Beyond Type Checking

Building Bulletproof TypeScript Applications

DevWorld Conference 2025

JOSEPH **ANSON**

Today's Journey

The Problem
Compile-time type safety

The Solution
Runtime validation

Implementation
Real-world examples

Live Demo
Runtime Validation in Action

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

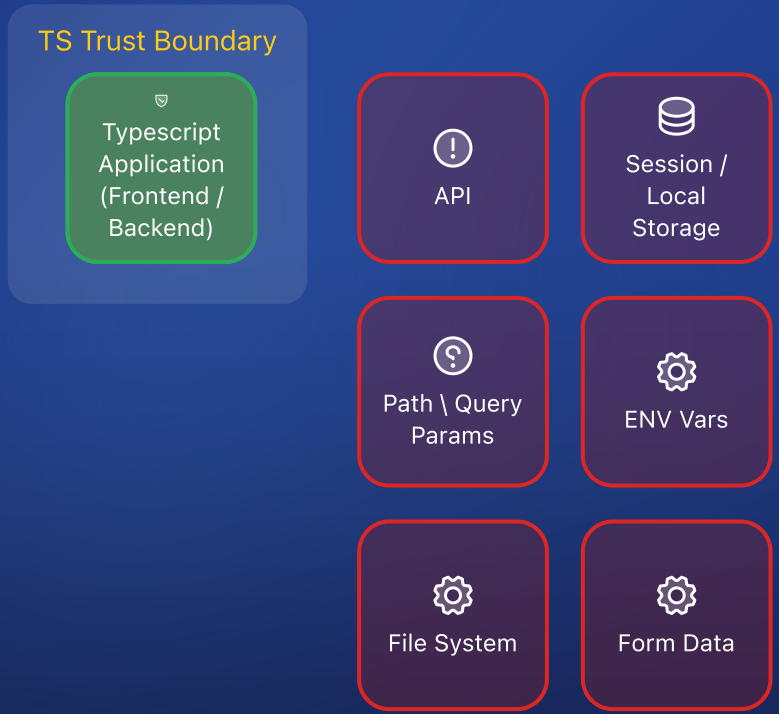
3

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
```



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

```typescript
// 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

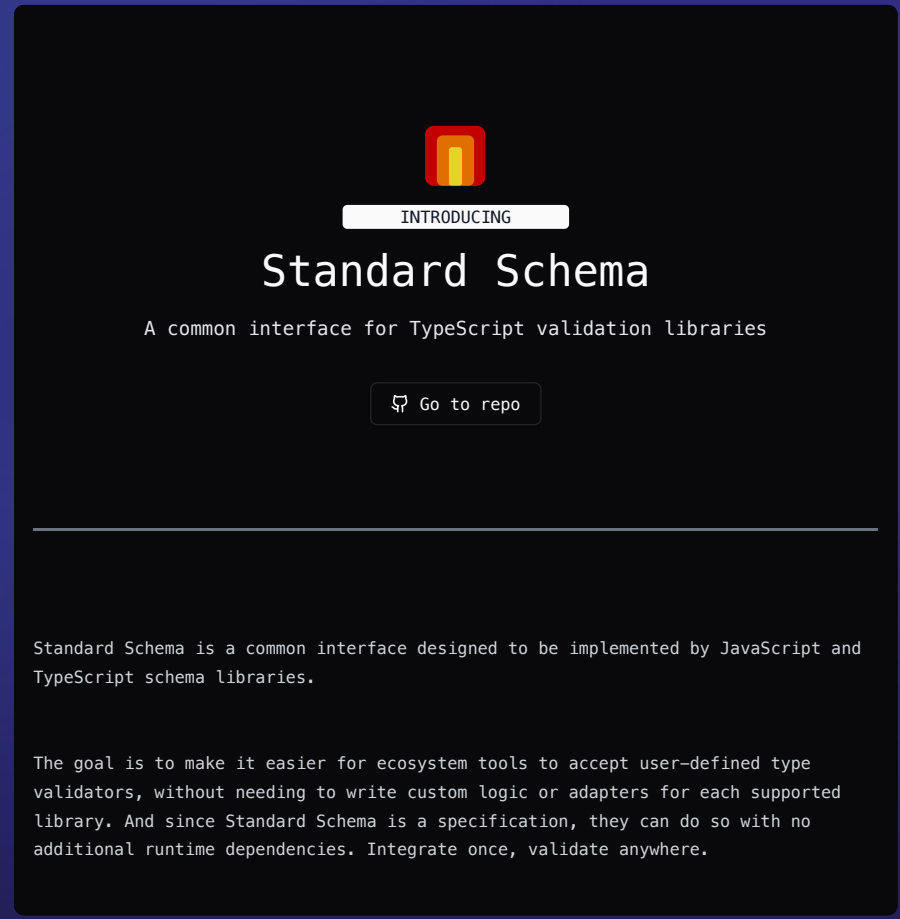
Early Warning

Early detection of mismatches, preventing deeper errors

What's Out There?

Runtime Validation Libraries

- Zod
- Valibot
- Arktype
- Joi
- Yup



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 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 API Bodies** with `nitro`
- **Mock Generation** with libraries like `zod-schema-faker`
- **Form Builders** hooking into real-time validation with `Veevalidate`
- **Generate Structured Data** with `vercel ai`
- **Validate LocalStorage Data**

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

```ts
// Example: Validate API Bodies with Nitro
import { userSchema } from './'
import { z } from 'zod'
import { defineEventHandler, readValidatedBody } from "h3";

return default defineEventHandler(async (event) => {
  const query = await readValidatedBody(event, userSchema.parse);

  return `Hello ${query.name}! You are ${query.age} years old.`;
})
```

Ecosystem Integration

1 Single Source of Truth

Zod schemas can be used in the frontend, backend, and generated from your database schema.

API Validation

Seamless integration with frameworks like Express, Fastify, Nitro, Hono to validate incoming requests.

Frontend Safety

Type-safe forms with React Hook Form, FormKit, Veevalidate, Shadcn, etc.

</> Schema Transformations

Transform, parse and validate data between different formats and structures.

Generate Mocks from Schemas

Generate realistic mock data for testing and development.

AI Structured Data Generation

Use schemas to generate structured data with AI.

Tooling Support

API Frameworks

- H3
- Nitro
- tRPC
- Hono
- oRPC
- GQLoom
- express-zod-api
- NestJS

Form Libraries

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

UI Frameworks

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

HTTP Clients

- upfetch
- rest-client
- better-fetch
- make-service
- zod-axios
- zodios

Utilities

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

Routing

- TanStack Router
- call-api
- Kitbag

Key Benefits

⬇️ Fewer Production Bugs
Runtime validation catches issues pre-deployment

⏱️ Faster Debugging
Detailed error paths & validation messages

✨ DevEx Improvement
Autocomplete & type safety across boundaries

🔄 1:1 Schema Parity
Single source of truth across all layers

Adoption Strategy

Incremental Approach
Validate one endpoint at a time, starting with critical paths

Schema Composition
Build a library of reusable schema components for consistency

Reuse Schemas
Create reusable schemas that serve as data models

Generate Assets
Auto-generate types, mocks, and API clients from schemas

What we learned

1

Powerful & Flexible

Zod is a powerful and flexible validation library that can be used in a variety of scenarios.

2

Easy Integration

Easily introduced into existing codebase with minimal effort.

3

Incremental Value

Provides a lot of value, even if you don't use it for everything.

4

Trust Boundaries

Can be added at any trust boundary in your codebase.

Thank You!

Let's Build Safer Systems Together

Slides, Demo Application & Resources

josephanson.com | josephanson