Kneel Before Zod

Unlocking the secrets to an error-free TypeScript experience

$whoami

Senior Software engineer_

Klarna.

Google Developers Experts 2019
Web Technologies GDE

MVP Microsoft® Most Valuable Professional

docker CAPTAINS

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

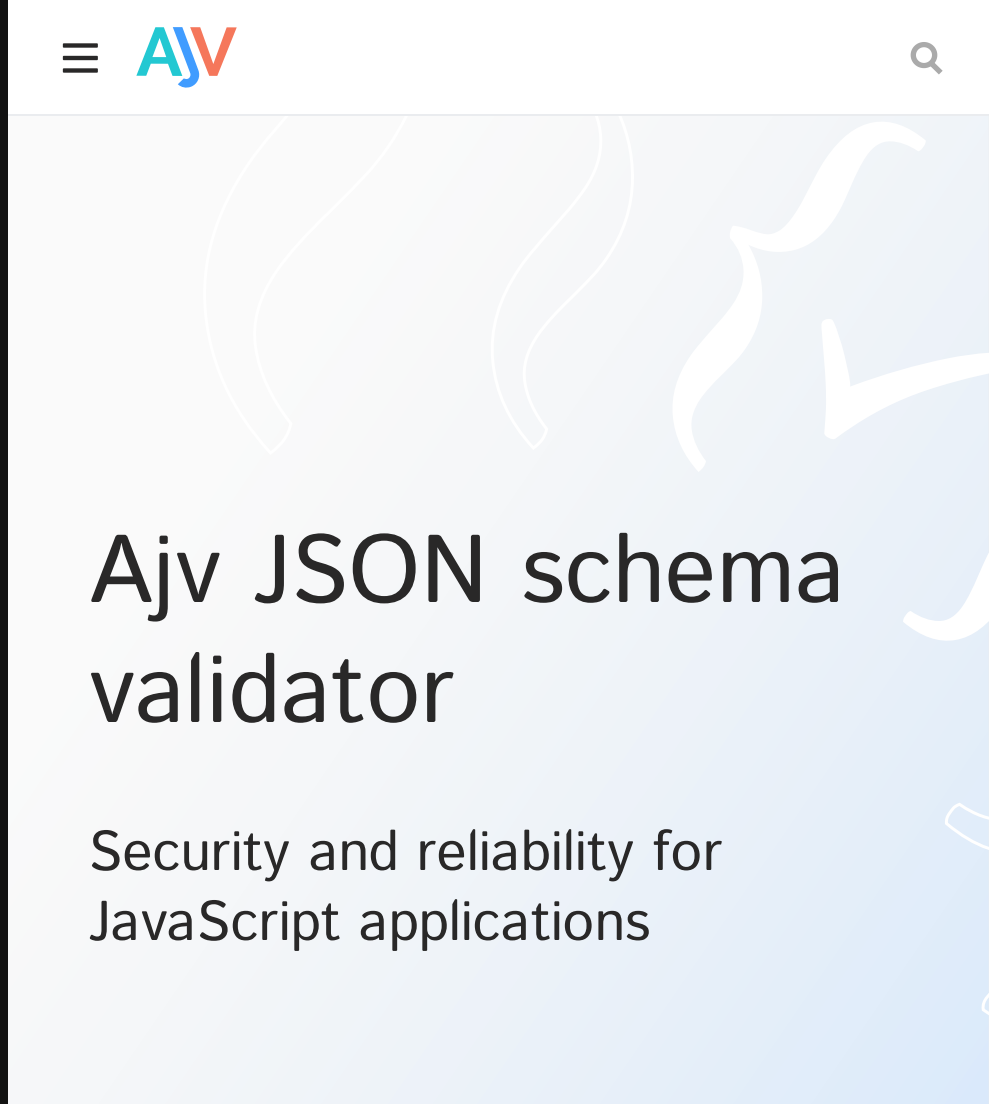
```ts
import { z } from "zod";

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);
```

```ts
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?



Zod is not the second

Example

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

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

```
const Joi = require('joi');

const schema = Joi.object({
    username: Joi.string()
        .alphanum()
        .min(3)
        .max(30)
        .required(),
    password: Joi.string()
        .pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')),
    repeat_password: Joi.ref('password'),
    access_token: [
        Joi.string(),
        Joi.number()
    ],
    birth_year: Joi.number()
        .integer()
        .min(1900)
    email: Joi.string()
        .email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] } })
})

schema.validate({ username: 'abc', birth_year: 1994 });
// → { 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

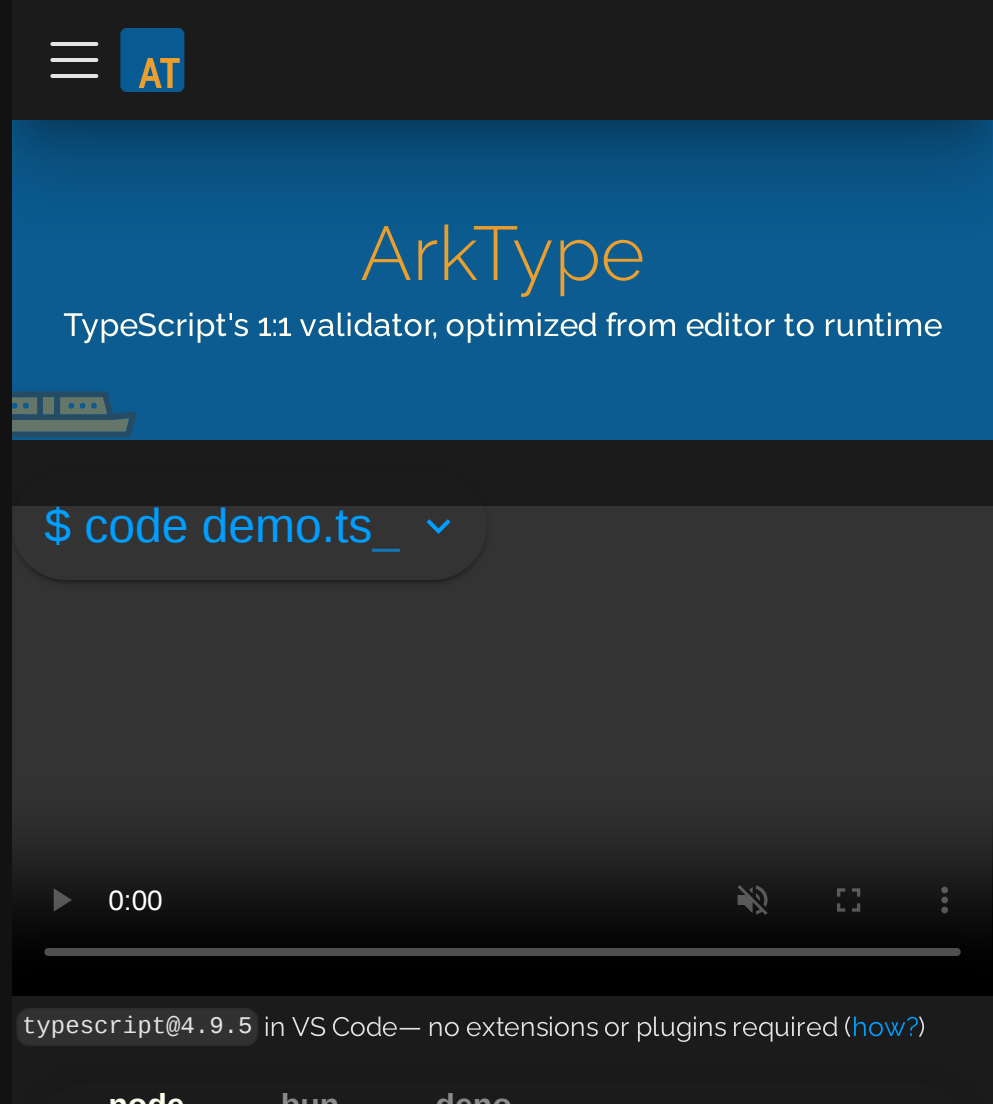
You can use the `.format()` method to convert this error into a nested object.

```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

```ts
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:

```ts
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 ✅

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

type UserType = z.infer<typeof userSchema>
```

```ts
type UserType = {
    user: string;
    pass: string;
}
```

Don't 🚫

```ts
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 ♻️

```ts
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>
```

```ts
type UserType = {
    id: string & z.BRAND<"ID">;
    email: string & z.BRAND<"EMAIL">;
}
```

Extend 🔎

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

type EmployeeType = z.infer<typeof employeeSchema>
```

```ts
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

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

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

```ts
type Config = z.infer<typeof configSchema>
```

```ts
type Config = {
    port: number;
}
```

Transform

But can also be applied at a type level:

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

.transform(({PORT}) => ({port: PORT})) // lowercasing

type Config = z.infer<typeof configSchema>
```

```ts
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

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

app.post('/users', async (req, res, next) => {
    const {success, data} = userSchema.safeParse(req.body)
    if (!success) {
      return res.status(422).json(err.message)
    }

    const user = await doSomething(data)
    return res.json(user)
})

```

Handling errors in APIs

Errors can also be handled in "unsafe" mode

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

app.post('/users', async (req, res, next) => {
  try {
    const parsed = userSchema.parse(req.body)
    const user = await doSomething(parsed)
    return res.json(user)
  } catch (err) {
    if (err instanceof ZodError) {
      return res.status(422).json(err.message)
    }
    next(err)
  }
})

```

Zod errors are complete

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

```
[
  {
    "code": "invalid_type", // the error code
    "expected": "string", // the expected type
    "received": "number", // the received type
    "path": [], // the path to the error (empty means root)
    "message": "Expected string, received number" // the error message
  }
]

```

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

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

```ts
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