ZOD Docs - English

Table of contents

Zod

✨ https://zod.dev ✨

TypeScript-first schema validation with static type inference

Documentation • Discord • npm • Issues • @colinhacks • tRPC

> These docs have been translated into Chinese.

Table of contents

Introduction

Zod is a TypeScript-first schema declaration and validation library. I'm using the term "schema" to broadly refer to any data type, from a simple `string` to a complex nested object.

Zod is designed to be as developer-friendly as possible. The goal is to eliminate duplicative type declarations. With Zod, you declare a validator *once* and Zod will automatically infer the static TypeScript type. It's easy to compose simpler types into complex data structures.

Some other great aspects:

- Zero dependencies
- Works in Node.js and all modern browsers
- Tiny: 8kb minified + zipped
- Immutable: methods (i.e. `.optional()`) return a new instance
- Concise, chainable interface
- Functional approach: parse, don't validate
- Works with plain JavaScript too! You don't need to use TypeScript.

Sponsors

Sponsorship at any level is appreciated and encouraged. For individual developers, consider the Cup of Coffee tier. If you built a paid product using Zod, consider one of the podium tiers.

Gold

Astro

astro.build

Astro is a new kind of static site builder for the modern web. Powerful developer experience meets lightweight output.

Glow Wallet

glow.app

Your new favorite Solana wallet.

Deletype

deletype.com

Proxy

proxy.com

Silver

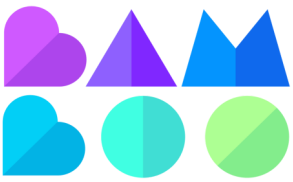
Numeric
numeric.io

Snaplet
snaplet.dev

Marcato Partners
marcatopartners.com

Interval
interval.com

Seasoned Software
seasoned.cc



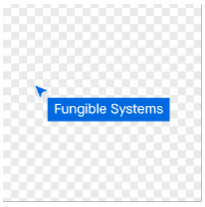
Bamboo Creative
bamboocreative.nz

Bronze

Brandon Bayer
@flybayer, creator of Blitz.js

Jiří Brabec
@brabeji

Alex Johansson
@alexdotjs



Fungible Systems
fungible.systems

Adaptable
adaptable.io

Avana Wallet
avanawallet.com
Solana non-custodial wallet

Ecosystem

There are a growing number of tools that are built atop or support Zod natively! If you've built a tool or library on top of Zod, tell me about it on Twitter or start a Discussion. I'll add it below and tweet it out.

Resources

- Total TypeScript Zod Tutorial by @mattpocockuk
- Fixing TypeScript's Blindspot: Runtime Typechecking by @jherr

API libraries

- `tRPC`: Build end-to-end typesafe APIs without GraphQL.
- `@anatine/zod-nestjs`: Helper methods for using Zod in a NestJS project.
- `zod-endpoints`: Contract-first strictly typed endpoints with Zod. OpenAPI compatible.
- `domain-functions`: Decouple your business logic from your framework using composable functions. With first-class type inference from end to end powered by Zod schemas.
- `@zodios/core`: A typescript API client with runtime and compile time validation backed by axios and zod.
- `express-zod-api`: Build Express-based APIs with I/O schema validation and custom middlewares.

Form integrations

- `react-hook-form`: A first-party Zod resolver for React Hook Form.
- `zod-validation-error`: Generate user-friendly error messages from `ZodError`s
- `zod-formik-adapter`: A community-maintained Formik adapter for Zod.

- `react-zorm`: Standalone `<form>` generation and validation for React using Zod.
- `zodix`: Zod utilities for FormData and URLSearchParams in Remix loaders and actions.

Zod to X

- `zod-to-ts`: Generate TypeScript definitions from Zod schemas.
- `zod-to-json-schema`: Convert your Zod schemas into JSON Schemas.
- `@anatine/zod-openapi`: Converts a Zod schema to an OpenAPI v3.x `SchemaObject`.
- `zod-fast-check`: Generate `fast-check` arbitraries from Zod schemas.
- `zod-dto`: Generate Nest.js DTOs from a Zod schema.
- `fastify-type-provider-zod`: Create Fastify type providers from Zod schemas.
- `zod-to-openapi`: Generate full OpenAPI (Swagger) docs from Zod, including schemas, endpoints & parameters.
- `nestjs-graphql-zod`: Generates NestJS GraphQL model classes from Zod schemas. Provides GraphQL method decorators working with Zod schemas.

X to Zod

- `ts-to-zod`: Convert TypeScript definitions into Zod schemas.
- `@runtyping/zod`: Generate Zod from static types & JSON schema.
- `json-schema-to-zod`: Convert your JSON Schemas into Zod schemas. Live demo.
- `json-to-zod`: Convert JSON objects into Zod schemas. Live demo.
- `graphql-codegen-typescript-validation-schema`: GraphQL Code Generator plugin to generate form validation schema from your GraphQL schema
- `zod-prisma`: Generate Zod schemas from your Prisma schema.
- `Supervillain`: Generate Zod schemas from your Go structs.
- `prisma-zod-generator`: Emit Zod schemas from your Prisma schema.
- `prisma-trpc-generator`: Emit fully implemented tRPC routers and their validation schemas using Zod.

Mocking

- `@anatine/zod-mock`: Generate mock data from a Zod schema. Powered by faker.js.
- `zod-mocking`: Generate mock data from your Zod schemas.

Powered by Zod

- `slonik`: Node.js Postgres client with strong Zod integration.
- `soly`: Create CLI applications with zod.
- `zod-xlsx`: A xlsx based resource validator using Zod schemas.

Installation

Requirements

- TypeScript 4.5+!

- You must enable `strict` mode in your `tsconfig.json`. This is a best practice for all TypeScript projects.

```
// tsconfig.json
{
  // ...
  "compilerOptions": {
    // ...
    "strict": true
  }
}
```

From `npm` (Node/Bun)

```
npm install zod       # npm
yarn add zod          # yarn
bun add zod           # bun
pnpm add zod          # pnpm
```

From `deno.land/x` (Deno)

Unlike Node, Deno relies on direct URL imports instead of a package manager like NPM. Zod is available on deno.land/x. The latest version can be imported like so:

```
import { z } from "https://deno.land/x/zod/mod.ts";
```

You can also specify a particular version:

```
import { z } from "https://deno.land/x/zod@v3.16.1/mod.ts";
```

> The rest of this README assumes you are using npm and importing directly from the `"zod"` package.

Basic usage

Creating a simple string schema

```
import { z } from "zod";

// creating a schema for strings
const mySchema = z.string();

// parsing
mySchema.parse("tuna"); // => "tuna"
mySchema.parse(12); // => throws ZodError

// "safe" parsing (doesn't throw error if validation fails)
mySchema.safeParse("tuna"); // => { success: true; data: "tuna" }
mySchema.safeParse(12); // => { success: false; error: ZodError }
```

Creating an object schema

```ts
import { z } from "zod";

const User = z.object({
  username: z.string(),
});

User.parse({ username: "Ludwig" });

// extract the inferred type
type User = z.infer<typeof User>;
// { username: string }
```

Primitives

```ts
import { z } from "zod";

// primitive values
z.string();
z.number();
z.bigint();
z.boolean();
z.date();
z.symbol();

// empty types
z.undefined();
z.null();
z.void(); // accepts undefined

// catch-all types
// allows any value
z.any();
z.unknown();

// never type
// allows no values
z.never();
```

Literals

```ts
const tuna = z.literal("tuna");
const twelve = z.literal(12);
const twobig = z.literal(2n); // bigint literal
const tru = z.literal(true);

const terrificSymbol = Symbol("terrific");
const terrific = z.literal(terrificSymbol);

// retrieve literal value
tuna.value; // "tuna"
```

Currently there is no support for Date literals in Zod. If you have a use case for this feature, please file an issue.

Strings

Zod includes a handful of string-specific validations.

```
z.string().max(5);
z.string().min(5);
z.string().length(5);
z.string().email();
z.string().url();
z.string().uuid();
z.string().cuid();
z.string().regex(regex);
z.string().startsWith(string);
z.string().endsWith(string);
z.string().trim(); // trim whitespace
z.string().datetime(); // defaults to UTC, see below for options
```

> Check out validator.js for a bunch of other useful string validation functions that can be used in conjunction with Refinements.

You can customize some common error messages when creating a string schema.

```
const name = z.string({
  required_error: "Name is required",
  invalid_type_error: "Name must be a string",
});
```

When using validation methods, you can pass in an additional argument to provide a custom error message.

```
z.string().min(5, { message: "Must be 5 or more characters long" });
z.string().max(5, { message: "Must be 5 or fewer characters long" });
z.string().length(5, { message: "Must be exactly 5 characters long" });
z.string().email({ message: "Invalid email address" });
z.string().url({ message: "Invalid url" });
z.string().uuid({ message: "Invalid UUID" });
z.string().startsWith("https://", { message: "Must provide secure URL" });
z.string().endsWith(".com", { message: "Only .com domains allowed" });
z.string().datetime({ message: "Invalid datetime string! Must be UTC." });
```

Coercion for primitives

Zod now provides a more convenient way to coerce primitive values.

```
const schema = z.coerce.string();
schema.parse("tuna"); // => "tuna"
schema.parse(12); // => "12"
schema.parse(true); // => "true"
```

During the parsing step, the input is passed through the `String()` function, which is a JavaScript built-in for coercing data into strings. Note that the returned schema is a `ZodString` instance so you can use all string methods.

```
z.coerce.string().email().min(5);
```

All primitive types support coercion.

```
z.coerce.string(); // String(input)
z.coerce.number(); // Number(input)
z.coerce.boolean(); // Boolean(input)
z.coerce.bigint(); // BigInt(input)
z.coerce.date(); // new Date(input)
```

Datetime validation

The `z.string().datetime()` method defaults to UTC validation: no timezone offsets with arbitrary sub-second decimal precision.

```
const datetime = z.string().datetime();

datetime.parse("2020-01-01T00:00:00Z"); // pass
datetime.parse("2020-01-01T00:00:00.123Z"); // pass
datetime.parse("2020-01-01T00:00:00.123456Z"); // pass (arbitrary precision)
datetime.parse("2020-01-01T00:00:00+02:00"); // fail (no offsets allowed)
```

Timezone offsets can be allowed by setting the `offset` option to `true`.

```
const datetime = z.string().datetime({ offset: true });

datetime.parse("2020-01-01T00:00:00+02:00"); // pass
datetime.parse("2020-01-01T00:00:00.123+02:00"); // pass (millis optional)
datetime.parse("2020-01-01T00:00:00Z"); // pass (Z still supported)
```

You can additionally constrain the allowable `precision`. By default, arbitrary sub-second precision is supported (but optional).

```
const datetime = z.string().datetime({ precision: 3 });

datetime.parse("2020-01-01T00:00:00.123Z"); // pass
datetime.parse("2020-01-01T00:00:00Z"); // fail
datetime.parse("2020-01-01T00:00:00.123456Z"); // fail
```

Numbers

You can customize certain error messages when creating a number schema.

```
const age = z.number({
  required_error: "Age is required",
  invalid_type_error: "Age must be a number",
});
```

Zod includes a handful of number-specific validations.

```
z.number().gt(5);
z.number().gte(5); // alias .min(5)
z.number().lt(5);
z.number().lte(5); // alias .max(5)
```

```
z.number().int(); // value must be an integer

z.number().positive(); //     > 0
z.number().nonnegative(); //  >= 0
z.number().negative(); //     < 0
z.number().nonpositive(); //  <= 0

z.number().multipleOf(5); // Evenly divisible by 5. Alias .step(5)

z.number().finite(); // value must be finite, not Infinity or -Infinity
```

Optionally, you can pass in a second argument to provide a custom error message.

```
z.number().lte(5, { message: "this👏is👏too👏big" });
```

NaNs

You can customize certain error messages when creating a nan schema.

```
const isNaN = z.nan({
  required_error: "isNaN is required",
  invalid_type_error: "isNaN must be not a number",
});
```

Booleans

You can customize certain error messages when creating a boolean schema.

```
const isActive = z.boolean({
  required_error: "isActive is required",
  invalid_type_error: "isActive must be a boolean",
});
```

Dates

Use z.date() to validate `Date` instances.

```
z.date().safeParse(new Date()); // success: true
z.date().safeParse("2022-01-12T00:00:00.000Z"); // success: false
```

You can customize certain error messages when creating a date schema.

```
const myDateSchema = z.date({
  required_error: "Please select a date and time",
  invalid_type_error: "That's not a date!",
});
```

Zod provides a handful of date-specific validations.

```
z.date().min(new Date("1900-01-01"), { message: "Too old" });
z.date().max(new Date(), { message: "Too young!" });
```

Supporting date strings

To write a schema that accepts either a `Date` or a date string, use `z.preprocess`.

```
const dateSchema = z.preprocess((arg) => {
  if (typeof arg == "string" || arg instanceof Date) return new Date(arg);
}, z.date());
type DateSchema = z.infer<typeof dateSchema>;
// type DateSchema = Date

dateSchema.safeParse(new Date("1/12/22")); // success: true
dateSchema.safeParse("2022-01-12T00:00:00.000Z"); // success: true
```

Zod enums

```
const FishEnum = z.enum(["Salmon", "Tuna", "Trout"]);
type FishEnum = z.infer<typeof FishEnum>;
// 'Salmon' | 'Tuna' | 'Trout'
```

`z.enum` is a Zod-native way to declare a schema with a fixed set of allowable *string* values. Pass the array of values directly into `z.enum()`. Alternatively, use `as const` to define your enum values as a tuple of strings. See the const assertion docs for details.

```
const VALUES = ["Salmon", "Tuna", "Trout"] as const;
const FishEnum = z.enum(VALUES);
```

This is not allowed, since Zod isn't able to infer the exact values of each element.

```
const fish = ["Salmon", "Tuna", "Trout"];
const FishEnum = z.enum(fish);
```

Autocompletion

To get autocompletion with a Zod enum, use the `.enum` property of your schema:

```
FishEnum.enum.Salmon; // => autocompletes

FishEnum.enum;
/*
=> {
  Salmon: "Salmon",
  Tuna: "Tuna",
  Trout: "Trout",
}
*/
```

You can also retrieve the list of options as a tuple with the `.options` property:

```
FishEnum.options; // ["Salmon", "Tuna", "Trout"]);
```

Native enums

Zod enums are the recommended approach to defining and validating enums. But if you need to validate against an enum from a third-party library (or you don't want to rewrite your existing enums) you can use `z.nativeEnum()`.

Numeric enums

```
enum Fruits {
  Apple,
  Banana,
}

const FruitEnum = z.nativeEnum(Fruits);
type FruitEnum = z.infer<typeof FruitEnum>; // Fruits

FruitEnum.parse(Fruits.Apple); // passes
FruitEnum.parse(Fruits.Banana); // passes
FruitEnum.parse(0); // passes
FruitEnum.parse(1); // passes
FruitEnum.parse(3); // fails
```

String enums

```
enum Fruits {
  Apple = "apple",
  Banana = "banana",
  Cantaloupe, // you can mix numerical and string enums
}

const FruitEnum = z.nativeEnum(Fruits);
type FruitEnum = z.infer<typeof FruitEnum>; // Fruits

FruitEnum.parse(Fruits.Apple); // passes
FruitEnum.parse(Fruits.Cantaloupe); // passes
FruitEnum.parse("apple"); // passes
FruitEnum.parse("banana"); // passes
FruitEnum.parse(0); // passes
FruitEnum.parse("Cantaloupe"); // fails
```

Const enums

The `.nativeEnum()` function works for `as const` objects as well. ⚠ `as const` required TypeScript 3.4+!

```
const Fruits = {
  Apple: "apple",
  Banana: "banana",
  Cantaloupe: 3,
} as const;

const FruitEnum = z.nativeEnum(Fruits);
type FruitEnum = z.infer<typeof FruitEnum>; // "apple" | "banana" | 3

FruitEnum.parse("apple"); // passes
FruitEnum.parse("banana"); // passes
FruitEnum.parse(3); // passes
FruitEnum.parse("Cantaloupe"); // fails
```

You can access the underlying object with the `.enum` property:

```
FruitEnum.enum.Apple; // "apple"
```

Optionals

You can make any schema optional with `z.optional()`. This wraps the schema in a `ZodOptional` instance and returns the result.

```ts
const schema = z.optional(z.string());

schema.parse(undefined); // => returns undefined
type A = z.infer<typeof schema>; // string | undefined
```

For convenience, you can also call the `.optional()` method on an existing schema.

```ts
const user = z.object({
  username: z.string().optional(),
});
type C = z.infer<typeof user>; // { username?: string | undefined };
```

You can extract the wrapped schema from a `ZodOptional` instance with `.unwrap()`.

```ts
const stringSchema = z.string();
const optionalString = stringSchema.optional();
optionalString.unwrap() === stringSchema; // true
```

Nullables

Similarly, you can create nullable types with `z.nullable()`.

```ts
const nullableString = z.nullable(z.string());
nullableString.parse("asdf"); // => "asdf"
nullableString.parse(null); // => null
```

Or use the `.nullable()` method.

```ts
const E = z.string().nullable(); // equivalent to nullableString
type E = z.infer<typeof E>; // string | null
```

Extract the inner schema with `.unwrap()`.

```ts
const stringSchema = z.string();
const nullableString = stringSchema.nullable();
nullableString.unwrap() === stringSchema; // true
```

Objects

```ts
// all properties are required by default
const Dog = z.object({
  name: z.string(),
  age: z.number(),
});

// extract the inferred type like this
type Dog = z.infer<typeof Dog>;
```

```
// equivalent to:
type Dog = {
  name: string;
  age: number;
};
```

.shape

Use `.shape` to access the schemas for a particular key.

```
Dog.shape.name; // => string schema
Dog.shape.age; // => number schema
```

.keyof

Use `.keyof` to create a `ZodEnum` schema from the keys of an object schema.

```
const keySchema = Dog.keyof();
keySchema; // ZodEnum<["name", "age"]>
```

.extend

You can add additional fields to an object schema with the `.extend` method.

```
const DogWithBreed = Dog.extend({
  breed: z.string(),
});
```

You can use `.extend` to overwrite fields! Be careful with this power!

.merge

Equivalent to `A.extend(B.shape)`.

```
const BaseTeacher = z.object({ students: z.array(z.string()) });
const HasID = z.object({ id: z.string() });

const Teacher = BaseTeacher.merge(HasID);
type Teacher = z.infer<typeof Teacher>; // => { students: string[], id: string }
```

> If the two schemas share keys, the properties of B overrides the property of A. The returned schema also inherits the "unknownKeys" policy (strip/strict/passthrough) and the catchall schema of B.

.pick/.omit

Inspired by TypeScript's built-in `Pick` and `Omit` utility types, all Zod object schemas have `.pick` and `.omit` methods that return a modified version. Consider this Recipe schema:

```ts
const Recipe = z.object({
  id: z.string(),
  name: z.string(),
  ingredients: z.array(z.string()),
});
```

To only keep certain keys, use `.pick` .

```ts
const JustTheName = Recipe.pick({ name: true });
type JustTheName = z.infer<typeof JustTheName>;
// => { name: string }
```

To remove certain keys, use `.omit` .

```ts
const NoIDRecipe = Recipe.omit({ id: true });

type NoIDRecipe = z.infer<typeof NoIDRecipe>;
// => { name: string, ingredients: string[] }
```

`.partial`

Inspired by the built-in TypeScript utility type Partial, the `.partial` method makes all properties optional.

Starting from this object:

```ts
const user = z.object({
  email: z.string()
  username: z.string(),
});
// { email: string; username: string }
```

We can create a partial version:

```ts
const partialUser = user.partial();
// { email?: string | undefined; username?: string | undefined }
```

You can also specify which properties to make optional:

```ts
const optionalEmail = user.partial({
  email: true,
});
/*
{
  email?: string | undefined;
  username: string
}
*/
```

`.deepPartial`

The `.partial` method is shallow — it only applies one level deep. There is also a "deep" version:

```ts
const user = z.object({
  username: z.string(),
  location: z.object({
```

```
    latitude: z.number(),
    longitude: z.number(),
  }),
  strings: z.array(z.object({ value: z.string() })),
});

const deepPartialUser = user.deepPartial();

/*
{
  username?: string | undefined,
  location?: {
    latitude?: number | undefined;
    longitude?: number | undefined;
  } | undefined,
  strings?: { value?: string}[]
}
*/
```

Important limitation: deep partials only work as expected in hierarchies of objects, arrays, and tuples.

`.required`

Contrary to the `.partial` method, the `.required` method makes all properties required.

Starting from this object:

```
const user = z.object({
  email: z.string()
  username: z.string(),
}).partial();
// { email?: string | undefined; username?: string | undefined }
```

We can create a required version:

```
const requiredUser = user.required();
// { email: string; username: string }
```

You can also specify which properties to make required:

```
const requiredEmail = user.required({
  email: true,
});
/*
{
  email: string;
  username?: string | undefined;
}
*/
```

`.passthrough`

By default Zod object schemas strip out unrecognized keys during parsing.

```
const person = z.object({
  name: z.string(),
});

person.parse({
  name: "bob dylan",
  extraKey: 61,
});
// => { name: "bob dylan" }
// extraKey has been stripped
```

Instead, if you want to pass through unknown keys, use `.passthrough()` .

```
person.passthrough().parse({
  name: "bob dylan",
  extraKey: 61,
});
// => { name: "bob dylan", extraKey: 61 }
```

`.strict`

By default Zod object schemas strip out unrecognized keys during parsing. You can *disallow* unknown keys with `.strict()` . If there are any unknown keys in the input, Zod will throw an error.

```
const person = z
  .object({
    name: z.string(),
  })
  .strict();

person.parse({
  name: "bob dylan",
  extraKey: 61,
});
// => throws ZodError
```

`.strip`

You can use the `.strip` method to reset an object schema to the default behavior (stripping unrecognized keys).

`.catchall`

You can pass a "catchall" schema into an object schema. All unknown keys will be validated against it.

```
const person = z
  .object({
    name: z.string(),
  })
  .catchall(z.number());

person.parse({
  name: "bob dylan",
  validExtraKey: 61, // works fine
});

person.parse({
```

```
  name: "bob dylan",
  validExtraKey: false, // fails
});
// => throws ZodError
```

Using `.catchall()` obviates `.passthrough()` , `.strip()` , or `.strict()`. All keys are now considered "known".

Arrays

```
const stringArray = z.array(z.string());

// equivalent
const stringArray = z.string().array();
```

Be careful with the `.array()` method. It returns a new `ZodArray` instance. This means the *order* in which you call methods matters. For instance:

```
z.string().optional().array(); // (string | undefined)[]
z.string().array().optional(); // string[] | undefined
```

`.element`

Use `.element` to access the schema for an element of the array.

```
stringArray.element; // => string schema
```

`.nonempty`

If you want to ensure that an array contains at least one element, use `.nonempty()`.

```
const nonEmptyStrings = z.string().array().nonempty();
// the inferred type is now
// [string, ...string[]]

nonEmptyStrings.parse([]); // throws: "Array cannot be empty"
nonEmptyStrings.parse(["Ariana Grande"]); // passes
```

You can optionally specify a custom error message:

```
// optional custom error message
const nonEmptyStrings = z.string().array().nonempty({
  message: "Can't be empty!",
});
```

`.min/.max/.length`

```
z.string().array().min(5); // must contain 5 or more items
z.string().array().max(5); // must contain 5 or fewer items
z.string().array().length(5); // must contain 5 items exactly
```

Unlike `.nonempty()` these methods do not change the inferred type.

Tuples

Unlike arrays, tuples have a fixed number of elements and each element can have a different type.

```
const athleteSchema = z.tuple([
  z.string(), // name
  z.number(), // jersey number
  z.object({
    pointsScored: z.number(),
  }), // statistics
]);

type Athlete = z.infer<typeof athleteSchema>;
// type Athlete = [string, number, { pointsScored: number }]
```

A variadic ("rest") argument can be added with the `.rest` method.

```
const variadicTuple = z.tuple([z.string()]).rest(z.number());
const result = variadicTuple.parse(["hello", 1, 2, 3]);
// => [string, ...number[]];
```

Unions

Zod includes a built-in `z.union` method for composing "OR" types.

```
const stringOrNumber = z.union([z.string(), z.number()]);

stringOrNumber.parse("foo"); // passes
stringOrNumber.parse(14); // passes
```

Zod will test the input against each of the "options" in order and return the first value that validates successfully.

For convenience, you can also use the `.or` method:

```
const stringOrNumber = z.string().or(z.number());
```

Discriminated unions

A discriminated union is a union of object schemas that all share a particular key.

```
type MyUnion =
  | { status: "success"; data: string }
  | { status: "failed"; error: Error };
```

Such unions can be represented with the `z.discriminatedUnion` method. This enables faster evaluation, because Zod can check the *discriminator key* (`status` in the example above) to determine which schema should be used to parse the input. This makes parsing more efficient and lets Zod report friendlier errors.

With the basic union method the input is tested against each of the provided "options", and in the case of invalidity, issues for all the "options" are shown in the zod error. On the other hand, the discriminated union allows for selecting just one of the "options", testing against it, and showing only the issues related to this "option".

```
const myUnion = z.discriminatedUnion("status", [
  z.object({ status: z.literal("success"), data: z.string() }),
  z.object({ status: z.literal("failed"), error: z.instanceof(Error) }),
]);

myUnion.parse({ type: "success", data: "yippie ki yay" });
```

Records

Record schemas are used to validate types such as `{ [k: string]: number }`.

If you want to validate the *values* of an object against some schema but don't care about the keys, use `z.record(valueType)`:

```
const NumberCache = z.record(z.number());

type NumberCache = z.infer<typeof NumberCache>;
// => { [k: string]: number }
```

This is particularly useful for storing or caching items by ID.

```
const userStore: UserStore = {};

userStore["77d2586b-9e8e-4ecf-8b21-ea7e0530eadd"] = {
  name: "Carlotta",
}; // passes

userStore["77d2586b-9e8e-4ecf-8b21-ea7e0530eadd"] = {
  whatever: "Ice cream sundae",
}; // TypeError
```

Record key type

If you want to validate both the keys and the values, use `z.record(keyType, valueType)`:

```
const NoEmptyKeysSchema = z.record(z.string().min(1), z.number());
NoEmptyKeysSchema.parse({ count: 1 }); // => { 'count': 1 }
NoEmptyKeysSchema.parse({ "": 1 }); // fails
```

(Notice how when passing two arguments, `valueType` is the second argument)

A note on numerical keys

While `z.record(keyType, valueType)` is able to accept numerical key types and TypeScript's built-in Record type is `Record<KeyType, ValueType>`, it's hard to represent the TypeScript type `Record<number, any>` in Zod.

As it turns out, TypeScript's behavior surrounding `[k: number]` is a little unintuitive:

```ts
const testMap: { [k: number]: string } = {
  1: "one",
};

for (const key in testMap) {
  console.log(`${key}: ${typeof key}`);
}
// prints: `1: string`
```

As you can see, JavaScript automatically casts all object keys to strings under the hood. Since Zod is trying to bridge the gap between static and runtime types, it doesn't make sense to provide a way of creating a record schema with numerical keys, since there's no such thing as a numerical key in runtime JavaScript.

Maps

```ts
const stringNumberMap = z.map(z.string(), z.number());

type StringNumberMap = z.infer<typeof stringNumberMap>;
// type StringNumberMap = Map<string, number>
```

Sets

```ts
const numberSet = z.set(z.number());
type NumberSet = z.infer<typeof numberSet>;
// type NumberSet = Set<number>
```

Set schemas can be further contrainted with the following utility methods.

```ts
z.set(z.string()).nonempty(); // must contain at least one item
z.set(z.string()).min(5); // must contain 5 or more items
z.set(z.string()).max(5); // must contain 5 or fewer items
z.set(z.string()).size(5); // must contain 5 items exactly
```

Intersections

Intersections are useful for creating "logical AND" types. This is useful for intersecting two object types.

```ts
const Person = z.object({
  name: z.string(),
});

const Employee = z.object({
  role: z.string(),
});

const EmployedPerson = z.intersection(Person, Employee);

// equivalent to:
const EmployedPerson = Person.and(Employee);
```

Though in many cases, it is recommended to use `A.merge(B)` to merge two objects. The `.merge` method returns a new `ZodObject` instance, whereas `A.and(B)` returns a less useful `ZodIntersection` instance that lacks common object methods like `pick` and `omit`.

```ts
const a = z.union([z.number(), z.string()]);
const b = z.union([z.number(), z.boolean()]);
const c = z.intersection(a, b);

type c = z.infer<typeof c>; // => number
```

Recursive types

You can define a recursive schema in Zod, but because of a limitation of TypeScript, their type can't be statically inferred. Instead you'll need to define the type definition manually, and provide it to Zod as a "type hint".

```ts
interface Category {
  name: string;
  subcategories: Category[];
}

// cast to z.ZodType<Category>
const Category: z.ZodType<Category> = z.lazy(() =>
  z.object({
    name: z.string(),
    subcategories: z.array(Category),
  })
);

Category.parse({
  name: "People",
  subcategories: [
    {
      name: "Politicians",
      subcategories: [{ name: "Presidents", subcategories: [] }],
    },
  ],
}); // passes
```

Unfortunately this code is a bit duplicative, since you're declaring the types twice: once in the interface and again in the Zod definition.

JSON type

If you want to validate any JSON value, you can use the snippet below.

```ts
const literalSchema = z.union([z.string(), z.number(), z.boolean(), z.null()]);
type Literal = z.infer<typeof literalSchema>;
type Json = Literal | { [key: string]: Json } | Json[];
const jsonSchema: z.ZodType<Json> = z.lazy(() =>
  z.union([literalSchema, z.array(jsonSchema), z.record(jsonSchema)])
);

jsonSchema.parse(data);
```

Thanks to ggoodman for suggesting this.

Cyclical objects

Despite supporting recursive schemas, passing cyclical data into Zod will cause an infinite loop.

Promises

```
const numberPromise = z.promise(z.number());
```

"Parsing" works a little differently with promise schemas. Validation happens in two parts:

1. Zod synchronously checks that the input is an instance of Promise (i.e. an object with `.then` and `.catch` methods.).
2. Zod uses `.then` to attach an additional validation step onto the existing Promise. You'll have to use `.catch` on the returned Promise to handle validation failures.

```
numberPromise.parse("tuna");
// ZodError: Non-Promise type: string

numberPromise.parse(Promise.resolve("tuna"));
// => Promise<number>

const test = async () => {
  await numberPromise.parse(Promise.resolve("tuna"));
  // ZodError: Non-number type: string

  await numberPromise.parse(Promise.resolve(3.14));
  // => 3.14
};
```

Instanceof

You can use `z.instanceof` to check that the input is an instance of a class. This is useful to validate inputs against classes that are exported from third-party libraries.

```
class Test {
  name: string;
}

const TestSchema = z.instanceof(Test);

const blob: any = "whatever";
TestSchema.parse(new Test()); // passes
TestSchema.parse("blob"); // throws
```

Function schemas

Zod also lets you define "function schemas". This makes it easy to validate the inputs and outputs of a function without intermixing your validation code and "business logic".

You can create a function schema with `z.function(args, returnType)` .

```
const myFunction = z.function();

type myFunction = z.infer<typeof myFunction>;
// => ()=>unknown
```

Define inputs and outputs.

```
const myFunction = z
  .function()
  .args(z.string(), z.number()) // accepts an arbitrary number of arguments
  .returns(z.boolean());
type myFunction = z.infer<typeof myFunction>;
// => (arg0: string, arg1: number)=>boolean
```

Function schemas have an `.implement()` method which accepts a function and returns a new function that automatically validates its inputs and outputs.

```
const trimmedLength = z
  .function()
  .args(z.string()) // accepts an arbitrary number of arguments
  .returns(z.number())
  .implement((x) => {
    // TypeScript knows x is a string!
    return x.trim().length;
  });

trimmedLength("sandwich"); // => 8
trimmedLength(" asdf "); // => 4
```

If you only care about validating inputs, just don't call the `.returns()` method. The output type will be inferred from the implementation.

> You can use the special `z.void()` option if your function doesn't return anything. This will let Zod properly infer the type of void-returning functions. (Void-returning functions actually return undefined.)

```
const myFunction = z
  .function()
  .args(z.string())
  .implement((arg) => {
    return [arg.length]; //
  });
myFunction; // (arg: string)=>number[]
```

Extract the input and output schemas from a function schema.

```
myFunction.parameters();
// => ZodTuple<[ZodString, ZodNumber]>

myFunction.returnType();
// => ZodBoolean
```

Preprocess

Typically Zod operates under a "parse then transform" paradigm. Zod validates the input first, then passes it through a chain of transformation functions. (For more information about transforms, read the .transform docs.)

But sometimes you want to apply some transform to the input *before* parsing happens. A common use case: type coercion. Zod enables this with the `z.preprocess()`.

```
const castToString = z.preprocess((val) => String(val), z.string());
```

This returns a `ZodEffects` instance. `ZodEffects` is a wrapper class that contains all logic pertaining to preprocessing, refinements, and transforms.

Schema methods

All Zod schemas contain certain methods.

`.parse`

`.parse(data: unknown): T`

Given any Zod schema, you can call its `.parse` method to check `data` is valid. If it is, a value is returned with full type information! Otherwise, an error is thrown.

> IMPORTANT: The value returned by `.parse` is a *deep clone* of the variable you passed in.

```
const stringSchema = z.string();
stringSchema.parse("fish"); // => returns "fish"
stringSchema.parse(12); // throws Error('Non-string type: number');
```

`.parseAsync`

`.parseAsync(data:unknown): Promise<T>`

If you use asynchronous refinements or transforms (more on those later), you'll need to use `.parseAsync`

```
const stringSchema1 = z.string().refine(async (val) => val.length < 20);
const value1 = await stringSchema.parseAsync("hello"); // => hello

const stringSchema2 = z.string().refine(async (val) => val.length > 20);
const value2 = await stringSchema.parseAsync("hello"); // => throws
```

`.safeParse`

`.safeParse(data:unknown): { success: true; data: T; } | { success: false; error: ZodError; }`

If you don't want Zod to throw errors when validation fails, use `.safeParse`. This method returns an object containing either the successfully parsed data or a ZodError instance containing detailed information about the validation problems.

```
stringSchema.safeParse(12);
// => { success: false; error: ZodError }

stringSchema.safeParse("billie");
// => { success: true; data: 'billie' }
```

The result is a *discriminated union* so you can handle errors very conveniently:

```
const result = stringSchema.safeParse("billie");
if (!result.success) {
  // handle error then return
  result.error;
} else {
  // do something
  result.data;
}
```

`.safeParseAsync`

> Alias: `.spa`

An asynchronous version of `safeParse`.

```
await stringSchema.safeParseAsync("billie");
```

For convenience, this has been aliased to `.spa`:

```
await stringSchema.spa("billie");
```

`.refine`

`.refine(validator: (data:T)=>any, params?: RefineParams)`

Zod lets you provide custom validation logic via *refinements*. (For advanced features like creating multiple issues and customizing error codes, see `.superRefine`.)

Zod was designed to mirror TypeScript as closely as possible. But there are many so-called "refinement types" you may wish to check for that can't be represented in TypeScript's type system. For instance: checking that a number is an integer or that a string is a valid email address.

For example, you can define a custom validation check on *any* Zod schema with `.refine`:

```
const myString = z.string().refine((val) => val.length <= 255, {
  message: "String can't be more than 255 characters",
});
```

> ⚠ Refinement functions should not throw. Instead they should return a falsy value to signal failure.

Arguments

As you can see, `.refine` takes two arguments.

1. The first is the validation function. This function takes one input (of type `T` — the inferred type of the schema) and returns `any`. Any truthy value will pass validation. (Prior to zod@1.6.2 the validation function had to return a boolean.)
2. The second argument accepts some options. You can use this to customize certain error-handling behavior:

```
type RefineParams = {
  // override error message
  message?: string;

  // appended to error path
  path?: (string | number)[];

  // params object you can use to customize message
  // in error map
  params?: object;
};
```

For advanced cases, the second argument can also be a function that returns `RefineParams`/

```
z.string().refine(
  (val) => val.length > 10,
  (val) => ({ message: `${val} is not more than 10 characters` })
);
```

Customize error path

```
const passwordForm = z
  .object({
    password: z.string(),
    confirm: z.string(),
  })
  .refine((data) => data.password === data.confirm, {
    message: "Passwords don't match",
    path: ["confirm"], // path of error
  })
  .parse({ password: "asdf", confirm: "qwer" });
```

Because you provided a `path` parameter, the resulting error will be:

```
ZodError {
  issues: [{
    "code": "custom",
    "path": [ "confirm" ],
    "message": "Passwords don't match"
  }]
}
```

Asynchronous refinements

Refinements can also be async:

```ts
const userId = z.string().refine(async (id) => {
  // verify that ID exists in database
  return true;
});
```

⚠ If you use async refinements, you must use the `.parseAsync` method to parse data! Otherwise Zod will throw an error.

Relationship to transforms

Transforms and refinements can be interleaved:

```ts
z.string()
  .transform((val) => val.length)
  .refine((val) => val > 25);
```

`.superRefine`

The `.refine` method is actually syntactic sugar atop a more versatile (and verbose) method called `superRefine`. Here's an example:

```ts
const Strings = z.array(z.string()).superRefine((val, ctx) => {
  if (val.length > 3) {
    ctx.addIssue({
      code: z.ZodIssueCode.too_big,
      maximum: 3,
      type: "array",
      inclusive: true,
      message: "Too many items 😡",
    });
  }

  if (val.length !== new Set(val).size) {
    ctx.addIssue({
      code: z.ZodIssueCode.custom,
      message: `No duplicates allowed.`,
    });
  }
});
```

You can add as many issues as you like. If `ctx.addIssue` is *not* called during the execution of the function, validation passes.

Normally refinements always create issues with a `ZodIssueCode.custom` error code, but with `superRefine` you can create any issue of any code. Each issue code is described in detail in the Error Handling guide: ERROR_HANDLING.md.

Abort early

By default, parsing will continue even after a refinement check fails. For instance, if you chain together multiple refinements, they will all be executed. However, it may be desirable to *abort early* to prevent later refinements from being executed. To achieve this, pass the `fatal` flag to `ctx.addIssue` and return `z.NEVER`.

```
const schema = z.number().superRefine((val, ctx) => {
  if (val < 10) {
    ctx.addIssue({
      code: z.ZodIssueCode.custom,
      message: "should be >= 10",
      fatal: true,
    });

    return z.NEVER;
  }

  if (val !== 12) {
    ctx.addIssue({
      code: z.ZodIssueCode.custom,
      message: "should be twelve",
    });
  }
});
```

`.transform`

To transform data after parsing, use the `transform` method.

```
const stringToNumber = z.string().transform((val) => val.length);
stringToNumber.parse("string"); // => 6
```

Chaining order

Note that `stringToNumber` above is an instance of the `ZodEffects` subclass. It is NOT an instance of `ZodString`. If you want to use the built-in methods of `ZodString` (e.g. `.email()`) you must apply those methods *before* any transforms.

```
const emailToDomain = z
  .string()
  .email()
  .transform((val) => val.split("@")[1]);

emailToDomain.parse("colinhacks@example.com"); // => example.com
```

Validating during transform

The `.transform` method can simultaneously validate and transform the value. This is often simpler and less duplicative than chaining `refine` and `validate`.

As with `.superRefine`, the transform function receives a `ctx` object with a `addIssue` method that can be used to register validation issues.

```ts
const Strings = z.string().transform((val, ctx) => {
  const parsed = parseInt(val);
  if (isNaN(parsed)) {
    ctx.addIssue({
      code: z.ZodIssueCode.custom,
      message: "Not a number",
    });

    // This is a special symbol you can use to
    // return early from the transform function.
    // It has type `never` so it does not affect the
    // inferred return type.
    return z.NEVER;
  }
  return parsed;
});
```

Relationship to refinements

Transforms and refinements can be interleaved. These will be executed in the order they are declared.

```ts
z.string()
  .transform((val) => val.toUpperCase())
  .refine((val) => val.length > 15)
  .transform((val) => `Hello ${val}`)
  .refine((val) => val.indexOf("!") === -1);
```

Async transforms

Transforms can also be async.

```ts
const IdToUser = z
  .string()
  .uuid()
  .transform(async (id) => {
    return await getUserById(id);
  });
```

> ⚠ If your schema contains asynchronous transforms, you must use .parseAsync() or .safeParseAsync() to parse data. Otherwise Zod will throw an error.

`.default`

You can use transforms to implement the concept of "default values" in Zod.

```ts
const stringWithDefault = z.string().default("tuna");

stringWithDefault.parse(undefined); // => "tuna"
```

Optionally, you can pass a function into `.default` that will be re-executed whenever a default value needs to be generated:

```ts
const numberWithRandomDefault = z.number().default(Math.random);

numberWithRandomDefault.parse(undefined); // => 0.4413456736055323
numberWithRandomDefault.parse(undefined); // => 0.1871840107401901
numberWithRandomDefault.parse(undefined); // => 0.7223408162401552
```

Conceptually, this is how Zod processes default values:

1. If the input is `undefined`, the default value is returned
2. Otherwise, the data is parsed using the base schema

`.catch`

Use `.catch()` to provide a "catch value" to be returned in the event of a parsing error.

```ts
const numberWithCatch = z.number().catch(42);

numberWithCatch.parse(5); // => 5
numberWithCatch.parse("tuna"); // => 42
```

Optionally, you can pass a function into `.catch` that will be re-executed whenever a default value needs to be generated:

```ts
const numberWithRandomCatch = z.number().catch(Math.random);

numberWithRandomDefault.parse("sup"); // => 0.4413456736055323
numberWithRandomDefault.parse("sup"); // => 0.1871840107401901
numberWithRandomDefault.parse("sup"); // => 0.7223408162401552
```

Conceptually, this is how Zod processes "catch values":

1. The data is parsed using the base schema
2. If the parsing fails, the "catch value" is returned

`.optional`

A convenience method that returns an optional version of a schema.

```ts
const optionalString = z.string().optional(); // string | undefined

// equivalent to
z.optional(z.string());
```

`.nullable`

A convenience method that returns a nullable version of a schema.

```ts
const nullableString = z.string().nullable(); // string | null

// equivalent to
z.nullable(z.string());
```

.nullish

A convenience method that returns a "nullish" version of a schema. Nullish schemas will accept both `undefined` and `null`. Read more about the concept of "nullish" in the TypeScript 3.7 release notes.

```
const nullishString = z.string().nullish(); // string | null | undefined

// equivalent to
z.string().optional().nullable();
```

.array

A convenience method that returns an array schema for the given type:

```
const nullableString = z.string().array(); // string[]

// equivalent to
z.array(z.string());
```

.promise

A convenience method for promise types:

```
const stringPromise = z.string().promise(); // Promise<string>

// equivalent to
z.promise(z.string());
```

.or

A convenience method for union types.

```
z.string().or(z.number()); // string | number

// equivalent to
z.union([z.string(), z.number()]);
```

.and

A convenience method for creating intersection types.

```
z.object({ name: z.string() }).and(z.object({ age: z.number() })); // { name:
string } & { age: number }

// equivalent to
z.intersection(z.object({ name: z.string() }), z.object({ age: z.number() }));
```

.brand

`.brand<T>() => ZodBranded<this, B>`

TypeScript's type system is structural, which means that any two types that are structurally equivalent are considered the same.

```ts
type Cat = { name: string };
type Dog = { name: string };

const petCat = (cat: Cat) => {};
const fido: Dog = { name: "fido" };
petCat(fido); // works fine
```

In some cases, its can be desirable to simulate *nominal typing* inside TypeScript. For instance, you may wish to write a function that only accepts an input that has been validated by Zod. This can be achieved with *branded types* (AKA *opaque types*).

```ts
const Cat = z.object({ name: z.string() }).brand<"Cat">();
type Cat = z.infer<typeof Cat>;

const petCat = (cat: Cat) => {};

// this works
const simba = Cat.parse({ name: "simba" });
petCat(simba);

// this doesn't
petCat({ name: "fido" });
```

Under the hood, this works by attaching a "brand" to the inferred type using an intersection type. This way, plain/unbranded data structures are no longer assignable to the inferred type of the schema.

```ts
const Cat = z.object({ name: z.string() }).brand<"Cat">();
type Cat = z.infer<typeof Cat>;
// {name: string} & {[symbol]: "Cat"}
```

Note that branded types do not affect the runtime result of `.parse`. It is a static-only construct.

Guides and concepts

Type inference

You can extract the TypeScript type of any schema with `z.infer<typeof mySchema>` .

```ts
const A = z.string();
type A = z.infer<typeof A>; // string

const u: A = 12; // TypeError
const u: A = "asdf"; // compiles
```

What about transforms?

In reality each Zod schema internally tracks **two** types: an input and an output. For most schemas (e.g. `z.string()`) these two are the same. But once you add transforms into the mix, these two values can diverge. For instance `z.string().transform(val => val.length)` has an input of `string` and an output of `number` .

You can separately extract the input and output types like so:

```
const stringToNumber = z.string().transform((val) => val.length);

// ⚠️ Important: z.infer returns the OUTPUT type!
type input = z.input<typeof stringToNumber>; // string
type output = z.output<typeof stringToNumber>; // number

// equivalent to z.output!
type inferred = z.infer<typeof stringToNumber>; // number
```

Writing generic functions

When attempting to write a functions that accepts a Zod schemas as an input, it's common to try something like this:

```
function makeSchemaOptional<T>(schema: z.ZodType<T>) {
  return schema.optional();
}
```

This approach has some issues. The `schema` variable in this function is typed as an instance of `ZodType`, which is an abstract class that all Zod schemas inherit from. This approach loses type information, namely *which subclass* the input actually is.

```
const arg = makeSchemaOptional(z.string());
arg.unwrap();
```

A better approach is for the generate parameter to refer to *the schema as a whole*.

```
function makeSchemaOptional<T extends z.ZodTypeAny>(schema: T) {
  return schema.optional();
}
```

> `ZodTypeAny` is just a shorthand for `ZodType<any, any, any>`, a type that is broad enough to match any Zod schema.

As you can see, `schema` is now fully and properly typed.

```
const arg = makeSchemaOptional(z.string());
arg.unwrap(); // ZodString
```

Constraining allowable inputs

The `ZodType` class has three generic parameters.

```
class ZodType<
  Output = any,
  Def extends ZodTypeDef = ZodTypeDef,
  Input = Output
> { ... }
```

By constraining these in your generic input, you can limit what schemas are allowable as inputs to your function:

```
function makeSchemaOptional<T extends z.ZodType<string>>(schema: T) {
  return schema.optional();
}

makeSchemaOptional(z.string());
// works fine

makeSchemaOptional(z.number());
// Error: 'ZodNumber' is not assignable to parameter of type 'ZodType<string,
ZodTypeDef, string>'
```

Error handling

Zod provides a subclass of Error called `ZodError`. ZodErrors contain an `issues` array containing detailed information about the validation problems.

```
const data = z
  .object({
    name: z.string(),
  })
  .safeParse({ name: 12 });

if (!data.success) {
  data.error.issues;
  /* [
      {
        "code": "invalid_type",
        "expected": "string",
        "received": "number",
        "path": [ "name" ],
        "message": "Expected string, received number"
      }
  ] */
}
```

For detailed information about the possible error codes and how to customize error messages, check out the dedicated error handling guide: ERROR_HANDLING.md

Zod's error reporting emphasizes *completeness* and *correctness*. If you are looking to present a useful error message to the end user, you should either override Zod's error messages using an error map (described in detail in the Error Handling guide) or use a third party library like `zod-validation-error`

Error formatting

You can use the `.format()` method to convert this error into a nested object.

```
const data = z
  .object({
    name: z.string(),
  })
  .safeParse({ name: 12 });

if (!data.success) {
  const formatted = data.error.format();
```

```
  /* {
    name: { _errors: [ 'Expected string, received number' ] }
  } */

  formatted.name?._errors;
  // => ["Expected string, received number"]
}
```

Comparison

There are a handful of other widely-used validation libraries, but all of them have certain design limitations that make for a non-ideal developer experience.

Joi

https://github.com/hapijs/joi

Doesn't support static type inference 😐

Yup

https://github.com/jquense/yup

Yup is a full-featured library that was implemented first in vanilla JS, and later rewritten in TypeScript.

- Supports casting and transforms
- All object fields are optional by default
- Missing object methods: (partial, deepPartial)
- Missing promise schemas
- Missing function schemas
- Missing union & intersection schemas

io-ts

https://github.com/gcanti/io-ts

io-ts is an excellent library by gcanti. The API of io-ts heavily inspired the design of Zod.

In our experience, io-ts prioritizes functional programming purity over developer experience in many cases. This is a valid and admirable design goal, but it makes io-ts particularly hard to integrate into an existing codebase with a more procedural or object-oriented bias. For instance, consider how to define an object with optional properties in io-ts:

```
import * as t from "io-ts";

const A = t.type({
  foo: t.string,
});

const B = t.partial({
  bar: t.number,
});
```

```
const C = t.intersection([A, B]);

type C = t.TypeOf<typeof C>;
// returns { foo: string; bar?: number | undefined }
```

You must define the required and optional props in separate object validators, pass the optionals through `t.partial` (which marks all properties as optional), then combine them with `t.intersection` .

Consider the equivalent in Zod:

```
const C = z.object({
  foo: z.string(),
  bar: z.number().optional(),
});

type C = z.infer<typeof C>;
// returns { foo: string; bar?: number | undefined }
```

This more declarative API makes schema definitions vastly more concise.

`io-ts` also requires the use of gcanti's functional programming library `fp-ts` to parse results and handle errors. This is another fantastic resource for developers looking to keep their codebase strictly functional. But depending on `fp-ts` necessarily comes with a lot of intellectual overhead; a developer has to be familiar with functional programming concepts and the `fp-ts` nomenclature to use the library.

- Supports codecs with serialization & deserialization transforms
- Supports branded types
- Supports advanced functional programming, higher-kinded types, `fp-ts` compatibility
- Missing object methods: (pick, omit, partial, deepPartial, merge, extend)
- Missing nonempty arrays with proper typing (`[T, ...T[]]`)
- Missing promise schemas
- Missing function schemas

Runtypes

https://github.com/pelotom/runtypes

Good type inference support, but limited options for object type masking (no `.pick` , `.omit` , `.extend` , etc.). No support for `Record` s (their `Record` is equivalent to Zod's `object`). They DO support branded and readonly types, which Zod does not.

- Supports "pattern matching": computed properties that distribute over unions
- Supports readonly types
- Missing object methods: (deepPartial, merge)
- Missing nonempty arrays with proper typing (`[T, ...T[]]`)
- Missing promise schemas
- Missing error customization

Ow

https://github.com/sindresorhus/ow

Ow is focused on function input validation. It's a library that makes it easy to express complicated assert statements, but it doesn't let you parse untyped data. They support a much wider variety of types; Zod has a nearly one-to-one mapping with TypeScript's type system, whereas ow lets you validate several highly-specific types out of the box (e.g. `int32Array` , see full list in their README).

If you want to validate function inputs, use function schemas in Zod! It's a much simpler approach that lets you reuse a function type declaration without repeating yourself (namely, copy-pasting a bunch of ow assertions at the beginning of every function). Also Zod lets you validate your return types as well, so you can be sure there won't be any unexpected data passed downstream.

Changelog

View the changelog at CHANGELOG.md

Colophon

This book is created by using the following sources:

- Zod - English
- GitHub source: colinhacks/zod
- Created: 2022-12-11
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf