On this page > drops `errorMap`

Migration guide

This migration guide aims to list the breaking changes in Zod 4 in order of highest to lowest impact. To learn more about the performance enhancements and new features of Zod 4, read the introductory post.

```
npm upgrade zod@^4.0.0
```

Note — Zod 3 exported a number of undocumented quasi-internal utility types and functions that are not considered part of the public API. Changes to those are not documented here.

Error customization

Zod 4 standardizes the APIs for error customization under a single, unified `error` param. Previously Zod's error customization APIs were fragmented and inconsistent. This is cleaned up in Zod 4.

deprecates `message`

Replaces `message` with `error`. The `message` parameter is still supported but deprecated.

Zod 4 Zod 3

Zod 4 is now stable! Read the announcement.

drops `invalid_type_error` and `required_error`

Ask AI

The `invalid_type_error` / `required_error` params have been dropped. These were hastily added years ago as a way to customize errors that was less verbose than `errorMap`. They came with all sorts of footguns (they can't be used in conjunction with `errorMap`) and do not align with Zod's actual issue codes (there is no `required` issue code).

These can now be cleanly represented with the new `error` parameter.

Zod 4 Zod 3

```
z.string({
  error: (issue) => issue.input === undefined
    ? "This field is required"
    : "Not a string"
});
```

drops `errorMap`

This is renamed to `error`.

Error maps can also now return a plain `string` (instead of `{message: string}`). They can also return `undefined`, which tells Zod to yield control to the next error map in the chain.

Zod 4 Zod 3

```
z.string().min(5, {
  error: (issue) => {
    if (issue.code === "too_small") {
      return `Value must be >${issue.minimum}`
    }
  },
});
```

`ZodError`

updates issue formats

The issue formats have been dramatically streamlined.

```
import * as z from "zod"; // v4
```

```
type IssueFormats =
  | z.core.$ZodIssueInvalidType
  | z.core.$ZodIssueTooBig
  | z.core.$ZodIssueTooSmall
  | z.core.$ZodIssueInvalidStringFormat
  | z.core.$ZodIssueNotMultipleOf
  | z.core.$ZodIssueUnrecognizedKeys
  | z.core.$ZodIssueInvalidValue
  | z.core.$ZodIssueInvalidUnion
  | z.core.$ZodIssueInvalidKey // new: used for z.record/z.map
  | z.core.$ZodIssueInvalidElement // new: used for z.map/z.set
  | z.core.$ZodIssueCustom;
```

Below is the list of Zod 3 issues types and their Zod 4 equivalent:

```
import * as z from "zod"; // v3

export type IssueFormats =
  | z.ZodInvalidTypeIssue // ♻️ renamed to z.core.$ZodIssueInvalidType
  | z.ZodTooBigIssue  // ♻️ renamed to z.core.$ZodIssueTooBig
  | z.ZodTooSmallIssue // ♻️ renamed to z.core.$ZodIssueTooSmall
  | z.ZodInvalidStringIssue // ♻️ z.core.$ZodIssueInvalidStringFormat
  | z.ZodNotMultipleOfIssue // ♻️ renamed to z.core.$ZodIssueNotMultipleOf
  | z.ZodUnrecognizedKeysIssue // ♻️ renamed to z.core.$ZodIssueUnrecognizedKeys
  | z.ZodInvalidUnionIssue // ♻️ renamed to z.core.$ZodIssueInvalidUnion
  | z.ZodCustomIssue // ♻️ renamed to z.core.$ZodIssueCustom
  | z.ZodInvalidEnumValueIssue // ❌ merged in z.core.$ZodIssueInvalidValue
  | z.ZodInvalidLiteralIssue // ❌ merged into z.core.$ZodIssueInvalidValue
  | z.ZodInvalidUnionDiscriminatorIssue // ❌ throws an Error at schema creation time
  | z.ZodInvalidArgumentsIssue // ❌ z.function throws ZodError directly
  | z.ZodInvalidReturnTypeIssue // ❌ z.function throws ZodError directly
  | z.ZodInvalidDateIssue // ❌ merged into invalid_type
  | z.ZodInvalidIntersectionTypesIssue // ❌ removed (throws regular Error)
  | z.ZodNotFiniteIssue // ❌ infinite values no longer accepted (invalid_type)
```

While certain Zod 4 issue types have been merged, dropped, and modified, each issue remains structurally similar to Zod 3 counterpart (identical, in most cases). All issues still conform to the same base interface as Zod 3, so most common error handling logic will work without modification.

```
export interface $ZodIssueBase {
  readonly code?: string;
  readonly input?: unknown;
  readonly path: PropertyKey[];
  readonly message: string;
}
```

changes error map precedence

The error map precedence has been changed to be more consistent. Specifically, an error map passed into `.parse()` *no longer* takes precedence over a schema-level error map.

```
const mySchema = z.string({ error: () => "Schema-level error" });

// in Zod 3
mySchema.parse(12, { error: () => "Contextual error" }); // => "Contextual error"

// in Zod 4
mySchema.parse(12, { error: () => "Contextual error" }); // => "Schema-level error"
```

deprecates `.format()`

The `.format()` method on `ZodError` has been deprecated. Instead use the top-level `z.treeifyError()` function. Read the Formatting errors docs for more information.

deprecates `.flatten()`

The `.flatten()` method on `ZodError` has also been deprecated. Instead use the top-level `z.treeifyError()` function. Read the Formatting errors docs for more information.

drops `.formErrors`

This API was identical to `.flatten()`. It exists for historical reasons and isn't documented.

deprecates `.addIssue()` and `.addIssues()`

Directly push to `err.issues` array instead, if necessary.

```
myError.issues.push({
  // new issue
});
```

`z.number()`

no infinite values

`POSITIVE_INFINITY` and `NEGATIVE_INFINITY` are no longer considered valid values for `z.number()` .

`.safe()` no longer accepts floats

In Zod 3, `z.number().safe()` is deprecated. It now behaves identically to `.int()` (see below). Importantly, that means it no longer accepts floats.

`.int()` accepts safe integers only

The `z.number().int()` API no longer accepts unsafe integers (outside the range of `Number.MIN_SAFE_INTEGER` and `Number.MAX_SAFE_INTEGER`). Using integers out of this range causes spontaneous rounding errors. (Also: You should switch to `z.int()` .)

`z.string()` updates

deprecates `.email()` etc

String formats are now represented as *subclasses* of `ZodString` , instead of simple internal refinements. As such, these APIs have been moved to the top-level `z` namespace. Top-level APIs are also less verbose and more tree-shakable.

```
z.email();
z.uuid();
z.url();
z.emoji();         // validates a single emoji character
z.base64();
z.base64url();
z.nanoid();
z.cuid();
z.cuid2();
z.ulid();
z.ipv4();
z.ipv6();
z.cidrv4();          // ip range
z.cidrv6();          // ip range
z.iso.date();
z.iso.time();
z.iso.datetime();
```

```
z.iso.duration();
```

The method forms (`z.string().email()`) still exist and work as before, but are now deprecated.

```
z.string().email(); // ❌ deprecated
z.email(); // ✅
```

stricter `.uuid()`

The `z.uuid()` now validates UUIDs more strictly against the RFC 9562/4122 specification; specifically, the variant bits must be `10` per the spec. For a more permissive "UUID-like" validator, use `z.guid()` .

```
z.uuid(); // RFC 9562/4122 compliant UUID
z.guid(); // any 8-4-4-4-12 hex pattern
```

no padding in `.base64url()`

Padding is no longer allowed in `z.base64url()` (formerly `z.string().base64url()`). Generally it's desirable for base64url strings to be unpadded and URL-safe.

drops `z.string().ip()`

This has been replaced with separate `.ipv4()` and `.ipv6()` methods. Use `z.union()` to combine them if you need to accept both.

```
z.string().ip() // ❌
z.ipv4() // ✅
z.ipv6() // ✅
```

updates `z.string().ipv6()`

Validation now happens using the `new URL()` constructor, which is far more robust than the old regular expression approach. Some invalid values that passed validation previously may now fail.

drops `z.string().cidr()`

Similarly, this has been replaced with separate `.cidrv4()` and `.cidrv6()` methods. Use `z.union()` to combine them if you need to accept both.

```
z.string().cidr() // ❌
z.cidrv4() // ✅
z.cidrv6() // ✅
```

`z.coerce` updates

The input type of all `z.coerce` schemas is now `unknown` .

```
const schema = z.coerce.string();
type schemaInput = z.input<typeof schema>;

// Zod 3: string;
// Zod 4: unknown;
```

`.default()` updates

The application of `.default()` has changed in a subtle way. If the input is `undefined` , `ZodDefault` short-circuits the parsing process and returns the default value. The default value must be assignable to the *output type*.

```
const schema = z.string()
  .transform(val => val.length)
  .default(0); // should be a number
schema.parse(undefined); // => 0
```

In Zod 3, `.default()` expected a value that matched the *input type*. `ZodDefault` would parse the default value, instead of short-circuiting. As such, the default value must be assignable to the *input type* of the schema.

```
// Zod 3
```

```
const schema = z.string()
  .transform(val => val.length)
  .default("tuna");
schema.parse(undefined); // => 4
```

To replicate the old behavior, Zod implements a new `.prefault()` API. This is short for "pre-parse default".

```
// Zod 3
const schema = z.string()
  .transform(val => val.length)
  .prefault("tuna");
schema.parse(undefined); // => 4
```

`z.object()`

deprecates `.strict()` and `.passthrough()`

These methods are generally no longer necessary. Instead use the top-level `z.strictObject()` and `z.looseObject()` functions.

```
// Zod 3
z.object({ name: z.string() }).strict();
z.object({ name: z.string() }).passthrough();

// Zod 4
z.strictObject({ name: z.string() });
z.looseObject({ name: z.string() });
```

These methods are still available for backwards compatibility, and they will not be removed. They are considered legacy.

deprecates `.strip()`

This was never particularly useful, as it was the default behavior of `z.object()` . To convert a strict object to a "regular" one, use `z.object(A.shape)` .

drops `.nonstrict()`

This long-deprecated alias for `.strip()` has been removed.

drops `.deepPartial()`

This has been long deprecated in Zod 3 and it now removed in Zod 4. There is no direct alternative to this API. There were lots of footguns in its implementation, and its use is generally an anti-pattern.

changes `z.unknown()` optionality

The `z.unknown()` and `z.any()` types are no longer marked as "key optional" in the inferred types.

```
const mySchema = z.object({
  a: z.any(),
  b: z.unknown()
});
// Zod 3: { a?: any; b?: unknown };
// Zod 4: { a: any; b: unknown };
```

deprecates `.merge()`

The `.merge()` method on `ZodObject` has been deprecated in favor of `.extend()`. The `.extend()` method provides the same functionality, avoids ambiguity around strictness inheritance, and has better TypeScript performance.

```
// .merge (deprecated)
const ExtendedSchema = BaseSchema.merge(AdditionalSchema);

// .extend (recommended)
const ExtendedSchema = BaseSchema.extend(AdditionalSchema.shape);

// or use destructuring (best tsc performance)
const ExtendedSchema = z.object({
  ...BaseSchema.shape,
  ...AdditionalSchema.shape,
});
```

Note: For even better TypeScript performance, consider using object destructuring instead of `.extend()`. See the API documentation for more details.

`z.nativeEnum()` deprecated

The `z.nativeEnum()` function is now deprecated in favor of just `z.enum()` . The `z.enum()` API has been overloaded to support an enum-like input.

```
enum Color {
  Red = "red",
  Green = "green",
  Blue = "blue",
}

const ColorSchema = z.enum(Color); // ✅
```

As part of this refactor of `ZodEnum` , a number of long-deprecated and redundant features have been removed. These were all identical and only existed for historical reasons.

```
ColorSchema.enum.Red; // ✅ => "Red" (canonical API)
ColorSchema.Enum.Red; // ❌ removed
ColorSchema.Values.Red; // ❌ removed
```

`z.array()`

changes `.nonempty()` type

This now behaves identically to `z.array().min(1)` . The inferred type does not change.

```
const NonEmpty = z.array(z.string()).nonempty();

type NonEmpty = z.infer<typeof NonEmpty>;
// Zod 3: [string, ...string[]]
// Zod 4: string[]
```

The old behavior is now better represented with `z.tuple()` and a "rest" argument. This aligns more closely to TypeScript's type system.

```
z.tuple([z.string()], z.string());
// => [string, ...string[]]
```

`z.promise()` deprecated

There's rarely a reason to use `z.promise()`. If you have an input that may be a `Promise`, just `await` it before parsing it with Zod.

> If you are using `z.promise` to define an async function with `z.function()`, that's no longer necessary either; see the `ZodFunction` section below.

`z.function()`

The result of `z.function()` is no longer a Zod schema. Instead, it acts as a standalone "function factory" for defining Zod-validated functions. The API has also changed; you define an `input` and `output` schema upfront, instead of using `args()` and `.returns()` methods.

Zod 4 Zod 3

```
const myFunction = z.function({
  input: [z.object({
    name: z.string(),
    age: z.number().int(),
  })],
  output: z.string(),
});

myFunction.implement((input) => {
  return `Hello ${input.name}, you are ${input.age} years old.`;
});
```

If you have a desperate need for a Zod schema with a function type, consider this workaround.

adds `.implementAsync()`

To define an async function, use `implementAsync()` instead of `implement()` .

```
myFunction.implementAsync(async (input) => {
  return `Hello ${input.name}, you are ${input.age} years old.`;
});
```

`.refine()`

ignores type predicates

In Zod 3, passing a type predicate as a refinement functions could still narrow the type of a schema. This wasn't documented but was discussed in some issues. This is no longer the case.

```
const mySchema = z.unknown().refine((val): val is string => {
  return typeof val === "string"
});

type MySchema = z.infer<typeof mySchema>;
// Zod 3: `string`
// Zod 4: still `unknown`
```

drops `ctx.path`

Zod's new parsing architecture does not eagerly evaluate the `path` array. This was a necessary change that unlocks Zod 4's dramatic performance improvements.

```
z.string().superRefine((val, ctx) => {
  ctx.path; // ❌ no longer available
});
```

drops function as second argument

The following horrifying overload has been removed.

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

`.superRefine()` deprecated

The `.superRefine()` method has been deprecated in favor of `.check()`. The `.check()` method provides the same functionality with a cleaner API. The `.check()` method is also available on Zod and Zod Mini schemas.

```
const UniqueStringArray = z.array(z.string()).check((ctx) => {
  if (ctx.value.length > 3) {
    ctx.issues.push({
      code: "too_big",
      maximum: 3,
      origin: "array",
      inclusive: true,
      message: "Too many items 😡",
      input: ctx.value
    });
  }

  if (ctx.value.length !== new Set(ctx.value).size) {
    ctx.issues.push({
      code: "custom",
      message: `No duplicates allowed.`,
      input: ctx.value
    });
  }
});
```

`z.ostring()`, etc dropped

The undocumented convenience methods `z.ostring()`, `z.onumber()`, etc. have been removed. These were shorthand methods for defining optional string schemas.

`z.literal()`

drops `symbol` support

Symbols aren't considered literal values, nor can they be simply compared with `===`. This was an oversight in Zod 3.

static `.create()` factories dropped

Previously all Zod classes defined a static `.create()` method. These are now implemented as standalone factory functions.

```
z.ZodString.create(); // ❌
```

`z.record()`

drops single argument usage

Before, `z.record()` could be used with a single argument. This is no longer supported.

```
// Zod 3
z.record(z.string()); // ✅

// Zod 4
z.record(z.string()); // ❌
z.record(z.string(), z.string()); // ✅
```

improves enum support

Records have gotten a lot smarter. In Zod 3, passing an enum into `z.record()` as a key schema would result in a partial type

```
const myRecord = z.record(z.enum(["a", "b", "c"]), z.number());
// { a?: number; b?: number; c?: number; }
```

In Zod 4, this is no longer the case. The inferred type is what you'd expect, and Zod ensures exhaustiveness; that is, it makes sure all enum keys exist in the input during parsing.

```
const myRecord = z.record(z.enum(["a", "b", "c"]), z.number());
// { a: number; b: number; c: number; }
```

To replicate the old behavior with optional keys, use `z.partialRecord()` :

```
const myRecord = z.partialRecord(z.enum(["a", "b", "c"]), z.number());
// { a?: number; b?: number; c?: number; }
```

`z.intersection()`

throws `Error` on merge conflict

Zod intersection parses the input against two schemas, then attempts to merge the results. In Zod 3, when the results were unmergable, Zod threw a `ZodError` with a special `"invalid_intersection_types"` issue.

In Zod 4, this will throw a regular `Error` instead. The existence of unmergable results indicates a structural problem with the schema: an intersection of two incompatible types. Thus, a regular error is more appropriate than a validation error.

Internal changes

The typical user of Zod can likely ignore everything below this line. These changes do not impact the user-facing `z` APIs.

There are too many internal changes to list here, but some may be relevant to regular users who are (intentionally or not) relying on certain implementation details. These changes will be of particular interest to library authors building tools on top of Zod.

updates generics

The generic structure of several classes has changed. Perhaps most significant is the change to the `ZodType` base class:

```
// Zod 3
class ZodType<Output, Def extends z.ZodTypeDef, Input = Output> {
  // ...
}

// Zod 4
class ZodType<Output = unknown, Input = unknown> {
  // ...
}
```

The second generic `Def` has been entirely removed. Instead the base class now only tracks `Output` and `Input`. While previously the `Input` value defaulted to `Output`, it now defaults to `unknown`. This allows generic functions involving `z.ZodType` to behave more intuitively in many cases.

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

inferSchema(z.string()); // z.ZodString
```

The need for `z.ZodTypeAny` has been eliminated; just use `z.ZodType` instead.

adds `z.core`

Many utility functions and types have been moved to the new `zod/v4/core` sub-package, to facilitate code sharing between Zod and Zod Mini.

```
import * as z from "zod/v4/core";

function handleError(iss: z.$ZodError) {
  // do stuff
}
```

For convenience, the contents of `zod/v4/core` are also re-exported from `zod` and `zod/mini` under the `z.core` namespace.

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

function handleError(iss: z.core.$ZodError) {
  // do stuff
}
```

Refer to the Zod Core docs for more information on the contents of the core sub-library.

moves `._def`

The `._def` property is now moved to `._zod.def`. The structure of all internal defs is subject to change; this is relevant to library authors but won't be comprehensively documented here.

drops `ZodEffects`

This doesn't affect the user-facing APIs, but it's an internal change worth highlighting. It's part of a larger restructure of how Zod handles *refinements*.

Previously both refinements and transformations lived inside a wrapper class called `ZodEffects`. That means adding either one to a schema would wrap the original schema in a `ZodEffects` instance. In Zod 4, refinements now live inside the schemas themselves. More accurately, each schema contains an array of "checks"; the concept of a "check" is new in Zod 4 and generalizes the concept of a refinement to include potentially side-effectful transforms like `z.toLowerCase()`.

This is particularly apparent in the Zod Mini API, which heavily relies on the `.check()` method to compose various validations together.

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

z.string().check(
  z.minLength(10),
  z.maxLength(100),
  z.toLowerCase(),
  z.trim(),
);
```

adds `ZodTransform`

Meanwhile, transforms have been moved into a dedicated `ZodTransform` class. This schema class represents an input transform; in fact, you can actually define standalone transformations now:

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

const schema = z.transform(input => String(input));

schema.parse(12); // => "12"
```

This is primarily used in conjunction with `ZodPipe`. The `.transform()` method now returns an instance of `ZodPipe`.

```
z.string().transform(val => val); // ZodPipe<ZodString, ZodTransform>
```

drops `ZodPreprocess`

As with `.transform()`, the `z.preprocess()` function now returns a `ZodPipe` instance instead of a dedicated `ZodPreprocess` instance.

```
z.preprocess(val => val, z.string()); // ZodPipe<ZodTransform, ZodString>
```

drops `ZodBranded`

Branding is now handled with a direct modification to the inferred type, instead of a dedicated `ZodBranded` class. The user-facing APIs remain the same.

Edit on GitHub

‹ Next
Introducing Zod 4

Next ›
Versioning