EcomLabs

STEP-BY-STEP GUIDE

# Zod validation in TypeScript: the complete example

A zod schema for an order, the function that checks the form, the component, a Next.js API route that answers 400 and a script that tests all of it. The same code we ran, ready to copy.

| | |
|---|---|
| FOR | TypeScript projects with forms and API routes, in Next.js (App Router) or any server built on Request and Response |
| TESTED WITH | zod 4.6.7 · Node 22.22.3 · tsc 5.7.3 --strict · tsx |
| UPDATED | November 19, 2026 |
| DOCS | zod.dev |
| ARTICLE | ecomlabs.dev/g/zod-en |
| AUTHOR | Luis Morales · @luismoralesdev |

# Contents


| | |
|---|---|
| Step 01 | The schema |
| Step 02 | The form |
| Step 03 | The API route |
| Step 04 | Test it |
| When you are done | Checklist |

STEP 01

## The schema

Install zod 4 with `npm install zod` (we tested 4.6.7); it needs TypeScript 5.5 or later with `"strict": true`. Imports use the `@/*` alias Next.js sets up by default.

```
// lib/order-schema.ts
import * as z from "zod";

export const OrderInput = z.object({
  name: z.string().trim().min(2, "Write your full name"),
  email: z.string().trim().pipe(z.email("Check your email address")),
  postalCode: z.string().trim().regex(/^\d{5}$/, "A postal code has 5 digits"),
  quantity: z.coerce.number("Enter a number").int("Whole units only").min(1, "Order at least 1").max(10, "Up
to 10 per order"),
  notes: z.string().trim().max(500, "Notes: 500 characters max").optional(),
});

export type OrderInput = z.infer<typeof OrderInput>;
```

- The email is trimmed before the format check. `z.email().trim()` rejects `" ana@example.com "`, because it checks first and trims after.
- `z.coerce.number()` converts the form's string. Watch out: an empty field becomes 0 and fails `.min(1)`.

STEP 02

## The form

A function with no React in it, so it can be tested in Node:

```
// lib/check-order.ts
import * as z from "zod";
import { OrderInput } from "@/lib/order-schema";

// Runs in the browser before anything is sent: same schema as the API.
// Returns the clean order, or the messages for each field.
export function checkOrder(data: FormData) {
  const result = OrderInput.safeParse(Object.fromEntries(data));
  if (!result.success) {
    return { ok: false as const, errors: z.flattenError(result.error).fieldErrors };
  }
  return { ok: true as const, order: result.data };
}
```

`safeParse` never throws: it returns `success` with either `data` or `error`. `z.flattenError` groups the messages by field, such as `{ email: ["Check your email address"] }`. The component uses it like this:

```tsx
// components/OrderForm.tsx
"use client";
import { useState } from "react";
import { checkOrder } from "@/lib/check-order";

export function OrderForm() {
  const [errors, setErrors] = useState<Record<string, string[] | undefined>>({});

  async function onSubmit(event: React.FormEvent<HTMLFormElement>) {
    event.preventDefault();
    const checked = checkOrder(new FormData(event.currentTarget));
    if (!checked.ok) return setErrors(checked.errors);
    const res = await fetch("/api/orders", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(checked.order),
    });
    setErrors(res.status === 400 ? (await res.json()).fieldErrors : {});
  }

  return (
    <form onSubmit={onSubmit} noValidate>
      <label htmlFor="email">Email</label>
      <input id="email" name="email" type="email" autoComplete="email"
        aria-invalid={Boolean(errors.email)} aria-describedby="email-error" />
      <p id="email-error">{errors.email?.[0]}</p>
      {/* name, postalCode, quantity and notes repeat the same pattern */}
      <button type="submit">Place order</button>
    </form>
  );
}
```

Repeat label, input and message for each schema key; every input's `name` must match its key.

---

STEP 03

## The API route

It answers 400 with the errors per field, also when the body is not JSON. Price and stock never come from the body.

```tsx
// app/api/orders/route.ts
import * as z from "zod";
import { OrderInput } from "@/lib/order-schema";

export async function POST(request: Request) {
  const body: unknown = await request.json().catch(() => null);
  const result = OrderInput.safeParse(body);
  if (!result.success) {
```

```
    return Response.json(z.flattenError(result.error), { status: 400 });
  }
  const order = result.data; // OrderInput: trimmed, typed, unknown keys dropped
  // Save the order here. Price and stock come from your database, not from the body.
  return Response.json({ ok: true, quantity: order.quantity }, { status: 201 });
}
```

STEP 04

## Test it

The script calls `checkOrder` with a `FormData` and `POST` with `Request` objects, with no server running; it needs `npm install -D tsx`. The `.mts` extension allows top-level `await` even in a CommonJS project.

```
// test-order.mts: run with npx tsx test-order.mts
import assert from "node:assert/strict";
import { checkOrder } from "@/lib/check-order";
import { POST } from "@/app/api/orders/route";

const form = new FormData();
const fields = { name: " Ana ", email: " ana@example.com ", postalCode: "06700", quantity: "2" };
for (const [k, v] of Object.entries(fields)) form.append(k, v);
const checked = checkOrder(form);
assert.ok(checked.ok);
assert.deepEqual(checked.order, { name: "Ana", email: "ana@example.com", postalCode: "06700", quantity: 2 });

const post = (body: string) => POST(new Request("http://localhost/api/orders", { method: "POST", body }));
const bad = await post(JSON.stringify({ name: "Ana", email: "x", postalCode: "06700", quantity: 50 }));
assert.equal(bad.status, 400);
console.log(await bad.json());
assert.equal((await post("{not json")).status, 400);
const good = await post(JSON.stringify({ ...fields, quantity: 2, price: 0 }));
assert.equal(good.status, 201);
console.log("OK");
```

It prints the bad order's `fieldErrors`, then `OK`. Add a case for every rule you change.

WHEN YOU ARE DONE

## Checklist

- [ ] The schema lives in one file that both the form and the route import.
- [ ] The route uses `safeParse` and answers 400 with `z.flattenError`, malformed JSON included.
- [ ] Types come from `z.infer`; no hand-written `interface`, no `as`.
- [ ] Price, stock and totals come from the server.