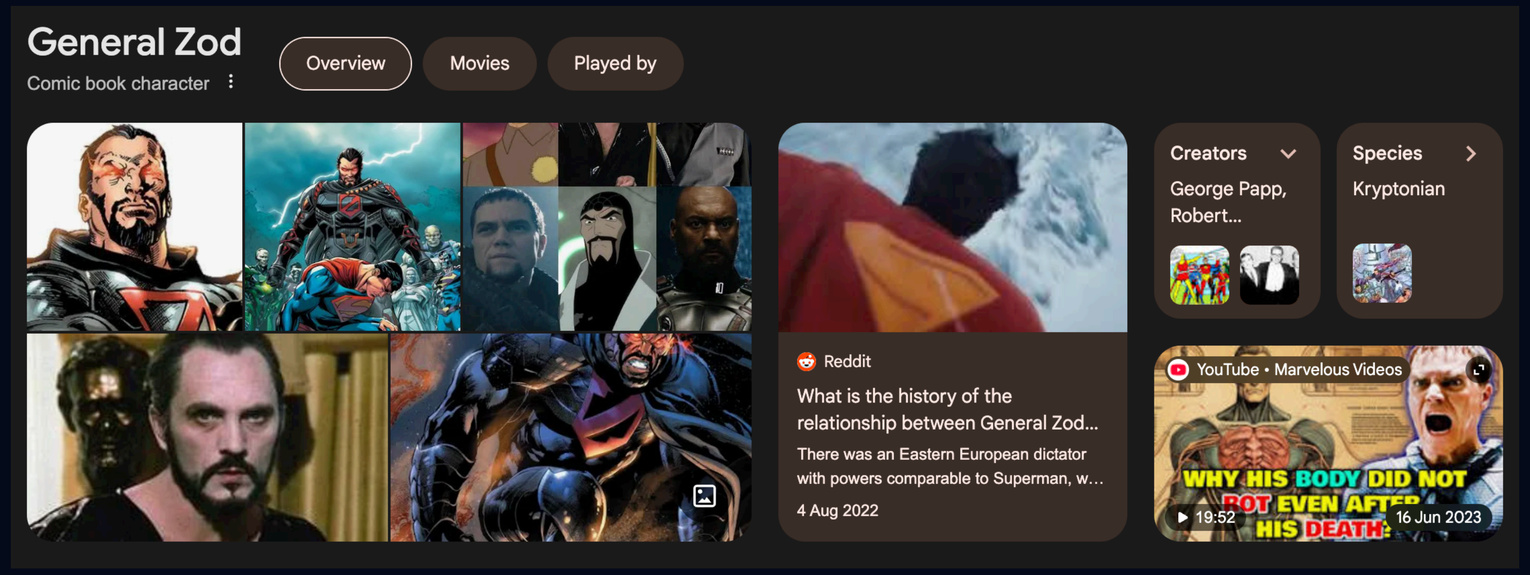

ZOD RETURNS
YOUR DATA WILL BE VALIDATED!
Required!
z.object({
name:z.string(),
email.z.string(.email)
);
}
}
Must be string!
Email format only!
SCHEMA VALIDATOR

SO WHAT IS ZOD?

General Zod is a supervillain appearing in American comic books published by DC Comics, commonly as an adversary of the superhero Superman. The character, who first appeared in Adventure Comics #283, was created by Robert Bernstein and initially designed by George Papp.



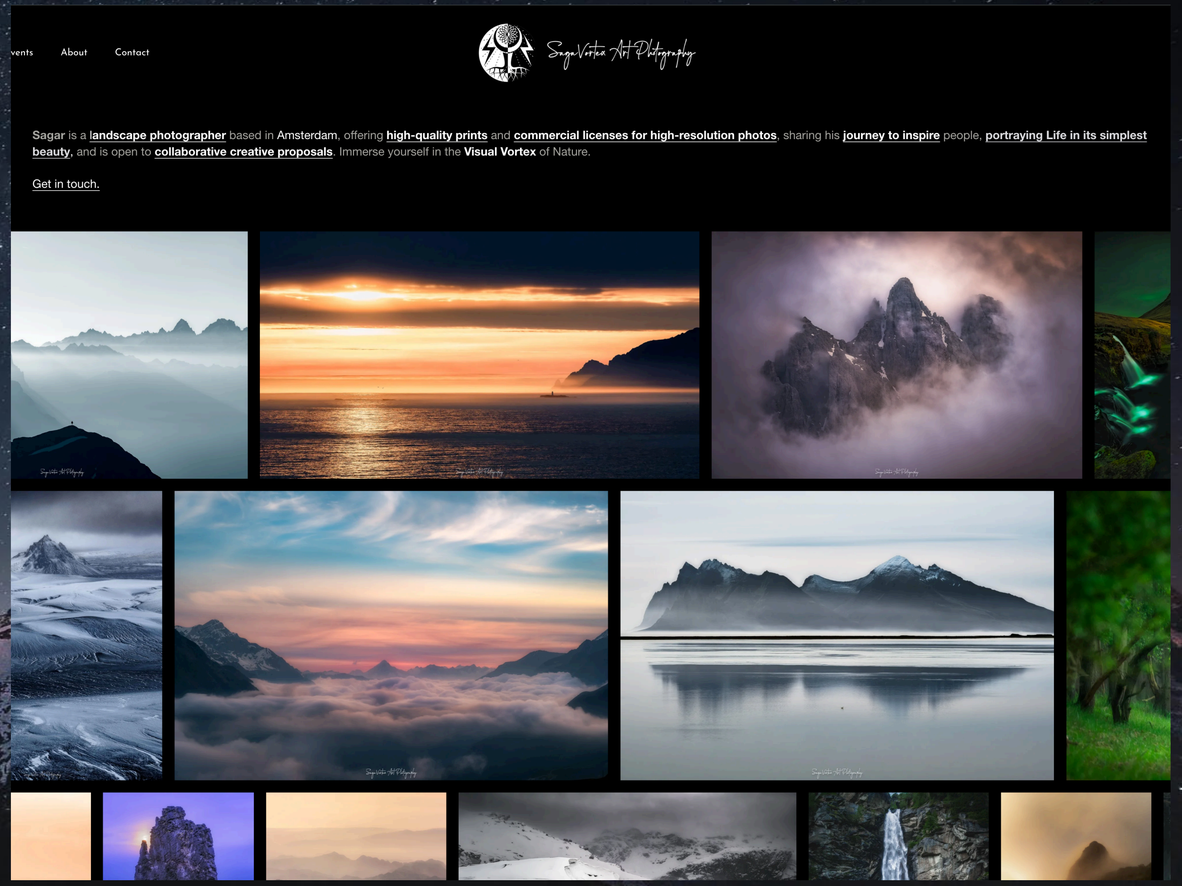
The context doesn’t sound quite right, eh? 🤔

SAGAR SAWUCK

SENIOR FRONTEND CONSULTANT AT BOL.COM

github.com/sagarsys
linkedin.com/in/sagarsys

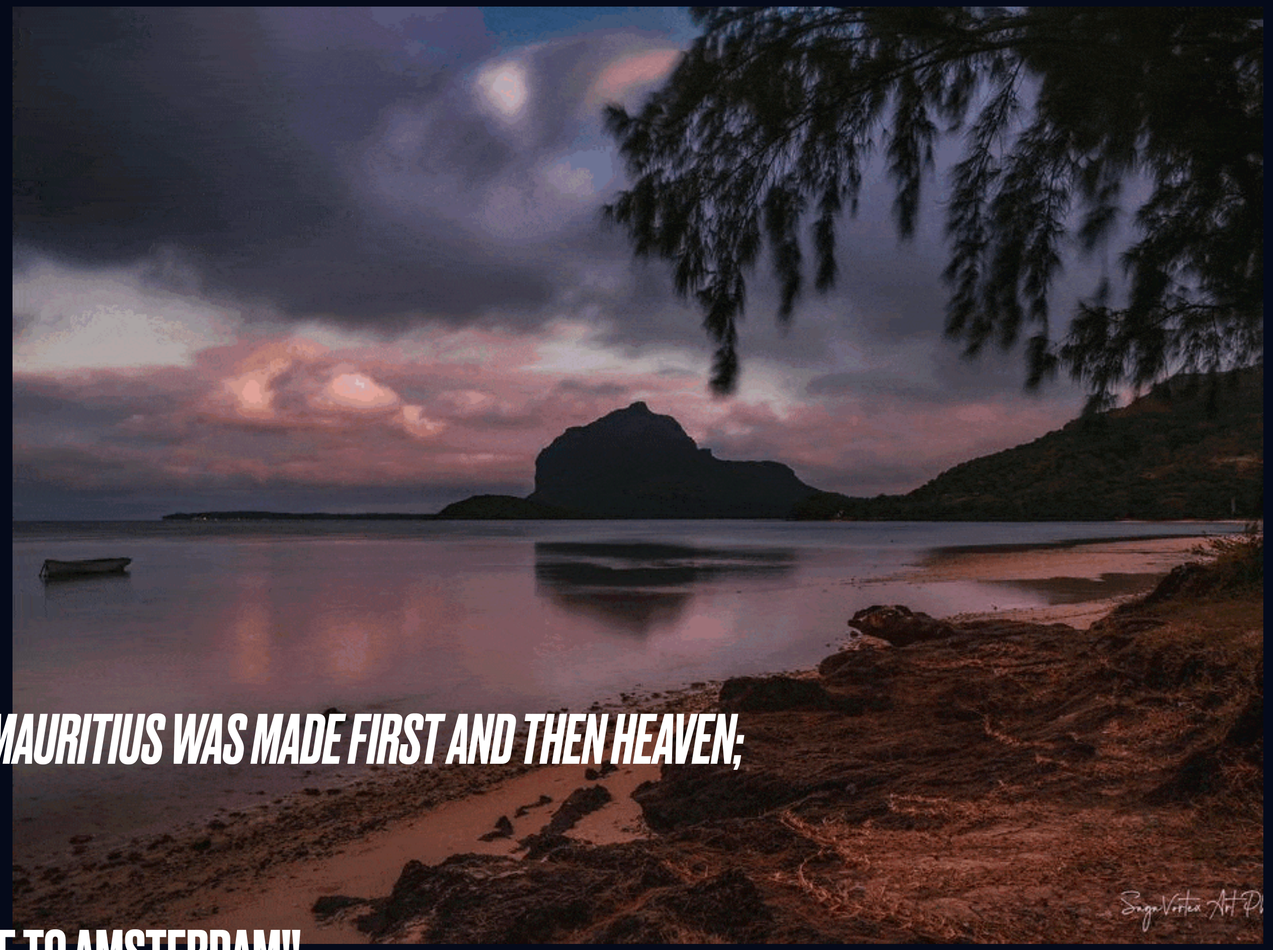
sagavortex.art



SagaVortex Art Photography

bio.site/sagavortex.art

ELEPHANT IN THE ROOM: MAURITIUS

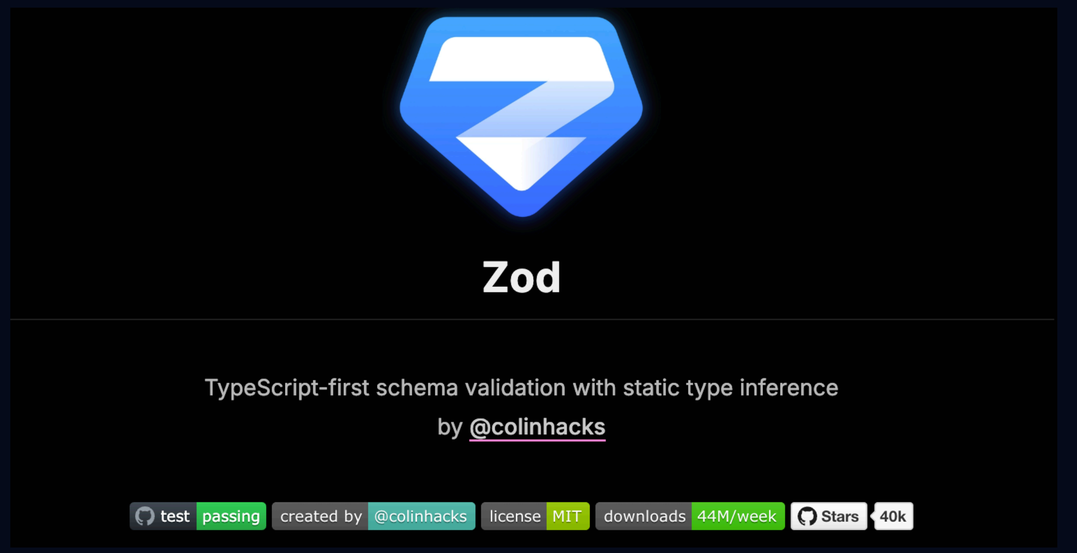


FAMOUS QUOTES ABOUT MAURITIUS INCLUDE MARK TWAIN'S ICONIC *"MAURITIUS WAS MADE FIRST AND THEN HEAVEN; AND HEAVEN WAS COPIED AFTER MAURITIUS"*

BUT EVEN HEAVEN GETS BORING AFTER A WHILE 🤷 WHICH BRINGS ME TO AMSTERDAM!!

ZOD IN JAVASCRIPT

- **Schema-first validation** - Define once, use everywhere
- **TypeScript integration** - Automatic type inference
- **Runtime safety** - Catch errors before they crash your app
- **Zero external dependencies**
- **Works in Node.js and all modern browsers**
- **Tiny: 2kb core bundle (gzipped)**
- **Immutable API: methods return a new instance**
- **Concise interface**
- **Works with TypeScript and plain JS**
- **Built-in JSON Schema conversion**
- **Extensive ecosystem**



WHAT PROBLEM DOES ZOD SOLVE?

- **The Problem**: Runtime errors from invalid data
  - Form submission crashes app
  - API returns unexpected structure
  - Environment variables missing
- **The Hero**: Zod - A new hope for type safety!
- **What you'll learn**: Concrete examples of Zod solving real problems

WHY USE ZOD IN TYPESCRIPT?

Zod bridges the gap between TypeScript's compile-time type checking and runtime validation. It ensures data matches expected types both during development and at runtime, catching errors that TypeScript alone can't prevent.

Key Benefits

- **Schema as Single Source of Truth**: Define your data structure once, get both TypeScript types and runtime validation. No duplication or sync issues.
- **Excellent Developer Experience**: Descriptive error messages, autocomplete, and seamless TypeScript integration make it pleasant to work with.
- **API Boundary Safety**: Perfect for validating external data (API requests, form submissions, environment variables) where you can't trust the input format.
- **Composable & Flexible**: Schemas can be combined, extended, and transformed easily. Built-in methods for common patterns like optional fields, defaults, and transformations.
- **Framework Integration**: Works excellently with React Hook Form, Next.js API routes, tRPC, and other popular tools in the TypeScript ecosystem.

SCHEMA DEFINITION

```
const LoginSchema = z.object({
  email: z.string().email('Invalid email'),
  password: z.string().min(6, 'Password must be at least 6 characters')
});

const OrderSchema = z.object({
  id: z.string().uuid(),
  customer: z.object({
    name: z.string(),
    address: z.object({
      street: z.string().optional(),
      city: z.string(),
      country: z.enum(['US', 'CA', 'UK'])
    })
  }),
  items: z.array(z.object({
    product: z.string(),
    quantity: z.number().positive(),
    price: z.number()
  })),
  metadata: z.record(z.unknown()).optional()
});
```

COMPOSING SCHEMAS

```
// schemas/common.ts
export const TimestampSchema = z.object({
  createdAt: z.date(),
  updatedAt: z.date()
})

export const IdSchema = z.object({
  id: z.string().uuid()
})

// schemas/user.ts
import { TimestampSchema, IdSchema } from './common'

export const UserSchema = IdSchema.merge(TimestampSchema).extend({
  name: z.string(),
  email: z.string().email()
})

// schemas/product.ts
export const ProductSchema = IdSchema.merge(TimestampSchema).extend({
  name: z.string(),
  price: z.number().positive(),
  owner: UserSchema.pick({ id: true, name: true })
})
```

POLYMORPHIC DATA SCHEMAS WITH DISCRIMINATED UNION

A discriminated union is a pattern where you have multiple object types that share a common "discriminator" field (usually called a "tag" or "type" field). This field acts as a "switch" that tells you exactly which variant of the union you're dealing with.

```
const ApiResponseSchema = z.discriminatedUnion('success', [
  z.object({
    success: z.literal(true),
    data: z.any(),
    message: z.string().optional()
  }),
  z.object({
    success: z.literal(false),
    error: z.string(),
    code: z.number()
  })
])
```

```
const NotificationBase = z.object({
  id: z.string(),
  timestamp: z.date(),
  read: z.boolean()
})

const EmailNotification = NotificationBase.extend({
  type: z.literal('email'),
  subject: z.string(),
  body: z.string()
})

const PushNotification = NotificationBase.extend({
  type: z.literal('push'),
  title: z.string(),
  message: z.string()
})

const SMSNotification = NotificationBase.extend({
  type: z.literal('sms'),
  phoneNumber: z.string(),
  text: z.string()
})

// Union with discriminator
const NotificationSchema = z.discriminatedUnion('type', [
  EmailNotification,
  PushNotification,
  SMSNotification
])
```

CONDITIONAL SCHEMA COMPOSITION

1. USING `.AND()` WITH DISCRIMINATED UNIONS

```
// Base schema with common fields
const BaseUserSchema = z.object({
  name: z.string(),
  email: z.string().email(),
  role: z.enum(['admin', 'user', 'moderator'])
})

// Conditional fields based on role
const UserSchema = BaseUserSchema.and(
  z.discriminatedUnion('role', [
    z.object({
      role: z.literal('admin'),
      permissions: z.array(z.string()),
      canDeleteUsers: z.boolean()
    }),
    z.object({
      role: z.literal('moderator'),
      moderatedForums: z.array(z.string())
    }),
    z.object({
      role: z.literal('user'),
      subscriptionLevel: z.enum(['free', 'premium'])
    })
  ])
)
```

2. DYNAMIC SCHEMA FACTORIES

```
function createConfigSchema(environment: 'development' | 'production') {
  const baseSchema = z.object({
    apiUrl: z.string().url(),
    database: z.object({
      host: z.string(),
      port: z.number()
    })
  })

  if (environment === 'development') {
    return baseSchema.extend({
      debugMode: z.boolean().default(true),
      mockData: z.boolean().optional()
    })
  }

  return baseSchema.extend({
    ssl: z.boolean().default(true),
    monitoring: z.object({
      endpoint: z.string().url(),
      apiKey: z.string()
    })
  })
}

const devConfigSchema = createConfigSchema('development')
const prodConfigSchema = createConfigSchema('production')
```

BASIC USAGE EXAMPLE

```
const UserSchema = z.object({
  name: z.string(),
  email: z.string().email(),
  age: z.number().min(18)
})

// extract the inferred type
type User = z.infer<typeof UserSchema>;

// use it in your code
const user: User = { name: "billie", email: "billie@email.com" , age: 30 };

const result = UserSchema.safeParse({ name: 42, age: "100" });
if (!result.success) {
  result.error;   // ZodError instance
} else {
  result.data;
}
```

COMMON USE CASES

- **Form validation in React applications**: validating user input with libraries like React Hook Form
- **API request/response validation**: Validating incoming requests and responses
- **Environment/Config file parsing**: Type-safe environment/config variables
- **Database schema validation**: With ORMs or raw database operations
- **tRPC Integration**: Zod is the default validator for tRPC procedures
- **Data transformation and coercion**: Converting and cleaning data
- **External API Response Validation**: Ensuring third-party APIs return expected data
- ...and much more!

FORM VALIDATION IN REACT APPLICATIONS WITH REACT HOOK FORM

```
import { z } from 'zod'
import { useForm } from 'react-hook-form'
import { zodResolver } from '@hookform/resolvers/zod'

const formSchema = z.object({
  email: z.string().email('Invalid email'),
  password: z.string().min(8, 'Password must be 8+ characters'),
  confirmPassword: z.string()
}).refine(data => data.password === data.confirmPassword, {
  message: "Passwords don't match",
  path: ["confirmPassword"]
})

function SignupForm() {
  const { register, handleSubmit, formState: { errors } } = useForm({
    resolver: zodResolver(formSchema)
  })

  const onSubmit = (data) => {
    // data is automatically typed and validated
  }
}
```

DEMO TIME.

https://github.com/sagarsys/zod-returns

https://zod-returns.vercel.app/

KEY TAKEAWAYS: THE TYPESCRIPT VALIDATION LIBRARY THAT STRIKES BACK!

- **Single source of truth for data shape** - One Schema, Two Powers:
  Define once → get TypeScript types + runtime validation
- **Runtime Safety**: Catch incorrect data before it crashes your app
- **Perfect Form Validation**: Seamless React Hook Form integration
- **Composable & Scalable**: Build complex schemas from simple building blocks

From hoping your data is right → knowing your data is right

THE ZOD HAS SPOKEN! 


QUESTIONS ABOUT YOUR VALIDATION DESTINY?
https://zod.dev/


const schema = z.object({
bugs. z.never(),
dataSafety: z.literal(true),
});
SagaVortex Art Photography
bio.site/sagavortex.art
THE
ZOD SPOKEN
YOUR VALIDATION DESTINY IS FULFILLED