Error Handling in Zod

This guide explains Zod's internal error handling system, and the various ways you can customize it for your purposes.

ZodError

All validation errors thrown by Zod are instances of `ZodError`.

```ts
class ZodError extends Error {
  issues: ZodIssue[];
}
```

ZodError is a subclass of `Error`; you can create your own instance easily:

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

const myError = new z.ZodError([]);
```

Each ZodError has an `issues` property that is an array of `ZodIssues`. Each issue documents a problem that occurred during validation.

ZodIssue

`ZodIssue` is *not* a class. It is a **discriminated union**.

The link above is the best way to learn about the concept. Discriminated unions are an ideal way to represent a data structures that may be one of many possible variants. You can see all the possible variants defined **here**. They are also described in the table below if you prefer.

Every ZodIssue has these fields:

field	type	details
`code`	`z.ZodIssueCode`	You can access this enum with `z.ZodIssueCode` A full breakdown of the possible values is below.
`path`	`(string | number)[]`	e.g, `['addresses', 0, 'line1']`
`message`	`string`	e.g. `Invalid type. Expected string, received number.`

However depending on the error code, there may be additional properties as well. Here is a full breakdown of the additional fields by error code:

ZodIssueCode

code	additiona
ZodIssueCode.invalid_type	`expected: Zoc` `received: Zoc` Jump to **this se** breakdown of th values of ZodPa
ZodIssueCode.unrecognized_keys	`keys: string|` The list of unrec
ZodIssueCode.invalid_union	`unionErrors:` The errors throv element of the u

code	additiona
ZodIssueCode.invalid_enum_value	`options: stri` The set of acce values for this e
ZodIssueCode.invalid_arguments	`argumentsErro ZodError` This is a specia only thrown by a function returne `ZodFunction.i` The `arguments` property is anot containing the v error details.
ZodIssueCode.invalid_return_type	`returnTypeErr ZodError` This is a specia only thrown by a function returne `ZodFunction.i` The `returnTyp` property is anot containing the v error details.
ZodIssueCode.invalid_date	*no additional pr*
ZodIssueCode.invalid_string	`validation: "` `"email" | "uu` Which built-in s failed
ZodIssueCode.too_small	`type: "string` `"number" | "ar` `"set" | "date"` The type of the validation `minimum: numb` The expected le `inclusive: bo` Whether the mi included in the r acceptable valu `exact: boolea` Whether the siz constrained to b value (used to p readable error r
ZodIssueCode.too_big	`type: "string` `"number" | "ar` `"set" | "date"` The type of the validation `maximum: numb`

code	additiona

	The expected le
	
	`inclusive: bo`
	Whether the ma
	included in the r
	acceptable valu
	
	`exact: boolea`
	Whether the siz
	constrained to b
	value (used to p
	readable error r

ZodIssueCode.not_multiple_of	`multipleOf: r` The value the n be a multiple of

ZodIssueCode.custom	`params: { [k`
	`any }`
	This is the error
	by refinements
	are using `supe`
	which case it's p
	throw issues of
	`ZodIssueCode`
	able to pass in a
	object here that
	in your custom e
	(see **ZodErrorM**

ZodParsedType

This is an enum used by Zod internally to represent the type of a parsed value. The possible values are:

- `string`
- `nan`
- `number`
- `integer`
- `float`
- `boolean`
- `date`
- `bigint`
- `symbol`
- `function`
- `undefined`

- `null`
- `array`
- `object`
- `unknown`
- `promise`
- `void`
- `never`
- `map`
- `set`

A demonstrative example

Here's a sample Person schema.

```ts
const person = z.object({
  names: z.array(z.string()).nonempty(), /
  address: z.object({
    line1: z.string(),
    zipCode: z.number().min(10000), // Ame
  }).strict(); // do not allow unrecognize
});
```

Let's pass in some improperly formatted data.

```ts
try {
  person.parse({
    names: ["Dave", 12], // 12 is not a st
    address: {
      line1: "123 Maple Ave",
      zipCode: 123, // zip code isn't 5 di
      extra: "other stuff", // unrecognize
    },
  });
} catch (err) {
  if (err instanceof z.ZodError) {
    console.log(err.issues);
  }
}
```

Here are the errors that will be printed:

```ts
[
  {
    code: "invalid_type",
    expected: "string",
    received: "number",
    path: ["names", 1],
    message: "Invalid input: expected stri
  },
  {
    code: "unrecognized_keys",
    keys: ["extra"],
    path: ["address"],
    message: "Unrecognized key(s) in objec
  },
  {
    code: "too_small",
    minimum: 10000,
    type: "number",
    inclusive: true,
    path: ["address", "zipCode"],
    message: "Value should be greater than
  },
];
```

As you can see three different issues were identified. Every ZodIssue has a `code` property and additional metadata about the validation failure. For instance the `unrecognized_keys` error provides a list of the unrecognized keys detected in the input.

Customizing errors with ZodErrorMap

You can customize **all** error messages produced by Zod by providing a custom "error map" to Zod, like so:

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

const customErrorMap: z.ZodErrorMap = (issu
  if (issue.code === z.ZodIssueCode.invali
    if (issue.expected === "string") {
      return { message: "bad type!" };
    }
  }
  if (issue.code === z.ZodIssueCode.custom
    return { message: `less-than-${(issue.
  }
  return { message: ctx.defaultError };
};

z.setErrorMap(customErrorMap);
```

`ZodErrorMap` is a special function. It accepts two arguments: `issue` and `ctx`. The return type is `{ message: string }`. Essentially the error map accepts some information about the validation that is failing and returns an appropriate error message.

- `issue: Omit<ZodIssue, "message">`

  As mentioned above, ZodIssue is a discriminated union.

- `ctx: { defaultError: string; data: any }`
  - `ctx.defaultError` is the error message generated by the default error map. If you only want to override the message for a single type

of error, you can do that. Just return `{ message: ctx.defaultError }` for everything else.

- `ctx.data` contains the data that was passed into `.parse`. You can use this to customize the error message.

As in the example, you can modify certain error messages and simply fall back to `ctx.defaultError` otherwise.

Error map priority

A custom error maps doesn't need to produce an error message for every kind of issue in Zod. Instead, your error map can override certain errors and return `ctx.defaultError` for everything else.

But how is the value of `ctx.defaultError` determined?

Error messages in Zod are generated by passing metadata about a validation issue through a chain of error maps. Error maps with higher priority override messages generated by maps with lower priority.

The lowest priority map is the `defaultErrorMap`, which defined in `src/errors.ts`. This produces the default error message for all issues in Zod.

Global error map

This message is then passed as `ctx.defaultError` into `overrideErrorMap`. This is a global error map you can set with `z.setErrorMap`:

ts

```
const myErrorMap: z.ZodErrorMap = /* ... *
z.setErrorMap(myErrorMap);
```

Schema-bound error map

The `overrideErrorMap` message is then passed as `ctx.defaultError` into any schema-bound error maps. Every schema can be associated with an error map.

ts

```
z.string({ errorMap: myErrorMap });

// this creates an error map under the hoo
z.string({
  invalid_type_error: "Invalid name",
  required_error: "Name is required",
});
```

Contextual error map

Finally, you can pass an error map as a parameter to any `parse` method. This error map, if provided, has highest priority.

ts

```
z.string().parse("adsf", { errorMap: myErr
```

A working example

Let's look at a practical example of of customized error map:

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

const customErrorMap: z.ZodErrorMap = (err
  /*
  This is where you override the various e
  */
  switch (error.code) {
    case z.ZodIssueCode.invalid_type:
      if (error.expected === "string") {
        return { message: `This ain't a st
      }
      break;
    case z.ZodIssueCode.custom:
      // produce a custom message using er
      // error.params won't be set unless y
      // a `params` arguments into a custor
      const params = error.params || {};
      if (params.myField) {
        return { message: `Bad input: ${pa
      }
      break;
  }

  // fall back to default message!
  return { message: ctx.defaultError };
};

z.string().parse(12, { errorMap: customErr

/* throws:
  ZodError {
    errors: [{
      code: "invalid_type",
      path: [],
      message: "This ain't a string!",
      expected: "string",
      received: "number",
    }]
  }
*/
```

Error handling for forms

If you're using Zod to validate the inputs from a web form, there is a convenient way to "flatten" a ZodError to a rich, structured format that can be easily rendered in your interface.

Consider this example of a simple signup form:

```ts
const FormData = z.object({
  name: z.string(),
  contactInfo: z.object({
    email: z.string().email(),
    phone: z.string().optional(),
  }),
});
```

Now lets pass in some invalid data:

```ts
const result = FormData.safeParse({
  name: null,
  contactInfo: {
    email: "not an email",
    phone: "867-5309",
  },
});
```

This will throw a ZodError with two issues:

```ts
if (!result.success) {
  console.log(result.error.issues);
}
/*
  [
    {
      "code": "invalid_type",
      "expected": "string",
      "received": "null",
      "path": ["name"],
      "message": "Expected string, receive
    },
    {
      "validation": "email",
      "code": "invalid_string",
      "message": "Invalid email",
      "path": ["contactInfo","email"]
    }
  ]
*/
```

Formatting errors

Using the `.format()` method on `ZodError`, we can make this error easier to work with.

```ts
if (!result.success) {
  console.log(result.error.format());
  /*
    {
      name: {
        _errors: ['Expected string, receiv
      },
      contactInfo: {
        email: {
          _errors: ['Invalid email']
        }
      }
    }
  */
}
```

As you can see, the result is an object that denormalizes the issues array into a nested object. This makes it easier to display error messages in your form interface.

```tsx
const FormData = z.object({ ... });

function Errors(props: {errors?: string[]}
  if(!props.errors?.length) return null;
  return <div>{props.errors.map(err => <p>
}

function MyForm(){
  const {register, data} = useForm({ ... }

  const result = FormData.safeParse(data);
  const errors = result.success ? {} : resı

  return <div>
    <label>Name<label>
    <input {...register('name')}>
    <Errors errors={errors?.name?._errors}
  </div>
}
```

Flattening errors

Because `.format` returns a deeply nested object, the errors are contained within the `_errors` property to avoid key collisions. However this isn't necessary if your object schema is only one level deep.

In this scenario, `.flatten()` may be more convenient.

ts

```
if (!result.success) {
  console.log(result.error.flatten());
}
/*
  {
    formErrors: [],
    fieldErrors: {
      name: ['Expected string, received nu
      contactInfo: ['Invalid email']
    },
  }
*/
```

The `fieldErrors` key points to an object that groups all issues by key.

The `formErrors` element is a list of issues that occurred on the "root" of the object schema. For instance: if you called `FormData.parse(null)` , `flatten()` would return:

ts

```
const result = FormData.safeParse(null);
if (!result.success) {
  result.error.flatten();
  /*
    {
      formErrors: ["Invalid input: expecte
      fieldErrors: {}
    }
  */
}
```

Post-processing issues

Both `.flatten()` and `.format()` accept an optional mapping function of `(issue: ZodIssue) => U` to `flatten()` , which can customize how each `ZodIssue` is transformed in the final output.

This can be particularly useful when integrating Zod with form validation, as it allows you to pass back whatever `ZodIssue` specific context you might need.

```ts
result.error.flatten((issue: ZodIssue) =>
  message: issue.message,
  errorCode: issue.code,
}));
/*
  {
    formErrors: [],
    fieldErrors: {
      name: [
        {message: "Expected string, receiv
      ]
      contactInfo: [
        {message: "Invalid email", errorCo
      ]
    },
  }
*/
```

Extract type signature

You can infer the return type signature of `.format()` and `.flatten()` with the following utilities:

```ts
type FormattedErrors = z.inferFormattedErr
/*
  {
    name?: {_errors?: string[]},
    contactInfo?: {
      _errors?: string[],
      email?: {
        _errors?: string[],
      },
      phone?: {
        _errors?: string[],
      },
    },
  }
*/

type FlattenedErrors = z.inferFlattenedErr
/*
  {
    formErrors: string[],
    fieldErrors: {
      email?: string[],
      password?: string[],
      confirm?: string[]
    }
  }
*/
```

These utilities also accept a second generic argument that corresponds to the result of any `ZodIssue` mapper function.

```ts
type FormDataErrors = z.inferFlattenedErro
  typeof FormData,
  { message: string; errorCode: string }
>;

/*
  {
    formErrors: { message: string, errorCo
    fieldErrors: {
      email?: { message: string, errorCode
      password?: { message: string, errorC
      confirm?: { message: string, errorCo
    }
  }
*/
```