

# KneeL BEFORE Zod

Unlocking the secrets to an error-free TypeScript experience

# \$whoami

Senior Software **engineer**  
**Klarna.**



<https://{{twitter,instagram/github,youtube,linkedin}}.lsantos.dev>



# What is Zod?

- Zod is a TypeScript-first schema declaration and validation library
- Allows you to define a schema and validate data against it



# How it looks like

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

const schema = z.object({
  id: z.string().uuid().brand("ID"),
  name: z.string(),
  email: z.string().email(),
});

const data = {
  id: "123e4567-e89b-12d3-a456-426614174000",
  name: "Lucas",
  age: 28,
}

const result = schema.parse(data);
```

```
const result: {
  id: string & z.BRAND<"ID">;
  name: string;
  email: string;
}
```



# ZOD IS NOT THE FIRST

There are others that do the same, like:

1. AJV: But then, where are the types?

## Ajv JSON schema validator

Security and reliability for  
JavaScript applications

# ZOD IS NOT THE SECOND

## Example

```
import { Type, type Static } from '@sinclair/typebox'

const T = Type.Object({
  x: Type.Number(), // const T = {
  y: Type.Number(), //   type: 'object',
  z: Type.Number() //   required: ['x', 'y', 'z'],
}) //   properties: {
      //     x: { type: 'number' },
      //     y: { type: 'number' },
      //     z: { type: 'number' }
      //   }
      // }

type T = Static<typeof T> // type T = {
                           //   x: number,
                           //   y: number,
                           //   z: number
                           // }
```



# ZOD IS NOT THE THIRD

```
1 const Joi = require('joi');
2
3 const schema = Joi.object({
4     username: Joi.string()
5         .alphanum()
6         .min(3)
7         .max(30)
8         .required(),
9     password: Joi.string()
10        .pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')),
11     repeat_password: Joi.ref('password'),
12     access_token: [
13         Joi.string(),
14         Joi.number()
15     ],
16     birth_year: Joi.number()
17         .integer()
18         .min(1900)
19     email: Joi.string()
20         .email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] } })
21 })
22
23 schema.validate({ username: 'abc', birth_year: 1994 });
24 // → { value: { username: 'abc', birth_year: 1994 } }
```





# ZOD IS NOT THE FOURTH

There are others that do the same, like:

1. AJV
2. Typebox
3. Joi
4. IO-TS

```
}
```

```
]
```

```
 */
```

```
}
```

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.

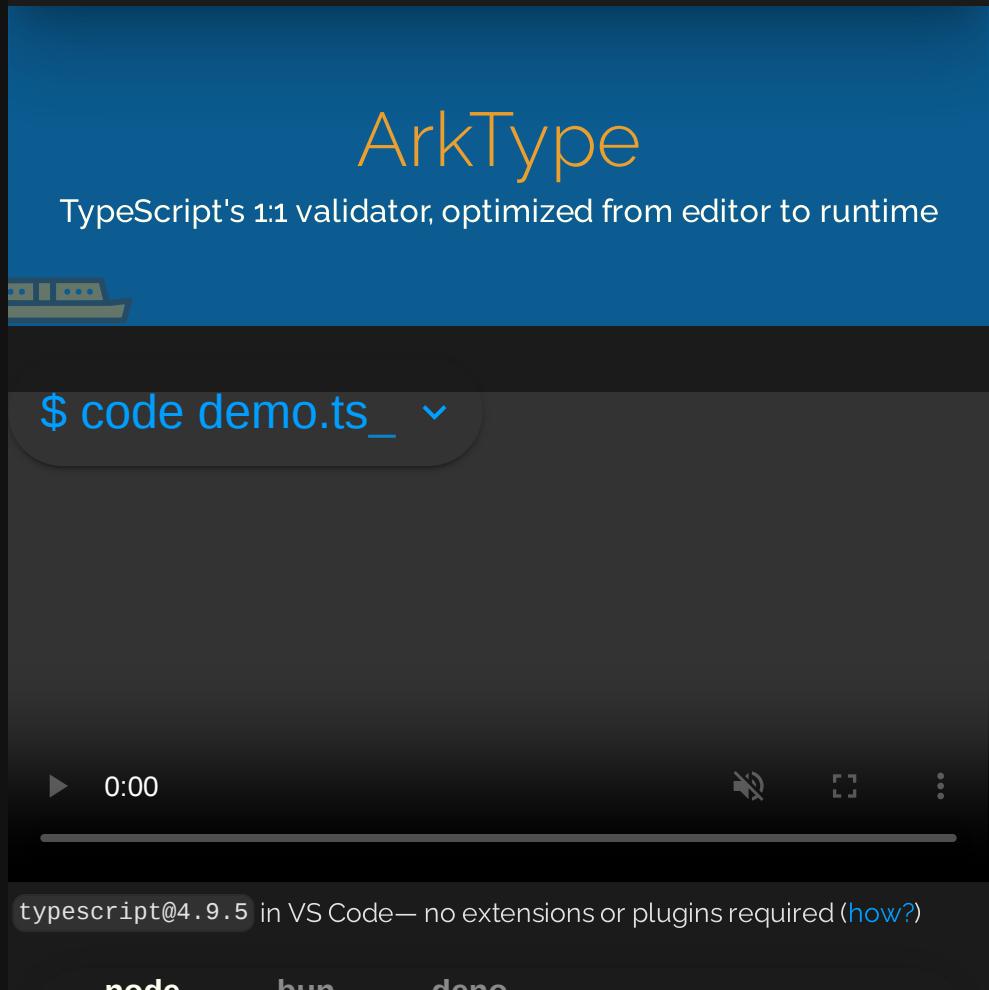


ts

# ZOD IS NOT THE FIFTH

There are others that do the same, like:

1. AJV
2. Typebox
3. Joi
4. IO-TS
5. ArkType



# Then, WHY Zod?

- AJV and TypeBox are not TypeScript-first and they need to be converted to JSON Schema
- Joi is not TypeScript-first, it's old and not very composable (also not super maintained)
- IO-TS is super composable, but also super complex, and it's not very easy to read or maintain
- ArkType is a new one, it's good, but it's not as mature as Zod
  - Plus it's also not as easy to read and maintain



# The Zod way

- Zod is a library that is built on top of the concept of "Zod Types"
- You compose Zod Types (which are standalone) to create a more complex schema
- You can transform, refine, and extend the schema with custom validations
- Has a very good error handling system
- Easy to read and maintain
- Generates types for you
- Has support for advanced TS features, like branded types, union, discriminated unions, enums, and more



# COMPARING ZOD

Let's create a type that represents a user, using Joi:

```
import Joi from 'joi'

const schema = Joi.object({
  username: Joi.string()
    .alphanum()
    .min(3)
    .max(30)
    .required(),
  password: Joi.string().required()
})

const user = schema.validate({ username: 'lsantosdev', password: '123456' })
```



# COMPARING ZOD

Oh no! We got `any`... This means we have to type it ourselves

```
import Joi from 'joi'

const schema = Joi.object({
  const schema: Joi.ObjectSchema<any>

  username: Joi.string()
    .alphanum()
    .min(3)
    .max(30)
    .required(),
  password: Joi.string().required()
})

const user = schema.validate({ username: 'lsantosdev', password: '123456' })
  const user: Joi.ValidationResult<any>
```

And typing it ourselves means that we have two sources of truth that *are not connected*. If the type changes, the schema **won't**... And vice-versa



# COMPARING ZOD

Let's do the same with Zod:

```
import { z } from 'zod'

const schema = z.object({
  username: z.string().min(3).max(30), // zod is required by default
  password: z.string()
})

const user = schema.parse({ username: 'lsantosdev', password: '123456' })
  const user: {
    username: string;
    password: string;
  }

const username = user.us // COMPLETIONS!
  ↩ username
```

Types 🍍 Schema. If the schema changes, the type will also change → *instant feedback*



# Schema-DRIVEN DEVELOPMENT

1. Schema-driven development is when you focus your data on schemas
2. It allows your application to stay consistent
3. Reduces the amount of errors and makes your code more reliable

# The RULES OF SCHEMA-DRIVEN-DEVELOPMENT

1. Schemas are the source of truth
2. Do not Repeat Yourself
3. Always validate, never cast
4. Data flows in one direction: outside → inside

# Schemas as THE SOURCE OF TRUTH

**NEVER** redefine a type that can be inferred from a schema.

If your schema changes, your type will **not** follow ➔ *Bad typing*

Do 

```
const userSchema = z.object({
  user: z.string(),
  pass: z.string()
})

type UserType = z.infer<typeof userSchema>
  type UserType = {
    user: string;
    pass: string;
  }
```

Don't 

```
const userSchema = z.object({
  user: z.string(),
  pass: z.string()
})

// DUPLICATE, not source of truth
interface UserType {
  user: string
  pass: string
}
```

Data IS **not** unique

Extend and reuse, **do not** re-create



# Reduce, Reuse, Recycle

Try to extend and reuse your types as much as possible without re-creating the schema

## Reuse

```
const idSchema = z.string().uuid().brand('ID')
const userSchema = z.object({
  id: idSchema,
  email: z.string().email().brand('EMAIL')
})
type UserType = z.infer<typeof userSchema>
type UserType = {
  id: string & z.BRAND<"ID">;
  email: string & z.BRAND<"EMAIL">;
}
```

## Extend

```
const employeeSchema = userSchema.extend({
  badge: z.number()
})

type EmployeeType = z.infer<typeof employeeSchema>
type EmployeeType = {
  id: string & z.BRAND<"ID">;
  email: string & z.BRAND<"EMAIL">;
  badge: number;
}
```

# **Extensibility\_**

# The extensibility of Zod

- If you need custom validations you can extend zod and refine your data
- You can also transform the data so you receive the final schema already converted

```
const configSchema = z.object({
  PORT: z.string().optional().default("3000"),
  DB_NAME: z.string().optional().default('my-db'),
  DB_HOST: z.string().optional().default('localhost'),
  DB_PORT: z.string().optional().default('27017'),
  DB_USER: z.string().optional(),
  DB_PASS: z.string().optional()
}).transform((val) => {
  const credentials = val.DB_USER && val.DB_PASS ? `${val.DB_USER}:${val.DB_PASS}@` : ''
  return {
    port: Number(val.PORT),
    connStr: `mongodb://${credentials}${val.DB_HOST}:${val.DB_PORT}/${val.DB_NAME}`
  }
})

const config = configSchema.parse({}) // { port: 3000, connStr: 'mongodb://localhost:27017/my-db' }

const config: {
  port: number;
  connStr: string;
}
```



# Refine

`.refine()` will extend your validation to create custom validators. It's a function that returns a boolean and a message

```
const configSchema = z.object({
  PORT: z
    .string()
    .optional()
    .default('3000')
    .refine((v) => Number(v) > 65535, 'Invalid port range')
})
type Config = z.infer<typeof configSchema>
  type Config = {
    PORT: string;
  }

const valid = configSchema.parse({ PORT: 3000 }) // ok
const invalid = configSchema.parse({PORT: 99999999}) // ZodError "Invalid Port Range"
```



# TRANSFORM

`.transform()` will modify the end result of the parsing. It can be applied at an object level

```
const configSchema = z.object({
  PORT: z
    .string()
    .optional()
    .default('3000')
    .refine((v) => Number(v) > 65535, 'Invalid port range')
})
  .transform((v) => ({ port: Number(v.PORT) }))
```

```
v: {
  PORT: string;
}
```

```
type Config = z.infer<typeof configSchema>
  type Config = {
    port: number;
  }
```



# TRANSFORM

But can also be applied at a type level:

```
const configSchema = z.object({
  PORT: z
    .string()
    .optional()
    .default('3000')
    .transform((v) => Number(v))
      v: string
    .refine((v) => v > 65535, 'Invalid port range')
  )
    v: number
  .transform(({PORT}) => ({port: PORT})) // lowercasing

type Config = z.infer<typeof configSchema>
  type Config = {
    port: number;
  }
```



# **ERROR HANDLING\_**

# The errors of Zod

- Zod has two methods:
  - `.parse()` will tryparse the schema and, if it fails, will throw a `ZodError`
  - `.safeParse()` will tryparse the schema but will always return with a `success` property that indicates errors
- Both of them have their `async` counterparts (`.parseAsync` and `.safeParseAsync`) for async flows
- `ZodError` is a powerful class that includes the error message, the issues found with the schema and the path for the error

```
1  const stringSchema = z.string()  
2  
3  stringSchema.safeParse(NaN);  
4  // => { success: false; error: ZodError }  
5  
6  stringSchema.safeParse("lsantos.dev");  
7  // => { success: true; data: 'lsantos.dev' }  
8
```



# HANDLING ERRORS IN APIs

```
1 const userSchema = z.object({
2   name: z.string(),
3   pass: z.string()
4 })
5
6 app.post('/users', async (req, res, next) => {
7   const {success, data} = userSchema.safeParse(req.body)
8   if (!success) {
9     return res.status(422).json(err.message)
10  }
11
12  const user = await doSomething(data)
13  return res.json(user)
14 })
15
```



# HANDLING ERRORS IN APIs

Errors can also be handled in "unsafe" mode

```
1  const userSchema = z.object({
2    name: z.string(),
3    pass: z.string()
4  })
5
6  app.post('/users', async (req, res, next) => {
7    try {
8      const parsed = userSchema.parse(req.body)
9      const user = await doSomething(parsed)
10     return res.json(user)
11   } catch (err) {
12     if (err instanceof ZodError) {
13       return res.status(422).json(err.message)
14     }
15     next(err)
16   }
17 })
18 )
```



# Zod errors are complete

A `ZodError` includes the complete error stack so you can send it over to the client:

```
1  [
2    {
3      "code": "invalid_type", // the error code
4      "expected": "string", // the expected type
5      "received": "number", // the received type
6      "path": [], // the path to the error (empty means root)
7      "message": "Expected string, received number" // the error message
8    }
9  ]
10 ]
```

But you can also format it to a more human-readable message using `.format()`:

```
const result = z.object({ name: z.string() }).safeParse({ name: 12 })
if (!result.success) {
  const formatted = result.error.format() // { name: { _errors: [ 'Expected string, received number' ] } }
  formatted.name?._errors // => ["Expected string, received number"]
  _errors: string[] | undefined
}
```



# ZOD IN REAL LIFE\_

# The REALITY OF ZOD

- Zod really shines when you have to create complex schemas
- This is a real example for a system that receives a JSON string that's another object which should be validated

Can you spot the problem?

```
const schema = z.object({
  maskedPan: z.string(),
  orderId: z.string().uuid(),
  clientInformation: z.string() // this is a json string, how to validate it?
})
```

We lose type inference for the `clientInformation` field. It's just a string, but we need it to be an object, a **validated object**



# Complex schemas

We can define a schema for our JSON:

```
const clientInformation = z.object({
  authenticationToken: z.string(),
  items: z
    .array(
      z.object({
        id: z.string().uuid(),
        quantity: z.number().int().positive()
      })
    )
    .nonempty(),
  orderId: z.string().uuid(),
  sourcePSP: z.string().optional(),
  sourcePSPTransactionId: z.string().optional()
})
```



# COMPLEX SCHEMAS

And then we can use it in our main schema:

```
const schema = z.object({
  maskedPan: z.string(),
  orderId: clientInformation.shape.orderId,
  clientInformation: z.string().transform((json, ctx) => {
    try {
      const obj = JSON.parse(json) // throws if not a valid JSON
      return clientInformation.parse(obj) // throws if not a valid clientInformation object
    } catch (error) {
      if (error instanceof z.ZodError) {
        error.issues.forEach(ctx.addIssue)
      } else {
        ctx.addIssue({
          code: z.ZodIssueCode.custom,
          message: 'Invalid clientInformation JSON string',
          path: [],
          fatal: true
        })
      }
    }
    return z.NEVER
  })
})
```



# COMPLEX SCHEMAS

This will give us a final schema that's fully typed and validated.

And, if the schema is not valid, either because of invalid JSON or invalid clientInformation, it will throw a `ZodError` with the issues

```
type SchemaType = z.infer<typeof schema>

type SchemaType = {
    maskedPan: string;
    orderId: string;
    clientInformation: {
        orderId: string;
        authenticationToken: string;
        items: [
            {
                id: string;
                quantity: number;
            }, ... {
                id: string;
                quantity: number;
            }[]
        ];
        sourcePSP?: string | undefined;
        sourcePSPTransactionId?: string | undefined;
    };
}
```



# Summary

1. Zod provides type-safe validations for objects or primitives
2. Zod allows you to extract types from schemas, do not re-create
3. Define your schemas in a single place, reuse from there
4. Types comes from schemas alone
5. Extend zod to create custom validators with `refine` and `transform`
6. Use `ZodError` to type your APIs and parse messages automatically
7. Zod shines when you have complex schemas



# THANK YOU!

<https://{{twitter,instagram/github,youtube,linkedin}}.lsantos.dev>



# See THIS TALK ON MY WEBSITE

<https://lsantos.dev/talks/kneel-before-zod>

