Zod

TypeScript-first schema validation with static type inference.

01 Install & import

```
npm install zod   # or pnpm add, bun add
```

```
import * as z from "zod"
const schema = z.string()
schema.parse("hello") // "hello"
schema.parse(42)      // throws ZodError
```

Needs TypeScript 5.5+ with "strict": true in tsconfig.json. No dependencies: runs in Node, Bun, Deno and the browser.

02 Primitive schemas

```
z.string(); z.number(); z.boolean()
z.bigint(); z.symbol(); z.date()
z.null(); z.undefined(); z.void()
z.any(); z.unknown(); z.never()
z.literal("foo") // exactly "foo"
z.nan()          // only NaN
```

z.date() wants a Date object, not a date string. Prefer z.unknown() to z.any() to keep type safety. Schemas are immutable: methods return a new schema.

03 String validators

```
z.string().min(3, "Too short")
z.string().max(20)
z.string().length(10)
z.string().regex(/^[a-z]+$/, "a-z only")
z.string().startsWith("https://")
z.string().trim().toLowerCase()
```

.trim(), .toLowerCase() and .toUpperCase() change the value as it parses. Checks run in order, so trim before .min().

04 String formats (Zod 4)

`z.email()`	Email address
`z.url()`	URL
`z.uuid()`	UUID, or z.uuidv4(), z.uuidv7()
`z.iso.date()`	Date like "2024-01-01"
`z.iso.datetime()`	ISO 8601 date and time

Top-level in Zod 4; z.string().email() is deprecated. Also z.ipv4(), z.jwt(), z.base64(), z.nanoid(), z.ulid().

05 Number validators

```
z.number().min(0).max(100)  // inclusive
z.number().gt(0).lt(100)    // exclusive
z.number().int()            // safe integer
z.number().positive()       // > 0
z.number().multipleOf(0.01) // 2 decimals
z.bigint().min(0n).max(1000n)
```

z.number() rejects NaN and Infinity. Zod 4: .safe() equals .int(), .finite() is a no-op, .step() is deprecated.

06 Objects & z.infer

```
const UserSchema = z.object({
  id: z.uuid(),
  name: z.string().min(1),
  email: z.email(),
  age: z.number().optional(),
})
type User = z.infer<typeof UserSchema>
// { id: string; name: string;
//   email: string; age?: number }
```

One schema gives runtime checks and the TypeScript type.

07 Unknown keys

`z.object()`	Strips unknown keys (default)
`z.strictObject()`	Throws on unknown keys
`z.looseObject()`	Keeps unknown keys as they are
`.catchall(schema)`	Checks unknown keys with schema

z.object({ name }).catchall(z.number()) accepts { name: "app", port: 3000 }.

08 Object methods

```
UserSchema.pick({ id: true, name: true })
UserSchema.omit({ password: true })
UserSchema.partial()       // all optional
UserSchema.partial({ email: true })
UserSchema.required()      // all required
UserSchema.extend({ role: z.string() })
UserSchema.extend(Timestamps.shape)
UserSchema.keyof()         // enum of keys
```

pick and omit take { key: true }, not an array. .merge() is deprecated in Zod 4: use A.extend(B.shape).

09 Arrays & tuples

```
z.array(z.string()) // z.string().array()
z.array(z.string()).min(1).max(10)
z.array(z.string()).nonempty()
const Coord = z.tuple([
  z.number(), z.number(),
])
z.tuple([z.string()]).rest(z.number())
// [string, ...number[]]
```

A tuple fixes the length and the type at each position. nonempty() is the same as min(1).

Zod

Unions, enums, error handling, refinements and transforms.

10 Records, maps & sets

```
const Key = z.enum(["pending", "done"])
z.record(z.string(), z.number())
z.record(Key, z.string()) // exhaustive
z.partialRecord(Key, z.string())
z.map(z.string(), z.number())
z.set(z.string()).min(1).max(100)
```

Enum keys are all required; partialRecord makes them optional. z.map() and z.set() need real Map and Set objects.

11 Unions

```
z.union([z.string(), z.number()])
z.string().or(z.number()) // same
const Shape = z.discriminatedUnion("kind", [
  z.object({ kind: z.literal("circle"),
             radius: z.number() }),
  z.object({ kind: z.literal("square"),
             size: z.number() }),
])
```

z.union() tries options in order. A discriminated union checks only the matching kind: faster, clearer errors.

12 Enums & literals

```
const Fish = z.enum(["tuna", "salmon"])
type Fish = z.infer<typeof Fish>
Fish.options   // ["tuna", "salmon"]
Fish.enum.tuna // "tuna"
Fish.exclude(["salmon"])
z.literal(["tuna", "salmon"]) // Zod 4
z.enum(Status)                // TS enum
```

z.nativeEnum() is deprecated in Zod 4: pass the enum to z.enum(). Share values with an as const array: z.enum(ROLES).

13 Optional, default & catch

.optional()	T | undefined
.nullable()	T | null
.nullish()	T | null | undefined
.default(value)	Used when the input is undefined
.catch(value)	Used when parsing fails

.default() ignores null; a default function runs on every parse. z.number().catch(3000) turns "x" into 3000.

14 parse & safeParse

```
const user = UserSchema.parse(data)
// throws z.ZodError if invalid
const result = UserSchema.safeParse(data)
if (result.success) {
  result.data         // typed User
} else {
  result.error.issues // what failed
}
```

safeParse never throws, so it suits API routes: return a 400 with result.error.issues. With async refinements use parseAsync() or safeParseAsync().

15 Format errors

error.issues	Each has code, path and message
z.flattenError()	{ formErrors, fieldErrors }
z.treeifyError()	Errors nested by path
z.prettifyError()	One readable string

flattenError suits forms: fieldErrors.age is a list of messages for age.

16 Custom error messages

```
z.string({ error: "Name is required" })
z.string().min(3, "Too short")
z.email({ error: "Invalid email" })
z.string().min(8, { error: (iss) =>
  `Got ${String(iss.input).length} chars` })
```

Zod 4 uses error; the v3 message key still works but is deprecated. z.config({ customError }) sets one global handler, handy for i18n.

17 refine & superRefine

```
const Signup = z.object({
  password: z.string().min(8),
  confirm: z.string(),
}).refine((d) => d.password === d.confirm, {
  error: "Passwords don't match",
  path: ["confirm"], // error on confirm
})
```

Async checks need parseAsync(); sync parse throws. superRefine() can add several issues with ctx.addIssue().

18 transform, pipe & coerce

```
const ToInt = z.string()
  .transform((val) => parseInt(val, 10))
  .pipe(z.number().int().positive())
type In = z.input<typeof ToInt>   // string
type Out = z.output<typeof ToInt> // number
z.coerce.number().parse("42")     // 42
z.coerce.date().parse("2024-01-01")
```

coerce suits query params and form data. z.coerce.boolean() uses Boolean(), so "false" is true.