Clean FE Architecture with Valid Data
The Problem
The Problem

```javascript
{ a: 'string' }
{ b: 7 }
```

Frontend → Data → Backend
The Problem

```javascript
{ a:
  { b:
    { c: string } } }

{ a:
  { b:
    { d: 5 } } }
```
When the data arrive in the frontend ...
... and are distributed to all components ...
... validity must be checked everywhere.
When parts of the data are passed on internally ...
... the issue multiplies.
The resulting code

```typescript
export type Data = { a: { b: { c: string } } };

const f = (d: Data): void => {
    if (d.a === undefined || d.a.b === undefined || typeof d.a.b.c !== "string") {
        throw new Error();
    }
    // do something
};
```

often riddled with

- error checks
- guard clauses
- error handling
Problems

- Excessively defensive code is not as readable or maintainable
- Not clear how to handle invalid data
  - Throw error? And what then?
- Not clear how to type the received data:
  - Missing types (represented by any, unknown, ...) is problematic
  - Weak typing (with optional fields etc.) is also problematic
  - Normal typing suggests that data is correct and blocks necessary checks:

```typescript
typeof d.a.b.c !== "string") {
  'typeof' check is always false: 'c' always has type 'string'
```

Simplify  More actions...
Better Alternative:
Approach

- Check all data right after receiving them
- Erroneous data can be rejected immediately
- No bad surprises at a later point due to unexpected data
- Domain code is free of data checks
- Types can exactly describe the expected data
- Provides good support for the devs
- No struggle with the type system
The Difficulty

- No runtime data check in JavaScript
- Not even TypeScript checks at runtime!
- Check needs to be implemented by the devs
Questions regarding

*Clean FE Architecture with Valid Data*
Approaches for Data Validation
General Approach

- Read and check data
- Standard tool: parser
First approach: Parser Generator

- Scanner and parser
- Scanner tokenizes the character stream
- Parser recognizes grammatical structures in the token stream
- Two stand-alone applications are generated
- Those are integrated into the own code as “black boxes”

Advantages:
- Can treat complex and ambiguous languages efficiently
- Widely known

Disadvantages:
- Sometimes annoying to process the scanner output
- Steep learning curve as it requires to learn the description languages for scanner and parser
Example: Thermostat Control

Possible commands:

```
heat on
  Heater on!
heat off
  Heater off!
target temperature 22
  New temperature set!
```

Quelle: https://tldp.org/HOWTO/Lex-YACC-HOWTO-4.html
Example: Thermostat Control

Scanner-Description (Lex):

```c
{%
#include <stdio.h>
#include "y.tab.h"
%
%%
[0-9]+ return NUMBER;
heat return TOKHEAT;
on|off return STATE;
target return TOKTARGET;
temperature return TOKTEMPERATURE;
\n  /* ignore end of line */;
[ \t]+ /* ignore whitespace */;
%%
```
Example: Thermostat Control

Parser-Description (yacc):

```
%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE

commands: /* empty */
    | commands command ;

command: heat_switch
    | target_set ;

heat_switch:
    TOKHEAT STATE ;

target_set:
    TOKTARGET TOKTEMPERATURE NUMBER ;
```
Second Approach: Parser Combinator

► Built from functions
► Simple parser functions take the role of the scanner
► Complex parser functions validate more powerful language constructs

► Advantages:
  ► Easy to use
  ► Implementation is straightforward without the need for a generator
  ► Separate definition languages are not required

► Disadvantages:
  ► Not very well suited for complex languages
  ► Parsing process is not easily optimizable
Example: Thermostat Control

```javascript
import * as z from "zod";
const ZState = z.union([z.literal("on"), z.literal("off")]);
const ZHeatSwitch = z.object(
    {heat: ZState
    }).required().strict();

const ZTemperature = z.object(
    {temperature: z.number()
    }).required().strict();
const ZTargetSet = z.object(
    {target: ZTemperature
    }).required().strict();

const ZCommand = z.union([ZHeatSwitch, ZTargetSet]);
export const ZCommands = z.array(ZCommand);
export type ICommands = z.infer<typeof ZCommands>;
```
Simple Building Blocks

Simple combinator functions deal with constants and variables:

```plaintext
z.literal("on")
z.number()
```
Basis: Parser

- `parse()` deserializes / parses
More Complex Building Blocks

\[
z.\text{union}([z.\text{literal}("on"), z.\text{literal}("off")])
\]
Structure of Schema and Datatypes

Modelling data structures with combinator functions:

```javascript
const ZUser = z.object({
  userId: z.number(),
  name: z.string()
});
```

TypeScript type generation:

```typescript
type IUUser = z.infer<typeof ZUser>;
```

```typescript
type IUUser = t.TypeOf<typeof IOUser>;
```

<table>
<thead>
<tr>
<th>Alias for:</th>
<th>t.TypeOf&lt;typeof IOUser&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial type:</td>
<td>{name: t.TypeOf&lt;StringC&gt;, userId: t.TypeOf&lt;NumberC&gt;}</td>
</tr>
</tbody>
</table>
Usage

Transform data (e.g. JSON string) to JavaScript data:

```javascript
const myData: unknown = JSON.parse(myString);
```

Usage when decoding data of unknown format:

```javascript
const myUserValidation: IUser = ZUser.parse(myData);
```

Accessing the data through the desired type

```javascript
try {
    const myUserValidation: IUser = ZUser.parse(myData);
} catch (e) {
    if (e instanceof Z.ZodError) {
        console.log(e);
    }
}
```
Advantage 1: Support through typing

- Shape of data is laid out in the type system
- Support for developers
- No need for example data to “peek at the structure”
Advantage 2: Receiver performs Contract Testing

- Unearthes misunderstandings in communication with data provider
- Points out errors in the creation of the received data
- Notifies when an external API was changed (e.g. when we are a conformist)
Questions regarding

*Approaches for Data Validation*

?
Practical application with ZOD
### Step 1

<table>
<thead>
<tr>
<th>Type</th>
<th>TypeScript</th>
<th>codec / combinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>literal</td>
<td>'s'</td>
<td>z.literal('s')</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>z.null()</td>
</tr>
<tr>
<td>undefined</td>
<td>undefined</td>
<td>z.undefined()</td>
</tr>
<tr>
<td>void</td>
<td>void</td>
<td>z.void()</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>z.string()</td>
</tr>
<tr>
<td>number</td>
<td>number</td>
<td>z.number()</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>z.boolean()</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>z.unknown()</td>
</tr>
<tr>
<td>integer</td>
<td>BigInt</td>
<td>z.bigint()</td>
</tr>
</tbody>
</table>

© 2023 Nicole Rauch. All rights reserved.
### Step 2

<table>
<thead>
<tr>
<th>Type</th>
<th>TypeScript</th>
<th>codec / combinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>array of type</td>
<td>Array(&lt;A&gt;)</td>
<td>z.array((A)) or A.array()</td>
</tr>
<tr>
<td>tuple</td>
<td>[ A, B ]</td>
<td>z.tuple([ A, B ])</td>
</tr>
</tbody>
</table>

© 2023 Nicole Rauch. All rights reserved. Practical application with ZOD – Page 53
## Step 3

<table>
<thead>
<tr>
<th>Type</th>
<th>TypeScript</th>
<th>codec / combinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>record of type</td>
<td>Record&lt;K, A&gt;</td>
<td>z.record(K, A)</td>
</tr>
<tr>
<td>type alias</td>
<td>type T = { name: A }</td>
<td>z.object({ name: A })</td>
</tr>
<tr>
<td>partial</td>
<td>Partial&lt;{ name: string }&gt;</td>
<td>z.object({ name: z.string }).partial()</td>
</tr>
<tr>
<td>strict</td>
<td>-</td>
<td>z.object({ name: A }).strict()</td>
</tr>
</tbody>
</table>

▶ strict: no unknown extra properties
## Step 4

<table>
<thead>
<tr>
<th>Type</th>
<th>TypeScript</th>
<th>codec / combinator</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>A</td>
<td>B</td>
<td>z.union([ A, B ])</td>
</tr>
<tr>
<td>intersection</td>
<td>A &amp; B</td>
<td></td>
<td>z.intersection( A, B )</td>
</tr>
<tr>
<td>keyof</td>
<td>keyof M</td>
<td>z.keyof(M)</td>
<td></td>
</tr>
</tbody>
</table>
Step 5 - Putting it all together
Questions regarding

Practical application with ZOD
Links
Runtime Validation

Basics

- Parser combinator:
  https://en.wikipedia.org/wiki/Parser_combinator

ZOD Documentation:

- https://zod.dev/

Alternative Libraries:

- Commented overview of Joi, Yup, io-ts, Runtypes, Ow:
  https://zod.dev/?id=comparison
Branding

Basics:

▶ https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d
Questions regarding

*Links*

?
Thank You!

E-Mail  info@nicole-rauch.de
Twitter @NicoleRauch
Web  http://www.nicole-rauch.de

Domain-Driven Design · Specification by Example
Software Craftsmanship
React & Redux · TypeScript
Functional Programming
Einführung: islandworks / 360 images
Parser: Uriel Soberanes
https://unsplash.com/photos/L1bAGEWYCtk
Aufgaben: congerdesign / 4188 images
https://pixabay.com/photos/puzzle-pieces-puzzle-patience-mesh-1925425/
Custom Types: Vishnu Mohanan
https://unsplash.com/photos/vtg8tAdoWVQ