Installing Node.js (the only must-have)

Node.js is a JavaScript runtime. We will use it to run Vite, which compiles our React code into something the browser can read.

1. Visit **nodejs.org**.

2. Click the big green button labelled **LTS** (long-term support). Currently this is version 22 - any version 20 or higher works.

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

4. Open a NEW terminal window (Command Prompt on Windows, Terminal on Mac).

5. Run **node --version** - it should print v20 or higher.

6. Run **npm --version** - it should print 10.x or higher.

> **NOTE**
>
> If **node** is "not found", close the terminal and open a fresh one - tools added to PATH are only picked up by new terminals. If it still fails, re-install with the "Add to PATH" checkbox ticked.

Installing VS Code

1. Visit **code.visualstudio.com**.
2. Click **Download**. The site auto-detects your operating system.
3. Run the installer with default options.
4. Open VS Code. You will see a clean welcome page.

VS Code extensions worth installing

Extensions are like browser add-ons for your editor. The right ones make React development much smoother. Open the Extensions panel (Ctrl+Shift+X or Cmd+Shift+X) and install:

• **ES7+ React/Redux/React-Native snippets** - type *rfc* and press Tab to instantly create a React component.

• **Prettier - Code formatter** - auto-formats your code on save.

• **Auto Rename Tag** - rename an opening tag and the closing tag updates with it.

• **Tailwind CSS IntelliSense** - if you use Tailwind (we will use Bootstrap, but install it anyway for future projects).

• **GitLens** - see who wrote each line and when, useful when working with others.

> **TIP**
>
> All these extensions are free. Search the Extensions panel (Ctrl+Shift+X) and install with one click each.

Bonus - React Developer Tools (browser extension)

This browser extension adds two new tabs to your DevTools - **Components** and **Profiler** - that let you inspect the React component tree, see props and state in real time, and find performance issues. It is the single most useful tool for debugging React apps.

1. Open the Chrome Web Store (or Firefox Add-ons).
2. Search **React Developer Tools**.
3. Click Install / Add to browser.
4. Now when you visit any React site (including your local dev server), the new tabs appear in DevTools (F12).

Chapter 4 - Creating Your First React Project with Vite

Time to make the project. We will use **Vite** (pronounced "veet", French for "fast") - the modern, lightning-quick build tool used by almost every new React project in 2026. Vite replaces the old Create React App tool, which Meta has officially retired.

Step 1 - Create the project

Open a terminal and navigate to wherever you keep code projects. Then run:

```
cd ~/code     # or wherever you put projects
npm create vite@latest pizza-billing -- --template react

cd pizza-billing
npm install
```

What just happened?

- **npm create vite@latest pizza-billing** - tells Vite to scaffold a new project in a folder called pizza-billing.
- **--template react** - we want the React starter, not vanilla JS or Vue.
- **cd pizza-billing** - move into the project folder.
- **npm install** - downloads all the libraries Vite and React need. Takes 30-60 seconds the first time. Creates a *node_modules* folder with thousands of files - never edit anything in there.

Step 2 - Start the dev server

```
npm run dev
```

Vite prints something like:

```
VITE v5.4.0  ready in 412 ms

-> Local:   http://localhost:5173/
-> Network: use --host to expose
-> press h + enter to show help
```

Open **http://localhost:5173** in your browser. You should see a default React + Vite welcome page with a counter button. If you see it - congratulations, you just installed and ran your first React app.

> **TIP**
>
> Keep that terminal window open. Vite's dev server will auto-reload the page in your browser every time you save a file. This is called **hot module replacement**, and it makes development feel almost magical.

Step 3 - Open the project in VS Code

From the same terminal, run:

```
code .
```

(That's *code* followed by a space and a dot.) VS Code opens with the entire project visible in the left sidebar. We are now ready to start writing real code.

> **NOTE**
>
> If *code .* does not open VS Code, you can install the *code* command from inside VS Code: open the Command Palette (Ctrl/Cmd + Shift + P) and type "Shell Command: Install code command in PATH".

Chapter 5 - Tour of the Files - What Each One Does

Before we change anything, let us walk through every file Vite created. Knowing where things live will save you frustration later when something does not work.

The full project structure

```
pizza-billing/
+-- node_modules/            # NEVER touch - thousands of installed packages
+-- public/                  # static files (favicon, robots.txt)
|   `-- vite.svg             # the V logo on the welcome page
+-- src/                     # YOU WILL EDIT - all your code lives here
|   +-- assets/              # images you import in components
|   +-- App.css              # styles for the App component
|   +-- App.jsx              # YOUR main component (we'll rewrite this)
|   +-- index.css            # global styles
|   `-- main.jsx             # entry point - mounts App into the page
+-- .gitignore               # files Git should ignore
+-- index.html               # the only HTML file in the whole project
+-- package.json             # project metadata & dependencies
+-- package-lock.json        # exact versions of installed packages
`-- vite.config.js           # Vite settings (rarely touch)
```

The two folders that matter

- **src/** - 99 percent of your work happens here. Every component file goes inside src.
- **public/** - for static files that should be served as-is, without any processing. Things like favicons and downloadable PDFs.

How a request becomes a page

When you visit *http:///localhost:5173*, this is what happens behind the scenes:

1. Browser asks Vite for the page.

2. Vite serves **index.html** at the project root.

3. index.html contains a single and a script tag pointing to **src/main.jsx**.

4. main.jsx imports React and the App component, then tells React to render App inside the root div.

5. App.jsx returns JSX that becomes the visible HTML.

6. If you change any source file, Vite re-compiles it and pushes the update to your browser within milliseconds.

What is in main.jsx

```
// src/main.jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
    <StrictMode>
        <App />
    </StrictMode>,
)
```

Reading main.jsx line by line

- **import { StrictMode } from 'react'** - StrictMode is a dev-only tool that helps catch common bugs.
- **import { createRoot } from 'react-dom/client'** - the function that connects React to the actual DOM.
- **import './index.css'** - load the global CSS file.
- **import App from './App.jsx'** - load our top-level component.
- **document.getElementById('root')** - find the empty div in index.html.
- **.render(<App />)** - tell React to mount our App component inside that div.

> **TIP**
>
> You will rarely edit main.jsx. It is the bootstrap code that starts your app. Once it works, leave it alone.

What is in App.jsx

Vite's default App.jsx has a counter and some logos - decoration we don't need. We will delete almost all of it. For now, just look at how it is structured:

```
// src/App.jsx (the default Vite version - we'll replace this)
import { useState } from 'react'
import './App.css'

function App() {
    const [count, setCount] = useState(0)

    return (
        <div>
            <h1>Vite + React</h1>
            <button onClick={() => setCount((c) => c + 1)}>
                count is {count}
            </button>
        </div>
    )
}

export default App
```

Step 1 - Clean out the default App

Open **src/App.jsx** in VS Code, select everything (Ctrl/Cmd + A), and delete it. Then paste in this clean starter:

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

function App() {
    return (
        <div>
            <h1>Pizza Express POS</h1>
            <p>Coming soon - the best pizza billing app in town.</p>
        </div>
    )
}

export default App
```

Save the file. Switch to your browser - the page automatically reloads with the new content. **That** is hot module replacement. You did not refresh anything.

Step 2 - Clean out App.css and index.css

Vite's default styles are designed for the demo page. Replace both files with empty stubs - we will add styles as we go.

Open **src/App.css**, delete everything, save the empty file. Open **src/index.css**, delete everything, save. The page now looks plain - black text on a white background. Perfect starting point.

Step 3 - Add Bootstrap 5 for styling

Bootstrap gives us nice-looking buttons, cards, forms, and grids without writing custom CSS. Open **index.html** at the project root and add this line just before the closing </head> tag:

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

Save the file. The browser reloads. Bootstrap is now available - any class names like *btn btn-primary* or *card* will work in our components.

> **NOTE**
>
> Loading Bootstrap from a CDN is the simplest way to add it. For production apps you would *npm install bootstrap* instead, but for learning this is faster and works the same.

Chapter 6 - Your First Component - Building the Navbar

Let us write our first real component. The navbar is a small, self-contained piece of UI - perfect for getting our hands dirty.

Step 1 - Create a components folder

Inside **src/**, create a new folder called **components**. All reusable component files will live here. The convention is one file per component, named the same as the component.

Right-click the **src/** folder in VS Code, choose New Folder, type **components**, hit Enter.

Step 2 - Create Navbar.jsx

Right-click the new **components/** folder, choose New File, type **Navbar.jsx**, and paste in this code:

```
// src/components/Navbar.jsx
function Navbar() {
    return (
        <nav className="navbar navbar-dark bg-danger px-3">
            <span className="navbar-brand fw-bold">
                Pizza Express POS
            </span>
            <div>
                <a href="/" className="text-white text-decoration-none me-3">
                    Menu
                </a>
                <a href="/history" className="text-white text-decoration-none">
                    Sales History
                </a>
            </div>
        </nav>
    )
}

export default Navbar
```

Reading the Navbar component

- **function Navbar()** - components are just functions. The name MUST start with a capital letter (React's rule).
- **className** instead of **class** - *class* is a reserved word in JavaScript, so React renamed it.
- **navbar navbar-dark bg-danger** - Bootstrap 5 classes that give us a red dark navbar.
- **export default Navbar** - makes the component importable from other files.

Step 3 - Use the Navbar in App

Open **src/App.jsx** and import the Navbar at the top, then use it in the JSX:

```
// src/App.jsx
import Navbar from './components/Navbar'
import './App.css'

function App() {
    return (
        <div>
            <Navbar />
            <main className="container py-4">
                <h1>Menu</h1>
                <p>Coming soon - pizza grid here.</p>
            </main>
        </div>
    )
}

export default App
```

Save. The browser reloads and you should see a red navbar at the top, with the brand name on the left and two links on the right. Below it, the Menu heading and a placeholder.

> **TIP**
>
> Notice how we use the component as if it were a custom HTML tag: **<Navbar />**. This is the magic of React - everything is a tag, and you can compose them just like Lego blocks.

Chapter 7 - The Pizza Menu - Lists and Loops

Now we render a list of pizzas. We will create two new components: **PizzaCard** (one pizza, looks like a card) and **PizzaList** (loops through the array of pizzas, renders one PizzaCard for each).

Step 1 - Create the pizzas data

We need an array of pizzas. In a real app this would come from a database, but we have no backend - so we will hard-code it as a JavaScript array. Create **src/data/pizzas.js** (you will need to make a *data* folder first):

```
// src/data/pizzas.js
export const pizzas = [
    {
        id: 1,
        name: 'Margherita',
        description: 'Tomato, mozzarella, basil',
        price: 1200,
    },
    {
        id: 2,
        name: 'Pepperoni',
        description: 'Spicy pepperoni & cheese',
        price: 1500,
    },
    {
        id: 3,
        name: 'BBQ Chicken',
        description: 'Grilled chicken, BBQ sauce',
        price: 1700,
    },
    {
        id: 4,
        name: 'Veggie',
        description: 'Peppers, olives, mushrooms',
        price: 1300,
    },
]
```

> **TIP**
>
> Every pizza has a unique **id**. React uses this ID as a *key* when rendering lists - it lets React efficiently update only the items that change. We will see this in a moment.

Step 2 - Create PizzaCard component

src/components/PizzaCard.jsx:

```
// src/components/PizzaCard.jsx
function PizzaCard({ pizza }) {
    return (
        <div className="card h-100 shadow-sm">
            <div className="card-body d-flex">
                <div className="me-3">
                    <div
                        className="rounded-circle bg-warning"
                        style={{ width: 80, height: 80 }}
                    />
                </div>
                <div className="flex-grow-1">
                    <h5 className="card-title">{pizza.name}</h5>
                    <p className="text-muted small mb-2">
                        {pizza.description}
                    </p>
                    <p className="h5 text-danger fw-bold mb-2">
                        Rs. {pizza.price.toLocaleString()}
                    </p>
                    <button className="btn btn-danger btn-sm">
                        + Add
                    </button>
                </div>
            </div>
        </div>
    )
}

export default PizzaCard
```

What is happening here?

- **function PizzaCard({ pizza })** - destructure the pizza prop directly in the function arguments. Cleaner than *props.pizza*.
- **{pizza.name}** - JSX uses curly braces for JavaScript expressions. Anything inside is evaluated as JS.
- **style={{ width: 80, height: 80 }}** - inline styles in React use a JavaScript object (camelCase keys, numbers without 'px'). The double braces are *{ a JS object }*.
- **pizza.price.toLocaleString()** - formats the number with commas: 1500 becomes 1,500.

Step 3 - Create PizzaList component

src/components/PizzaList.jsx:

```
// src/components/PizzaList.jsx
import { pizzas } from '../data/pizzas'
import PizzaCard from './PizzaCard'

function PizzaList() {
    return (
        <div className="row g-3">
            {pizzas.map((pizza) => (
                <div key={pizza.id} className="col-md-6">
                    <PizzaCard pizza={pizza} />
                </div>
            ))}
        </div>
    )
}

export default PizzaList
```

The .map() pattern - rendering lists

This is one of the most common patterns in React. **.map()** is JavaScript's way of transforming each item in an array into something else. Here we transform each pizza object into a PizzaCard component.

- **{pizzas.map(...)}** - curly braces because we are inside JSX and want to run JavaScript.
- **(pizza) => (...)** - arrow function: for each pizza, return some JSX.
- **key={pizza.id}** - this is critical. React uses keys to track which items changed. Without keys, React shows a warning and the app may behave weirdly when items are added/removed.
- **pizza={pizza}** - pass the pizza object as a prop to PizzaCard.

> **WARNING**
>
> **Always provide a unique *key* when rendering a list.** Use the item's database ID (or the index as a last resort, but only if the list never reorders).

Step 4 - Show the list in App

Update **App.jsx** to use PizzaList:

```
// src/App.jsx
import Navbar from './components/Navbar'
import PizzaList from './components/PizzaList'
import './App.css'

function App() {
    return (
        <div>
            <Navbar />
            <main className="container py-4">
                <h1>Menu</h1>
                <p className="text-muted">
                    Tap a pizza to add it to the bill
                </p>
                <PizzaList />
            </main>
        </div>
    )
}

export default App
```

Save. You should now see four pizza cards in a 2-column grid. Each card shows the name, description, price, and an Add button. The Add button does not do anything yet - that is chapter 8.

Chapter 8 - useState - Adding Pizzas to the Cart

Now we make the Add button do something. When the cashier clicks Add, the pizza should appear in a cart. The cart needs to remember what is in it - that is what state is for.

Step 1 - Where should the cart live?

The cart is shared between two components: PizzaList (where you add to it) and a Cart sidebar (where you see what is in it). Whenever two components need the same data, that data lives in their **common parent**. In our case, that's **App**. This is the lifting state up rule from chapter 2.

Step 2 - Add useState to App

Update **src/App.jsx**:

```
// src/App.jsx
import { useState } from 'react'
import Navbar from './components/Navbar'
import PizzaList from './components/PizzaList'
import './App.css'

function App() {
    const [cartItems, setCartItems] = useState([])

    function addToCart(pizza) {
        setCartItems((current) => {
            // Already in cart? Increment quantity.
            const existing = current.find((item) => item.id === pizza.id)
            if (existing) {
                return current.map((item) =>
                    item.id === pizza.id
                        ? { ...item, quantity: item.quantity + 1 }
                        : item
                )
            }
            // Not in cart yet? Add with quantity 1.
            return [...current, { ...pizza, quantity: 1 }]
        })
    }

    return (
        <div>
            <Navbar />
            <main className="container py-4">
                <h1>Menu</h1>
                <p className="text-muted">
                    Tap a pizza to add it to the bill
                </p>
                <PizzaList onAdd={addToCart} />
                <pre>{JSON.stringify(cartItems, null, 2)}</pre>
            </main>
        </div>
    )
}

export default App
```

Reading the addToCart function

• **setCartItems((current) => ...)** - the setter accepts a function that receives the current state. This is the safest way to update state based on the previous value.

• **current.find(...)** - check if the pizza is already in the cart.

• **current.map(...)** - if found, return a new array where that item has its quantity incremented. We never mutate - we always return a NEW array.

• **{ ...item, quantity: item.quantity + 1 }** - the spread operator copies all the properties of *item*, then overrides *quantity* with a new value.

• **[...current, { ...pizza, quantity: 1 }]** - if not found, return a new array with the existing items plus a new entry.

> **WARNING**
>
> **Never mutate state.** Don't use *cartItems.push(...)* or *cartItems[0].quantity*++. React only re-renders if you give it a NEW array or NEW object. Always use spread (...) to create copies.

The temporary <pre> tag

We added **<pre>{JSON.stringify(cartItems, null, 2)}</pre>** as a temporary debug helper. It dumps the cart array as formatted JSON onto the page so we can see the cart fill up while we test. We will remove this in chapter 9.

Step 3 - Pass onAdd through PizzaList to PizzaCard

PizzaList receives **onAdd** as a prop and passes it through to each PizzaCard. Update **PizzaList.jsx**:

```
// src/components/PizzaList.jsx
import { pizzas } from '../data/pizzas'
import PizzaCard from './PizzaCard'

function PizzaList({ onAdd }) {
    return (
        <div className="row g-3">
            {pizzas.map((pizza) => (
                <div key={pizza.id} className="col-md-6">
                    <PizzaCard pizza={pizza} onAdd={onAdd} />
                </div>
            ))}
        </div>
    )
}

export default PizzaList
```

Step 4 - Wire up the click handler in PizzaCard

Update **PizzaCard.jsx**:

```
// src/components/PizzaCard.jsx
function PizzaCard({ pizza, onAdd }) {
    return (
        <div className="card h-100 shadow-sm">
            <div className="card-body d-flex">
                <div className="me-3">
                    <div
                        className="rounded-circle bg-warning"
                        style={{ width: 80, height: 80 }}
                    />
                </div>
                <div className="flex-grow-1">
                    <h5 className="card-title">{pizza.name}</h5>
                    <p className="text-muted small mb-2">
                        {pizza.description}
                    </p>
                    <p className="h5 text-danger fw-bold mb-2">
                        Rs. {pizza.price.toLocaleString()}
                    </p>
                    <button
                        className="btn btn-danger btn-sm"
                        onClick={() => onAdd(pizza)}
                    >
                        + Add
                    </button>
                </div>
            </div>
        </div>
    )
}

export default PizzaCard
```

The only change is the button: it now has an **onClick** handler that calls **onAdd(pizza)**. Click the Add button on any pizza - watch the JSON below the menu fill up. Click the same pizza twice - the quantity goes from 1 to 2.

> **TIP**
>
> Try clicking buttons on different pizzas. Notice how the <pre> debug area updates instantly - no refresh, no manual DOM work. That is the React magic. Behind the scenes, every click calls *setCartItems*, which tells React the state changed, which makes React re-render the parts of the page that depend on *cartItems*.

Chapter 9 - Lifting State Up - The Cart Sidebar

Time to replace that ugly JSON dump with a real cart sidebar. We will create two new components: **Cart** (the sidebar wrapper) and **CartItem** (one row in the cart).

Step 1 - Create CartItem.jsx

src/components/CartItem.jsx:

```
// src/components/CartItem.jsx
function CartItem({ item }) {
    const total = item.price * item.quantity

    return (
        <div className="d-flex justify-content-between align-items-center mb-3">
            <div>
                <div className="fw-bold">{item.name}</div>
                <div className="text-muted small">
                    {item.quantity} x Rs. {item.price.toLocaleString()}
                </div>
            </div>
            <div className="fw-bold">
                Rs. {total.toLocaleString()}
            </div>
        </div>
    )
}

export default CartItem
```

Simple component - takes an *item* prop, calculates its line total, displays the name + quantity on the left and the line total on the right. No state, no events yet - just rendering.

Step 2 - Create Cart.jsx

src/components/Cart.jsx:

```
// src/components/Cart.jsx
import CartItem from './CartItem'

function Cart({ items }) {
    if (items.length === 0) {
        return (
            <div className="card">
                <div className="card-body">
                    <h4 className="card-title">Current Bill</h4>
                    <p className="text-muted">
                        Cart is empty. Pick a pizza to start.
                    </p>
                </div>
            </div>
        )
    }

    const subtotal = items.reduce(
        (sum, item) => sum + item.price * item.quantity,
        0
    )

    return (
```

```
        <div className="card">
            <div className="card-body">
                <h4 className="card-title">Current Bill</h4>
                <p className="text-muted small">
                    {items.length} item{items.length !== 1 ? 's' : ''}
                </p>
                <hr />
                {items.map((item) => (
                    <CartItem key={item.id} item={item} />
                ))}
                <hr />
                <div className="d-flex justify-content-between">
                    <strong>Subtotal</strong>
                    <strong>Rs. {subtotal.toLocaleString()}</strong>
                </div>
            </div>
        </div>
    )
}

export default Cart
```

What is going on here?

- **Early return for empty state** - if there are zero items, render the empty message and stop. Otherwise continue to the real cart.
- **items.reduce(...)** - sums up *price x quantity* for every item. *reduce* walks through an array, accumulating a single value.
- **{items.length !== 1 ? 's' : ''}** - tiny ternary to pluralise: "1 item" or "3 items".
- **items.map(...)** - render one CartItem per item. Same pattern as the pizza grid.

Step 3 - Two-column layout in App

We want the menu on the left, the cart on the right. Bootstrap's grid makes this easy. Update **App.jsx**:

```
// src/App.jsx
import { useState } from 'react'
import Navbar from './components/Navbar'
import PizzaList from './components/PizzaList'
import Cart from './components/Cart'
import './App.css'

function App() {
    const [cartItems, setCartItems] = useState([])

    function addToCart(pizza) {
        setCartItems((current) => {
            const existing = current.find((item) => item.id === pizza.id)
            if (existing) {
                return current.map((item) =>
                    item.id === pizza.id
                        ? { ...item, quantity: item.quantity + 1 }
                        : item
                )
            }
            return [...current, { ...pizza, quantity: 1 }]
        })
    }
```

```
    return (
        <div>
            <Navbar />
            <main className="container-fluid py-4">
                <div className="row">
                    <div className="col-md-8">
                        <h1>Menu</h1>
                        <p className="text-muted">
                            Tap a pizza to add it to the bill
                        </p>
                        <PizzaList onAdd={addToCart} />
                    </div>
                    <div className="col-md-4">
                        <Cart items={cartItems} />
                    </div>
                </div>
            </main>
        </div>
    )
}

export default App
```

Save. You should now see the menu on the left, the cart on the right. Click Add on any pizza - it appears in the cart. Click the same pizza twice - the line shows quantity 2 with the doubled price.

What we just built (component recap)

App owns the cart state. PizzaList receives an *onAdd* callback prop and passes it down to each PizzaCard. PizzaCard's button calls *onAdd(pizza)*, which calls App's *addToCart*, which calls *setCartItems*, which makes React re-render the App. The new *cartItems* array flows DOWN to Cart as a prop, which renders the items.

> **TIP**
>
> Read that paragraph twice. This is the entire core loop of any React app: **state lives somewhere, props flow down, callbacks flow up**. Once you can trace this loop in your head, you understand React.

Chapter 10 - Quantity Controls and Removing Items

The cart works, but the cashier needs to be able to change quantities and remove items. We will add three controls per cart row: minus, plus, and remove (x).

Step 1 - Add helper functions in App

Open **App.jsx** and add three new functions next to *addToCart*:

```
// inside App, alongside addToCart

function increaseQuantity(id) {
    setCartItems((current) =>
        current.map((item) =>
            item.id === id
                ? { ...item, quantity: item.quantity + 1 }
                : item
        )
    )
}

function decreaseQuantity(id) {
    setCartItems((current) =>
        current
            .map((item) =>
                item.id === id
                    ? { ...item, quantity: item.quantity - 1 }
                    : item
            )
            .filter((item) => item.quantity > 0)
    )
}

function removeItem(id) {
    setCartItems((current) =>
        current.filter((item) => item.id !== id)
    )
}
```

How each helper works

- **increaseQuantity(id)** - find the matching item by id, return a copy with quantity + 1.
- **decreaseQuantity(id)** - decrement quantity, then filter out any item that hit zero. Decrementing past zero would be weird, so removing it is cleaner.
- **removeItem(id)** - filter out the item entirely. *filter* returns a new array with only items that pass the test.

Step 2 - Pass the helpers to Cart

Update the JSX inside App to pass the new functions as props:

```
<Cart
    items={cartItems}
    onIncrease={increaseQuantity}
    onDecrease={decreaseQuantity}
    onRemove={removeItem}
/>
```

Step 3 - Pass them through Cart to CartItem

Update **Cart.jsx**:

```
// src/components/Cart.jsx
import CartItem from './CartItem'

function Cart({ items, onIncrease, onDecrease, onRemove }) {
    if (items.length === 0) {
        return (
            <div className="card">
                <div className="card-body">
                    <h4 className="card-title">Current Bill</h4>
                    <p className="text-muted">
                        Cart is empty. Pick a pizza to start.
                    </p>
                </div>
            </div>
        )
    }

    const subtotal = items.reduce(
        (sum, item) => sum + item.price * item.quantity,
        0
    )

    return (
        <div className="card">
            <div className="card-body">
                <h4 className="card-title">Current Bill</h4>
                <p className="text-muted small">
                    {items.length} item{items.length !== 1 ? 's' : ''}
                </p>
                <hr />
                {items.map((item) => (
                    <CartItem
                        key={item.id}
                        item={item}
                        onIncrease={onIncrease}
                        onDecrease={onDecrease}
                        onRemove={onRemove}
                    />
                ))}
                <hr />
                <div className="d-flex justify-content-between">
                    <strong>Subtotal</strong>
                    <strong>Rs. {subtotal.toLocaleString()}</strong>
                </div>
            </div>
        </div>
    )
}
```

```
export default Cart
```

Step 4 - Add the buttons in CartItem

Update **CartItem.jsx**:

```
// src/components/CartItem.jsx
function CartItem({ item, onIncrease, onDecrease, onRemove }) {
    const total = item.price * item.quantity

    return (
        <div className="d-flex justify-content-between align-items-center mb-3">
            <div className="flex-grow-1">
                <div className="fw-bold">{item.name}</div>
                <div className="d-flex align-items-center mt-1">
                    <button
                        className="btn btn-outline-secondary btn-sm"
                        onClick={() => onDecrease(item.id)}
                    >
                        -
                    </button>
                    <span className="mx-2">{item.quantity}</span>
                    <button
                        className="btn btn-outline-secondary btn-sm"
                        onClick={() => onIncrease(item.id)}
                    >
                        +
                    </button>
                    <button
                        className="btn btn-link text-danger btn-sm ms-2"
                        onClick={() => onRemove(item.id)}
                    >
                        x
                    </button>
                </div>
            </div>
            <div className="fw-bold ms-2">
                Rs. {total.toLocaleString()}
            </div>
        </div>
    )
}

export default CartItem
```

Save. Click the - button - the quantity goes down. Click + - it goes up. Click x - the item is gone. The subtotal updates automatically because it is calculated from the cart on every render. **Beautiful.**

> **TIP**
>
> Notice we never wrote any DOM update code. We just told React what the cart should be. React figured out which cart rows to add, remove, or update. This is the React way.

Chapter 11 - Calculating Totals with Tax

Right now the cart shows only a subtotal. Real bills also need tax and a grand total. We will also add a Generate Bill button that we'll wire up in chapter 12.

Step 1 - Add tax constant

Tax rates change. We will keep the rate in one place so it is easy to update. Create **src/data/config.js**:

```
// src/data/config.js
export const TAX_RATE = 0.10  // 10% tax

export const SHOP_INFO = {
    name: 'PIZZA EXPRESS',
    address: 'Sample Street, Colombo 03',
    phone: '+94 11 1234567',
}
```

> **TIP**
>
> Constants you might tweak (tax rate, shop info, currency symbol) should live in a config file. When the tax rate changes, you edit one line - not search-and-replace through 12 components.

Step 2 - Update Cart to show tax and total

Update **Cart.jsx**. The only changes are the totals section at the bottom and the new import:

```
// src/components/Cart.jsx
import { TAX_RATE } from '../data/config'
import CartItem from './CartItem'

function Cart({ items, onIncrease, onDecrease, onRemove, onCheckout }) {
    if (items.length === 0) {
        return (
            <div className="card">
                <div className="card-body">
                    <h4 className="card-title">Current Bill</h4>
                    <p className="text-muted">
                        Cart is empty. Pick a pizza to start.
                    </p>
                </div>
            </div>
        )
    }

    const subtotal = items.reduce(
        (sum, item) => sum + item.price * item.quantity,
        0
    )
    const tax = Math.round(subtotal * TAX_RATE)
    const total = subtotal + tax

    return (
        <div className="card">
            <div className="card-body">
                <h4 className="card-title">Current Bill</h4>
```

```
                <p className="text-muted small">
                    {items.length} item{items.length !== 1 ? 's' : ''}
                </p>
                <hr />
                {items.map((item) => (
                    <CartItem
                        key={item.id}
                        item={item}
                        onIncrease={onIncrease}
                        onDecrease={onDecrease}
                        onRemove={onRemove}
                    />
                ))}
                <hr />
                <div className="d-flex justify-content-between text-muted small">
                    <span>Subtotal</span>
                    <span>Rs. {subtotal.toLocaleString()}</span>
                </div>
                <div className="d-flex justify-content-between text-muted small">
                    <span>Tax ({(TAX_RATE * 100).toFixed(0)}%)</span>
                    <span>Rs. {tax.toLocaleString()}</span>
                </div>
                <hr />
                <div className="d-flex justify-content-between mb-3">
                    <h5 className="mb-0">Total</h5>
                    <h5 className="mb-0 text-danger">
                        Rs. {total.toLocaleString()}
                    </h5>
                </div>
                <button
                    className="btn btn-danger w-100"
                    onClick={onCheckout}
                >
                    Generate Bill
                </button>
            </div>
        </div>
    )
}

export default Cart
```

Reading the calculations

- **subtotal** - sum of price x quantity for every item, same as before.
- **tax = Math.round(subtotal * TAX_RATE)** - 10 percent of subtotal, rounded to whole rupees so we don't get fractional cents.
- **total = subtotal + tax** - the grand total.
- **(TAX_RATE * 100).toFixed(0)** - displays "10%" instead of "0.1%". *toFixed(0)* rounds to zero decimal places.

Step 3 - Wire up the Generate Bill button

We added an **onCheckout** prop to Cart. For now we will just log to the console - in chapter 12 we will navigate to the checkout page. Update **App.jsx**:

```
// inside App component
function handleCheckout() {
    console.log('Checkout clicked! Cart:', cartItems)
    // We'll navigate to /checkout in chapter 12
}

// in the JSX, update Cart usage:
<Cart
    items={cartItems}
    onIncrease={increaseQuantity}
    onDecrease={decreaseQuantity}
    onRemove={removeItem}
    onCheckout={handleCheckout}
/>
```

Click Generate Bill and open the browser DevTools console (F12 -> Console tab). You should see your cart array logged. The button is wired up, just not doing anything useful yet.

Chapter 12 - React Router - The Checkout Page

Right now our app is one page. We need at least three: the menu (home), the checkout page, and the bill. To navigate between them we use **React Router** - the standard routing library.

Step 1 - Install React Router

Stop the dev server (Ctrl+C in the terminal). Then run:

```
npm install react-router-dom
```

Restart the server with **npm run dev**.

Step 2 - Set up the router in main.jsx

Open **src/main.jsx** and wrap App in a BrowserRouter:

```
// src/main.jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import { BrowserRouter } from 'react-router-dom'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
    <StrictMode>
        <BrowserRouter>
            <App />
        </BrowserRouter>
    </StrictMode>,
)
```

BrowserRouter is what gives our app the ability to know which URL the user is on, and to change it without a full page reload.

Step 3 - Move the menu UI into a MenuPage

Create **src/pages/MenuPage.jsx** (you will need a **pages** folder). Cut everything from inside the <main> tag in App.jsx and paste it here:

```
// src/pages/MenuPage.jsx
import PizzaList from '../components/PizzaList'
import Cart from '../components/Cart'

function MenuPage({
    cartItems,
    addToCart,
    increaseQuantity,
    decreaseQuantity,
    removeItem,
    onCheckout,
}) {
    return (
        <main className="container-fluid py-4">
            <div className="row">
                <div className="col-md-8">
                    <h1>Menu</h1>
                    <p className="text-muted">
                        Tap a pizza to add it to the bill
```

```
                    </p>
                    <PizzaList onAdd={addToCart} />
                </div>
                <div className="col-md-4">
                    <Cart
                        items={cartItems}
                        onIncrease={increaseQuantity}
                        onDecrease={decreaseQuantity}
                        onRemove={removeItem}
                        onCheckout={onCheckout}
                    />
                </div>
            </div>
        </main>
    )
}

export default MenuPage
```

Step 4 - Create CheckoutPage.jsx

src/pages/CheckoutPage.jsx - this collects customer info before generating the bill:

```
// src/pages/CheckoutPage.jsx
import { useState } from 'react'
import { useNavigate } from 'react-router-dom'
import { TAX_RATE } from '../data/config'

function CheckoutPage({ cartItems, onConfirm }) {
    const [name, setName] = useState('')
    const [phone, setPhone] = useState('')
    const [table, setTable] = useState('')
    const [payment, setPayment] = useState('Cash')
    const navigate = useNavigate()

    const subtotal = cartItems.reduce(
        (s, i) => s + i.price * i.quantity, 0
    )
    const tax = Math.round(subtotal * TAX_RATE)
    const total = subtotal + tax

    function handleSubmit(e) {
        e.preventDefault()
        const bill = {
            id: Date.now(),
            createdAt: new Date().toISOString(),
            customer: name || 'Walk-in',
            phone,
            table,
            payment,
            items: cartItems,
            subtotal,
            tax,
            total,
        }
        onConfirm(bill)
        navigate(`/bill/${bill.id}`)
    }

    if (cartItems.length === 0) {
        return (
            <main className="container py-4">
                <h1>Cart is empty</h1>
                <p>Go back to the menu and add some pizzas.</p>
```

```
            </main>
        )
    }

    return (
        <main className="container py-4">
            <h1>Customer Details</h1>
            <p className="text-muted">
                Final step before generating the bill
            </p>

            <form className="row g-4" onSubmit={handleSubmit}>
                <div className="col-md-6">
                    <div className="card">
                        <div className="card-body">
                            <div className="mb-3">
                                <label className="form-label">Customer name</label>
                                <input
                                    className="form-control"
                                    value={name}
                                    onChange={(e) => setName(e.target.value)}
                                />
                            </div>
                            <div className="mb-3">
                                <label className="form-label">Phone number</label>
                                <input
                                    className="form-control"
                                    value={phone}
                                    onChange={(e) => setPhone(e.target.value)}
                                />
                            </div>
                            <div className="mb-3">
                                <label className="form-label">Table number</label>
                                <input
                                    className="form-control"
                                    value={table}
                                    onChange={(e) => setTable(e.target.value)}
                                />
                            </div>
                            <div className="mb-3">
                                <label className="form-label d-block">
                                    Payment method
                                </label>
                                {['Cash', 'Card', 'Mobile'].map((opt) => (
                                    <div key={opt} className="form-check form-check-inline">
                                        <input
                                            className="form-check-input"
                                            type="radio"
                                            checked={payment === opt}
                                            onChange={() => setPayment(opt)}
                                        />
                                        <label className="form-check-label">{opt}</label>
                                    </div>
                                ))}
                            </div>
                            <button
                                type="button"
                                className="btn btn-secondary me-2"
                                onClick={() => navigate('/')}
                            >
                                Back to Menu
                            </button>
                            <button type="submit" className="btn btn-danger">
                                Print Bill
                            </button>
```

```
                        </div>
                    </div>
                </div>

                <div className="col-md-6">
                    <div className="card">
                        <div className="card-body">
                            <h4 className="card-title">Order Summary</h4>
                            {cartItems.map((item) => (
                                <div
                                    key={item.id}
                                    className="d-flex justify-content-between"
                                >
                                    <span>{item.name} x {item.quantity}</span>
                                    <span>
                                        Rs. {(item.price * item.quantity).toLocaleString()}
                                    </span>
                                </div>
                            ))}
                            <hr />
                            <div className="d-flex justify-content-between text-muted">
                                <span>Subtotal</span>
                                <span>Rs. {subtotal.toLocaleString()}</span>
                            </div>
                            <div className="d-flex justify-content-between text-muted">
                                <span>Tax</span>
                                <span>Rs. {tax.toLocaleString()}</span>
                            </div>
                            <hr />
                            <div className="d-flex justify-content-between">
                                <h4>Total</h4>
                                <h4 className="text-danger">
                                    Rs. {total.toLocaleString()}
                                </h4>
                            </div>
                        </div>
                    </div>
                </div>
            </form>
        </main>
    )
}

export default CheckoutPage
```

What is new here

- **useState for each form field** - controlled inputs. The input's value comes from state, and onChange updates state.
- **useNavigate()** - hook from React Router. Calling *navigate('/bill/123')* changes the URL without a page reload.
- **e.preventDefault()** - stop the form's default browser behaviour (which would refresh the page).
- **Date.now()** - quick unique ID. Returns milliseconds since 1970, so each bill gets a different number.
- **onConfirm(bill)** - tell the parent (App) about the new bill so it can save it.

Step 5 - Set up routes in App.jsx

Update **App.jsx** to use Routes:

```
// src/App.jsx
import { useState } from 'react'
import { Routes, Route, useNavigate } from 'react-router-dom'
import Navbar from './components/Navbar'
import MenuPage from './pages/MenuPage'
import CheckoutPage from './pages/CheckoutPage'
import './App.css'

function App() {
    const [cartItems, setCartItems] = useState([])
    const [bills, setBills] = useState([])
    const navigate = useNavigate()

    function addToCart(pizza) {
        setCartItems((current) => {
            const existing = current.find((item) => item.id === pizza.id)
            if (existing) {
                return current.map((item) =>
                    item.id === pizza.id
                        ? { ...item, quantity: item.quantity + 1 }
                        : item
                )
            }
            return [...current, { ...pizza, quantity: 1 }]
        })
    }

    const increaseQuantity = (id) =>
        setCartItems((c) =>
            c.map((i) => (i.id === id ? { ...i, quantity: i.quantity + 1 } : i))
        )

    const decreaseQuantity = (id) =>
        setCartItems((c) =>
            c.map((i) => (i.id === id ? { ...i, quantity: i.quantity - 1 } : i))
              .filter((i) => i.quantity > 0)
        )

    const removeItem = (id) =>
        setCartItems((c) => c.filter((i) => i.id !== id))

    function handleCheckout() {
        navigate('/checkout')
    }

    function handleConfirmBill(bill) {
        setBills((current) => [bill, ...current])
        setCartItems([])  // clear the cart
    }

    return (
        <div>
            <Navbar />
            <Routes>
                <Route
                    path="/"
                    element={
                        <MenuPage
                            cartItems={cartItems}
                            addToCart={addToCart}
```

```
                            increaseQuantity={increaseQuantity}
                            decreaseQuantity={decreaseQuantity}
                            removeItem={removeItem}
                            onCheckout={handleCheckout}
                        />
                    }
                />
                <Route
                    path="/checkout"
                    element={
                        <CheckoutPage
                            cartItems={cartItems}
                            onConfirm={handleConfirmBill}
                        />
                    }
                />
            </Routes>
        </div>
    )
}

export default App
```

Save. Add some pizzas to the cart, click Generate Bill - the URL changes to /checkout and the customer form appears. Fill it in, click Print Bill - the URL changes to /bill/<number>. The bill page does not exist yet (404), but the navigation works.

Chapter 13 - Generating the Bill (Receipt)

Now we render the bill page. It needs to look like a real receipt and have a Print button that opens the browser's print dialog.

Step 1 - Create BillReceipt component

src/components/BillReceipt.jsx - the printable receipt:

```
// src/components/BillReceipt.jsx
import { SHOP_INFO } from '../data/config'

function BillReceipt({ bill }) {
    return (
        <div className="card mx-auto" style={{ maxWidth: 460 }}>
            <div className="card-body">
                <div className="text-center mb-3">
                    <h3 className="mb-1">{SHOP_INFO.name}</h3>
                    <div className="text-muted small">{SHOP_INFO.address}</div>
                    <div className="text-muted small">{SHOP_INFO.phone}</div>
                </div>
                <hr style={{ borderStyle: 'dashed' }} />

                <div className="small">
                    <div className="d-flex">
                        <div style={{ width: 100 }} className="text-muted">
                            Bill #:
                        </div>
                        <div>{bill.id}</div>
                    </div>
                    <div className="d-flex">
                        <div style={{ width: 100 }} className="text-muted">
                            Date:
                        </div>
                        <div>{new Date(bill.createdAt).toLocaleString()}</div>
                    </div>
                    <div className="d-flex">
                        <div style={{ width: 100 }} className="text-muted">
                            Customer:
                        </div>
                        <div>{bill.customer}</div>
                    </div>
                    {bill.phone && (
                        <div className="d-flex">
                            <div style={{ width: 100 }} className="text-muted">
                                Phone:
                            </div>
                            <div>{bill.phone}</div>
                        </div>
                    )}
                    {bill.table && (
                        <div className="d-flex">
                            <div style={{ width: 100 }} className="text-muted">
                                Table:
                            </div>
                            <div>{bill.table}</div>
                        </div>
                    )}
                    <div className="d-flex">
                        <div style={{ width: 100 }} className="text-muted">
                            Payment:
                        </div>
```

```
                        <div>{bill.payment}</div>
                    </div>
                </div>

                <hr style={{ borderStyle: 'dashed' }} />

                <table className="table table-sm">
                    <thead>
                        <tr>
                            <th>Item</th>
                            <th className="text-center">Qty</th>
                            <th className="text-end">Price</th>
                            <th className="text-end">Total</th>
                        </tr>
                    </thead>
                    <tbody>
                        {bill.items.map((item) => (
                            <tr key={item.id}>
                                <td>{item.name}</td>
                                <td className="text-center">{item.quantity}</td>
                                <td className="text-end">
                                    {item.price.toLocaleString()}
                                </td>
                                <td className="text-end">
                                    {(item.price * item.quantity).toLocaleString()}
                                </td>
                            </tr>
                        ))}
                    </tbody>
                </table>

                <hr style={{ borderStyle: 'dashed' }} />

                <div className="d-flex justify-content-between small text-muted">
                    <span>Subtotal</span>
                    <span>Rs. {bill.subtotal.toLocaleString()}</span>
                </div>
                <div className="d-flex justify-content-between small text-muted">
                    <span>Tax</span>
                    <span>Rs. {bill.tax.toLocaleString()}</span>
                </div>
                <hr style={{ borderStyle: 'dashed' }} />
                <div className="d-flex justify-content-between">
                    <h5>TOTAL</h5>
                    <h5 className="text-danger">
                        Rs. {bill.total.toLocaleString()}
                    </h5>
                </div>

                <div className="text-center text-muted small mt-3">
                    Thank you, come again!
                </div>
            </div>
        </div>
    )
}

export default BillReceipt
```

Step 2 - Create BillPage

src/pages/BillPage.jsx:

```
// src/pages/BillPage.jsx
import { useParams, useNavigate, Link } from 'react-router-dom'
import BillReceipt from '../components/BillReceipt'

function BillPage({ bills }) {
    const { id } = useParams()
    const navigate = useNavigate()
    const bill = bills.find((b) => String(b.id) === id)

    if (!bill) {
        return (
            <main className="container py-4">
                <h1>Bill not found</h1>
                <Link to="/" className="btn btn-primary">
                    Back to Menu
                </Link>
            </main>
        )
    }

    return (
        <main className="container py-4">
            <div className="d-flex justify-content-between align-items-center mb-3">
                <div>
                    <h1>Bill #{bill.id}</h1>
                    <p className="text-muted">
                        Generated {new Date(bill.createdAt).toLocaleString()}
                    </p>
                </div>
                <div>
                    <button
                        className="btn btn-dark me-2"
                        onClick={() => window.print()}
                    >
                        Print
                    </button>
                    <button
                        className="btn btn-danger"
                        onClick={() => navigate('/')}
                    >
                        + New Order
                    </button>
                </div>
            </div>

            <BillReceipt bill={bill} />
        </main>
    )
}

export default BillPage
```

What is new

- **useParams()** - reads the dynamic part of the URL. For */bill/123*, params.id is "123".
- **bills.find(...)** - search the array for the matching bill.
- **String(b.id) === id** - URL params are always strings, so we convert the numeric id to a string before comparing.

• **window.print()** - opens the browser's native print dialog. No library needed - this is a standard browser API.

Step 3 - Add the bill route

Open **App.jsx** and add the new route:

```
// at the top of App.jsx
import BillPage from './pages/BillPage'

// inside <Routes>:
<Route
    path="/bill/:id"
    element={<BillPage bills={bills} />}
/>
```

Save. Now go through the whole flow: add pizzas, click Generate Bill, fill in customer info, click Print Bill. You should land on a beautifully formatted receipt page. Click Print - the browser's print dialog opens. Click + New Order - you go back to a fresh menu (cart is cleared).

> **TIP**
>
> Try opening the browser DevTools and switching to mobile view (F12, then toggle device toolbar). The receipt should still look good on a phone-sized screen because we used Bootstrap's responsive classes.

Chapter 14 - Sales History with localStorage

Right now bills disappear when you refresh the page. That is obviously not OK for a real shop. We will use the browser's **localStorage** to save bills permanently.

What is localStorage?

Every browser has a tiny built-in key-value database called **localStorage**. You give it a string key and a string value, and the browser remembers it forever - even after closing the browser, restarting the computer, or rebooting. Each website has its own private localStorage.

The localStorage API

```
// Save
localStorage.setItem('myKey', 'someValue')

// Read
const value = localStorage.getItem('myKey')

// Delete
localStorage.removeItem('myKey')

// Clear everything
localStorage.clear()
```

> **TIP**
>
> localStorage only stores strings. To save objects or arrays, convert them with **JSON.stringify()**. To read them back, use **JSON.parse()**.

Step 1 - Load bills from localStorage on startup

Open **App.jsx** and change the bills useState to read the initial value from localStorage:

```
// inside App, replace the existing useState for bills:
const [bills, setBills] = useState(() => {
    const saved = localStorage.getItem('bills')
    return saved ? JSON.parse(saved) : []
})
```

useState can take a function as its argument. React only calls that function once, on the first render. This is called a *lazy initializer* - perfect for loading from localStorage.

Step 2 - Save bills whenever they change

We need to write to localStorage every time bills changes. The **useEffect** hook lets us run code AFTER the component renders. Add this near the top of App, alongside useState:

```
// at the top of App.jsx, add to imports:
import { useState, useEffect } from 'react'

// inside App, after useState:
useEffect(() => {
    localStorage.setItem('bills', JSON.stringify(bills))
}, [bills])
```

How useEffect works

- **useEffect(() => { ... }, [bills])** - run the function whenever bills changes.
- The array **[bills]** is the *dependency list*. React compares new dependencies to old ones. If they are different, the effect runs again.
- First render: effect runs (saves the initial bills, possibly empty array).
- User generates a new bill: *bills* changes, effect runs, writes the new bills array to localStorage.
- User refreshes the page: lazy initializer reads bills back from localStorage. The cycle is complete.

> **TIP**
>
> useState + useEffect is the most common combination in React. Get comfortable with this pattern - you will use it constantly. Whenever you have state that should also live somewhere outside React (localStorage, an API, the URL), useEffect is how you sync it.

Step 3 - Create the SalesHistoryPage

src/pages/SalesHistoryPage.jsx:

```
// src/pages/SalesHistoryPage.jsx
import { Link } from 'react-router-dom'

function SalesHistoryPage({ bills }) {
    // Today's bills only - filter to bills generated today.
    const today = new Date().toDateString()
    const todayBills = bills.filter(
        (b) => new Date(b.createdAt).toDateString() === today
    )
    const todayRevenue = todayBills.reduce((s, b) => s + b.total, 0)
    const avgBill = todayBills.length > 0
        ? Math.round(todayRevenue / todayBills.length)
        : 0

    return (
        <main className="container py-4">
            <h1>Sales History</h1>
            <p className="text-muted">
                All bills from this device, newest first
            </p>

            <div className="row g-3 mb-4">
                <div className="col-md-4">
                    <div className="card">
                        <div className="card-body">
```

```
                    <div className="text-muted small">Today's Bills</div>
                    <div className="h2 text-primary">{todayBills.length}</div>
                </div>
            </div>
        </div>
        <div className="col-md-4">
            <div className="card">
                <div className="card-body">
                    <div className="text-muted small">Today's Revenue</div>
                    <div className="h2 text-success">
                        Rs. {todayRevenue.toLocaleString()}
                    </div>
                </div>
            </div>
        </div>
        <div className="col-md-4">
            <div className="card">
                <div className="card-body">
                    <div className="text-muted small">Avg Bill</div>
                    <div className="h2">
                        Rs. {avgBill.toLocaleString()}
                    </div>
                </div>
            </div>
        </div>
    </div>

    {bills.length === 0 ? (
        <p className="text-muted">No bills yet. Make a sale!</p>
    ) : (
        <table className="table">
            <thead className="table-dark">
                <tr>
                    <th>Bill #</th>
                    <th>Time</th>
                    <th>Customer</th>
                    <th>Table</th>
                    <th>Items</th>
                    <th>Total</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                {bills.map((bill) => (
                    <tr key={bill.id}>
                        <td>{bill.id}</td>
                        <td>
                            {new Date(bill.createdAt).toLocaleTimeString()}
                        </td>
                        <td>{bill.customer}</td>
                        <td>{bill.table || '-'}</td>
                        <td>
                            {bill.items.length} item
                            {bill.items.length !== 1 ? 's' : ''}
                        </td>
                        <td>Rs. {bill.total.toLocaleString()}</td>
                        <td>
                            <Link
                                to={`/bill/${bill.id}`}
                                className="btn btn-sm btn-primary"
                            >
                                View
                            </Link>
                        </td>
                    </tr>
```

```
                        ))}
                    </tbody>
                </table>
            )}
        </main>
    )
}

export default SalesHistoryPage
```

Step 4 - Add the history route

```
// at the top of App.jsx
import SalesHistoryPage from './pages/SalesHistoryPage'

// add inside <Routes>:
<Route
    path="/history"
    element={<SalesHistoryPage bills={bills} />}
/>
```

Save. Visit **http://localhost:5173/history** or click the Sales History link in the navbar. You should see all your bills with daily stats at the top. Refresh the page - the bills are still there. Close the browser, reopen, visit again - still there. **That** is localStorage.

View past bills

Click the View button on any row - you go to that bill's receipt page. Print is still available. The whole flow works for any bill, old or new.

> **TIP**
>
> If you ever want to wipe all bills (for testing), open DevTools (F12), go to Application tab -> Local Storage -> http://localhost:5173, right-click and delete the *bills* key. Or run *localStorage.clear()* in the Console.

Chapter 15 - Deploying Your App to the Internet

Your app works locally. Time to put it online so anyone can use it. There are several free options - we will cover the easiest two.

Step 1 - Build for production

Running *npm run dev* uses Vite's development server. For deployment we need a production build. Stop the dev server (Ctrl+C) and run:

```
npm run build
```

Vite creates a **dist/** folder containing everything needed to host the app: HTML, CSS, JS, images. The output is minified and optimised. Test it locally first:

```
npm run preview
```

This serves the production build at **http://localhost:4173**. Make sure the app works there before deploying.

Option A - Deploy to Netlify (easiest)

Netlify hosts static sites for free with HTTPS, custom domains, and global CDN. The drag-and-drop method takes about 60 seconds.

1. Visit **app.netlify.com** and sign up (free).
2. On the dashboard, find the box that says "Want to deploy a new site without connecting to Git?" - drag your **dist** folder onto it.
3. Wait 10 seconds while Netlify uploads.
4. Done. Your site has a URL like *amazing-pizza-12345.netlify.app*. Share it with the world.

> **TIP**
>
> Netlify will offer you a custom domain like **my-pizza-shop.netlify.app** (free) or you can connect your own domain (like **mypizzashop.com**) in five minutes via the settings.

Option B - Deploy to GitHub Pages

If you want to put your project in a public GitHub repo and host from there, GitHub Pages is free and reliable.

1. Push your project to GitHub (see chapter 14 of our Laravel tutorial for git basics).

2. Install the gh-pages helper: **npm install gh-pages --save-dev**

3. Open **vite.config.js** and add **base: '/your-repo-name/'** inside the config object.

4. Open **package.json** and add to scripts: *"deploy": "npm run build && gh-pages -d dist"*.

5. Run **npm run deploy**. The script builds and pushes the dist folder to a special *gh-pages* branch.

6. On GitHub, go to your repo's Settings -> Pages -> Source: *gh-pages* branch. Wait 30 seconds for it to go live.

7. Visit **https://your-username.github.io/your-repo-name**.

Step 2 - Important: localStorage is per-domain

When you deploy your app, it gets a new URL. localStorage data from your local development (*localhost:5173*) does NOT transfer to the deployed version - they are different domains to the browser. The deployed app starts with no bills. This is fine - in production, you and your real users will use the deployed URL from day one.

Step 3 - Going further

You have built and deployed a working React app. Here are 10 ideas to take it further:

1. Add an admin panel - manage the pizza menu (add, edit, delete items) instead of hard-coding pizzas.

2. Discounts - input a discount percentage in checkout, subtract from total.

3. Daily/weekly charts - install Recharts and show sales trends over time.

4. Search and filter on the sales history page - find bills by customer name or date range.

5. Print only the receipt - use a CSS @media print rule to hide the navbar and buttons when printing.

6. Export sales to CSV - one button to download all bills as a spreadsheet.

7. Multiple cashiers - require login, track which user made each bill.

8. Add a backend - replace localStorage with a real API (see our Laravel tutorial for the backend side).

9. PWA support - make it installable as an app on mobile devices.

10. Dark mode - add a theme toggle.

Final thoughts

You started this tutorial knowing nothing about React. You finished by building a complete point-of-sale app, deploying it to the internet, and learning the patterns that every React developer uses every day - components, props, state, events, routing, and persistence.

The next step is **build something else**. Pick anything that interests you - a todo list, a habit tracker, a quiz game, a weather dashboard - and build it from scratch. Reading more tutorials helps, but the real learning happens when you face your own problems.

Share your project on LinkedIn, post a screenshot, link to the live URL. That visibility is how careers in tech start. We at **egotechworld.com** would love to feature your work - send us a link.

Happy coding!

Quick Reference Card - React 19

Hooks you used in this tutorial

Hook	What it does
useState	Declare state. Returns [value, setter]. Setter triggers re-render.
useEffect	Run side effects after render. Sync with localStorage, APIs, etc.
useNavigate	From React Router. Returns a function to programmatically change URL.
useParams	From React Router. Read dynamic URL params (like /bill/:id).

Most-used npm commands

Command	What it does
npm create vite@latest my-app	Create a new Vite project.
npm install	Install dependencies (one-time after cloning a repo).
npm run dev	Start the dev server with hot reload.
npm run build	Build for production into dist/.
npm run preview	Serve the production build locally to test.
npm install package-name	Add a new package.

Final project structure

```
pizza-billing/
+-- node_modules/            # auto-generated, never edit
+-- public/
|   `-- vite.svg
+-- src/
|   +-- components/
|   |   +-- BillReceipt.jsx
|   |   +-- Cart.jsx
|   |   +-- CartItem.jsx
|   |   +-- Navbar.jsx
|   |   +-- PizzaCard.jsx
|   |   `-- PizzaList.jsx
|   +-- data/
|   |   +-- config.js
|   |   `-- pizzas.js
|   +-- pages/
|   |   +-- BillPage.jsx
|   |   +-- CheckoutPage.jsx
|   |   +-- MenuPage.jsx
|   |   `-- SalesHistoryPage.jsx
|   +-- App.css
|   +-- App.jsx
|   +-- index.css
|   `-- main.jsx
+-- index.html
+-- package.json
`-- vite.config.js
```

Common React mistakes and fixes

Mistake	Fix
Mutating state directly (cartItems.push)	Always create a new array/object: [...cartItems, newItem].
Forgetting key in a list	Always add key={item.id} to the outermost looped element.
Component name lowercase (function navbar)	React components MUST start with a capital letter: Navbar.
Using class= instead of className=	JSX uses className for the CSS class attribute.
Calling a function instead of passing it	**onClick={handleClick}** is correct. **onClick={handleClick()}** calls it on render.
State not updating	Setter is async. Use the function form: setCount(c => c + 1).

Where to go next

- **react.dev** - the official React docs. Excellent and regularly updated.
- **react-router.com** - everything about routing.
- **tailwindcss.com** - if you want to try Tailwind instead of Bootstrap.
- **vitejs.dev** - more about Vite.
- **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, Laravel, PHP, and Python tutorials. We post new content regularly and welcome guest articles from learners like you.

Happy coding!