**Zod: Why you’re using TypeScript wrong**

[[](https://medium.com/@adrgautier?source=post_page-----b0c1583df089--------------------------------)](https://medium.com/@adrgautier?source=post_page-----b0c1583df089--------------------------------)

[[ekino-france](https://medium.com/ekino-france?source=post_page-----b0c1583df089--------------------------------)](https://medium.com/ekino-france?source=post_page-----b0c1583df089--------------------------------)

[Adrien Gautier](https://medium.com/@adrgautier?source=post_page-----b0c1583df089--------------------------------)

·

Follow

Published in

[ekino-france](https://medium.com/ekino-france?source=post_page-----b0c1583df089--------------------------------)

·

6 min read

·

Nov 10

659

14



Photo by [Ash Edmonds](https://unsplash.com/@badashphotos?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**TL;DR:** External data must be validated on runtime.

If you have some experience in web development, you have inevitably encountered runtime errors when working with external data coming from an API. Working with TypeScript helps to significantly reduce these errors by reminding the structure and type of any data, across the entire application. However, while TypeScript is good at preventing impossible operations on data **known** during compilation, it can be too permissive regarding **external** (in other words, **unknown**) data.

In this article, I will explain why TypeScript allows you to write code that can fail on runtime and how Zod can prevent these data-related errors.

**The goal of TypeScript**

As I said in the introduction, the idea behind TypeScript is to track the structure and type of any data across the entire code. This not only helps to provide autocompletion in the IDE, but it also prevents invalid operations that would cause errors during runtime. In theory, every possible runtime error can be predicted and identified during TypeScript **compilation**. But this is not the case.

**Is TypeScript missing its goal?**

In reality, TypeScript's primary objective is to improve productivity. This means that TypeScript will always chose productivity over “safety”.

A good demonstration of this is the type any. It exists, yet it’s widely accepted that we shouldn’t use it. However, not writing a single any in our code doesn't mean our application is immune from runtime errors. Take a look at the following snippet :

const obviouslyAnArticle: Article = JSON.parse(input); // input is a string

Because JSON.parse return type is any, it can be associated with a variable **explicitly typed** (as Articlein this example). Without writing any ourselves, we are telling TypeScript to ignore **the runtime probability** where the parsed content does not satisfy the type Article.

We have to keep in mind that any is often used in external definition files ([and it doesn't look like this is going to change](https://github.com/microsoft/TypeScript/issues/26188)) which means we must be even more careful.

unknown and assertions

If unknown was used instead of any the above snippet would not be possible. We would be required to write explicit [assertions](https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-assertions) using the as keyword :

const shouldBeAnArticle = JSON.parse(input) as Article;

With this syntax, we explicitly tell TypeScript to lower its guard. It is still bad, but no longer hidden!

**Type narrowing expressions**

Instead of relying on unsafe type assertions, we can use **type narrowing**expressions.

*“the process of refining types to more specific types than declared is called narrowing”* — [TypeScript documentation](https://www.typescriptlang.org/docs/handbook/2/narrowing.html)

For example, the typeof operator (provided by JavaScript) can determine an object’s type during **runtime**.

console.log(typeof 42);  
// Expected log: "number"

When used in a condition, TypeScript is able to narrow the type of the object.

if(typeof input === "string") {  
 // input is narrowed to the type string  
 submit(input.toLowerCase());  
}

This expression allows TypeScript to predict that input can only be a string in this scope.

While assertions tells TypeScript to trust the developer,  
narrowing expressions infers types from the runtime logic.

**Discrimination**

While TypeScript can narrow types with [many other expressions](https://www.typescriptlang.org/docs/handbook/2/narrowing.html), this only makes sense when refining either unions or primitive types. I call that “types discrimination”.

type Fish = { swim: () => void };  
type Bird = { fly: () => void };  
   
function move(animal: Fish | Bird) {  
 if ("swim" in animal) {  
 // input is narrowed to the type Fish  
 return animal.swim();  
 }  
 // input is narrowed to the type Bird  
 return animal.fly();  
}

With the above example, the keyword in allows TypeScript to discriminate the type of the animal object.

With unkown data, types discrimination can be a waste of time:

if(typeof input !== "string") {  
 // input is still unkown  
}

This means we **cannot** rely exclusively on type narrowing expressions for external data. We need another way to narrow the type down: **data validation**.

**Zod to the rescue**

[Zod](https://github.com/colinhacks/zod) at its core, is an object schema validation tool. This means that it can ensure, on **runtime**, the validity of any given object with a defined schema.

**Declaring the schema**

The first thing to do with Zod is to define the schema.

import \* as z from "zod";  
  
const userSchema = z.object({  
 id: z.number(),  
 name: z.string(),  
 age: z.number().optional()  
}).strict();

You might be familiar with this approach if you’ve used other validation tools like [yup](https://github.com/jquense/yup) or [joi](https://github.com/sideway/joi) before. Zod provides multiple functions (like object(), string()), each returning a Zod schema that **can be combined** in a bigger schema.

Each piece of schema can be “refined” using methods (like .optional()) which can be queued to obtain complex validation rules.

Thanks to Zod popularity, you can find [many tools](https://zod.dev/?id=x-to-zod) to help you convert your existing types and interfaces into Zod schemas. I also like the [transform.tools](https://transform.tools/json-to-typescript) website to quickly convert a JSON file into Zod schemas.

**Using the schema**

The schema provides essentially two ways to validate data. The .parse() method, which can throw errors, and the .safeParse() method:

const result = userSchema.safeParse(input);  
if (!result.success) {  
 result.error;  
} else {  
 result.data; // data type is infered from userSchema  
}

Either the parse fails or the parse returns the object which matches the defined validation schema. In that case, **the object inherits** the type inferred from **the schema structure**.

**Inferring a type from a schema**

Generally, data is shared across multiple scopes and contexts. For this reason, we usually declare a type alias once and then use it everywhere the data goes. Zod provides the z.infer<> generic to access the type inferred from a schema.

type Article = z.infer<typeof articleSchema>;

Zod allows to define complex validation rules, such as checking the length of the string: *z.string().min(6)*

Those rules have no equivalent in TypeScript logic and are translated into the closest type, in this case: string.

**Practical Zod uses**

So, where would you use Zod in your TypeScript project?

**Parsing API response**

*“Never trust the backend”*— Every front-end developers

The main source of unknown/unpredictable data is an API response. You can manually validate the data coming from a fetch promise.

fetch(getArticle)  
 .then((response) => response.json())  
 .then((data) => {  
 return articleSchema.parse(data);  
 })  
 .catch(console.error);

You can also use a dedicated library like [Zodios](https://github.com/ecyrbe/zodios) (a lib based on axios) to do that.

Because you have no control over the incoming data structure, you may want to make the external data more consistent with your project conventions and logic. Zod provides the [.transform() method](https://github.com/colinhacks/zod/tree/v3#transform) to do this during validation.

**Validating form data**

Another source of external data is user inputs. Zod provides some built-in utils for string validation. You can, of course, implement your own rules using the [.refine() method](https://github.com/colinhacks/zod/tree/v3#refine).

const myString = z.string().refine((val) => val.length <= 255, {  
 message: "String can't be more than 255 characters",  
});

If you are using React, you can use Zod schemas for form validation in [react hook form](https://react-hook-form.com/docs/useform#resolver).

**Discriminating types?**

I started using Zod with [version 1](https://github.com/colinhacks/zod/tree/v1#type-guards). This version included a .check() method which allowed to use a schema as a **type guard**, which would be used in a condition for type discrimination.

Because of this feature, it was tempting to go for the “full-schema” approach and use Zod **for both** validation and type discrimination. However, taking this approach quickly turned out to be **a waste of time**.

This method has since been removed from the next version of the library. This is a good thing because **Zod now focuses on its original purpose**: external data validation. Type narrowing expressions prove sufficient for most situations for type discrimination.

If you really feel limited with native expressions and the if/else statements syntax, you might also want to take a look at [ts-pattern](https://github.com/gvergnaud/ts-pattern)

**Summary**

TypeScript is too permissive out of the box. For safer code, external data (unknown by nature) must be validated with tools like Zod. Zod is most useful for validating unpredictable data like form inputs or API response. For most other scenarios, however, type narrowing expressions should be sufficient.