# How APIs Work

## Frontend + API + Backend

### A Beginner's Step-by-Step Tutorial

Build a temperature converter and learn every moving part

**No database. No frameworks. Just the essentials.**


## What you will learn

What an API is, in plain English.
How HTTP requests and responses are shaped.
What JSON is and why APIs love it.
Building a backend with Node.js and Express.
Building a frontend with HTML, CSS, JavaScript.
Calling an API from the browser with fetch().
Handling errors and HTTP status codes.
Testing your API with Thunder Client.

**egotechworld.com**

*Free coding tutorials, project source code, and AI tools*

Beginner-Friendly - Step by Step - 2026 Edition

# Table of Contents

# Chapter 1 - What Is an API, in Plain English

Welcome! In this tutorial we are going to demystify **APIs** - a word you hear constantly in the tech world but rarely see explained simply. By the end you will have built a working API with your own hands and you will understand exactly what it is, what it does, and how it fits into a real web app.

## What is an API?

**API** stands for **Application Programming Interface**. That sounds intimidating, but the idea is simple. An API is a **contract** between two pieces of software. It says: "if you ask me in this exact way, I will answer in this exact way".

That is it. APIs are agreements about how software talks to other software.

## A real-world analogy

Imagine walking into a restaurant. You sit down. The waiter hands you a **menu**. The menu lists everything the kitchen can make and how to ask for it ("Margherita pizza - Rs. 1,200"). You say what you want. The waiter takes your order to the kitchen. The kitchen cooks. The waiter brings the food back.

In this analogy:

- **The customer** is the frontend - whatever the user interacts with.
- **The kitchen** is the backend - where the work actually happens.
- **The menu** is the API - the list of things you can ask for, and how.
- **The waiter** is HTTP - the protocol that carries messages back and forth.

You don't walk into the kitchen and start cooking. You don't shout your order over a wall. You use the menu. You follow the agreed system. The menu hides the complexity of the kitchen from you - you don't need to know which chef cooks the pizza, what brand of cheese they use, or where the oven is. You just know how to *order*.

## Why have APIs at all?

- **Separation of concerns** - the team building the website does not need to be the same team building the database. They agree on the API and work in parallel.
- **Reuse** - one backend can serve a website, a mobile app, and a smart-watch app. They all talk to the same API.
- **Security** - the database, your secrets, your business logic stay safely on the server. The browser only sees the API.
- **Stability** - you can rewrite the backend in a different language tomorrow. As long as the API stays the same, the frontend never knows.

## The kind of API we are building

There are many kinds of APIs - libraries, operating system APIs, hardware APIs. We are focused on the most common kind in modern web development: the **web API**. Specifically a **REST API** that uses **HTTP** and **JSON**.

Don't worry about those words yet. We will unpack each one in the next few chapters.

## What you will build

To keep the focus on APIs (not business logic), we will build the simplest possible useful app - a **temperature converter**. The user types a number in Celsius, clicks Convert, and sees the value in Fahrenheit.

Why so simple? Because the conversion math is one line of code:

```
fahrenheit = celsius * 9 / 5 + 32
```

With the calculation out of the way, every other line of code we write is doing some part of the API plumbing - which is exactly what we want to learn.

### The two screens

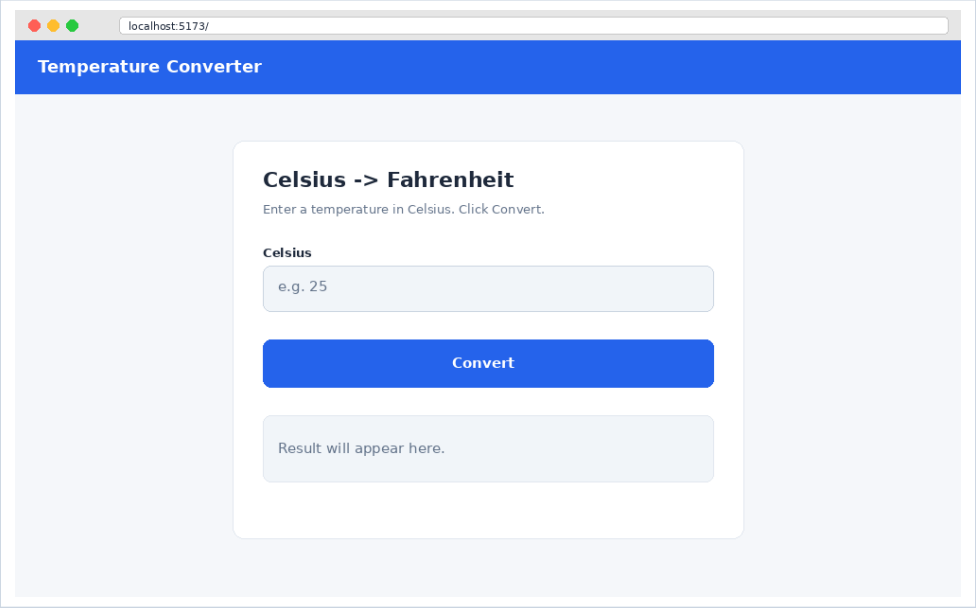


*Figure 1 - The starting screen. One input, one button, one result area.*

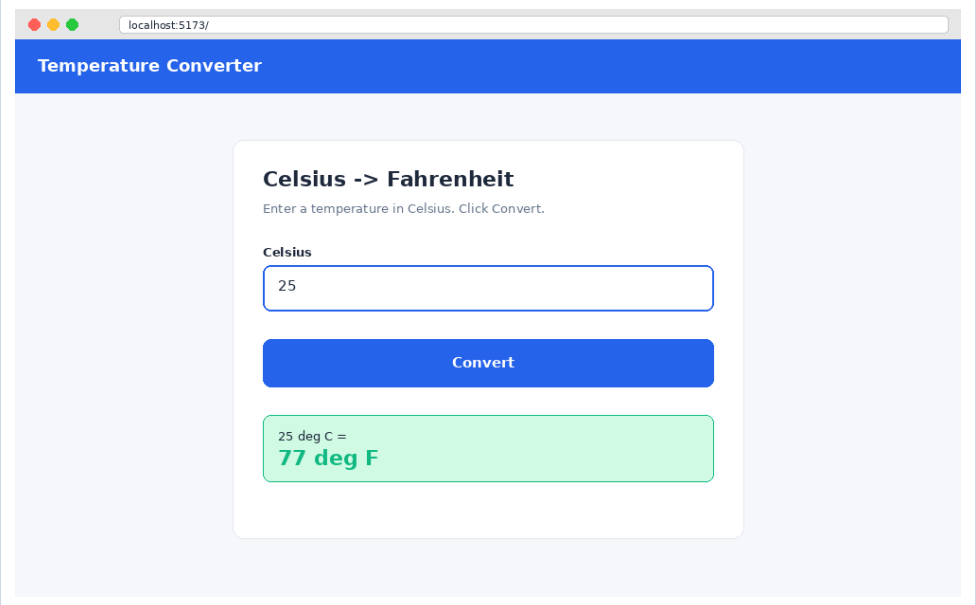


*Figure 2 - After the user types 25 and clicks Convert, the API responds with 77 deg F.*

## What you will NOT need

- **No database** - we are calculating, not storing.
- **No login or accounts** - anyone can use it.
- **No frameworks** on the frontend - just HTML, CSS, and vanilla JavaScript.
- **No deployment** - we run it locally to learn.

> **TIP**
>
> By avoiding those layers, we strip the project down to the *API mechanics* alone. Every line of code you write will be helping you understand how APIs work. Once you have this foundation, adding databases and frameworks later is the easy part.

## Who this tutorial is for

- Beginners who have heard the word "API" and want to know what it really means.
- Frontend developers who have called APIs but never built one.
- Self-taught developers connecting the dots between separate skills.
- Anyone preparing for their first developer job.

If you know basic HTML and have written even ten lines of JavaScript, you have enough background. Let us begin.

# Chapter 2 - The Three Pieces - Frontend, API, Backend

Every web app like ours has three pieces. They are separate. They talk to each other. Understanding what each one does and why it is separate from the others is the foundation of everything that follows.

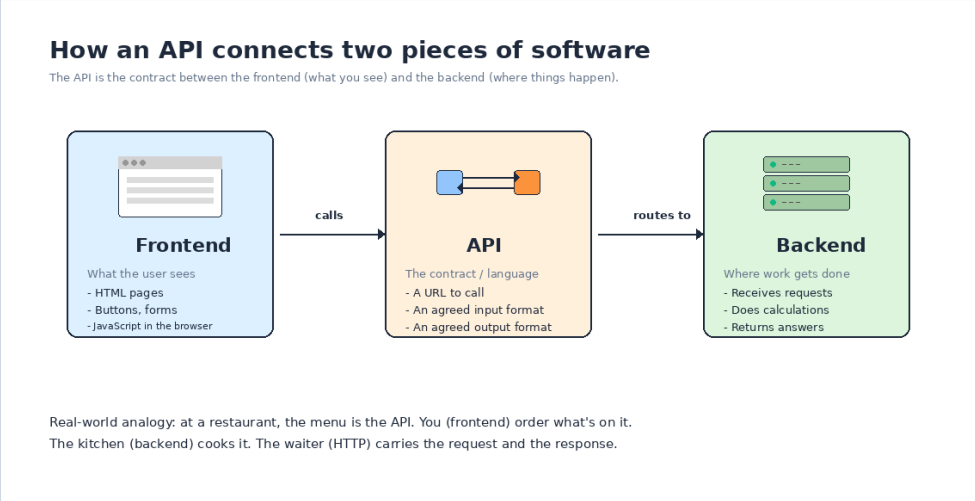


*Figure 3 - The frontend, the API, and the backend - three separate parts of the same app.*

## The frontend

The **frontend** is everything the user can see and click. It runs in the user's web browser. For our app it is just one HTML file with some CSS and JavaScript.

- It draws the input box, button, and result area.
- It listens for clicks on the Convert button.
- It calls the API when the button is clicked.
- It displays whatever the API sends back.

The frontend does **no** calculation. It does not know how to convert Celsius to Fahrenheit. It just collects input, asks the API, and shows the answer.

## The backend

The **backend** is the program that runs on a server somewhere. For now "somewhere" is your own laptop. The backend is where the work happens.

- It receives requests from the frontend.
- It does whatever calculation or processing is needed.
- It sends back a response.

For our app the backend's job is tiny - take a Celsius number, do the math, return Fahrenheit. But the SAME pattern handles much bigger jobs: querying databases, sending emails, processing payments, calling other APIs. The shape never changes.

## The API

The **API** is the agreed-upon way the frontend and backend talk. It is not a third program - it is a contract between the first two. Specifically, our API will define:

- **The URL**: *POST http://localhost:4000/api/convert*
- **The input format**: a JSON object like *{ "celsius": 25 }*
- **The output format**: a JSON object like *{ "celsius": 25, "fahrenheit": 77 }*

That is the whole contract. Three things. Once both sides agree, the frontend and backend can be built independently and they will fit together perfectly.

## Why three pieces?

The same question came up in chapter 1, but it deserves a second look now that you can see the pieces:

- **The browser cannot be trusted**. A clever user can change anything in the browser. So sensitive logic must live on the server.
- **One backend, many clients**. We could build a mobile app tomorrow that talks to the same API. The backend never has to change.
- **Each piece scales differently**. If the frontend is slow, fix the JavaScript. If the backend is slow, optimise the server. They do not affect each other.
- **Different teams, different skills**. The frontend team needs design + JavaScript skills. The backend team needs server + database skills. The API lets them work in parallel.

## Where things run - the ports

When you develop locally, both your frontend and backend run on your own laptop, but they each listen on a different **port** - a numbered "door" into your computer.

| Piece | URL | Notes |
|---|---|---|
| Frontend (React/Vite or HTML) | http://localhost:5173 | Vite default. Or 8080 for plain HTML via XAMPP. |
| Backend (Express) | http://localhost:4000 | We will pick this. Could be 3000, 5000, anything free. |
| Database (we don't use one today) | (no port, no DB) | Just here for context. MySQL would be 3306. |

> **TIP**
>
> Ports under 1024 are reserved (ports 80 and 443 are HTTP and HTTPS). For development, anything 3000+ is safe. Just don't pick the same port for two different services.

# Chapter 3 - HTTP - The Language of the Web

**HTTP** stands for **Hypertext Transfer Protocol**. It is the language browsers and servers use to talk to each other. Every API call you will ever make - from converting temperatures to processing credit card payments - travels over HTTP.

## How HTTP conversations work

HTTP is a **request and response** protocol. The client (browser, in our case) sends a request. The server sends a response. That is it. Every interaction is one request and one response.

## HTTP methods (verbs)

Every HTTP request starts with a **method** (also called a verb) that tells the server what kind of action you want. There are about a dozen methods, but in practice you use four:

| Method | Meaning | Example |
|---|---|---|
| GET | Read data. No side effects. | Get the menu of tours. |
| POST | Create something new. | Submit a booking. |
| PUT | Update something that exists. | Change a user's email. |
| DELETE | Remove something. | Cancel a booking. |

Our temperature converter does a calculation, not a read or save. We will use **POST** because we are sending data in the request body. Some APIs use GET for read-only calculations - both are valid choices.

## Anatomy of a request

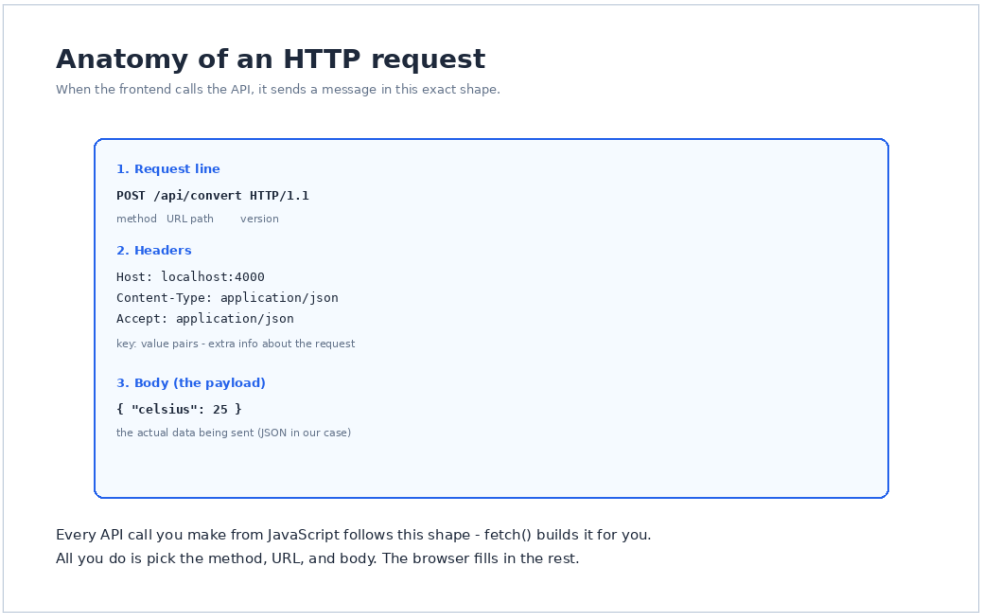


*Figure 4 - Three parts of every HTTP request.*

### Reading each part

• **Request line**: tells the server what you want and where. *POST /api/convert HTTP/1.1* means "a POST request to /api/convert, using HTTP version 1.1".

• **Headers**: extra information about the request. *Content-Type: application/json* means "the body is JSON". *Host* tells the server which website on this IP the request is for.

• **Body**: the actual data you are sending. For our app, *{ "celsius": 25 }*. GET requests don't usually have a body; POST and PUT do.

> **TIP**
>
> When you call *fetch(url, options)* in JavaScript, you specify the method, headers, and body in the options object. fetch() builds the actual request behind the scenes. We will see this in chapter 12.

## Anatomy of a response


*Figure 5 - The matching three-part response from the server.*

• **Status line**: the server's verdict. *HTTP/1.1 200 OK* means "all good". *404 Not Found* means the URL doesn't exist. We will look at every common code in chapter 5.

• **Headers**: same idea as in the request. *Content-Type* tells the browser what kind of data is coming back.

• **Body**: the actual answer. For our API, *{ "celsius": 25, "fahrenheit": 77 }*.

## Why JSON for the body?

HTTP can carry any kind of data in its body - HTML, plain text, images, video. For APIs, we use **JSON** because it is compact, easy for both sides to parse, and JavaScript-friendly. The next chapter is all about JSON.

# Chapter 4 - JSON - How Data Travels

**JSON** stands for **JavaScript Object Notation**. Despite the name, JSON is used by every programming language - Python, PHP, Java, Go, Rust, you name it. It is the most common data format on the web.

JSON's job is to take data and turn it into a string of text that can be sent over the network or saved in a file. The receiving side parses the text back into data they can use.

## The whole language - six data types

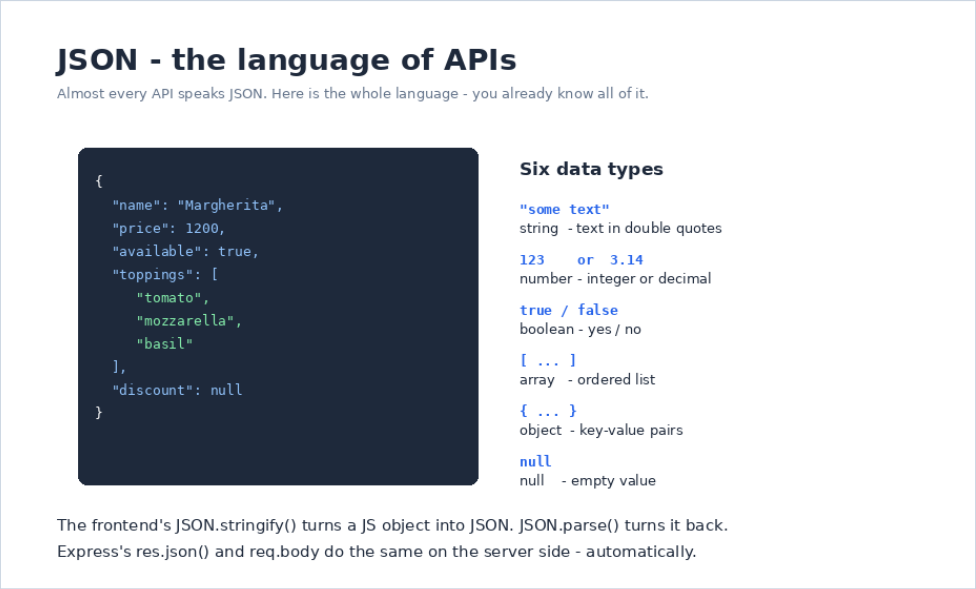


*Figure 6 - JSON has just six data types. You already know them all from JavaScript.*

- **String** - text in double quotes: *"hello"*
- **Number** - integer or decimal: *42* or *3.14*
- **Boolean** - *true* or *false*
- **Array** - ordered list in square brackets: *[1, 2, 3]*
- **Object** - key-value pairs in curly braces: *{ "name": "Saman" }*
- **Null** - the empty value: *null*

That is it. The whole format. No classes, no functions, no comments, no fancy syntax - just data.

## JSON in JavaScript

JavaScript has two built-in helpers for working with JSON. These are the only two functions you ever need:

```
// 1. JS object -> JSON string
const obj = { celsius: 25, hot: true };
const text = JSON.stringify(obj);
// text is now: '{"celsius":25,"hot":true}'


// 2. JSON string -> JS object
const text = '{"celsius":25,"hot":true}';
const obj = JSON.parse(text);
// obj is now: { celsius: 25, hot: true }
```

When you send data to an API, you call **JSON.stringify**. When the API sends data back, you call **JSON.parse**. Or use **response.json()** from fetch, which calls parse for you.

## Common JSON gotchas

- **Double quotes only**. Single quotes are not valid JSON. *'celsius'* won't parse - it must be *"celsius"*.
- **No trailing commas**. *{ "a": 1, }* is invalid. *{ "a": 1 }* is fine.
- **Keys must be strings**. Even keys that look like numbers: *{ "42": "answer" }*, with quotes.
- **No comments allowed**. JSON is data, not code.
- **No undefined**. JavaScript has *undefined*; JSON does not. Use *null* instead, or omit the key.

> **TIP**
>
> When debugging, paste your JSON into **jsonlint.com**. It will tell you exactly which character is wrong. The same for any JSON validator built into your editor.

## Why JSON beats other formats

| Format | Verdict |
|---|---|
| JSON | Compact, readable, every language supports it. Modern winner. |
| XML | Verbose, complex. Common in legacy enterprise APIs. |
| CSV | Good for tables. Bad for nested data. |
| Form data | Used in old HTML forms. Limited types. |
| Protocol Buffers | Binary, very fast. Used in high-performance backends. |

For 99 percent of web APIs in 2026, the answer is JSON.

# Chapter 5 - HTTP Status Codes - The Grammar

Every HTTP response carries a 3-digit **status code**. It tells the client whether the request worked, and if not, what kind of failure it was. Knowing these codes is one of the highest-leverage skills in web development.

## The five families

The first digit puts the response into one of five families. You can guess the meaning of any new code from its first digit:

| Range | Family | Meaning |
|---|---|---|
| 1xx | Informational | Rare. Used for protocol-level signals. |
| 2xx | Success | Worked. |
| 3xx | Redirect | Look elsewhere - the resource moved. |
| 4xx | Client error | The request was wrong (your fault). |
| 5xx | Server error | The server crashed (their fault). |

We will focus on 2xx, 4xx, and 5xx - the codes you encounter every single day.

## The codes you must know

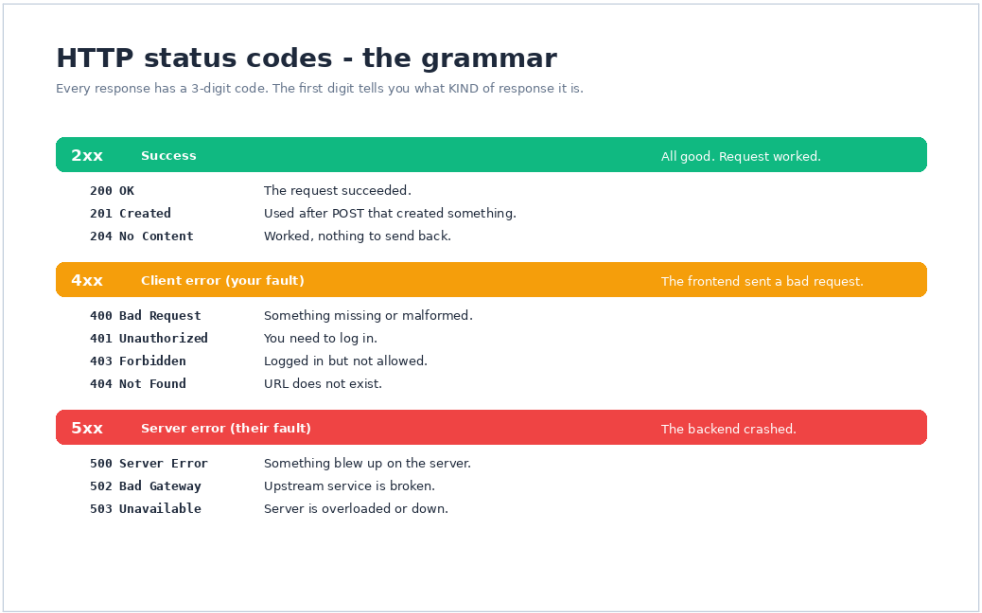


*Figure 7 - The HTTP status codes you will see in practice.*

## Reading the diagram

### 2xx Success

• **200 OK** - default success. The request worked.

• **201 Created** - used after a POST that created a new resource. "I made it. Here is its location."

• **204 No Content** - used after a DELETE that succeeded. "Done. Nothing to send back."

### 4xx Client error - the frontend's fault

• **400 Bad Request** - your request was malformed or missing data. "You sent garbage."

• **401 Unauthorized** - you are not logged in. The name is misleading - it really means "unauthenticated".

• **403 Forbidden** - you ARE logged in but you cannot do this action. "I know who you are; you are not allowed."

• **404 Not Found** - the URL does not match any known endpoint. The most famous code on the web.

• **422 Unprocessable Entity** - your data is well-formed but doesn't pass validation. "You sent JSON correctly, but the email is missing."

• **429 Too Many Requests** - rate-limited. "Slow down."

### 5xx Server error - the backend's fault

• **500 Internal Server Error** - the server crashed while handling your request. Probably a bug in the backend code.

• **502 Bad Gateway** - one server tried to ask another server for help, and that other server failed.

• **503 Service Unavailable** - the server is overloaded or down for maintenance.

• **504 Gateway Timeout** - one server waited too long for another to respond.

> **TIP**
>
> **4xx is your fault, 5xx is their fault.** When you debug an API problem, this is the first question to ask. Did the frontend send something bad (4xx), or did the backend itself crash (5xx)? It changes which code you need to fix.

## How to set status codes in Express

We will see this in detail in chapter 9. As a teaser:

```
// success
res.status(200).json({ celsius: 25, fahrenheit: 77 });

// missing input
res.status(400).json({ error: 'celsius is required' });

// not a number
res.status(422).json({ error: 'celsius must be a number' });

// our code crashed
res.status(500).json({ error: 'something went wrong' });
```

## How to read status codes in fetch()

```
const response = await fetch('/api/convert', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ celsius: 25 }),
});

if (response.status === 200) {
    const data = await response.json();
    console.log(data.fahrenheit);   // 77
}
if (response.status === 400) {
    const err = await response.json();
    console.error('Bad request:', err.error);
}
if (response.status >= 500) {
    console.error('Server is having problems, try again later');
}

// Or use the simple .ok shortcut: response.ok is true for 2xx
if (response.ok) {
    // success path
} else {
    // some kind of error
}
```

> **WARNING**
>
> Always handle the failure case in your frontend code. APIs fail. The network goes down. Servers crash. Hard-coding for the happy path is how you ship apps that look broken to users.

# Chapter 6 - Tools You Need - Node and a Code Editor

We have covered all the theory. Time to build. The good news: this project needs only two tools, both free, both quick to install.

## What you need

| Tool | Why we need it | Where to get it |
|---|---|---|
| Node.js 22+ | Runs the backend (Express server) and a tiny static frontend server. | nodejs.org |
| VS Code | Code editor with great JavaScript support. | code.visualstudio.com |

That is the whole list. No database, no Apache, no Docker - just Node and an editor.

## Installing Node.js

**Node.js** lets you run JavaScript outside a browser. It is what powers our backend.

1. Visit **nodejs.org**.
2. Click the big green button labelled **LTS** (currently 22 or 24).
3. Run the installer with default options. Tick "Add to PATH" if asked.
4. Open a NEW terminal and run **node --version**. It should print v22 or higher.
5. Run **npm --version**. *npm* is Node's package manager and ships with Node.

## Installing VS Code

1. Visit **code.visualstudio.com**.
2. Click **Download**.
3. Run the installer.

### VS Code extensions worth installing

- **Prettier** - auto-formats your code on save.
- **Thunder Client** - lets you test APIs from inside VS Code. We will use it heavily in chapter 10.
- **ESLint** - flags common JavaScript mistakes.

## Quick check - both tools at once

Open a terminal and run:

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

> **NOTE**
>
> If a command says "not found", close the terminal and open a fresh one - tools added to PATH only show up in NEW terminals. If it still fails, re-install with the "Add to PATH" checkbox ticked.

## The folder structure

Our project has two folders: one for the backend, one for the frontend. Open a terminal:

```
cd ~/code            # or wherever you keep projects
mkdir api-demo
cd api-demo
mkdir backend frontend
```

By the end of the tutorial, the structure will look like:

```
api-demo/
+-- backend/
|   +-- node_modules/      # auto-generated, never edit
|   +-- index.js           # the Express server
|   `-- package.json       # backend dependencies
`-- frontend/
    +-- index.html         # the UI
    +-- style.css          # the styles
    `-- app.js             # the fetch() code
```

We will build the backend first. Once it works (tested on its own), we'll build the frontend that calls it.

# Chapter 7 - Building the Backend with Express

Time to write our backend. We will use **Express** - the most popular web framework for Node.js. Tens of thousands of real APIs run on it. Express turns a few lines of code into a working web server.

## Step 1 - Initialise the npm project

From the **backend/** folder:

```
cd ~/code/api-demo/backend
npm init -y
```

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

## Step 2 - Install Express and CORS

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

### What each package does

| Package | Purpose |
|---|---|
| express | The web framework. Routes URLs to handler functions. |
| cors | Allows the frontend (different port) to call the backend. Without this, browsers block the request. |
| nodemon | Dev tool. Restarts the server automatically when you save. |

> **TIP**
>
> **What is CORS?** Cross-Origin Resource Sharing is a browser security rule that blocks JavaScript on one website from calling another website. Since our frontend (port 5173) and backend (port 4000) are different, the browser thinks they are different websites. The cors package tells the browser "this combination is allowed". Without it, every fetch() would fail.

## Step 3 - Tell Node we want modern syntax

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

```
{
    "name": "backend",
    "version": "1.0.0",
    "type": "module",
    "main": "index.js",
    "scripts": {
        "dev": "nodemon index.js",
        "start": "node index.js"
    },
    "dependencies": {
        "cors": "^2.8.5",
        "express": "^4.21.0"
    }
}
```

The key parts: **"type": "module"** enables import/export. The **scripts** let us run *npm run dev* (uses nodemon, restarts on save) or *npm start* (plain node).

## Step 4 - The smallest Express server

Create **backend/index.js**:

```
// backend/index.js
import express from 'express';
import cors from 'cors';

const app = express();
const PORT = 4000;

// Middleware
app.use(cors());            // allow requests from the frontend
app.use(express.json());    // parse JSON request bodies

// A test route - proves the server works
app.get('/api/health', (req, res) => {
    res.json({ status: 'ok' });
});

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

### Reading line by line

- **import express from 'express'** - load the framework.
- **const app = express()** - create the Express application.
- **app.use(cors())** - tell Express to add CORS headers to every response.
- **app.use(express.json())** - tell Express to parse incoming JSON bodies. Without this, *req.body* would be undefined.
- **app.get('/api/health', (req, res) => ...)** - register a route. "When a GET request comes in for /api/health, run this function."

- **req** - the incoming request (URL, headers, body).
- **res** - the outgoing response. We send back JSON.
- **res.json({...})** - send a JSON response. Express handles the Content-Type header for us.
- **app.listen(PORT, ...)** - actually start the HTTP server on port 4000.

## Step 5 - Run it!

```
npm run dev
```

You should see:

```
[nodemon] starting `node index.js`
Server running on http://localhost:4000
```

Open **http://localhost:4000/api/health** in your browser. You should see **{"status":"ok"}**. **That** is your first working API. Empty, but real.

> **TIP**
>
> Keep this terminal open. nodemon watches for file changes - every time you save, it restarts the server. No manual restart needed.

## What is middleware?

Notice the two **app.use(...)** calls. Anything passed to *app.use* is a **middleware** - a function that runs for every request before the route handler. Middleware is how Express handles cross-cutting concerns:

- **cors** - adds the right CORS headers.
- **express.json()** - parses JSON bodies into *req.body*.
- **express.urlencoded()** - parses form data (we don't need this).
- **express.static('public')** - serves static files.
- Custom middleware - logging, authentication, rate-limiting.

Middleware runs in order. Always put cors() and express.json() BEFORE your routes - otherwise routes won't see the parsed body or the CORS headers.

# Chapter 8 - The Convert Endpoint - Reading the Input

Time for the real work. We will add a **POST /api/convert** route that takes a Celsius value and returns Fahrenheit.

## Step 1 - Add the route

Open **backend/index.js** and add this above the health route:

```
// POST /api/convert - converts Celsius to Fahrenheit
app.post('/api/convert', (req, res) => {
    const celsius = req.body.celsius;
    const fahrenheit = celsius * 9 / 5 + 32;

    res.json({
        celsius: celsius,
        fahrenheit: fahrenheit,
    });
});
```

Save. nodemon restarts the server. The endpoint is live.

### Reading the route

- **app.post('/api/convert', ...)** - register a POST handler for /api/convert.
- **req.body.celsius** - read the *celsius* field from the request body. **express.json()** middleware (chapter 7) already parsed the JSON for us.
- **celsius * 9 / 5 + 32** - the conversion formula.
- **res.json({...})** - send the response.

Notice we send back BOTH celsius and fahrenheit. This is a common API pattern - return the full picture so the client can display "25 deg C = 77 deg F" without remembering what they sent.

## Step 2 - Try it from the browser?

You cannot easily test a POST from the browser address bar - browsers send GET when you type a URL. To test our POST, we need a tool. Let's use **curl** from the terminal first, then **Thunder Client** in chapter 10.

```
curl -X POST http://localhost:4000/api/convert \
     -H "Content-Type: application/json" \
     -d '{"celsius": 25}'
```

Response:

```
{"celsius":25,"fahrenheit":77}
```

### Reading the curl command

- **-X POST** - use the POST method.
- **-H "Content-Type: application/json"** - add a header saying the body is JSON.
- **-d '...'** - the request body. Single quotes around the JSON to avoid shell escaping headaches.

> **TIP**
>
> **curl** ships with most operating systems. On Windows you may need to use *curl.exe* in PowerShell or CMD. It is the universal tool every backend developer reaches for to test an API quickly.

## Step 3 - Notice what happens with bad input

Try sending a non-number:

```
curl -X POST http://localhost:4000/api/convert \
     -H "Content-Type: application/json" \
     -d '{"celsius": "hello"}'
```

Response:

```
{"celsius":"hello","fahrenheit":NaN}
```

**NaN** means "Not a Number" - the multiplication *"hello" * 9 / 5 + 32* doesn't produce a number. We returned a 200 OK with garbage in it. **That is a bug.**

### What about no body at all?

```
curl -X POST http://localhost:4000/api/convert
```

Without the Content-Type header and body, our handler crashes because *req.body* is undefined and we try to read *undefined.celsius*. The user gets a 500 error.

Both of these problems are **validation issues**. The next chapter is about fixing them properly.

# Chapter 9 - Validation and Error Responses

Real APIs cannot trust user input. The frontend might send wrong data, and a malicious user might bypass the frontend entirely. So the backend must validate every request and respond clearly when something is wrong.

## What we want

- If **celsius** is missing - return 400 with a clear message.
- If **celsius** is not a number - return 422 with a clear message.
- If **celsius** is a valid number - return 200 with the result.

## Step 1 - Update the convert route

Replace the route in **index.js** with this validating version:

```
// POST /api/convert - converts Celsius to Fahrenheit (with validation)
app.post('/api/convert', (req, res) => {
    // 1. The body must exist
    if (!req.body) {
        return res.status(400).json({
            error: 'Request body is required',
        });
    }

    // 2. celsius field must be present
    if (req.body.celsius === undefined) {
        return res.status(400).json({
            error: 'celsius is required',
        });
    }

    const celsius = Number(req.body.celsius);

    // 3. celsius must be a real number
    if (Number.isNaN(celsius)) {
        return res.status(422).json({
            error: 'celsius must be a number',
        });
    }

    // 4. (Optional) sanity range
    if (celsius < -273.15) {
        return res.status(422).json({
            error: 'celsius cannot be below absolute zero',
        });
    }

    // All good - do the math
    const fahrenheit = celsius * 9 / 5 + 32;

    res.json({
        celsius: celsius,
        fahrenheit: Math.round(fahrenheit * 100) / 100,
    });
});
```

### Reading the validation

• **return res.status(400).json({...})** - send a 400 response and STOP. *return* is critical - without it, the function would continue past the validation.

• **req.body.celsius === undefined** - check the field is present. We don't say *!req.body.celsius* because that would also reject the legitimate value 0.

• **Number(req.body.celsius)** - convert string "25" to number 25. If it can't be converted, we get NaN.

• **Number.isNaN(celsius)** - reliable way to check for NaN. Don't use plain *isNaN*; it has confusing edge cases.

• **celsius < -273.15** - the laws of physics. Below absolute zero is impossible. Always think about real-world constraints.

• **Math.round(fahrenheit * 100) / 100** - round to 2 decimal places.

> **TIP**
>
> Notice the pattern: **validate first, do the work last**. Each validation check returns early on failure. The actual logic only runs once everything has passed. This is called the "guard clause" pattern - cleaner than nested if statements.

## Step 2 - Test the error paths

### Test 1: empty body

```
curl -X POST http://localhost:4000/api/convert \
     -H "Content-Type: application/json" \
     -d '{}'

# Response: 400
# {"error":"celsius is required"}
```

### Test 2: not a number

```
curl -X POST http://localhost:4000/api/convert \
     -H "Content-Type: application/json" \
     -d '{"celsius": "hello"}'

# Response: 422
# {"error":"celsius must be a number"}
```

### Test 3: too cold

```
curl -X POST http://localhost:4000/api/convert \
     -H "Content-Type: application/json" \
     -d '{"celsius": -300}'

# Response: 422
# {"error":"celsius cannot be below absolute zero"}
```

### Test 4: a string number works

```
curl -X POST http://localhost:4000/api/convert \
     -H "Content-Type: application/json" \
     -d '{"celsius": "25"}'

# Response: 200
# {"celsius":25,"fahrenheit":77}
```

Number("25") returns 25. We accept it. Many real APIs are this forgiving - if the input *can* be a number, accept it. Strict APIs would reject this with 422 "celsius must be a number, not a string". Both choices are valid - just be consistent.

## Why error messages matter

Compare two error responses:

```
// Bad
{ "error": "validation failed" }

// Better
{ "error": "celsius must be a number" }

// Best
{
    "error": "validation failed",
    "details": [
        { "field": "celsius", "message": "must be a number" }
    ]
}
```

The first message is useless to debug. The second tells the user exactly what to fix. The third lets the frontend show field-level errors in the UI. Pick the level of detail that matches your needs.

> **TIP**
>
> When something goes wrong on a real production API, your error message is the only signal the frontend developer has. Future-you will thank past-you for clear, specific error messages.

# Chapter 10 - Testing the API with Thunder Client

Curl works. But it gets cumbersome for repeated tests. Time to introduce **Thunder Client**, a free VS Code extension that lets you save requests and inspect responses inside your editor. (You may also know **Postman** - same idea, standalone app.)

## Step 1 - Install Thunder Client

If you didn't install it in chapter 6: open VS Code, go to Extensions (Ctrl+Shift+X or Cmd+Shift+X), search "Thunder Client", click Install. A lightning-bolt icon appears in the left sidebar.

## Step 2 - Make sure your server is running

In a terminal:

```
cd ~/code/api-demo/backend
npm run dev
```

Leave it running. nodemon will keep watching.

## Step 3 - Test the happy path

1. Click the lightning-bolt icon in VS Code's left sidebar.
2. Click **New Request**.
3. Set the method to **POST**.
4. URL: **http://localhost:4000/api/convert**
5. Click the **Body** tab below the URL.
6. Choose **JSON** as the body type.
7. Paste: *{ "celsius": 25 }*
8. Click **Send**.

On the right you should see:

```
Status: 200 OK
Time: 12 ms

{
    "celsius": 25,
    "fahrenheit": 77
}
```

> **TIP**
>
> Notice the time: 12 ms. That is how long the round trip took from VS Code to the server and back. Real production APIs should be 50-200 ms locally. Slower means investigation.

## Step 4 - Test error paths

The whole point of validation is making sure errors look right. Save several Thunder Client requests so you can re-test every time you change the code:

| Body | Expected status | Expected message |
|---|---|---|
| {} | 400 | celsius is required |
| { "celsius": "hello" } | 422 | celsius must be a number |
| { "celsius": -300 } | 422 | celsius cannot be below absolute zero |
| { "celsius": 0 } | 200 | fahrenheit: 32 |
| { "celsius": -40 } | 200 | fahrenheit: -40 (the magic temperature where C and F meet!) |

## Step 5 - Save your requests as a collection

In Thunder Client, click **Collections** in the left sidebar -> **New Collection** -> name it "API Demo". Drag your saved requests into it. Now they live together and you can re-run them all with one click - perfect for catching regressions when you change code.

## Why test before building the frontend?

If your API has bugs and you start building the frontend, you won't know whether bugs are in the API, the frontend, or the connection between them. Solidify the API first, then build the frontend on top with confidence.

> **TIP**
>
> Always test the backend BEFORE building the frontend. If your API is broken, no amount of React magic will save you.

## Other tools you'll meet later

| Tool | Notes |
|---|---|
| Postman | Standalone app, the original. Heavier than Thunder Client. |
| Insomnia | Open-source Postman alternative. Slick UI. |
| Hoppscotch | Free, browser-based. Good for quick checks. |
| HTTPie | Beautiful command-line tool. Like curl but easier to read. |

All of them do the same job - send HTTP requests, show responses. Pick whichever you like. We'll stick with Thunder Client because it's already in VS Code.

# Chapter 11 - Building the Frontend HTML and CSS

The backend works and is tested. Time to build the user interface that calls it. Pure HTML, CSS, and JavaScript - no frameworks, no build tools. Just three files that any browser can run.

## Step 1 - Create index.html

In your **frontend/** folder, create **index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
    <title>Temperature Converter</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <h1>Temperature Converter</h1>
    </header>

    <main class="container">
        <div class="card">
            <h2>Celsius -&gt; Fahrenheit</h2>
            <p class="hint">
                Enter a temperature in Celsius. Click Convert.
            </p>

            <label for="celsius">Celsius</label>
            <input
                type="number"
                id="celsius"
                placeholder="e.g. 25"
                step="any"
            >

            <button id="convert-btn">Convert</button>

            <div id="result" class="result empty">
                Result will appear here.
            </div>
        </div>
    </main>

    <script src="app.js"></script>
</body>
</html>
```

### Reading the HTML

- **<input id="celsius">** - the value the user types. We will read it from JavaScript using its id.
- **type="number"** - the browser shows a numeric keypad on phones and prevents non-numeric typing. **step="any"** allows decimals (e.g. 22.5).
- **<button id="convert-btn">** - we will attach a click handler in JavaScript.

• ** ** - empty for now, JavaScript will fill it after the API call.

• **<script src="app.js">** - load our JavaScript at the END of body so the elements above it exist by the time the script runs.

## Step 2 - Create style.css

```
/* style.css */
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
}

body {
    font-family: system-ui, -apple-system, sans-serif;
    background: #f5f7fa;
    color: #1f2937;
    line-height: 1.5;
}

header {
    background: #2563EB;
    color: white;
    padding: 1rem 1.5rem;
}

header h1 {
    font-size: 1.4rem;
}

.container {
    max-width: 540px;
    margin: 3rem auto;
    padding: 0 1rem;
}

.card {
    background: white;
    border: 1px solid #e2e8f0;
    border-radius: 12px;
    padding: 2rem;
}

.card h2 {
    margin-bottom: 0.5rem;
}

.hint {
    color: #64748b;
    font-size: 0.9rem;
    margin-bottom: 1.5rem;
}

label {
    display: block;
    font-weight: bold;
    margin-bottom: 0.5rem;
}

input {
    display: block;
    width: 100%;
    padding: 0.75rem 1rem;
    border: 1px solid #cbd5e1;
    border-radius: 8px;
    font-size: 1rem;
    margin-bottom: 1.5rem;
}
```

```
input:focus {
    outline: 2px solid #2563EB;
    border-color: #2563EB;
}

button {
    display: block;
    width: 100%;
    padding: 0.85rem;
    background: #2563EB;
    color: white;
    border: none;
    border-radius: 8px;
    font-size: 1rem;
    font-weight: bold;
    cursor: pointer;
    margin-bottom: 1.5rem;
}

button:hover { background: #1d4ed8; }
button:disabled { background: #94a3b8; cursor: not-allowed; }

.result {
    padding: 1rem;
    border-radius: 8px;
    border: 1px solid #e2e8f0;
}

.result.empty {
    background: #f1f5f9;
    color: #64748b;
}

.result.success {
    background: #d1fae5;
    border-color: #10b981;
    color: #065f46;
}

.result.error {
    background: #fee2e2;
    border-color: #ef4444;
    color: #991b1b;
}

.result .big {
    font-size: 1.6rem;
    font-weight: bold;
    margin-top: 0.25rem;
}
```

## Step 3 - Run the frontend

We need a tiny web server to serve the files. The simplest is **npx serve** - it ships with Node and starts a static server in any folder. Open a NEW terminal:

```
cd ~/code/api-demo/frontend
npx serve
```

It will print something like:

```
Serving!

   Local:    http://localhost:3000
   Network:  http://192.168.1.5:3000
```

Open **http://localhost:3000** in your browser. You should see your blue header, the form card, the input, the Convert button, and the empty result placeholder. Beautiful but lifeless - the button does nothing yet. Chapter 12 fixes that.

> **TIP**
>
> You now have THREE things running: the backend on :4000, the frontend on :3000, and the editor. Three terminals. Welcome to web development.

# Chapter 12 - Connecting the Frontend with fetch()

Time to make the button work. We will use the browser's built-in **fetch()** function to call our API. Fetch is the modern, standard way to make HTTP requests from JavaScript.

## Step 1 - Create app.js

In your **frontend/** folder, create **app.js**:

```
// frontend/app.js

// Grab references to the elements we will use
const input = document.getElementById('celsius');
const button = document.getElementById('convert-btn');
const resultEl = document.getElementById('result');

// Listen for clicks on the Convert button
button.addEventListener('click', async () => {
    const celsius = input.value;

    // Disable the button while we wait
    button.disabled = true;
    resultEl.className = 'result empty';
    resultEl.textContent = 'Converting...';

    try {
        const response = await fetch('http://localhost:4000/api/convert', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ celsius: Number(celsius) }),
        });

        const data = await response.json();

        if (response.ok) {
            // Success - show the result
            resultEl.className = 'result success';
            resultEl.innerHTML = `
                <div>${data.celsius} deg C =</div>
                <div class="big">${data.fahrenheit} deg F</div>
            `;
        } else {
            // The server returned an error status (4xx or 5xx)
            resultEl.className = 'result error';
            resultEl.textContent = data.error || 'Something went wrong';
        }
    } catch (err) {
        // Network error - couldn't reach the server at all
        resultEl.className = 'result error';
        resultEl.textContent =
            'Cannot reach the server. Is the backend running?';
    } finally {
        button.disabled = false;
    }
});
```

## Reading the fetch call line by line

### Setup

• **document.getElementById(...)** - find the HTML elements by their id. We grab references once, at the top, so we can use them throughout.

• **button.addEventListener('click', async () => ...)** - run the function whenever the button is clicked. We use **async** so we can use **await** inside.

### While the request is in flight

• **button.disabled = true** - prevent double-clicks while the API call is happening.

• **resultEl.textContent = 'Converting...'** - show a loading message. Real API calls take time; users need feedback.

### The fetch() call itself

• **fetch(url, options)** - returns a **Promise**. We await it to get a Response object.

• **method: 'POST'** - matches our backend's *app.post(...)* route.

• **headers: { 'Content-Type': 'application/json' }** - tells the server the body is JSON. Without this, *express.json()* on the server won't parse the body.

• **body: JSON.stringify({ celsius: Number(celsius) })** - convert our JS object to a JSON string. Note: **fetch does NOT do this for you**. You must call JSON.stringify yourself.

• **Number(celsius)** - convert the string from the input field to a real number. Otherwise the server gets the string "25" and we get extra string-handling we don't need.

### Reading the response

• **await response.json()** - parse the response body as JSON. Returns a Promise that resolves to the actual data.

• **response.ok** - true for status 200-299, false for everything else. Quick way to check success.

• **response.status** - the actual code (200, 400, 500, ...).

### Three kinds of failure

Notice we handle three different cases:

• **response.ok = true** - 2xx success. Show the result.

• **response.ok = false** - the server responded but with an error status (400, 422, 500). The body usually has details. Show the error.

• **catch (err)** - the network failed entirely. The server is down, the user is offline, DNS failed. Fetch never even got a response.

> **WARNING**
>
> **fetch() does NOT throw on HTTP errors.** A 404 or 500 response is still a successful fetch - the server replied. fetch() only throws when there is no response at all (network down). This trips up everyone the first time.

## What is async/await again?

**fetch** returns a **Promise** - an object that represents a future value. The two ways to use a Promise:

```
// Old way - .then() chains
fetch(url)
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.error(err));

// New way - async/await (much cleaner)
async function doIt() {
    try {
        const response = await fetch(url);
        const data = await response.json();
        console.log(data);
    } catch (err) {
        console.error(err);
    }
}
```

Both are equivalent. We use async/await because it reads top-to-bottom like normal code, with a try/catch for errors. The chained version works fine - you'll see it in older tutorials.

# Chapter 13 - Showing the Result and Handling Errors

Our app already shows results in chapter 12's code. Let's now really walk through the user-facing details - the loading state, the success display, the error display, and what each looks like.

## Try the happy path first

1. Make sure the backend is still running (*npm run dev* in **backend/**).
2. Make sure the frontend is still running (*npx serve* in **frontend/**).
3. Visit **http://localhost:3000**.
4. Type **25** in the input field.
5. Click **Convert**.
6. You should see **25 deg C = 77 deg F** in a green success box.

If you see this, congratulations - your frontend successfully called your backend through the API. The full round trip works.

## Try the error paths

### 1. Empty input

Click Convert without typing anything. *input.value* is empty string. *Number("")* is 0, which is a valid number, so the API returns 32 deg F. Maybe not what we want!

### 2. Letters in the input

Because we used *type="number"*, the browser blocks letters. But if a user pastes "hello", *input.value* becomes empty (the browser silently rejects). So this case is mostly handled by HTML.

### 3. Backend down

Stop the backend server (Ctrl+C in its terminal). Click Convert. After a couple of seconds you should see **"Cannot reach the server"** in red. Restart the backend; click Convert again - works.

## Adding client-side validation

We can stop bad inputs from even reaching the backend. Add this guard at the top of the click handler in **app.js**:

```
button.addEventListener('click', async () => {
    const celsius = input.value;

    // Client-side validation
    if (celsius === '') {
        resultEl.className = 'result error';
        resultEl.textContent = 'Please enter a Celsius value.';
        return;
    }

    if (Number.isNaN(Number(celsius))) {
        resultEl.className = 'result error';
        resultEl.textContent = 'That is not a valid number.';
        return;
    }

    // ... rest of the function as before
});
```

> **WARNING**
>
> **Client-side validation is a UX improvement, not a security control.** It gives users instant feedback. But a malicious user can disable JavaScript and still call your API. **Always** validate on the server too. Both, always. We did this in chapter 9.

## Inspecting the request in DevTools

Open Chrome DevTools (F12) and go to the **Network** tab. Click Convert. You will see one row appear - the API call. Click it to see:

- **Headers** tab: status code, request URL, request and response headers.
- **Payload** tab: the JSON you sent.
- **Response** tab: the JSON you got back.
- **Timing** tab: how long each phase took.

DevTools' Network tab is your most important debugging tool when calling an API. When something doesn't work, look there first. The status code will tell you immediately whose fault it is.

## Showing helpful errors

Compare these two error displays:

- **Bad**: "Error" - useless to the user.
- **Better**: "Cannot reach the server. Is the backend running?" - tells them what's wrong AND what to check.
- **Best**: also include a Retry button.

Always show the backend's error message when there is one - it is more specific than anything we could write generically.

# Chapter 14 - The Full Request Lifecycle - Recap

We have built the whole thing. Let's walk through one click from end to end and watch each part of the system do its job. If you can trace this in your head, you understand APIs.

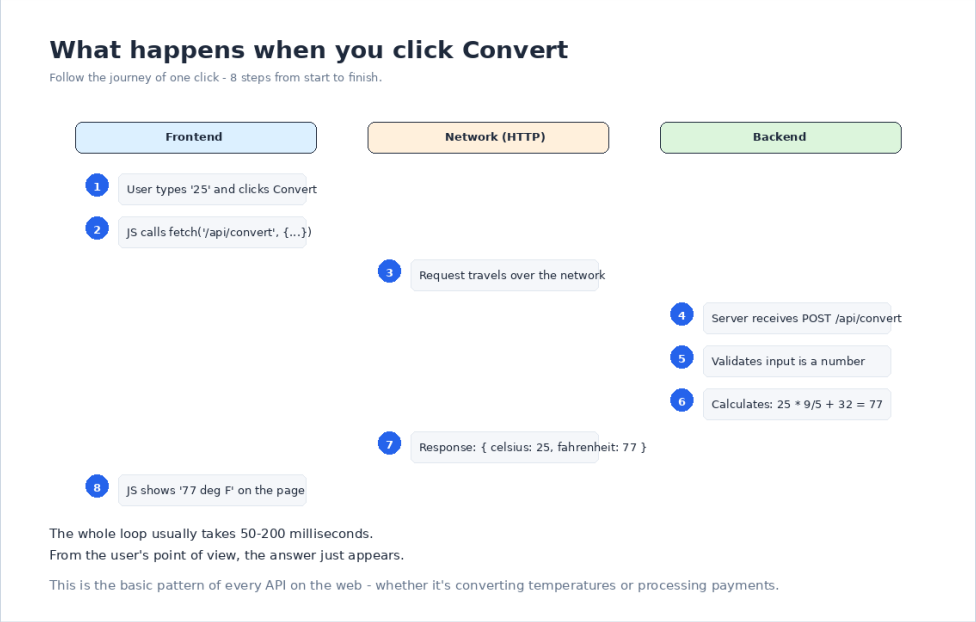


*Figure 8 - Eight steps from button click to result on screen.*

## Step by step with our actual code

### Step 1: User types and clicks

The user types **25** in the <input id="celsius"> and clicks the <button id="convert-btn">. The browser fires a click event.

### Step 2: JavaScript calls fetch()

Our **app.js** click handler runs. It reads *input.value*, builds an options object with method POST, the JSON body, and the Content-Type header, then calls **fetch('http://localhost:4000/api/convert', options)**.

### Step 3: Request travels over the network

The browser sends an HTTP request to localhost:4000. Locally this is microseconds; over a real network it might be tens of milliseconds. The request is exactly the shape we saw in chapter 3 - request line, headers, body.

### Step 4: Express receives the request

Express matches the URL */api/convert* and method POST to our handler in **backend/index.js**. The **express.json()** middleware parses the body, putting it in **req.body**. The **cors** middleware adds the right response headers.

### Step 5: Validation

Our route function runs. It checks *req.body.celsius* exists, can be converted to a number, and is above absolute zero. All good - we proceed.

### Step 6: The actual work

**fahrenheit = 25 * 9 / 5 + 32 = 77**. We round to 2 decimal places to keep things tidy.

### Step 7: Response travels back

**res.json({ celsius: 25, fahrenheit: 77 })** sends an HTTP 200 response with a JSON body. Express handles the Content-Type and Content-Length headers automatically.

### Step 8: Browser displays the result

Our fetch's *await response.json()* resolves with the parsed object. *response.ok* is true, so we set *resultEl.className = 'result success'* and inject the HTML showing both numbers. The user sees "25 deg C = 77 deg F" in green.

Total time: typically 50-200 ms locally. The user just sees the answer appear instantly.

## What changes when we add real features?

Imagine we now wanted to add: history of conversions, user accounts, a database, multiple unit types, real-time chat... **The shape never changes.** The frontend always:

1. Captures user input.
2. Calls fetch with the right method, URL, and JSON body.
3. Awaits the response.
4. Parses JSON.
5. Shows the data, or shows the error.

And the backend always:

1. Receives a request.
2. Runs middleware (parse, auth, log).
3. Validates the input.
4. Does the work (calculate, query DB, call another API).
5. Sends a JSON response with a sensible status code.

Master this loop. Every API in the world follows it.

# Chapter 15 - Where to Go Next

You now understand how APIs work. You can build a backend, expose it as an API, and call it from a frontend. That is a huge milestone. Here are 10 ways to extend what you have built and keep learning.

## 10 next steps

**1. Add a second endpoint** - *POST /api/convert/f-to-c* for Fahrenheit -> Celsius. Same shape, opposite direction.

**2. Add other unit conversions** - kilograms to pounds, kilometres to miles. One backend, many endpoints.

**3. Add a history list** - frontend keeps the last 10 conversions in localStorage. (See our React Pizza Billing tutorial for localStorage examples.)

**4. Add a database** - save every conversion to MySQL. (See our Todo App tutorial for full database integration.)

**5. Add authentication** - require an API key in the *Authorization* header. Reject requests without it.

**6. Build a real frontend in React** - replace the vanilla JS with React components. Same backend, prettier UI.

**7. Deploy the backend** - put it on Railway, Render, or a VPS so anyone can call it.

**8. Deploy the frontend** - drop the static files on Netlify or Vercel. Don't forget to update the API_BASE URL.

**9. Add rate limiting** - use the *express-rate-limit* package. Protect your API from abuse.

**10. Add a Swagger / OpenAPI spec** - generates beautiful API docs automatically.

## Common gotchas to remember

| Symptom | Likely cause |
|---|---|
| CORS error in browser console | **app.use(cors())** missing or after the routes. |
| req.body is undefined | **app.use(express.json())** missing. |
| fetch() returns the response, not the data | Forgot to call **await response.json()**. |
| fetch resolved but the data looks wrong | Status was 4xx or 5xx. Check **response.ok**. |
| Server crashes on bad input | Validation missing. Always validate **req.body**. |
| Body shows up as a string on the server | Forgot to set **Content-Type: application/json**. |
| Hardcoded localhost in production | Move the URL to an env variable or config file. |

## Final thoughts

You started this tutorial with a vague sense of what an API was. You finished by building one - validation, error handling, a UI that calls it, the works. That puts you ahead of most people who say they "know APIs" without having actually built one.

**Now build something else.** Pick anything you actually use - a unit converter, a tip calculator, a password generator, a quote-of-the-day API. Use the same patterns. Hit problems. Solve them. That is how skill grows.

Share what you build. Post it on LinkedIn with the GitHub link. Add it to your CV. Send us a link at **egotechworld.com** - we love featuring projects from learners.

Welcome to API development. Happy building!

# Quick Reference Card

## API endpoints we built

| Method | URL | Body | Response |
|---|---|---|---|
| POST | /api/convert | { celsius: 25 } | { celsius, fahrenheit } |
| GET | /api/health | - | { status: "ok" } |

## HTTP status codes you must know

| Code | Meaning | When you'll see it |
|---|---|---|
| 200 | OK | Successful GET or POST |
| 201 | Created | POST that made a new thing |
| 204 | No Content | Successful DELETE |
| 400 | Bad Request | Missing or malformed input |
| 401 | Unauthorized | Not logged in |
| 403 | Forbidden | Logged in but not allowed |
| 404 | Not Found | URL doesn't exist |
| 422 | Unprocessable Entity | Validation failed |
| 429 | Too Many Requests | Rate-limited |
| 500 | Internal Server Error | Backend crashed |

## JSON quick syntax

```
{
    "string":  "hello",
    "number":  42,
    "decimal": 3.14,
    "boolean": true,
    "null":    null,
    "array":   [1, 2, 3],
    "object":  { "nested": "yes" }
}
```

## Most-used Express patterns

```
// Setup
import express from 'express';
import cors from 'cors';
const app = express();
app.use(cors());
app.use(express.json());

// GET (read)
app.get('/api/things', (req, res) => {
    res.json([{ id: 1 }, { id: 2 }]);
});

// POST (create)
app.post('/api/things', (req, res) => {
    const data = req.body;
    if (!data.name) {
        return res.status(400).json({ error: 'name required' });
    }
    res.status(201).json({ id: 1, ...data });
});

// PUT (update)
app.put('/api/things/:id', (req, res) => {
    const id = req.params.id;
    res.json({ id, ...req.body });
});

// DELETE
app.delete('/api/things/:id', (req, res) => {
    res.status(204).end();
});

app.listen(4000);
```

## fetch() patterns

```
// GET
const res = await fetch('/api/things');
const data = await res.json();

// POST
const res = await fetch('/api/things', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name: 'Saman' }),
});
const data = await res.json();

// Always check response.ok
if (!res.ok) {
    const err = await res.json();
    throw new Error(err.error || 'Request failed');
}
```

## Common gotchas

| Problem | Fix |
|---|---|
| CORS error | **app.use(cors())** before the routes. |
| req.body undefined | **app.use(express.json())**. |
| JSON not parsed | Set **Content-Type: application/json** in the request. |
| fetch returns Response, not data | **await response.json()**. |
| fetch doesn't throw on 404 | Check **response.ok** manually. |

## Final project structure

```
api-demo/
+-- backend/
|   +-- node_modules/      # auto, never edit
|   +-- index.js           # Express server with all routes
|   +-- package.json
|   `-- package-lock.json
`-- frontend/
    +-- index.html         # the UI structure
    +-- style.css          # the styles
    `-- app.js             # the fetch() code
```

## Where to learn more

- **expressjs.com** - the official Express documentation.
- **developer.mozilla.org/docs/Web/API/Fetch_API** - everything about fetch().
- **httpstatuses.com** - quick reference for every status code.
- **jsonlint.com** - validates JSON you can paste in.
- **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 tutorials on PHP, Python, React, Laravel, Node, and the SDLC. We post new content regularly and welcome guest articles from learners like you.

Happy coding!