



# Beyond Type Checking

Building **Bulletproof** TypeScript Applications



DevWorld Conference 2025

JOSEPH ANSON

BEYOND TYPE CHECKING: BUILDING BULLETPROOF TYPESCRIPT APPLICATIONS

# Today's Journey



## The Problem

Compile-time type safety limitations



## The Solution

Runtime validation with Zod



## Implementation

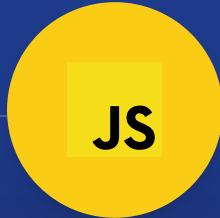
Real-world examples



## Ecosystem

Tools & adoption strategies

# JS Type Safety Journey



JavaScript's  
"Trust Me" Era



TypeScript  
Compile-Time Safety



Runtime  
Safety Gap

# Where Things Go Wrong

1

## The Vanishing Act

TypeScript types disappear at runtime, leaving no safety net for real-world data

2

## The API Illusion

API responses often don't match their types, leading to silent failures

3

## The Casting Trap

Overusing `as` or generics bypasses type checks, creating false confidence

# The Trust Boundary

```
// JavaScript's "Trust Me" Era
const user = {
  id: '123',
  name: 'Alice'
} // Let's hope this is a user
```

## TS Trust Boundary



# Why This Matters

## ⚠ The Cost of Assumptions

Silent Failures, Loud Consequences

Mismatched data leads to crashes, corrupted state, and costly debugging

## ⚠ The Illusion of Safety

Compile-time ≠ Runtime Safety

TypeScript's types vanish at runtime, leaving critical gaps in data validation

## ✓ The Safety Net

Catching Errors Early

Runtime validation acts as a safety net, preventing type-related crashes in production

# The Runtime Validation Gap

```
// Real-world example
interface LoanStatusResponse {
  id: string
  loanStatus: {
    status: 'MANUAL_REVIEW' | 'APPROVED' | 'REJECTED'
    interestRate: number
    total: number
  }
}
```

# Real-World Impact

- \$ **Data-related bugs are costly:** Direct financial impact from production issues
- ⌚ **Significant debugging time:** Hours spent tracking down type-related issues
- 😢 **User experience degradation:** Broken features and unexpected behavior
- 🔒 **Potential security risks:** Vulnerabilities from incorrect data handling

```
interface User {  
    age: number  
    role: 'ADMIN' | 'USER'  
    isActive: boolean  
}  
  
const response = await fetch<User>('/api/user')
```

# From Blind Faith to Bulletproof Validation

Building Trust Through Runtime Validation

# What is Runtime Validation?

1

## Runtime Verification

Verification of data types, structure, and constraints at runtime

2

## External Guards

Guard rails for external data like APIs, DB results, user input

3

## Early Warning

Early detection of mismatches, preventing deeper errors

# What's Out There?

## Runtime Validation Libraries



Zod



Valibot



Arktype



Joi

Yup



INTRODUCING

## Standard Schema

A common interface for TypeScript validation libraries

[Go to repo](#)

Standard Schema is a common interface designed to be implemented by JavaScript and TypeScript schema libraries.

The goal is to make it easier for ecosystem tools to accept user-defined type validators, without needing to write custom logic or adapters for each supported library. And since Standard Schema is a specification, they can do so with no additional runtime dependencies. Integrate once, validate anywhere.

# What is Standard Schema?

## Universal Standard

Unified approach to schema validation across the JavaScript ecosystem

## Collaborative Initiative

Created through collaboration between Zod, Valibot, and ArkType teams

## Ecosystem Integration

Designed for seamless adoption across frameworks and tools, by providing a standard interface for library authors

## Community-Driven

Evolving best practices shaped by real-world implementation

# Schema Validation

## Why Zod?

- ★ **Popular Choice:** Most widely adopted in the TypeScript ecosystem
- ✓ **Type Safety:** Very Simple Type Inference
- </> **Developer Experience:** Intuitive API and excellent documentation
- 👤 **Ecosystem:** Rich set of utilities and community support
- 🔧 **Functionality:** Used for validation, transformations, data generation, and more

# Zod Essentials

All of the zod utilities are available in the `z` object returned by the `zod` library.

```
// zod types
z.string();
z.number();
z.bigint();
z.boolean();
z.date();
z.array(z.string());
z.object({
  name: z.string(),
  age: z.number()
});
z.symbol();
z.undefined();
z.null();
z.any();
z.unknown();
z.never();
```

## Schema Definition

Craft a schema to represent any data structure

```
const userSchema = z.object({
  name: z.string().trim(),
  age: z.number().min(18).max(100),
  email: z.string().email().optional(),
  role: z.enum(['USER', 'ADMIN']),
  address: z.object({
    street: z.string(),
    city: z.string().min(1).max(50),
  })
})
```

## Type Inference

TypeScript infers the type of the validated data

```
type User = z.infer<typeof UserSchema>
/* => {
  name: string,
  age: number,
  email?: string,
  address: { street: string, city: string },
  role: 'USER' | 'ADMIN'
}
```

# Zod Validation

## .parse()

Returns typed data when successful or throws an error when validation fails

```
try {
  const result = userSchema.parse(data) // => Type User
  return result
}
catch (error) {
  // handle error
  if (error instanceof z.ZodError) {
    logger.analyticsError(error)
  }
}
```

## .safeParse()

Returns result pattern, after checking .success access to data is possible, otherwise error is available

```
const result = userSchema.safeParse(data)
// => { success: true, data: User }
// | { success: false, error: ZodError }

if (!result.success) {
  logger.analyticsError(result.error)
  return
  // ✖ result.data does not exist, because it failed
}

return result.data // => Type User
```

# Zod Errors

```
import { z } from 'zod'

const schema = z.object({
  name: z.string(),
  age: z.number()
})

const result = schema.safeParse({ name: 'Joseph', age: '31' })

console.log(result)
```

```
{
  "success": false,
  "error": {
    "issues": [
      {
        "code": "invalid_type",
        "expected": "number",
        "received": "string",
        "path": [
          "age"
        ],
        "message": "Expected number, received string"
      }
    ],
    "name": "ZodError"
  }
}
```

# Zod In Practice

```
type Product = {  
  id: string  
  price: number  
  variants: {  
    size: "S" | "M" | "L"  
  }[]  
}
```

```
// Runtime Validation - No thrown error  
async function fetchProduct(id: string): Promise<Product> {  
}
```

# Schema-Driven Types

1

## Automatic Type Generation

Use `z.infer<typeof Schema>` to automatically generate TypeScript types

2

## Single Source of Truth

Schema changes automatically reflect in TypeScript types, ensuring perfect sync

3

## DRY Principle

Eliminate duplication between runtime validation and type declarations

# Beyond Basics

- Validate Form Data with `Veevalidate`
- Validate API Bodies with `nitro`
- Mock Generation with libraries like `zod-schema-faker`
- Generate Structured Data with `vercel ai`
- Validate LocalStorage Data

```
// Example: Reusable User Schema
export const userSchema = z.object({
  id: z.string().uuid(),
  name: z.string(),
  email: z.string().email()
})

// Example: Form Validation with Veevalidate
<script setup>
import { userSchema } from './'
import { z } from 'zod'
import { toTypedSchema } from '@vee-validate/zod'

const form = useForm({
  validationSchema: toTypedSchema(userSchema)
})

const [name, nameAttrs] = defineField('name');
const [email, emailAttrs] = defineField('email');
</script>

<template>
  <input v-model="name" v-bind="nameAttrs" />
  <div>{{ errors.name }}</div>
  <input v-model="email" v-bind="emailAttrs" />
  <div>{{ errors.email }}</div>
</template>
```

# Tooling Support

## API Frameworks

- ⚡ H3
- ⚡ Nitro
- RPC tRPC
- 🔥 Hono
- RPC oRPC
- ⚡ GQLoom
- express-zod-api
- ⚡ NestJS

## Form Libraries

- ⚛ TanStack Form
- ⚡ Formwerk
- ⚡ Veevalidate
- ⚡ Regle
- ⚡ Superforms
- ⚛ React Hook Form
- ⚡ FormKit

## UI Frameworks

- ⚡ qwik Qwik
- ⚡ Nuxt UI
- ⚡ Mage
- ⚡ Primevue
- ⚡ Shadcn
- ⚡ Shadcn-vue
- ⚡ renoun
- ⚡ Nuxt Content
- ⚡ Astro Content

## HTTP Clients

- ⚡ upfetch
- ⚡ rest-client
- ⚡ better-fetch
- ⚡ make-service
- A X ⚡ O S zod-axios
- ⚡ zodios

## Utilities

- TS T3 Env
- ⚛ cachified
- ⚡ UploadThing
- TS OpenAuth
- ⚡ zod-schema-faker
- ⚡ Prisma zod generator
- ⚡ Drizzle ORM

## Routing

- ⚡ TanStack Router
- TS call-api
- ⚡ Kitbag

# Key Benefits



## Bulletproof Applications

Catch errors at your system's boundaries



## Developer Confidence

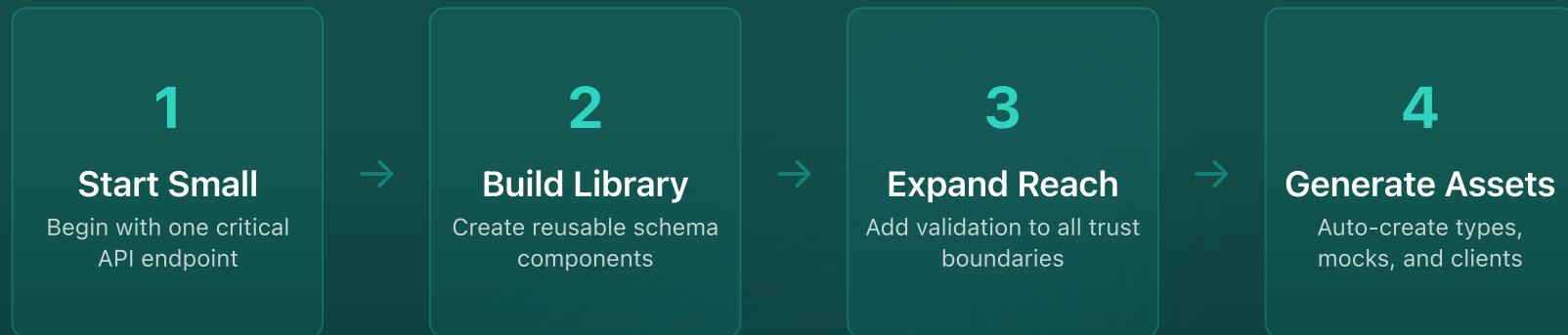
Know exactly when and how your data fails



## Single Source of Truth

Schema-driven types across your entire stack

# 4-Step Adoption Plan



# Key Takeaways

## TypeScript alone isn't enough

Runtime validation is essential

## Validate at trust boundaries

Prevent runtime errors at data entry points

## Schemas for validation

Powerful, flexible with excellent TypeScript integration

## Incremental adoption

Start small, prove value, expand systematically

# Shared Demo 🚀

## Runtime Schema Validation

Scan QR for demo repo



Nuxt



Nitro



Zod



Chrome based Gemini



### API Validation

Request / Response safety with Zod

### Frontend Safety

Form validation with Zod

### Data Generation

Generate Data using Zod and AI

# Thank You!

Let's Build Safer Systems Together

Slides, Demo Application & Resources



[josephanson.com](http://josephanson.com) | [@josephanson](https://twitter.com/josephanson)

