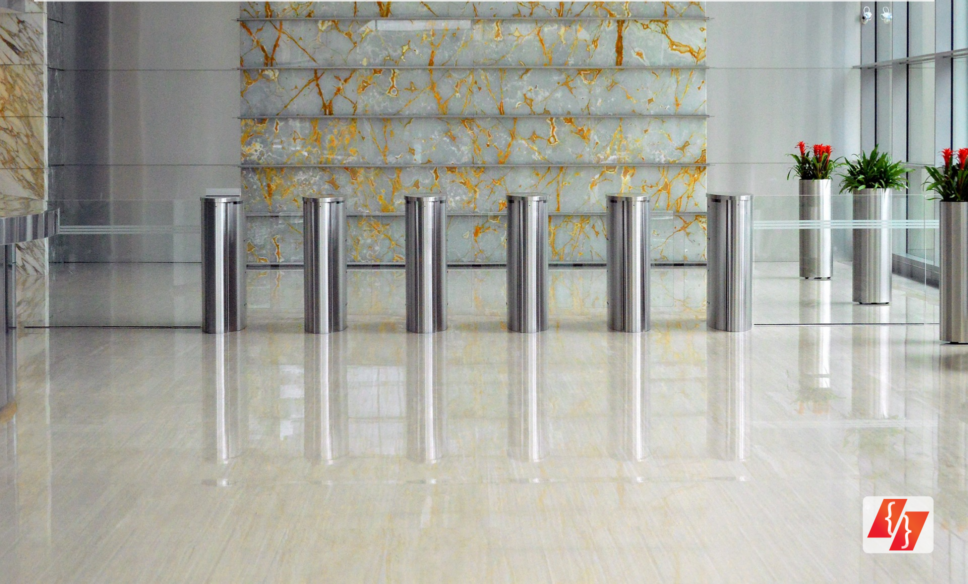
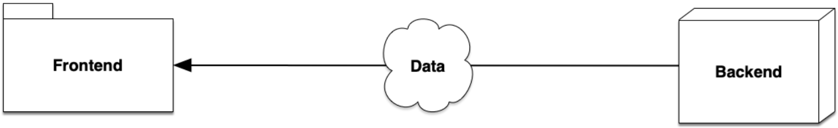
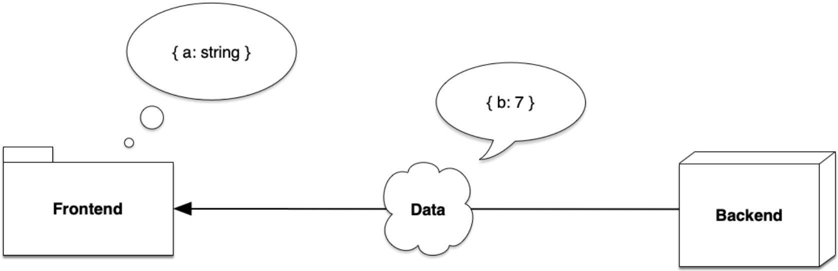
Clean FE Architecture with Valid Data



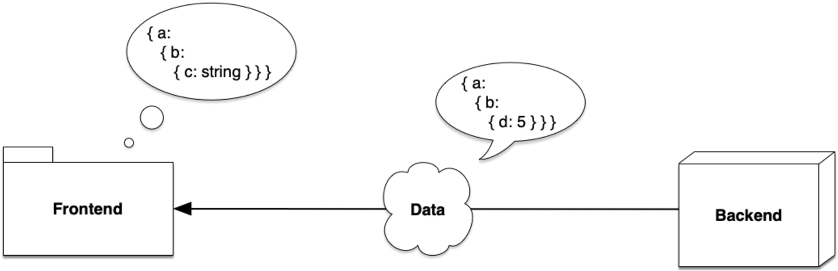
The Problem



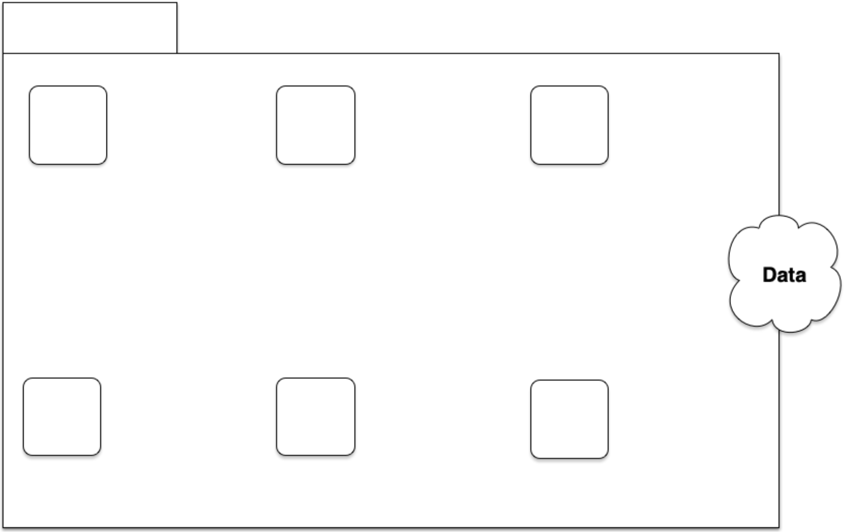
The Problem



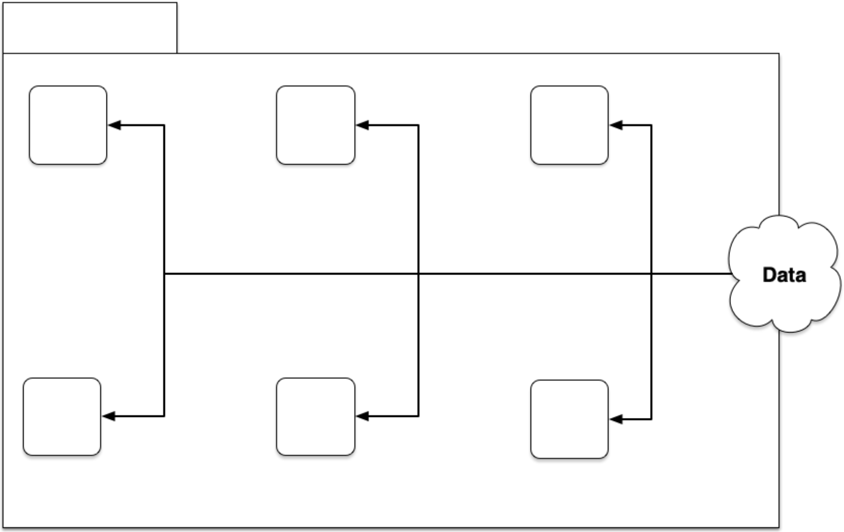
The Problem



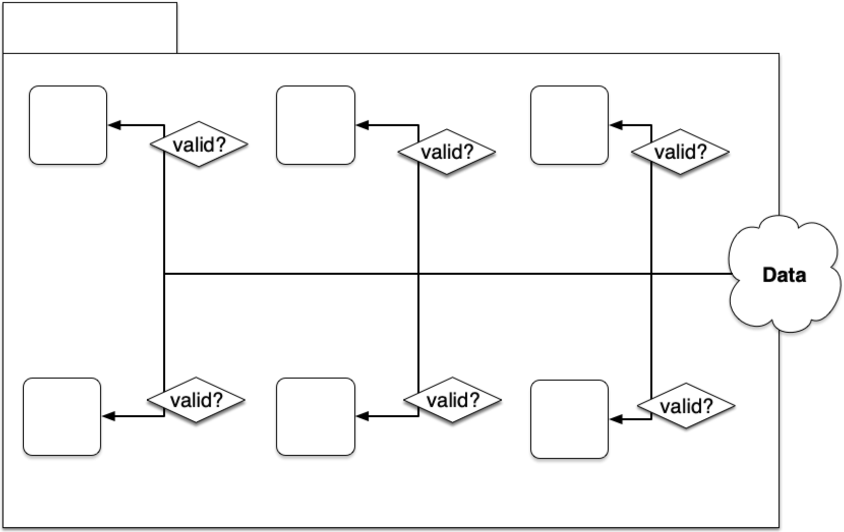
When the data arrive in the frontend ...



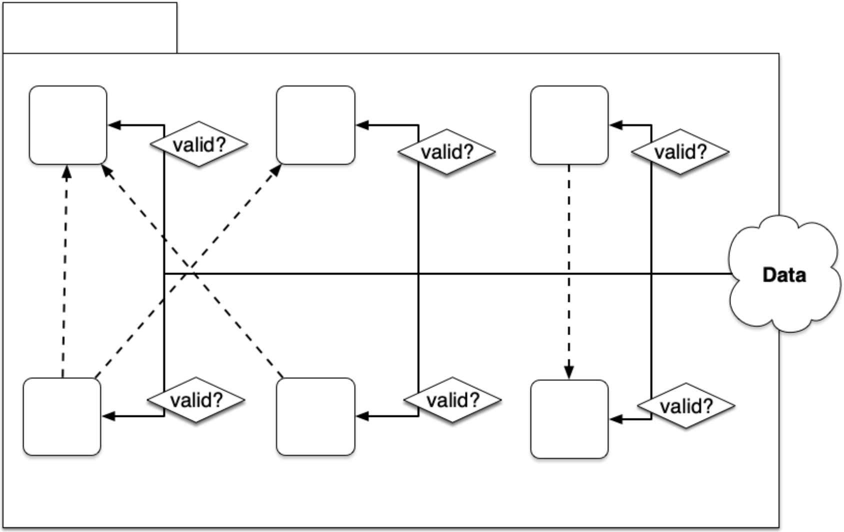
... and are distributed to all components ...



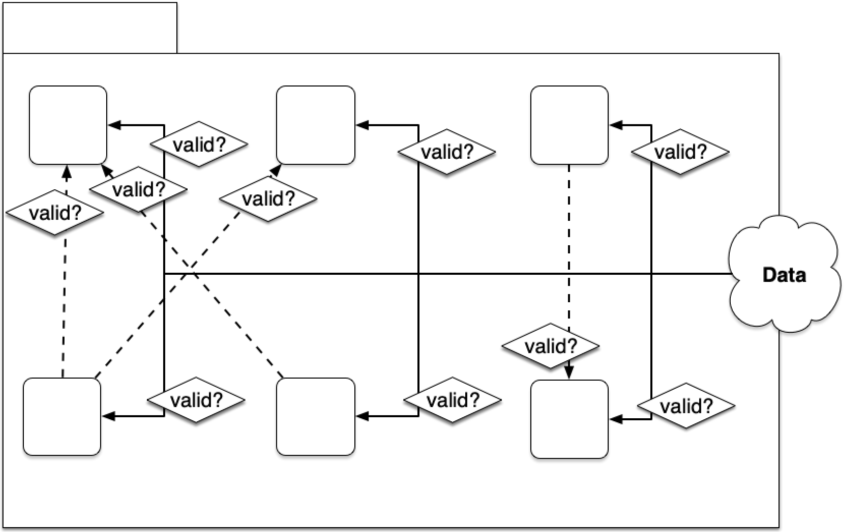
... validity must be checked everywhere.



When parts of the data are passed on internally ...



... the issue multiplies.



The resulting code

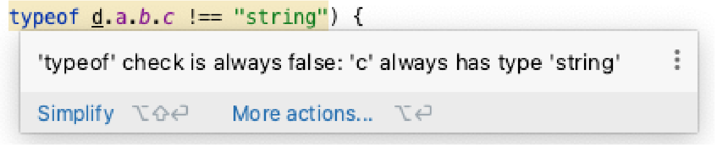
```
export type Data = { a: { b: { c: string } } };

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

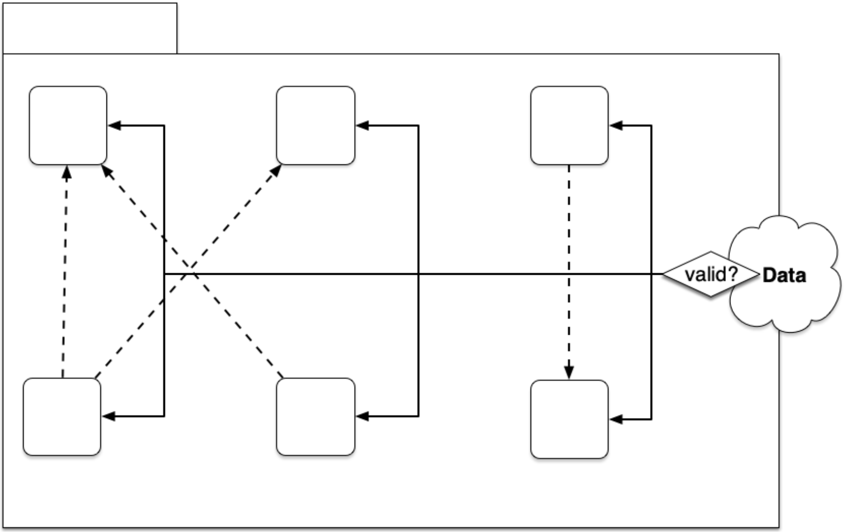
- error checks
- guard clauses
- error handling

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:



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):

```
%{
#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

```
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:

```
z.literal("on")
z.number()
```

Basis: Parser

- `parse()` deserializes / parses

More Complex Building Blocks

```
z.union([z.literal("on"), z.literal("off")])
```

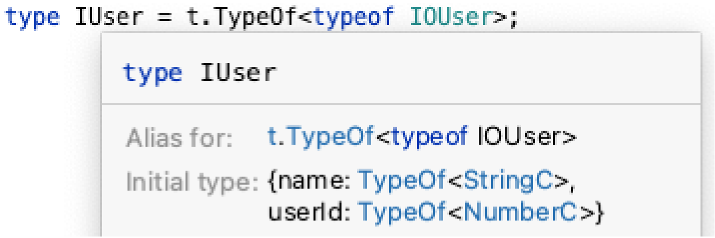
Structure of Schema and Datatypes

Modelling data structures with combinator functions:

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

TypeScript type generation:

```
type IUser = z.infer<typeof ZUser>;
```



Usage

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

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

Usage when decoding data of unknown format:

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

Accessing the data through the desired type

```
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

Setup

`https://github.com/NicoleRauch/ValidationCode`

Node 14 `https://nodejs.org/` `npm install`

Step 1

Type	TypeScript	codec / combinator
literal	’s’	z.literal(’s’)
null	null	z.null()
undefined	undefined	z.undefined()
void	void	z.void()
string	string	z.string()
number	number	z.number()
boolean	boolean	z.boolean()
unknown	unknown	z.unknown()
integer	BigInt	z.bigint()

Step 2

Type	TypeScript	codec / combinator
array of type	Array<A>	z.array(A) or A.array()
tuple	[A, B]	z.tuple([A, B])

Step 3

Type	**TypeScript**	**codec / combinator**
record of type	Record<K, A>	z.record(K, A)
type alias	type T = { name: A }	z.object({ name: A })
partial	Partial<{ name: string }>	z.object({ name: z.string }).partial()
strict	-	z.object({ name: A }).strict()

- strict: no unknown extra properties

Step 4

Type	TypeScript	codec / combinator	Remark
union	A | B	z.union([A, B])	
intersection	A & B	z.intersection(A, B)	only two types
keyof	keyof M	z.keyof(M)	creates an enum schema

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:

- Schema - https://github.com/Effect-TS/schema
- 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

NICOLE RAUCH
software development &
development coaching

Domain-Driven Design · Specification by Example
Software Craftsmanship
React & Redux · TypeScript
Functional Programming

Credits

Einführung: islandworks / 360 images

`https://pixabay.com/photos/inside-business-center-interior-1499606/`

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`