Zod

Version 4.6 · verified 2026-11-02

Zod 4 schemas as parsers at every trust boundary — inferred types, refinements, coercion, errors, Hono/Express/forms and JSON Schema.

Contents

Mental model 2

Schemas and parsing 2

Types from schemas 3

Refinements and transforms 4

Unions and discriminated unions 5

Coercion and env vars 6

Errors: messages and formatting 7

At the boundary: requests and forms 9

JSON Schema for LLM tools 10

Zod 3 vs 4, and performance 12

Further reading 12

Mental model

A Zod schema is a parser: give it `unknown` and it returns either a typed value or a structured error, never a half-checked object. That makes it the tool for every trust boundary — request bodies, form posts, env vars, LLM output, third-party APIs. Write the schema first and derive the TypeScript type from it; the reverse direction lets the two drift apart silently.

Schemas and parsing

Call	Returns
`schema.parse(x)`	Value, or throws `ZodError`
`schema.safeParse(x)`	`{ success, data }` or `{ success, error }`
`await schema.parseAsync(x)`	Needed for async refinements
`z.object({...})`	Strips unknown keys
`z.strictObject({...})`	Rejects unknown keys

Import with `import * as z from "zod"`. Prefer `safeParse` at boundaries, where invalid input is expected and should become a 400, not an exception. String formats such as `z.email()` and `z.url()` are top-level functions in Zod 4.

```
import * as z from "zod";


const User = z.object({
```

```
  email: z.email(),
  name: z.string().min(1).max(80),
  age: z.number().int().min(0).optional(),
});

const r = User.safeParse({ email: "a@b.co" });
if (r.success) r.data.email; // string
else r.error.issues; // name: expected string
```

> **Gotcha:** `z.object()` silently drops keys it does not know. Your typo `{ emial: ... }` passes as `{}` plus "required" errors, but an extra `isAdmin` key just disappears. Use `z.strictObject()` when unexpected input should fail loudly.

Types from schemas

Helper	Gives
`z.infer<typeof S>`	Output type (after transforms)
`z.output<typeof S>`	Same as `z.infer`
`z.input<typeof S>`	Input type (before transforms)

One schema, one source of truth: the type follows every edit to the schema. Input and output differ only when the schema transforms or defaults values, which is exactly when you need both — for example, a form's raw input versus the parsed result.

```ts
import * as z from "zod";

const Port = z.string().transform(Number);
type In = z.input<typeof Port>;   // string
type Out = z.output<typeof Port>; // number

const Todo = z.object({
  title: z.string(),
  done: z.boolean().default(false),
});
type Todo = z.infer<typeof Todo>;
// { title: string; done: boolean }
```

Refinements and transforms

Method	Use for
`.refine(fn, { error })`	Custom rule, value unchanged
`.superRefine((v, ctx) => …)`	Several issues from one check
`.transform(fn)`	Change the value or type
`.pipe(schema)`	Validate the transformed value
`.default(v)` / `.prefault(v)`	Fallback for `undefined`

Refinements are skipped while the value has a type error, so an object-level `.refine` never sees malformed data — and a form shows "Passwords differ" only once every field has the right type. Attach the issue to a field with `path` so form libraries show it next to the input.

```ts
import * as z from "zod";

const Signup = z.object({
  password: z.string().min(12),
  confirm: z.string(),
}).refine((d) => d.password === d.confirm, {
  error: "Passwords differ",
  path: ["confirm"],
});

const Csv = z.string()
  .transform((s) => s.split(","))
  .pipe(z.array(z.string().min(1)).max(10));
```

> **Gotcha:** In Zod 4, `.default()` short-circuits on `undefined` and returns the default without parsing it, so it must match the *output* type. Use `.prefault()` for the Zod 3 behavior, where the default goes through the schema.

Unions and discriminated unions

API	Matches
`z.union([A, B])`	First option that parses
`z.discriminatedUnion("k", [A, B])`	Option chosen by key `k`
`z.literal("x")` / `z.enum([...])`	Exact values

A discriminated union reads one key and parses only the matching option. It is faster than trying each branch, and its errors name the real problem instead of listing every branch's failure. Use it for events, API results and message types.

```
import * as z from "zod";

const Event = z.discriminatedUnion("type", [
  z.object({ type: z.literal("tap"), x: z.int() }),
  z.object({ type: z.literal("nav"), to: z.url() }),
]);
type Event = z.infer<typeof Event>;

function handle(e: Event) {
  if (e.type === "nav") return e.to; // narrowed
  return e.x;
}
```

Coercion and env vars

Schema	Converts with
`z.coerce.number()`	`Number(x)`
`z.coerce.boolean()`	`Boolean(x)`
`z.coerce.date()`	`new Date(x)`
`z.stringbool()`	`"true"` / `"1"` / `"yes"` / `"on"` … → `true`

Query strings, form fields and env vars arrive as strings. `z.coerce` runs the built-in constructor first, then validates. Parse `process.env` once at startup and export the typed result, so a missing variable crashes the deploy instead of the first request that needs it.

```
import * as z from "zod";

const Env = z.object({
  DATABASE_URL: z.url(),
  PORT: z.coerce.number().int().positive()
    .default(3000),
  DEBUG: z.stringbool().default(false),
});

export const env = Env.parse(process.env);
// PORT="" → 0 → fails .positive()
```

> **Gotcha:** `z.coerce.boolean()` is `Boolean(x)`, so the string `"false"` becomes `true`. Use `z.stringbool()` for flags, and note that `Number("")` is `0`, which is why the sample needs `.positive()`.

Errors: messages and formatting

Function	Output
`z.flattenError(err)`	`{ formErrors, fieldErrors }`

Function	Output
`z.treeifyError(err)`	Nested `{ errors, properties }`
`z.prettifyError(err)`	Multi-line string for logs
`{ error: "..." }` param	Custom message on any check

`error.issues` is the raw list: each issue has a `code`, a `path` and a `message`. Pick a formatter by consumer — flat for a form with one level of fields, tree for nested data, pretty for logs and CLIs.

```
import * as z from "zod";


const S = z.object({
  name: z.string().min(1, { error: "Required" }),
});
const r = S.safeParse({ name: "" });
if (!r.success) {
  z.flattenError(r.error).fieldErrors;
  // { name: ["Required"] }
}
```

> **Note:** Zod 4 replaces `message`, `invalid_type_error`, `required_error` and `errorMap` with a single `error` parameter, and deprecates `.format()` and `.flatten()` in favor of the functions above.

At the boundary: requests and forms

Where	Pattern
Hono	`validator("json", …)` from `hono/validator`
Express	`safeParse(req.body)` in the handler
React Hook Form	`zodResolver(schema)`
Server action	`safeParse` the `FormData` entries

Validate before any business logic touches the data, and return the formatted error with a 400. In Hono, the validator's return value becomes the typed `c.req.valid("json")`.

```
import { Hono } from "hono";
import { validator } from "hono/validator";
import * as z from "zod";

const Body = z.object({ title: z.string().min(1) });
const app = new Hono();

app.post("/todos", validator("json", (v, c) => {
  const r = Body.safeParse(v);
  if (!r.success)
    return c.json(z.flattenError(r.error), 400);
  return r.data;
}), (c) => c.json(c.req.valid("json"), 201));
```

The same schema drives a form on the client:

```tsx
import { useForm } from "react-hook-form";
import { zodResolver }
  from "@hookform/resolvers/zod";
import * as z from "zod";
const Login = z.object({ email: z.email() });
export function LoginForm() {
  const { register, handleSubmit, formState } =
    useForm({ resolver: zodResolver(Login) });
  return (
    <form onSubmit={handleSubmit(console.log)}>
      <input {...register("email")} />
      <p>{formState.errors.email?.message}</p>
    </form>
  );
}
```

> **Warning:** Client-side validation is a convenience, not a control. Parse the same input again on the server — anyone can post to your endpoint without your form.

JSON Schema for LLM tools

Option	Effect
`z.toJSONSchema(S)`	Draft 2020-12, output type
`{ io: "input" }`	Describe what callers send

Option	Effect
`{ target: "draft-07" }`	Older consumers, e.g. some SDKs
`.describe("…")`	Becomes `description`

LLM tool definitions and structured-output APIs take JSON Schema. Generate it from the same schema you use to parse the model's reply, so the contract and the check cannot drift. Descriptions matter: the model reads them.

```
import * as z from "zod";

const Args = z.object({
  city: z.string().describe("City name"),
  days: z.number().int().min(1).max(7),
});
const tool = {
  name: "get_forecast",
  input_schema: z.toJSONSchema(Args),
};
// later: Args.parse(modelToolCall.input)
```

> **Gotcha:** `z.toJSONSchema()` throws on types JSON Schema cannot express — `z.date()`, `z.bigint()`, `z.map()`, transforms. Model those fields as strings, or pass `{ io: "input" }` when a transform sits after a representable input.

Zod 3 vs 4, and performance

Zod 3	Zod 4
`z.string().email()`	`z.email()` (method deprecated)
`.strict()` / `.passthrough()`	`z.strictObject` / `z.looseObject`
`{ message }`, `errorMap`	`{ error }`
`z.record(V)`	`z.record(K, V)`
`.default()` parses the default	`.default()` short-circuits
`err.flatten()`, `err.format()`	`z.flattenError`, `z.treeifyError`

Define schemas once at module scope and reuse them; building a schema inside a hot handler repeats work on every request. For browser bundles where size matters, `zod/mini` offers the same checks through a tree-shakable functional API.

```
import * as z from "zod/mini";


const User = z.object({
  name: z.string().check(z.minLength(1)),
  tags: z.optional(z.array(z.string())),
});
User.parse({ name: "Al" }); // { name: "Al" }
```

Further reading

- Zod documentation
- Zod 4 migration guide
- Formatting errors

- JSON Schema conversion
- Zod Mini