HAW HAMBURG

Bachelorarbeit

Andre Köpke

Minimalredundante Validierung von Benutzereingaben in Front- und Backend eines interaktiven Softwaresystems

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Andre Köpke

Minimalredundante Validierung von Benutzereingaben in Front- und Backend eines interaktiven Softwaresystems

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Axel Schmolitzky
Zweitgutachter: Prof. Dr. Jens von Pilgrim

Eingereicht am: 11. März 2020

Letzte Änderung am: 9. März 2021

Andre Köpke

Thema der Arbeit

Minimalredundante Validierung von Benutzereingaben in Front- und Backend eines interaktiven Softwaresystems

Stichworte

Validierung, Vermeidung von Redundanzen, REST-API

Kurzzusammenfassung

In interaktiven Softwaresystemen müssen die Eingaben von Nutzern validiert werden. Damit wird verhindert, dass ein Nutzer z. B. „ABC!!!“ als Postleitzahl einträgt.
Moderne interaktive Softwaresysteme sind keine sog. *Fat Applications*, sondern bestehen oftmals aus einem Backend (Serveranwendung) und einem Frontend (Clientanwendung, z. B. Webseiten, Desktopclients, Apps).
Aufgrund dieser Architektur müssen Validierungsprüfungen in verschiedenen Komponenten redundant programmiert werden.
Ziel dieser Bachelorarbeit ist eine Lösung zu finden, mit der diese Redundanz verhindert wird.

Andre Köpke

Title of Thesis

Minimal redundant validation of user input in front- and backend of an interactive software system

Keywords

Validation, Avoiding redundancies, REST-API

Abstract

In interactive softwaresystems, user input must be validated. This prevents a user from entering e.g. "ABC!!!" as a postcode.

Modern interactive software systems are not so-called *Fat Applications*, but rather consist – as is often the case – a backend (server application) and a frontend (client application, e.g. websites, desktop clients, apps).
Due to this architecture, validation checks must be programmed redundantly in different components.
The aim of this Bachelor's thesis is to find a solution to avoid such redundancies.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **viii**

Quellcodeverzeichnis **ix**

Abkürzungen **x**

1 Einleitung **1**

2 Cysmo® **5**

3 Mögliche Lösungen für minimalredundante Validierungen **7**

3.1 Kotlin als IDL 7
3.2 OpenAPI als IDL 9
3.3 Endpunkt für Validierungsregeln 13
3.4 Vergleich der Lösungen und Wahl eines Favoriten 15

4 Vorarbeiten für OpenAPI **17**

4.1 Analyse der Codegeneratoren von OpenAPI 17
4.2 Bisherige Validierung 18
4.3 Erweiterung des *typescript-angular*-Codegenerators 20

5 Einbau in ein bestehendes Produkt **23**

5.1 Auslagerung der API 23
5.2 CI/CD zur automatischen Veröffentlichung 25
5.3 Partielle Umstellung im *Frontend* auf *OpenAPI Specification* 28
5.4 Partielle Umstellung im *Backend* auf *OpenAPI Specification* 33

6 Analyse des Einbaus **37**

7 Zusammenfassung **41**

Literaturverzeichnis **44**

Glossar **46**

Selbstständigkeitserklärung **50**

Abbildungsverzeichnis

1.1 Screenshot aus *Cysmo*® bei Eingabe einer ungültigen Postleitzahl. 2
1.2 UML-Sequenzdiagramm einer redundanten Validierung. 3

2.1 UML-Komponentendiagramm von *Cysmo*®. 5

3.1 Vorschau einer Übersetzung von OpenAPI Specification nach Typescript. . 10

5.1 Screenshot aus IntelliJ mit dem neuem Unternehmensobjekt. 31
5.2 Screenshot aus IntelliJ mit Git-Diff von alter zu neuer Validierung. 33
5.3 Screenshot von Postman mit leerer Rückmeldung. 35
5.4 Screenshot von Postman mit nicht leerer Rückmeldung. 36

Tabellenverzeichnis

3.1 Vor- und Nachteile von Kotlin als IDL 9
3.2 Vor- und Nachteile von OpenAPI als IDL 13
3.3 Vor- und Nachteile eines Endpunkt für Validierungen 15

Quellcodeverzeichnis

3.1 Datenklasse für Userregistrierung (Kotlin). 8
3.2 Pet-Beispiel für OpenAPI (Yaml). 11
3.3 Pet-Klasse generiert von OpenAPI (Java). 11
3.4 ExceptionHandler mit erweiterten Fehlermeldungen (Java). 12
3.5 HttpInterceptor für Validierungsprüfungen (Typescript). 14

4.1 Cysmo Frontend-Validierung (Html+ng2). 18
4.2 Cysmo Frontend-Validierung (Typescript). 19
4.3 Spring Template für Validierung in Java (Mustache) 21
4.4 Code mit neuem Generator (Typescript). 22

5.1 Annotationen für OpenAPI in *Cysmo®* (Java). 24
5.2 Neues Dockerfile für OpenAPI Generator (Dockerfile). 26
5.3 Gitlab-CI-Konfiguration, um OpenAPI nach Java zu übesetzen (Yaml). . 27
5.4 Unternehmen für OpenAPI (Json). 30
5.5 Generiertes Modell für Unternehmen (Typescript). 32
5.6 Neue Abhängigkeiten in Cysmo für OpenAPI (Groovy). 34
5.7 Neuer Methodenkopf für den Cysmo Controller mit OpenAPI (Java). . . . 34

Abkürzungen

API Application Programming Interface, *siehe Glossar.*

CI/CD Continuous Integration / Continuous Development, *siehe Glossar.*

DTO Datentransferobjekt, *siehe Glossar.*

IDE Integrierte Entwicklungsumgebung.

IDL Interface Definition Language *siehe Glossar.*

JVM Java Virtual Maschine, *siehe Glossar.*

OOP Objektorientierte Programmierung.

PoC Proof of Concept.

REST Representational State Transfer, *siehe Glossar.*

RTT Round Trip Time, *siehe Glossar.*

TTV Time-to-Value, *siehe Glossar.*

UML Unified Modeling Language.

1 Einleitung

Fast jede Software benötigt Benutzereingaben. Die Daten, die eingeben werden können, unterscheiden sich stark je nach Anwendungsfall. Es kann eine Liste sein, eine bestimmte Taste, ein Button zum Anklicken oder einfach nur Text. Wenn es sich um letzteres handelt, dann sollte die Software diese Texteingabe validieren.
Als Beispiel, wenn eine Software den Endnutzer dazu auffordert eine Postleitzahl einzutippen, dann könnte der Nutzer eine ungültige Postleitzahl eintippen. Dies könnte dazu führen, dass die Software nicht mehr ordnungsgemäß funktioniert. Aus diesem Grund ist eine Validierung notwendig.

> **Begriffabgrenzung Validierung**
> In dieser Bachelorarbeit wird der Begriff *Validierung* oft verwendet. Damit sind rein syntaktische Validierungen gemeint. Im Falle einer Postleitzahl wird nicht validiert, ob diese existiert, sondern nur ob die Form gültig ist. Fachliche Validierungen, die mitunter auch sehr komplex ausfallen können, werden nicht betrachtet.

Moderne Softwarelösungen bestehen aus mehreren Komponenten.
Die gröbste Unterscheidung kann zwischen dem *Backend* und *Frontend* gezogen werden. Benutzereingaben, die der Endnutzer im *Frontend* tätigt, werden über ein *Application Programming Interface (API)* an das *Backend* übermittelt.
Damit keine invaliden Daten in die fachliche Logik des *Backends* kommen, um somit eine ordnungsgemäße Funktionalität sicherzustellen, muss das *Backend* jene Daten validieren. Denn es ist möglich, dass der Endnutzer falsche Daten in das *Frontend* eingegeben hat und im *Frontend* diese Daten nicht ausreichend validiert wurden. Außerdem ist die *API* des *Backends* i.d.R. öffentlich erreichbar, wodurch die Möglichkeit besteht, dass Hacker versuchen gezielt invalide Daten an das *Backend* übermitteln, um dessen Betrieb zu stören.

Jeder Aufruf vom *Frontend* zum *Backend* verbraucht Zeit und Ressourcen. Zeitlich wird mindestens die *Round Trip Time (RTT)* zwischen *Frontend* und *Backend* benötigt. Zusätzlich werden noch Ressourcen (wie Netzwerkbandbreite, CPU-Rechenzeit und Arbeitsspeicher) verbraucht.
Verschärft wird der Ressourcenverbrauch durch den Fakt, dass Validierungen oftmals sehr früh oder sogar live durchgeführt werden. Dadurch können die Endnutzer schon während des Tippens sehen, ob ihre Eingabe gültig ist. Dies ermöglicht qualitativ hochwertige Echtzeitrückmeldungen wie am folgendem Bild dargestellt:

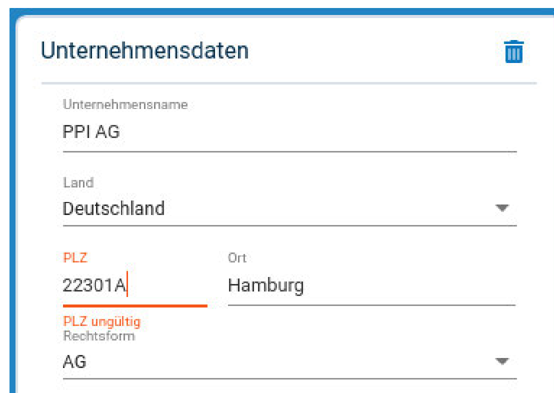

Abbildung 1.1: Screenshot aus *Cysmo*® bei Eingabe einer ungültigen Postleitzahl.

Aufgrund des hohen Ressourcenverbrauches werden solche Aufrufe zum *Backend* möglichst vermieden. Um so wenig Aufrufe wie möglich durchführen zu müssen, kann das *Frontend* die Daten ebenfalls validieren. Dies führt jedoch zu einer redundanten Implementierung der Validierungsprüfung, da nun sowohl das *Frontend* als auch das *Backend* Validierungen durchführen.
Im folgendem Bild ist der komplette Ablauf in *Unified Modeling Language (UML)* [14] dargestellt:

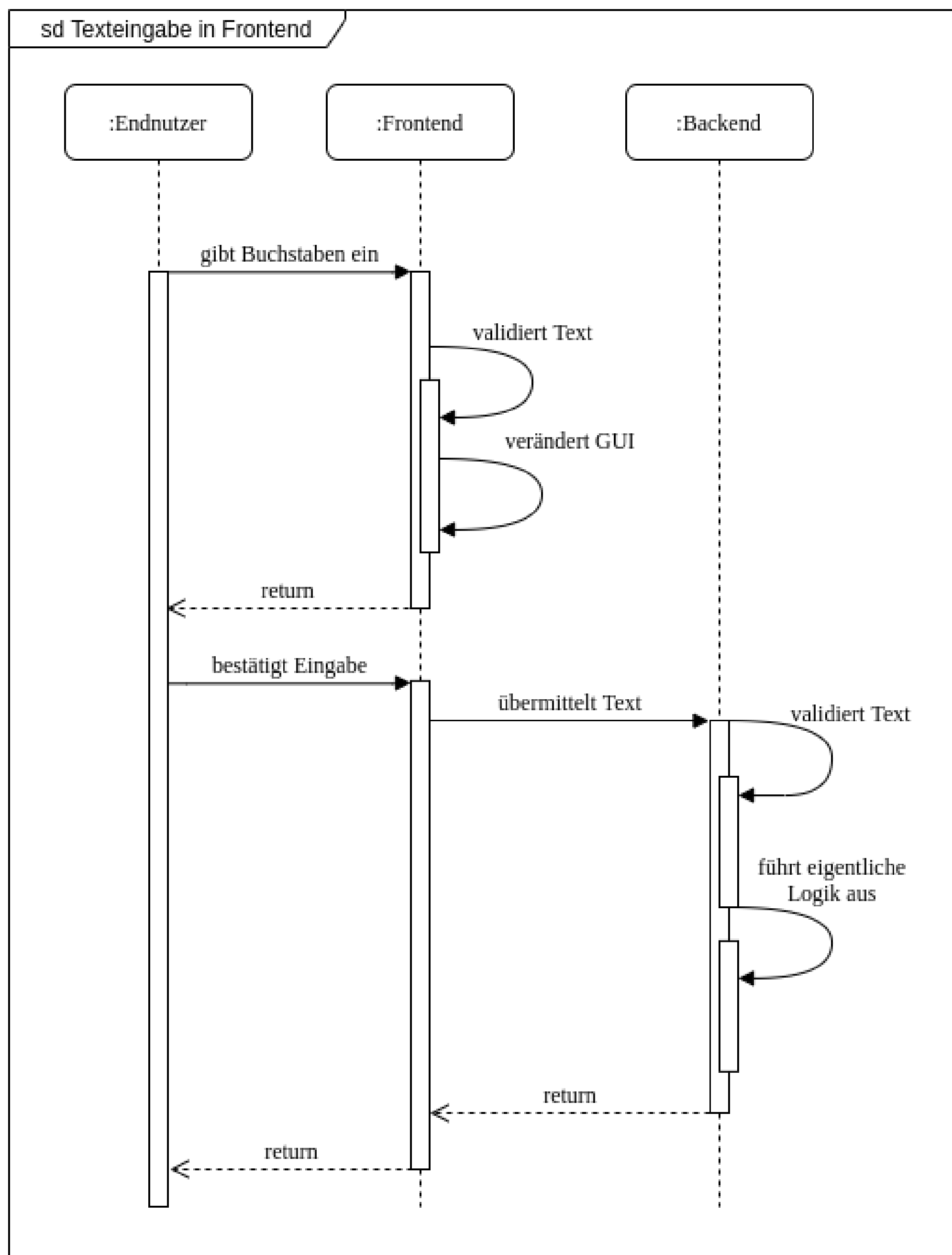

Abbildung 1.2: UML-Sequenzdiagramm einer redundanten Validierung.

Alle Komponenten liegen üblicherweise in unterschiedlichen Repositories und können unabhängig voneinander bearbeitet werden. Dadurch kann es vorkommen, dass redundante Validierungsprüfungen in verschieden Komponenten nicht mehr identisch sind. Verschärft

wird dieses Problem dadurch, wenn an den verschiedenen Komponenten unterschiedliche Entwickler arbeiten. Verschiedene Validierungen können zu Inkonsistenzen der Systeme führen und erschweren die Nachvollziehbarkeit, da nicht klar ist, welche Validierung den tatsächlichen Anforderungen entspricht. Wenn sich eine Validierungsprüfung ändert, dann muss diese Änderung in alle Komponenten übertragen werden. Jedoch gibt es keine Prozesse, die sicherstellen, dass die Validierung oder gar die *API* in allen Komponenten identisch bleibt.

Aufgrund der Tatsache, dass *Backend*aufrufe viele Ressourcen verbrauchen, implementieren de facto alle Komponenten die genutzten *APIs* selbst. Dies betrifft sowohl konsumierende als auch anbietende Komponenten und dadurch entsteht das Problem der redundanten Implementierung. Dieses Problem ist kein theoretisches Problem, sondern tritt in der realen Welt auf. So finden sich z. B. im Produkt *Cysmo®* des Softwareunternehmens *PPI AG* redundant implementierte Validierungen, für die als *Proof of Concept (PoC)* eine minimalredundante Lösung implementiert werden soll.

Deswegen lauten die Ziele dieser Bachelorarbeit:

1. Identifizieren verschiedener Lösungsansätze für eine minimalredundante Implementierung von Validierungsprüfungen (siehe Kapitel 3).
2. Bewerten, vergleichen und Wahl eines Favoriten der Lösungen (siehe Kapitel 3).
3. Favorisierte Lösung in *Cysmo®* in Form eines *PoC* implementieren (siehe Kapitel 5).
4. Den Einbau in *Cysmo®* analysieren und Vor- und Nachteile identifizieren (siehe Kapitel 6).
5. Zusammenhängende Vorteile herauskristallisieren und somit Gewinn für die *PPI AG* schaffen (siehe Kapitel 7).

2 Cysmo®

Bei *Cysmo®* handelt es sich um ein Produkt der *PPI AG*, welches nur für Versicherer gedacht ist. Mit *Cysmo®* können Ratings für Unternehmen durchgeführt werden, die die IT-Sicherheit jener Unternehmen bewerten. *Cysmo®* verwendet hierbei externe Datenquellen, sodass kein Eingriff in die IT der Unternehmen nötig ist und somit kein Risiko für den Betrieb der IT entsteht. [12]

Technisch besteht *Cysmo®* aus mehreren Komponenten und trifft somit den typischen Aufbau, welcher in der *Einleitung* beschrieben wurde. Die obersten Komponenten von *Cysmo®* stellen sich wie folgt dar:

Abbildung 2.1: UML-Komponentendiagramm von *Cysmo®*.

Cysmo-Frontend

Das *Frontend* ist in *Typescript* implementiert und verwendet das *Angular*-Framework. Dieses *Frontend* wird direkt von den Sacharbeitern der Versicherungen verwendet. Hier können neue Ratings anhand der Eingaben der Sacharbeiter angelegt werden. Diese Eingaben müssen unbedingt validiert werden.
Außerdem werden alle Ratings grafisch aufbereitet und dargestellt. Im Hintergrund wird ausschließlich der *Cysmo-Core* verwendet.

Cysmo-Core (Backend)

Der *Cysmo-Core* kann auch als *Backend* beschrieben werden. Es handelt sich um eine Anwendung die in Java geschrieben wurde und das *Spring*-Framework verwendet. Mithilfe von *APIs* kann der *Cysmo-Core* vom *Cysmo-Frontend* sowie auch direkt vom Kunden angesprochen werden. *Cysmo-Core* übernimmt eine steuerende Rolle auf fachlicher Ebene und kapselt die *Rating-Engine* ab.

Rating-Engine

Die *Rating-Engine* ist das Herzstück von *Cysmo®* und bewertet die von außen sichtbaren Teile der IT-Infrastruktur von Unternehmen, um passende Ratings zu erstellen. Die *Rating-Engine* wurde in *Go* implementiert und teilt sich in viele weitere Komponenten.
Für die bewerteten Unternehmen ist es nicht möglich den bei der Analyse durch die *Rating-Engine* entstanden Traffic von alltäglichem, normalem Traffic zu unterscheiden, da fast keine Aufrufe direkt zum Unternehmen durchgeführt werden, sondern fast ausschließlich externe Datenquellen verwendet werden.

Diese Bachelorarbeit fokusziert sich gegen die *Representational State Transfer (REST)*-Schnittstelle zwischen *Backend* und *Frontend*. Diese Schnittstelle wurde ursprünglich als „mündlicher Vertrag“ definiert und wurde manuell im *Backend* und im *Frontend* implementiert. Da inzwischen die *API* auch von Kunden verwendet wird, hat man sich dazu entschlossen, die *API* mithilfe von der *OpenAPI Specification* zu beschreiben. Dies wird mit Annotationen erreicht, die im *Backend* definiert wurden. Dieser Fakt war sehr nützlich für diese Bachelorarbeit, dazu mehr in Abschnitt 5.1.

Außerdem besitzt *Cysmo®* ein ausgeklügeltes System für *Continuous Integration / Continuous Development (CI/CD)* und gilt Vorlage für andere Produkte innerhalb der *PPI AG*. Dieses System kann u. a. statische Codeanalysen durchführen und Umgebungen automatisiert hoch- und runterfahren.

3 Mögliche Lösungen für minimalredundante Validierungen

Es gibt verschiedene Lösungsansätze, damit die Validierungsprüfungen in einem System minimalredundant sind. Die relevantesten Ansätze werden in diesem Kapitel näher erläutert und bewertet. Aufgrund der Bewertung wird eine Lösung favorisiert, die dann als *PoC* implementiert wird.

3.1 Kotlin als IDL

Bei einem interaktiven Softwaresystem, welches aus mindestens einem *Backend* und *Frontend* besteht, handelt es sich um ein System, das aus mehreren Rechnersystemen besteht. Zum einen aus einem Server und zum anderem aus dem Computer, der das *Frontend* (z. B. eine Webseite) anzeigt.
Diese Rechnersysteme erscheinen dem Endnutzer als ein einziges, kohärentes System. Es ist dabei nicht ohne weiteres erkennbar, dass das *Frontend* mit dem *Backend* kommuniziert.

> *A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.*
>
> — van Steen und Tanenbaum

Daraus lässt sich folgern, dass es sich bei einem interaktiven Softwaresystem, welches zumindest aus einem *Backend* und einem *Frontend* besteht, um ein *verteiltes System* handelt. Somit lassen sich die Regeln anwenden, die *Tannenbaum* und *van Steen* für *verteile Systeme* aufgestellt haben. Eine der Regeln besagt, dass *verteile Systeme* standardirisierten Regeln folgen sollten, die die Syntax und Semantik der angeboten Dienste beschreiben. [18, Seite 12]
Mithilfe einer *Interface Definition Language (IDL)* lässt sich dieses Ziel erreichen, denn

in einer *IDL* lassen sich diese Regeln beschreiben. Zudem empfehlen auch *Tannenbaum* und *van Steen* die Nutzung einer *IDL*. [18, Seite 12]
Aufgrund dieser Abstraktion mit einer unabhängigen *IDL*, können alle Systeme diese Definition, unabhängig von ihrer Programmiersprache, verwenden.

Eine Form der *IDL* lässt sich mit *Kotlin* umsetzen. Alle Methoden der *API* benötigen dabei unterschiedliche Parameter, die sich mithilfe von Klassen beschreiben lassen.
Als Beispiel, wenn eine Methode zur Registrierung neuer Benutzer als Parameter die Texte *Benutzernamen* und *Passwort* benötigt, dann lässt sich dies mit folgender Kotlin-Klasse beschreiben.

```
1 data class UserRegistrationParameter(
2     val username: String,
3     val password: String
4 )
```

Quellcode 3.1: Datenklasse für Userregistrierung (Kotlin).

Ein großer Vorteil an Kotlin-Klassen ist, dass diese nativ in der *Java Virtual Maschine (JVM)* nutzbar sind.
Da die *PPI AG* oftmals Java-Programme für *Backends* verwendet, wären diese Modelle nativ nutzbar. Weiterhin lassen sich Regeln definieren, die mehrere Felder betreffen, z. B. wenn als Land Deutschland ausgewählt ist, dann soll die Postleitzahl fünfstellig sein.

Kotlin-Klassen lassen sich ohne großen Aufwand nach *Typescript* übersetzen [10], jedoch ist diese Funktion noch experimentell. *Typescript* findet sich immer häufiger in *Frontends* von der *PPI AG*. Leider lassen sich mit *Kotlin* nur die Parameter von Methoden beschreiben, jedoch nicht die eigentlichen Methoden. Es handelt sich um keine *IDL*.

Es entsteht außerdem ein anderes Problem, denn Regex-Patterns zwischen Java und *Typescript* unterscheiden sich. Z. B. unterstützt *Typescript* bzw. JavaScript erst seit *ES2018* die *Lookbehind*-Funktion. Es kann somit vorkommen, dass Webbrowser diese Funktion noch nicht unterstützen.
Deshalb muss sichergestellt sein, dass sich Regex-Pattern sowohl in *Typescript* als auch in *Kotlin* identisch verhalten.

Alle Argumente lassen sich in folgender Tabelle zusammenfassen:

Tabelle 3.1: Vor- und Nachteile von Kotlin als IDL

Vorteile	Nachteile
Nutzbar in *JVM*	Keine vollwertige *IDL*
Übersetzbar nach *Typescript*	Wenig Kenntnis von *Kotlin* in der *PPI AG*
Regeln über mehrfache Felder hinweg	Keine Änderungen zur Laufzeit möglich
IDL von Kunden nutzbar	Unterscheidungen bei Regex
	Übersetzung nach *Typescript* experimentell

3.2 OpenAPI als IDL

Statt *Kotlin* als *IDL* existieren vollwertige *IDLs* wie z. B. die *OpenAPI Specification*. Im Gegensatz zu *Kotlin* wurde die *OpenAPI Specification* speziell für diesen Zweck konzipiert und ermöglicht es, dass die komplette Schnittstelle eines Systems definiert werden kann. Die *OpenAPI Specification* ist eine der bekanntesten *IDL* und bietet viele Features, die es in Kotlin nicht gibt.

Z. B. ist es möglich, aus der *IDL* direkt Code (sowohl für *Backends* als auch *Frontends*) zu generieren.

```yaml
paths:
  /pet:
    post:
      tags:
      - "pet"
      summary: "Add a new pet to the store"
      description: ""
      operationId: "addPet"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/xml"
      - "application/json"
      parameters:
      - in: "body"
        name: "body"
        description: "Pet object that needs to be added to the store"
        required: true
```

Automatische Generierung
von IDL zu Typescript

```typescript
    /**
     * Add a new pet to the store
     *
     * @param body Pet object that needs to be added to the store
     * @param observe set whether or not to return the data Observable as
     * @param reportProgress flag to report request and response progress.
     */
    public addPet(body: Pet, observe?: 'body', reportProgress?: boolean):
    public addPet(body: Pet, observe?: 'response', reportProgress?: boolea
    public addPet(body: Pet, observe?: 'events', reportProgress?: boolean)
    public addPet(body: Pet, observe: any = 'body', reportProgress: boolea

        if (body === null || body === undefined) {
```

Abbildung 3.1: Vorschau einer Übersetzung von OpenAPI Specification nach Typescript.

Außerdem bietet die *OpenAPI Specification* die Möglichkeit, direkt Regeln für die Validierung anzugeben.

```yaml
Pet:
  type: "object"
  required:
  - "species"
  properties:
    species:
      type: "string"
      example: "Dog"
      pattern: "[A-Z][a-z]+"   # <- Rule for validation
```

Quellcode 3.2: Pet-Beispiel für OpenAPI (Yaml).

Dieser Code für die *OpenAPI Specification* erstellt eine Regel, wonach die Rasse eines Tieres mit einem Großbuchstaben anfängt und ansonsten nur Kleinbuchstaben enthält. Übersetzt nach Java für *Backends*, entsteht für diese Regel folgender Code:

```java
// ...
public class Pet {
    // ...
   @ApiModelProperty(example = "Dog", required = true, value = "")
   @NotNull
   @Pattern(regexp="[A-Z][a-z]+")
   public String getSpecies() {
     return species;
   }
   // ..
 }
```

Quellcode 3.3: Pet-Klasse generiert von OpenAPI (Java).

Die *@Pattern*-Annotation wird von *Spring* automatisch erfasst und ausgewertet. Es muss kein zusätzlicher Code geschrieben werden, denn standardmäßig gibt *Spring* eine Fehlermeldung aus, wenn die Validierung verletzt wird. Ein Aufruf mit ungültigen Parametern wird von *Spring* mit dem Http-Statuscode 400 [13, Kapitel 6] beantwortet.
Die Rückmeldung lässt sich verfeinern, sodass eine Liste der fehlgeschlagen Prüfungen zurückgeben wird. Das vereinfacht die Nutzung der Schnittstelle, da Entwickler dann genau nachvollziehen können, weshalb ein 400er Statuscode entstanden ist.
Im folgendem befindet sich ein Beispiel eines ExceptionHandlers, der genau diese Aufgabe erfüllt:

```java
/**
 * This handler prevents the default-behavior of Spring.
 * Spring will return a 400 http-status-code
 * with a empty response-body per default.
 * This handler will add a detailed list of validation-
 * exceptions.
 */
@ControllerAdvice
public class ExceptionHandlerController
extends ResponseEntityExceptionHandler {

    /**
     * Catch all {@link MethodArgumentNotValidException}s
     * and add a response-body.
     */
    @Override
    @NonNull
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
        @NonNull final MethodArgumentNotValidException ex,
        @NonNull final HttpHeaders headers,
        @NonNull final HttpStatus status,
        @NonNull final WebRequest request) {

        final var body = ex.getBindingResult().getFieldErrors()
                .stream()
                .map(fieldError -> String.format("%s %s",
                    fieldError.getField(),
                    fieldError.getDefaultMessage()))
                .collect(Collectors.toList());

        return ResponseEntity.badRequest()
            .headers(new HttpHeaders())
            .body(new ValidationErrorDto(body));
    }
}
```

Quellcode 3.4: ExceptionHandler mit erweiterten Fehlermeldungen (Java).

Analog zu der simplen Umsetzung in *Spring* ist es nicht möglich, dass die Regeln für Validierungen in Klassen von *Typescript* übernommen werden. Wenn die Lösung gewählt wird, dann muss dieses Feature zusätzlich entwickelt werden. Dafür müssen die Client-Generatoren von der *OpenAPI Specification* [17] angepasst werden.

Tabelle 3.2: Vor- und Nachteile von OpenAPI als IDL

Vorteile	Nachteile
Code-Generatoren vorhanden	Generator für *Typescript* unvollständig
Einsetzbar in vielen Projekten	Keine Änderungen zur Laufzeit
IDL von Kunden nutzbar	

3.3 Endpunkt für Validierungsregeln

Anders als in den bisherigen Lösungen wäre es möglich, einen Endpunkt anzubieten, der Informationen zu den Validierungen bereitstellt. Dieser neue Endpunkt wird vom *Backend* angeboten und wäre für *Frontends* aufrufbar. Dort wird die gesamte *API* abgebildet und liefert zusätzlich zu allen Endpunkten Informationen zur Validierung.

Diese Architektur bietet den großen Vorteil, dass Änderungen der Validierungen zur Laufzeit möglich sind.
Jedoch wäre der Aufwand für eine solche Lösung enorm, denn im *Frontend* müsste ein Verzeichnis für alle Endpunkte im *Backend* vorhanden sein und es muss automatisch erkannt werden, wann ein Endpunkt aufgerufen wird, um dann eine Zwischenprüfung durchführen.
In *Angular* wäre dies mit einem *HTTP-Interceptor* möglich, wie im folgendem Codebeispiel dargestellt:

```
1  /**
2   * Short example of a http-interceptor which intercepts
3   * http-calls and validate them before continue the call.
4   * It will throws an error if the validations not passed successfully.
5   */
6  @Injectable()
7  export class ValidationInterceptor implements HttpInterceptor {
8
9      // map from url to validation-function
10     private rules: Map<string, (req: HttpRequest<any>) => boolean> = new Map();
11
12     intercept(req: HttpRequest<any>, next: HttpHandler):
13         Observable<HttpEvent<any>> {
14         this.checkValidations(req);
15         return next.handle();
16     }
17
18     /**
19      * Checking validation-rules for current http-request.
20      * Loading new validations, if are none present.
21      * @param req Intercepted Http-Request
22      */
23     checkValidations(req: HttpRequest<any>): void {
24         if (!this.rules.has(req.url)) {
25             this.rules.set(req.url, this.loadValidations(req.url));
26         }
27
28         if (!this.rules.get(req.url).apply(req)) {
29             // validation failed
30             throw new Error("validation failed");
31         }
32     }
33
34     /**
35      * Load and parse a validation-rule from backend.
36      * @param url Get rule for this url.
37      */
38     loadValidations(url:string): (req: HttpRequest<any>) => boolean {
39         // ...
40     }
41 }
```

Quellcode 3.5: HttpInterceptor für Validierungsprüfungen (Typescript).

Zu diesem Codebeispiel muss es jedoch noch ein passendes *Backend* geben, welches für jeden Endpunkt Validierungsregeln zurückgibt. In diesem Beispiel fehlt ein Ablaufmechanismus, denn aktuell sind die geladenen Validierungsprüfungen solange gültig, bis das *Frontend* beendet wird. Weiterhin wäre es noch wichtig, dass der Interceptor nur

auf HTTP-Aufrufe reagiert, die tatsächlich an das *Backend* geschickt werden. Für die Umsetzung dieser Lösung müssen diese Punkte noch beachtet werden.

Nicht alle *Frontends* basieren auf *Angular*, wodurch ein gemeinsamer Standard unwahrscheinlich wird. Für reine JavaScript-*Frontends* ist ein solcher *HTTP-Interceptor* außerdem sehr aufwändig.

Diese Methode ist kein bekannter Standard und wird in keiner Literatur erwähnt. Aus diesem Grund ist die Akzeptanz innerhalb von der *PPI AG* nicht sehr hoch.

Tabelle 3.3: Vor- und Nachteile eines Endpunkt für Validierungen

Vorteile	Nachteile
Änderungen zur Laufzeit	Zusätzliche Aufrufe nötig
Simpel mit *Angular*	Sehr aufwendig ohne *Angular*
	Geringe Akzeptanz bei der *PPI AG*
	Keine Standardlösung

3.4 Vergleich der Lösungen und Wahl eines Favoriten

Um eine Entscheidung für eine der aufgezählten Lösungen zu treffen, wurden folgende Kriterien definiert.

1. Die Implementierung der Validierung soll minimalredundant sein.
2. Änderungen der Validerungen erfordern keinen manuellen Mehraufwand beim Frontend.
3. Die Lösung ist in Java und Typescript verwendbar.
4. Es werden keine zusätzlichen *Backend*aufrufe durchgeführt.
5. Validierungen im *Frontend* sind in Echtzeit möglich.

Durch den vierten Punkt der Kriterien entfällt bereits die Lösung *Endpunkt für Validierungsregeln*, welche in Abschnitt 3.3 beschrieben wurde. Somit muss eine Entscheidung zwischen *Kotlin als IDL* und *OpenAPI als IDL* getroffen werden.

Beide Lösungen erfüllen zunächst sämtliche Kriterien. Da die *OpenAPI Specification* bereits in mehreren Projekten der *PPI AG* verwendet wird und außerdem die Tabelle *Vor- und Nachteile von OpenAPI als IDL* mehr positive als negative Punkte aufweist, fällt die Entscheidung auf *OpenAPI als IDL*.

4 Vorarbeiten für OpenAPI

4.1 Analyse der Codegeneratoren von OpenAPI

Der Codegenerator von der *OpenAPI Specification* ist in der Lage, für viele verschiedene Programmiersprachen automatisch Code zu generieren. Hierbei wird zwischen *API clients* und *Server stubs* unterschieden.

Server stubs
: *Server stubs* produzieren Code, um eine Schnittstelle anzubieten.

API clients
: *API clients* produzieren Code, um eine Schnittstelle zu konsumieren.

Um die Validierung via *OpenAPI Specification* in *Cysmo®* einzubauen, werden die Generatoren *Server stub „Spring“* und *API client „typescript-angular“* benötigt.
Der Generator *Spring* beinhaltet bereits alles Nötige, damit eine Validierung über die *OpenAPI Specification* möglich ist. So wurden die Bespiele in Abschnitt 3.2 mithilfe des *Spring*-Generators erstellt.

Anders ist es in dem Generator *typescript-angular*. Dort werden sämtliche Angaben zur Validierung komplett ignoriert. Bevor ein *PoC* gebaut werden kann, muss der Generator also um die Validierung erweitert werden.
In *Spring* werden Regeln für die Validierung mithilfe von Annotationen aus *javax.validation* angeben, die dann zur Laufzeit geprüft werden können.
Leider gibt es keine vergleichbaren, nativen Annotationen in *Typescript*, aber in dem Zusatzpaket *class-validator* [19] sind ähnliche Annotationen vorhanden.

Damit der Aufwand gering bleibt und nicht jede *Typescript*-Klasse um Methoden zur Validierung erweitert werden müssen, ist es sinnvoll, Annotationen bzw. *decorators* [3] zu verwenden. Eine endgültige Implementierung sollte aber erst vorgenommen werden, sobald die Einbindung in *Cysmo®* geprüft wurde.

4.2 Bisherige Validierung

Aktuell wird in *Cysmo®* für die Validierung *angular-form-validation* [9] verwendet. Folgende Codeschnipsel stammen aus *Cysmo®* und zeigen die derzeitige Implementierung:

```
<mat-form-field class="col-12" hideRequiredMarker>
  <input id="inputCompanyName" formControlName="companyName" matInput
         [attr.selenium-id]="'inputCompanyName'"
         placeholder="{{'company.name' | translate}}"
         type="text" required>
  <mat-error
    *ngIf="companyDataForm.get('companyName').hasError('required')">
    {{"validationMessage.companyNameRequired" | translate}}
  </mat-error>
  <mat-error
    *ngIf="companyDataForm.get('companyName').hasError('empty')">
    {{"validationMessage.companyNameRequired" | translate}}
  </mat-error>
  <!-- more fields-->

</mat-form-field>
```

Quellcode 4.1: Cysmo Frontend-Validierung (Html+ng2).

```typescript
export class CompanyDataCardComponent implements OnInit {

  // ...

  startEdit(): void {
    this.resetForm();
    this.editMode$.next(true);
  }

  resetForm() {
    this.companyDataForm = this.buildForm();
    this.editMode$.next(this.company.needsAdditionalInformation);
  }

  buildForm(): FormGroup {
    return this.fb.group({
      companyName: [
        this.company.name,
        [ Validators.required,
          Validators.maxLength(200),
          CustomValidators.notEmptyValidation
        ]],
      // more fields
    }
  }
  // ...
}
```

Quellcode 4.2: Cysmo Frontend-Validierung (Typescript).

Einfache Annotationen an Klassen von *Datentransferobjekten (DTO)* lassen sich hier nicht ohne weiteres verwenden. Denn die Validierung wird innerhalb der *Typescript*-Klasse manuell vorgegeben und somit ist ein dynamisches Laden der Validierungen nicht möglich. Mithilfe des Zusatzpaketes *ngx-dynamic-form-builder* [6] lässt sich *angular-form-validation* um diese fehlende Funktion erweitern.

Ein Einbau der Annotationen wäre somit in *Cysmo®* möglich. Mithilfe des *class-validator* lässt sich die Validierung auch aufwandsarm in anderen Projekten einbinden, selbst wenn diese kein *Angular* verwenden.

4.3 Erweiterung des *typescript-angular*-Codegenerators

Der Codegenerator von der *OpenAPI Specification* ist OpenSource und unter der *Apache License 2.0* lizenziert. Somit ist die rechtliche Grundlage gegeben, sodass der Generator im Rahmen dieser Bachelorarbeit modifiziert wird und auch später in kommerziellen Produkten der *PPI AG* verwendet werden darf. [7]

Der Quellcode des Codegenerators befindet sich auf Github [17] und wurde im Verlauf dieser Bachelorarbeit auf Github in einem *Fork* bearbeitet [1].

Sämtliche Generatoren des Codegenerators arbeiten mit der Template-Engine *Mustache* [5]. Jeder Generator hat einen eigenen Ordner im Projekt, in dem sämtliche Vorlagen gepflegt werden. Die relevanten Stellen für die Validerungen in *Spring* sind hier aufgeführt:

```
{{!
Auszüge aus dem Ordner
modules/openapi-generator/src/main/resources/JavaSpring/
}}

{{!
---------------------------
pojo.mustache
---------------------------
}}
  /**
  {{#description}}
   * {{{description}}}
  {{/description}}
  {{^description}}
   * Get {{name}}
  {{/description}}
  {{#minimum}}
   * minimum: {{minimum}}
  {{/minimum}}
  {{#maximum}}
   * maximum: {{maximum}}
  {{/maximum}}
   * @return {{name}}
  */
{{#useBeanValidation}}{{>beanValidation}}{{/useBeanValidation}}
public {{>nullableDataType}} {{getter}}() {
    return {{name}};
  }

{{!
---------------------------
beanValidation.mustache
---------------------------
}}
{{#required}}
  @NotNull
{{/required}}{{#isContainer}}{{^isPrimitiveType}}{{^isEnum}}
  @Valid{{/isEnum}}{{/isPrimitiveType}}{{/isContainer}}
  {{^isContainer}}{{^isPrimitiveType}}
  @Valid{{/isPrimitiveType}}{{/isContainer}}
{{>beanValidationCore}}

{{!
---------------------------
beanValidationCore.mustache
---------------------------
}}
{{#pattern}}{{^isByteArray}}@Pattern(regexp="{{{pattern}}}")
{{/isByteArray}}{{/pattern}}{{!
minLength && maxLength set
}}{{#minLength}}{{#maxLength}}@Size(min={{minLength}},max={{maxLength}})
{{/maxLength}}{{/minLength}}{{!
minLength set, maxLength not
}}{{#minLength}}{{^maxLength}}@Size(min={{minLength}})
{{/maxLength}}{{/minLength}}
```


Quellcode 4.3: Spring Template für Validierung in Java (Mustache)

Da die Validierung bereits in *Spring* vorhanden ist, lässt sich diese als Ausgangspunkt für den Einbau in den *Typescript-Angular*-Generator verwenden. Es müssen nur die Annotationen ausgetauscht werden. Auf Basis des Beispieles aus Quellcode 3.2 entsteht folgender *Typescript*-Code:

```
1  export class Pet {
2      @IsDefined()
3      @Matches(/[A-Z][a-z]+/)
4      species: string;
5  }
```

Quellcode 4.4: Code mit neuem Generator (Typescript).

Weiterhin war es nötig, die Typen der Klassen anzupassen. Vorher wurden hier *interfaces* statt *classes* verwendet. Innerhalb eines *interface* lassen sich jedoch keine *decorators* verwenden. Dieser Aufbau erzeugt leider keine Laufzeit- oder Kompilierfehler. Solche werden nur mit einer Warnung zur Kompilierzeit quittiert.
Dies liegt daran, dass in *Typescript interfaces* etwas anderes sind als es in der *Objektorientierten Programmierung (OOP)* üblich ist. *interfaces* werden ausschließlich zur Kompilierzeit verwendet, um die Typsicherheit zu garantieren. Jedoch gibt es in JavaScript kein passendes Equivalent, sodass *interfaces* zur Laufzeit komplett verloren gehen, da *Typescript* nach JavaScript kompiliert wird. [11]
Das Problem ist bereits bekannt und möglicherweise wird es dafür in Zukunft eine alternative Lösung geben. [4]
Da zu diesem Zeitpunkt keine Alternative in Sicht war, werden alle *interfaces* somit zu *classes* geändert. Es entstehen keine gravierenden Nachteile, wenn *class* statt *interface* verwendet wird. Jedoch entsteht beim Kompilieren der Klassen einiges an Overhead, der sich jedoch erstmal nicht vermeiden lässt.

Somit sind sämtliche Vorarbeiten abgeschlossen und die Validierung via *OpenAPI Specification* kann in Projekte, die auf Java oder *Typescript* basieren, eingebunden werden.

5 Einbau in ein bestehendes Produkt

Die erarbeitete Lösung soll als *PoC* in *Cysmo®* eingebaut werden.
Cysmo® generiert aktuell aus dem bestehendem Javacode die *OpenAPI Specification*. Diese Specification wird derzeit intern nicht verwendet, sondern nur für Kunden bereitgestellt, die die *API* nutzen wollen.

Somit ergeben sich folgende Unteraufgaben für das *Cysmo®*-Projekt:

1. Auslagerung der API.
2. CI/CD zur automatischen Veröffentlichung.
3. Partielle Umstellung im *Frontend* auf *OpenAPI Specification*.
4. Partielle Umstellung im *Backend* auf *OpenAPI Specification*.

5.1 Auslagerung der API

Der automatisch generierte Code sollte niemals manuell angepasst werden. Solche Änderungen verändern die *API* und widersprechen der vorherigen Definition. Die tatsächlich implementierte *API* und die Definition der *API* sind somit nicht mehr identisch. Um das Risiko von manuellen Anpassungen zu minimieren wird, in den einzelnen generierten Codedateien des Codegenerators darauf hingewiesen, dass solche Änderungen nicht durchgeführt werden sollten. Leider stellt dieser Hinweis nicht sicher, dass jemand Änderungen vornimmt.

Um dies zu gewährleisten, sollten die generierten Codedateien in ein eigenes Repository ausgelagert werden. Dieses Repository kann dann von den einzelnen Komponenten als Abhängigkeit definiert werden, sodass Paketmanager (z. B. *NPM* oder Maven) automatisch den generierten Code installieren können.

Dadurch ist der generierte Code vom Entwickler nicht änderbar und Diskrepanzen zwischen der Implementierung und der *API*-Definition werden verhindert.

In *Cysmo*® ist die *API* nicht als *OpenAPI Specification* definiert. Stattdessen sind die einzelnen Controller mit Annotationen versehen, um daraus die *API* zu generieren.

```
public class CompanyController {

    // some openapi-annotations
    // long strings are shorted with [...]
    @SecurityRequirement(name = "X-cysmo-auth", scopes = "write")
    @Operation(summary = "Creates a new company and triggers "
                   + "the first rating for this company",
            description = "This call creates a new company, "
                    + "generates a report about the company, "
                    + "saves it to the database and returns all combined data",
                    tags = "Companies")
    @ApiResponses({
            @ApiResponse(responseCode = "200", description = "OK",
                         content = @Content(mediaType = "application/json",
                         schema = @Schema(implementation = CompanyDto.class))),
            @ApiResponse(responseCode = "404", description = "Not found",
                         content = @Content(mediaType = "application/json"))
    })
    public CompanyDto create(
        @Parameter(
            description = "CompanyDto with all information about the [...]",
            required = true)
        @RequestBody
        final CompanyDto company,

        @Parameter(
            description = "Indicates if the sub domain warning [...]")
        @RequestParam(required = false)
        final boolean acceptSubDomain,

        // ...
        ) {
            // ...
        }
    // ...
}
```

Quellcode 5.1: Annotationen für OpenAPI in *Cysmo*® (Java).

Dieser Zustand soll bei *Cysmo*® nur eine Zwischenlösung sein. Denn die Annotationen blähen die Controller-Klassen unnötig auf. Aufgrund dieser Annotationen ist die Klasse

CompanyController 1.700 Zeilen lang. Dadurch ist die Klasse nur schwer wartbar. Eine vollständige Umstellung auf *OpenAPI Specification* würde dieses Problem lösen.

Zum Extrahieren der *OpenAPI Specification* aus den Annotationen musste lediglich das *Backend* gestartet werden. Über einen definierten Endpunkt war dann die *API* als .json-Datei verfügbar. Bis auf einen Syntaxfehler, der manuell rasch gelöst werden konnte, war die gesamte *API* sofort verwendbar. Die Validierung ist aber noch nicht vorhanden und muss noch eingepflegt werden.

Durch die Auslagerung entsteht ein weiterer Vorteil. Mittels *CI/CD* lassen sich Änderungen an der *OpenAPI Specification* automatisiert in den Komponenten testen. So kann verhindert werden, dass z. B. ein Feld entfernt wird, welches jedoch noch verwendet wird. Theoretisch könnten Änderungen an der Validierung vollautomatisch in die Projekte gebracht werden, ohne dass eine manuelle Änderung erforderlich ist. Damit lässt sich der zweite Punkt der Kriterien in Abschnitt 3.4 lösen. Eine nichtrepräsentative Umfrage unter den Entwicklern der *PPI AG* hat ergeben, dass diese automatische Änderung nicht erwünscht ist. Alternativ wäre es möglich, dass Änderungen an der *API* neue *Merge Requests* bei den entsprechenden Projekten erstellen.

Die Auslagerung eröffnet einige Möglichkeiten, die individuell und nach Bedarf in den Projekten umgesetzt werden können. Im nächsten Abschnitt wird eine der grundlegenden Möglichkeiten implementiert.

5.2 CI/CD zur automatischen Veröffentlichung

Durch die Auslagerung in Abschnitt 5.1 können Änderungen an der *API* an verschiedene *CI/CD*-Prozesse gekoppelt werden. Im folgendem wird eine automatische *Gitlab*-Pipeline [8] aufgebaut, die die *API*-Definitionen kompiliert und in verschiedenen Repositories veröffentlicht.

Dazu muss der Codegenerator in ein Dockerimage gebracht werden, da der Generator erst dann in der *Gitlab*-Pipeline verwendet werden kann. Von den Machern der *OpenAPI Specification* existiert bereits ein Image [15], welches den Codegenerator enthält. Die Änderungen aus Kapitel 4 sind jedoch in dem Dockerimage noch nicht vorhanden. Eben jene Änderungen lassen sich aber problemlos in ein neues Dockerimage bringen, wie im folgendem Dockerfile aufgezeigt.

```
1  # use the exiting dockerimage of openapi-generator
2  FROM openapitools/openapi-generator-cli:v5.0.0
3
4  # path as variable for improved readability
5  ENV JAR_PATH /opt/openapi-generator/modules/\
6  openapi-generator-cli/target/openapi-generator-cli.jar
7
8  # override old jar with modified one
9  COPY openapi-generator-cli.jar ${JAR_PATH}
10
11 # register an alias, so it is possible to use just "open-api-gen"
12 # as command in the image
13 RUN echo -e '#!/bin/bash\njava -jar ${JAR_PATH} "$@"' > /usr/bin/open-api-gen && \
14     chmod +x /usr/bin/open-api-gen
```

Quellcode 5.2: Neues Dockerfile für OpenAPI Generator (Dockerfile).

Das neue Dockerimage lässt sich in der *Gitlab*-Pipeline verwenden und ermöglicht es, dass relativ einfach der Code generiert werden kann. Mit gängigen Mitteln wie *NPM* und Maven ist der komplette Prozess trivial wie in folgender *Gitlab*-Pipeline-Konfiguration dargestellt:

```yaml
# this job generate java-code for given open-api-spec
# it will be cached in "output"-folder, so other jobs
# can access it
build-master-java-stub:
  only:
    - master
  stage: build
  # use new own image of openapi-generator-cli
  image: ppi/x/baseimages/openapi-generator-cli:v5.0.1
  script:
    # delete possible old outputs, but ensure that the folder exists
    - rm -rf output
    - mkdir output
    # create java-code via openAPI
    - open-api-gen generate
      -i src/api-docs.json
      -g spring
      -c src/java-spring-boot/config-file.yml
      -o output
  # save the results
  cache:
    key: "${CI_BUILD_REF_NAME}-spring"
    paths:
      - "output/**"

# build the output of previous job to .jar-files
# and publish them to ppi-maven
push-master-java-stub:
  only:
    - master
  stage: push
  # use standard gradle-image
  image: gradle:jdk11
  # fetch secrets first to access ppi-maven
  needs: ['fetch-artifactory-secret']
  before_script:
    # set this to prevent spawning a daemon, otherwise gradle will start the daemon
    - export GRADLE_OPTS="-Xmx2g -XX:MaxMetaspaceSize=512m -Dorg.gradle.daemon=false
      -XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8"
  script:
    - cd output
    # copy gradle-settings into generated code
    - cp ../src/java-spring-boot/*.gradle .
    # publish to ppi-maven-repository
    - gradle publish
  # load the results from build-master-java-stub
  cache:
    key: "${CI_BUILD_REF_NAME}-spring"
    paths:
      - "output/**"
```

Quellcode 5.3: Gitlab-CI-Konfiguration, um OpenAPI nach Java zu übesetzen (Yaml).

Der komplette Vorgang lässt sich noch verbessern, indem Git-Tags verwendet werden. Durch die Tags lässt sich eine Version an einen bestimmten Stand des Codes koppeln. Bei der Veröffentlichung kann dann diese Version angegeben werden.
Somit ist sichergestellt, dass Projekte Versionen verwenden mit denen sie kompatibel sind. Aufgrund der Komplexität wird diese Möglichkeit im Rahmen dieser Bachelorarbeit nicht durchgeführt.

Andere Projekte können nun die veröffentlichten Stände inkludieren und mit weiteren Pipelines ausbauen. Z. B. könnten neue *API*-Versionen, die keine Breaking-Changes enthalten, automatisch in den benutzten Projekten getestet werden. Oder die Entwickler der anderen Projekten könnten per E-Mail darüber informiert werden, dass es eine neue Version der *API* gibt. Oder in den entsprechenden Projekten werden Issues oder *Merge Requests* erstellt. Es ergeben sich eine Menge an Möglichkeiten, die einiges an Mehrwert bieten können.

5.3 Partielle Umstellung im *Frontend* auf *OpenAPI Specification*

Im ersten Schritt wird ein generiertes *Typescript*-Modell ins *Frontend* integriert. Danach wird die Validierung so verändert, dass die Informationen des Modells verwendet werden.

Diese Arbeit konzentriert sich auf das Unternehmensobjekt von *Cysmo*®. Dabei handelt es sich um ein *DTO*, das alle relevanten, zu speichernden Daten eines von *Cysmo*® bewerteten Unternehmens zusammenfasst. Dazu gehören z. B. der Name des Unternehmens, die Postleitzahl, das Land und die Internetadresse, die alle eine verschiedende Form der Validierung erfordern. Diese Daten können über eine interaktive, grafische Oberfläche eingeben werden. Ein Teil der Eingabemaske ist in der *Einleitung* zu finden.
Die grafische Oberfläche soll im Zuge dieser Arbeit – trotz des Einbaues einer neuen Validierungslogik – gleich bleiben und für den Endanwender soll es keinen sichtbaren Unterschied geben.

Cysmo® besitzt bereits eine vollständige *API*-Definition, wie in Abschnitt 5.1 beschrieben. Jedoch enthält diese Definition noch keine Angaben zur Validierung. Leider sind die Regeln für die Validierung noch verteilt im *Frontend* und *Backend*. Im ersten Schritt

werden alle syntaktischen Validerungen für Unternehmensdaten in *OpenAPI Specification* gebündelt. Dadurch entsteht folgendes Modell (reduziert auf Name und Postleitzahl):

```json
{
  "Company": {
    "required": ["domains"],
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "Name of the company",
        "nullable": true,
        "example": "PPI AG",
        "maxLength": 200
      },
      "domains": {
        "type": "array",
        "nullable": false,
        "minItems": 1,
        "maxItems": 5000,
        "items": {
          "type": "string",
          "description": "List of domains",
          "example": "ppi.de"
        }
      },
      "zipCode": {
        "type": "string",
        "description": "Zip code of the company's head office",
        "nullable": true,
        "example": "22305",
        "pattern": "^[0-9]{5}$"
      }
    }
  }
}
```

Quellcode 5.4: Unternehmen für OpenAPI (Json).

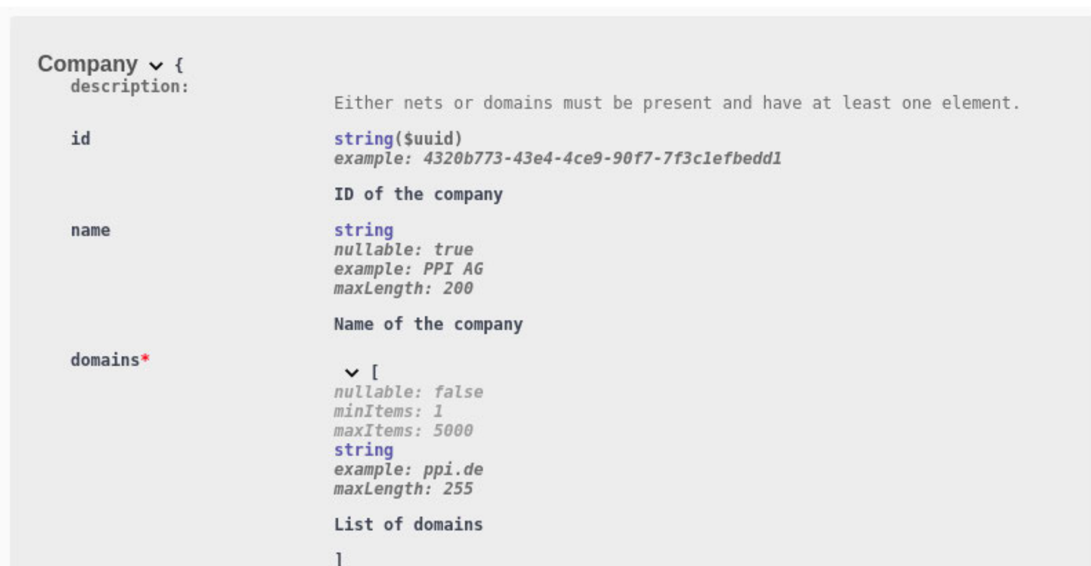

Abbildung 5.1: Screenshot aus IntelliJ mit dem neuem Unternehmensobjekt.

Die Informationen für die Felder *pattern*, *maxLength*, *minItems* und *maxItems* wurden aus dem Quellcode von *Cysmo*® extrahiert.

Mit der neuen Version des Codegenerators aus Kapitel 4 entsteht für das Unternehmen folgendes Objekt:

```typescript
/**
 * NOTE: This class is auto generated by OpenAPI Generator.
 * https://openapi-generator.tech
 * Do not edit the class manually.
 */
import { Comment } from './comment';
import { CompanyData } from './companyData';
import { BasicReport } from './basicReport';
import { IsDefined, Matches, Length, MinLength, MaxLength,
         ArrayMinSize, ArrayMaxSize, Min, Max } from 'class-validator';


export class Company {

    /**
     * Name of the company
     */
    @MaxLength(200)
    name?: string;

    @IsDefined()
    @ArrayMinSize(1)
    @ArrayMaxSize(5000)
    domains: Array<string>;

    /**
     * Zip code of the company\'s head office
     */
    @Matches(/^[0-9]{5}$/)
    zipCode?: string | null;


    // ...
}
```

Quellcode 5.5: Generiertes Modell für Unternehmen (Typescript).

Übersetzt ins Deutsche sagt das Modell aus, dass zwingend mindestens eine Domain angegeben werden muss, aber maximal 5.000 Domains angegeben werden dürfen. Die Postleitzahl sowie der Name sind optional, aber unterliegen Regeln, sofern diese Datensätze angeben werden. In diesem Beispiel müssen Postleitzahlen fünfstellig sein und somit dem deutschem Schema entsprechen.

Anstatt das bisherige Unternehmensobjekt mit dem neuem generiertem Objekt zu ersetzen, wird das neue Objekt parallel eingeführt. Es entstehen somit zwei unabhängige Unternehmensobjekte. Dieses Vorgehen verhindert, dass das *Frontend* sofort komplett

umgestellt werden muss. Der Umzug zu *OpenAPI Specification* kann somit schrittweise umgesetzt werden.

Da in dem *Frontend* von *Cysmo®* bisher das *NPM*-Paket *ngx-reactive-form-class-validator* nicht verwendet wurde, war es notwendig diese Abhängigkeit hinzufügen. Denn dieses Paket wird benötigt, damit eine Validierung in Echtzeit möglich ist. Dieses Paket ermöglicht, dass die Validierungsregeln aus den *class decorators* verwendet werden, anstatt diese statisch anzugeben. Die restlichen Änderungen waren geringfügig, wie an folgender Abbildung zu erkennen ist:

Abbildung 5.2: Screenshot aus IntelliJ mit Git-Diff von alter zu neuer Validierung.

Das *Frontend* ließ sich problemlos starten, nachdem diese Änderungen umgesetzt worden waren. Ein kurzer Test der Oberfläche hat gezeigt, dass die Funktionalität identisch zur ursprünglichen Version ist. Ein Unterschied war nicht feststellbar.
Somit war der *PoC* für das *Frontend* erfolgreich und eine komplette Umstellung ist in Zukunft möglich.

5.4 Partielle Umstellung im *Backend* auf *OpenAPI Specification*

Damit der *PoC* komplett ist, muss auch das *Backend* die Validierung von der *OpenAPI Specification* übernehmen. Wie auch vorher im *Frontend*, wird eine partielle Umstellung durchgeführt. Somit kann auch im *Backend* eine schritteweise Umstellung durchgeführt werden. Wieder liegt der Fokus auf dem Unternehmensobjekt.

Es wird eine Abhängigkeit in Gradle definiert, sodass das Artefakt aus Abschnitt 5.2 verwendet werden kann. Die Angabe der Abhängigkeit ist mittels Gradle sehr einfach, wie im folgendem dargestellt:

```groovy
// ...

dependencies {
    // ...

    // openapi-specification of the api as java-dep
    compile 'de.ppi.cysmo:core-api:2.0'

    // ...
}

// ...
```

Quellcode 5.6: Neue Abhängigkeiten in Cysmo für OpenAPI (Groovy).

Nun ist die komplette *OpenAPI Specification* in Form von Java Interfaces und Objekten verfügbar. Für jeden Controller gibt es nun ein Interface, welches einfach implementiert werden kann. Die Komplexität der bisherigen Controller nimmt dadurch stark ab, da die Annotationen für *Spring* nicht mehr den Code für die Controller aufblähen. Eine vollständige Umstellung wäre somit sehr vorteilhaft.
Die *DTOs* liegen nun auch als Javaobjekte vor und lassen sich bereits in den alten Controllern verwenden. Im folgendem Quellcode wurde das alte Unternehmensobjekt mit dem von der *OpenAPI Specification* ausgetauscht:

```java
/**
 * Old controller method which takes requests
 * to create new companies
 * (reduced to relevant fields)
 */
@PostMapping
public CompanyDto create(
    @Parameter(description = "CompanyDto with all information "
     + "about the company to create", required = true)
    @Valid // this annotation tells spring to validate the object
    @RequestBody // POST-body contains data
    final de.ppi.cysmo.coreApi.model.Company company) {

    // unchanged body
}
```

Quellcode 5.7: Neuer Methodenkopf für den Cysmo Controller mit OpenAPI (Java).

Durch diese Umstellung funktioniert die Validierung bereits. Jedoch fehlt noch eine Erweiterung des Exception-Handler-Controllers, denn fehlerhafte Validierungen erzeugen

eine leere Rückmeldung. Lediglich ein 400er Statuscode wird zurückgeben. Bei großen Objekten ist es somit quasi unmöglich herauszufinden, welches Feld die Validierung verletzt hat. Mit folgendem Screenshot soll das Problem verdeutlicht werden:

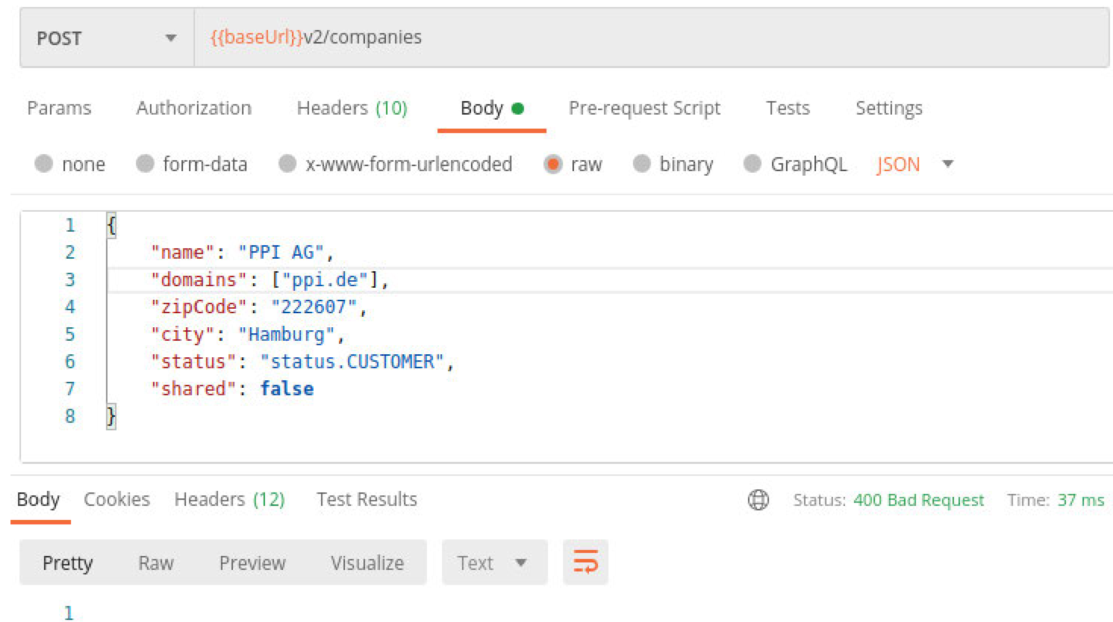

Abbildung 5.3: Screenshot von Postman mit leerer Rückmeldung.

Deswegen wird der bereits angesprochene Controller aus Abschnitt 3.2 eingebaut. Dadurch ändert sich die Rückmeldung, was die Arbeit mit der *API* wesentlich verbessert. Im nächsten Screenshot ist derselbe Aufruf zu sehen, nur dass diesmal eine lesbare Rückmeldung vorhanden ist.

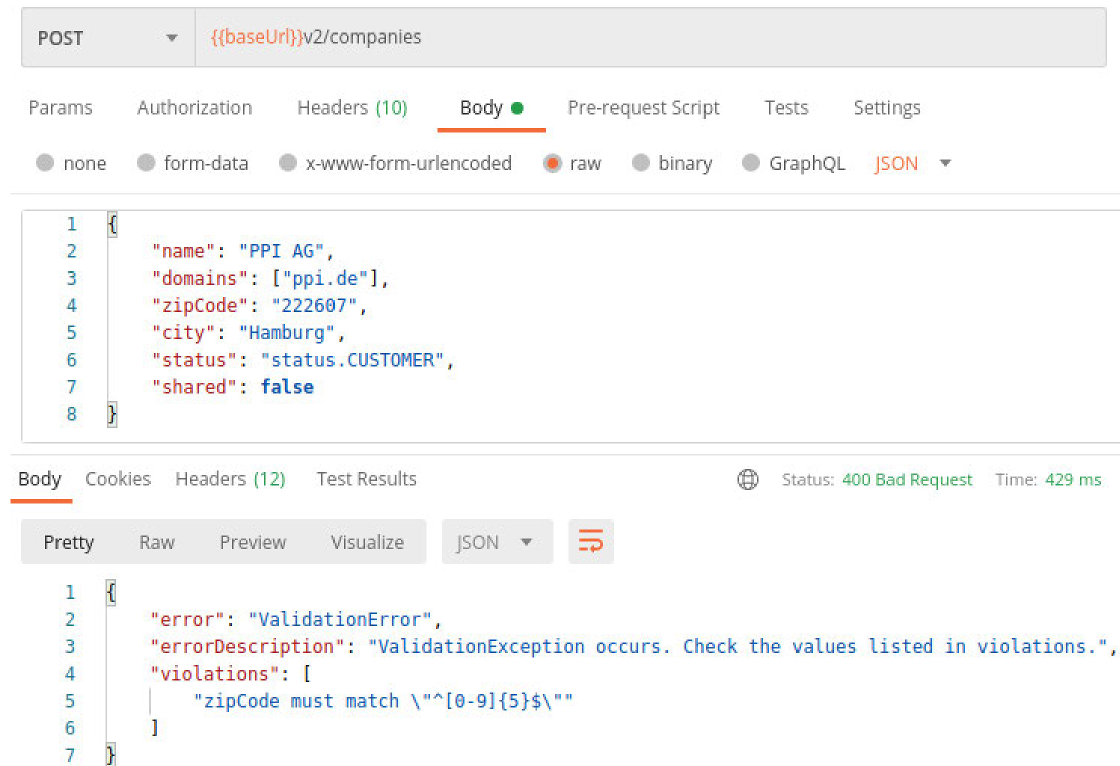

Abbildung 5.4: Screenshot von Postman mit nicht leerer Rückmeldung.

Das *Cysmo®-Backend* nutzt nun auch die Validierung von der *OpenAPI Specification* für das Unternehmensobjekt. Ein erster Test zeigt: Die Validierung funktioniert. Der komplette *PoC* ist somit erfolgreich beendet.

6 Analyse des Einbaus

Der Einbau war erfolgreich, auch wenn eine Prüfung angepasst werden musste.
In *Cysmo®* wurden die Daten nicht nur rein syntaktisch geprüft, sondern es wurden Felder miteinander kombiniert. Die Postleitzahl wurde an das Land geknüpft: Wenn das Unternehmen in Deutschland liegt, dann darf die Postleitzahl nur aus Zahlen bestehen und muss genau fünfstellig sein.

Diese Prüfung wurde ausschließlich im *Frontend* durchgeführt und kann dort auch bestehen bleiben. Zwar sollte das *Backend* diese Prüfung als fachliche Prüfung ebenfalls mit aufnehmen, was jedoch wenig mit einer syntaktischen Validierung zu tun hat und nicht mit *OpenAPI Specification* abbildbar ist.

Außerdem musste die bisherige *API* leicht angepasst werden. Es war nötig, dass entweder eine Domain, eine IP-Adresse oder beides zu einem Unternehmen angegeben wurde. Solche *Cross Field Validations* sind zwar mittels *OpenAPI Specification* abbildbar [16], aber nicht in Java und Typescript. Eine bessere Alternative wäre es, wenn Abhängigkeiten mehrerer Felder vermieden werden könnten, indem z. B. verschiedene Pfade verwendet werden. In diesem Fall würden dann drei unabhängige Endpunkte entstehen, die entweder Domains, IP-Adressen oder beides akzeptieren. Alternativ wäre es auch möglich, dass diese Felder zu einem einzigen Feld zusammengefasst werden.
Im Rahmen dieses *PoC* wurde das Feld der IP-Adresse deaktiviert, sodass nicht mehr Unternehmen mit IP-Adressen gespeichert werden können.

Identifizierung von Problemen

Obwohl der *PoC* einige Vorteile mit sich bringt, birgt die *OpenAPI Specification* auch diverse Nachteile:

1. **Lange und komplexe OpenAPI-Definition**
   Die komplette Definition in OpenAPI (exkl. der Validierungsregeln) ist für *Cysmo®*

beinahe 13.000 Zeilen lang. Das Problem wird zwar dadurch gemindert, dass es gute Editoren gibt (welche in modernen *Integrierten Entwicklungsumgebungen (IDEs)* nutzbar sind), die die gesamte *API* grafisch aufbereitet anzeigen. Alternativ wäre es möglich, dass die Definition komplett in Java beschrieben wird. Jedoch nimmt es den Entwicklern die Freiheit, denn sie sind dann darauf angewiesen, dass die Übersetzung von Java nach *OpenAPI Specification* und wieder zurück zu Java gut funktioniert.

2. **„Vendor-Lock-in"**
   Wenn alle Projekte intensiv mit der *OpenAPI Specification* arbeiten, dann entsteht ein sog. „Vendor-Lock-in". Die *PPI AG* ist stark an die *OpenAPI Specification* gebunden und ein Wechsel zu einer anderen Lösung für alle Projekte ist meist nicht wirtschaftlich. Das Problem wird jedoch dadurch entschärft, dass sowohl die Spezifizierungssprache als auch der Codegenerator OpenSource sind und theoretisch von Entwicklern der *PPI AG* angepasst werden könnten.

3. **Hohe *Time-to-Value (TTV)***
   Anpassungen an der *API* erfordern, dass zuerst die Auslagerung (beschrieben in Abschnitt 5.1) der *API* geändert werden muss. Diese Änderung muss zuerst sämtliche Qualitätssicherungsprozesse durchlaufen, wozu auch der *Merge Request* zählt. Dieser Prozess sichert zwar die Qualität, aber er kostet jedoch Zeit. Erst wenn dieser Prozess komplett abgeschlossen ist, kann mit der eigentlichen Entwicklung an den anderen Komponenten begonnen werden. Dieser Overhead ist für kleinere Projekte (mit einem oder zwei Entwicklern) möglicherweise schädlich, aber für größere Projekte ist diese Qualitätssicherung wichtig.

4. **Abhängigkeit zu Codegenerator**
   Es ist wichtig, dass der Codegenerator regelmäßig Aktualisierungen erhält. Denn mit jeder neuen Version von *Spring* muss potentiell der Codegenerator angepasst werden. Unter Umständen wird dadurch wieder die *TTV* für neue Versionen von *Spring* länger. Allgemein ist der Generator von der *OpenAPI Specification* mit knapp 8.000 Sternen auf Github recht populär [17], wodurch Updates von *Spring* wahrscheinlich schnell im Codegenerator verfügbar sind. Ein Restrisiko für dieses Problem bleibt dennoch bestehen.

In Abschnitt 3.4 wurden fünf Kriterien definiert, die von in dieser Bachelorarbeit erarbeiteten Lösung erfüllt werden müssen. Die Erfüllung dieser Kriterien wird nachfolgend einzeln bewertet.

Analyse der Kriterien

1. Die Validierung sollte minimalredundant implementiert sein.
Die Validierung wird nun einmalig und zentral definiert. Dadurch ist sichergestellt, dass Validierungsprüfungen minimalredundant implementiert sind.
Manuelle Änderungen an der *API* innerhalb der Komponenten sind nicht mehr möglich.

2. Änderungen der Validerungen erfordern keinen manuellen Mehraufwand beim *Frontend*.
Es besteht zwar noch ein manueller Mehraufwand beim *Frontend*, dieser wurde jedoch im Gegensatz zur vorherigen Version erheblich reduziert. Es muss nur noch eine Versionsnummer angepasst werden, anstatt der gesamten *API*-Implementierung.

Außerdem wurde durch den Aufbau einer *CI/CD*-Strecke der Grundstein gelegt, damit dieses Kriterium vollständig erfüllt werden kann. Es wäre möglich, dass ein Patch- oder Minor-Update der *API* vollautomatisch einen Neubau des *Backends* und *Frontends* auslöst. Für Major-Updates (die i.d.R. *Breaking Changes* enthalten) kann dieses vollautomatische System sicherstellen, dass die Anwendung keine veralteten Ressourcen nutzt. Das Entfernen von Komponenten, die im *Backend* oder *Frontend* noch verwendet werden, führen zu Fehlern bei der Kompilierung. Diese Fehler können im *CI/CD*-Prozess automatisch detektiert und entsprechend behandelt werden.

> **Beispiel**
> Wenn ein Breaking-Change entsteht (z. B. wenn der Pfad einer Ressource angepasst wird), dann erfolgt dies in zwei Schritten.
> Zuerst wird eine neue Minor-Version der *API* veröffentlicht, die den alten und neuen Pfad enthält. Diese neue Version wird in die verschiedenen Systeme eingebaut und die alte Variante wird nicht mehr aufgerufen.
> In einem zweitem Schritt kann in einem Major-Release die alte Variante komplett entfernt werden. [2]
> Durch dieses Vorgehen wird das Fehlerpotenzial reduziert.

3. Die Lösung ist in Java und Typescript verwendbar.
Durch die Vorarbeiten (beschrieben in Kapitel 4) ist die Validierung in beiden Sprachen verfügbar.

4. Es werden keine zusätzlichen *Backend*-Aufrufe durchgeführt.
Da die *API*-Definitionen zentral definiert sind, sind keine zusätzlichen Aufrufe nötig.

Sämtliche Systeme kennen die komplette *API*-Definition zur Kompilierzeit. Dadurch erfordern Änderungen eine erneute Auslieferung sämtlicher Komponenten.

5. Validierungen im *Frontend* sind in Echtzeit möglich.
Wie in Abschnitt 5.3 beschrieben, kann das *Frontend* anhand der *API*-Definition eine Validierung in Echtzeit durchführen. Während des Tippens können Endanwender somit auf eine Verletzung der Validierung hingewiesen werden.

Ableitung

Fast alle Kriterien wurden erfüllt. Eine Ausnahme bildet nur Kriterium zwei. Aufgrund des hohen Aufwands wurden hier nur mögliche Wege beschrieben, wie dieses Ziel zu erreichen ist. Außerdem wurde das Problem weiter geschwächt, indem der Mehraufwand drastisch reduziert wurde.

7 Zusammenfassung

Der Einbau eines *PoC* in *Cysmo*® war erfolgreich.
Als Lösungsweg wurde *OpenAPI als IDL* (s. Abschnitt 3.2) verwendet. Die Lösung erfüllt fast alle geforderten Kriterien und bietet somit eine gute Möglichkeit, um das Problem der redundanten Implementierung von Validierungen Herr zu werden.

Im Rahmen dieser Bachelorarbeit sind einige Vorteile bemerkt wurden, die erstmal nichts mit dem Thema der Bachelorarbeit zu tun haben, aber trotzdem eine wesentliche Qualitätssteigerung für die *PPI AG* bedeuten. Unter anderem betrifft das die Auslagerung der *API* (beschrieben in Abschnitt 5.1). Damit können die Komponenten, die die *API* nutzen, nicht mehr die *API* modifizieren. Dadurch werden Entwickler gezwungen sich an den Vertrag der *API* zu halten. Außerdem werden dadurch viele weitere Möglichkeiten eröffnet, sodass Projekte in der Lage sind, die *API* mit weiteren *CI/CD*-Prozessen zu kombinieren.

Weiterhin wurde festgestellt, dass eine vollständige Umstellung auf *OpenAPI Specification* nicht nur die Validierung vereinheitlicht, sondern auch in *Cysmo*® die Komplexität der Controller wesentlich reduzieren würde, da störende Annotationen komplett wegfallen. Außerdem werden manuell implementierte syntaktische Validerungen obsolet, wodurch die Controller auf das Wesentliche reduziert werden können.

Der Aufbau aus einem *Angular-Frontend* und einem *Spring-Backend* ist bei der *PPI AG* zum favorisiertem Standard geworden und einige andere Projekte und Produkte verwenden diesen bereits.

Aus Sicht des Autors ergeben sich durch die erfolgreiche Umsetzung einer minimalredundanten Validierung und der daraus resultierenden Erkenntnisse einige Vorteile für die *PPI AG*, die einen deutlichen Mehrwert liefern und die Qualität der Projekte und Produkte nachhaltig verbessern können. Es existieren zwar einige Nachteile (siehe Abschnitt 6). Jedoch werden die Nachteile von den Vorteilen deutlich überwogen.

Ausblick

Aufgrund der Ergebnisse dieser Bachelorarbeit sollte die *PPI AG* die minimalredundante Lösung, mithilfe der *OpenAPI Specification*, weiter vorantreiben. Dazu kann in *Cysmo®* ein vollständiger Umzug zur *OpenAPI Specification* durchgeführt werden. Dadurch gewinnt das Produkt einiges an Qualität, da dann die Validierung minimalredundant ist und somit potentielle Fehler verhindert werden. Außerdem wäre dann der Vertrag der Schnittstelle klarer definiert und in einem eigenem Repository als *Single Point of Truth* verfügbar. Basierend auf der Auslagerung sind verschiedene weitere Ausbaustufen des *CI/CD*-Systems möglich, wie es in Abschnitt 5.2 beschrieben wurde. Auch andere Schnittstellen, z. B. zwischen dem *Cysmo-Core* und der *Rating-Engine*, könnten als *OpenAPI Specification* definiert werden.

Durch die vollständige Umstellung auf die *OpenAPI Specification* gewinnen die Entwickler von *Cysmo®* einiges an Know-How. Dieses Wissen kann dann eingesetzt werden, damit andere Projekte ebenfalls den Umbau durchführen können. Auf diese Art und Weise können viele Projekte von der *OpenAPI Specification* profitieren und langfristig die Qualität der Produkte der *PPI AG* steigern.

Wenn viele Projekte die *OpenAPI Specification* einsetzen, dann steigt die Wahrscheinlichkeit, dass in der *OpenAPI Specification* Features fehlen. Sollte dieser Fall eintreten, dann könnten Entwickler der *PPI AG* die fehlenden Features nachpflegen und somit die weltweite OpenSource-Community fördern. Das könnte sich wiederrum positiv für die *PPI AG* auswirken, da dadurch die Reputation von der Firma steigen kann.

Danksagung

Mit dieser Seite möchte ich mich bei allen Personen bedanken, die mich bei dieser Bachelorarbeit unterstützt haben. Dies betrifft meine Arbeitskollegen und den Prof. Dr. Schmolitzky. Ganz besonders gilt meinem Dank Michael, der viele wertvolle Anmerkungen beim Korrekturlesen geben konnte.
Bei der *PPI AG* möchte ich mich dafür bedanken, dass sie diese Bachelorarbeit überhaupt möglich gemacht hat.

Literaturverzeichnis

[1] Andre Köpke: *Fork von openapi-generator*. 2021. – URL https://github.com/OpenAPITools/openapi-generator/compare/master...AndreKoepke:typescript-validation. – Geforkt am 12.09.2020

[2] Baeldung: Versioning a REST API. (2020). – URL https://www.baeldung.com/rest-versioning. – Eingesehen am 05.02.2021

[3] Barzilay, E. et al: Decorators. (2021). – URL https://www.typescriptlang.org/docs/handbook/decorators.html. – Eingesehen am 20.02.2021

[4] Cavanaugh, Ryan: Regex-validated string types (feedback reset). (2020). – URL https://github.com/microsoft/TypeScript/issues/41160. – Eingesehen am 21.02.2021

[5] Chris Wanstrath: *Mustache Manual*. 2014. – URL https://mustache.github.io/mustache.5.html. – Eingesehen am 18.01.2021

[6] Endy Kaufmann: *ngx-dynamic-form-builder auf Github*. 2020. – URL https://github.com/EndyKaufman/ngx-dynamic-form-builder. – Eingesehen am 05.01.2021

[7] Fondation, Apache S.: Apache License, Version 2.0. (2004). – URL https://www.apache.org/licenses/LICENSE-2.0

[8] GitLab Inc.: *Gitlab Pipelines*. – URL https://docs.gitlab.com/ee/ci/pipelines/. – Eingesehen am 01.02.2021

[9] Google: *Validating input in reactive forms*. 2021. – URL https://angular.io/guide/form-validation#validating-input-in-reactive-forms. – Eingesehen am 04.01.2021

[10] JetBrains: Using the Kotlin/JS IR compiler. (2020). – URL https://kotlinlang.org/docs/reference/js-ir-compiler.html#preview-generation-of-typescript-declaration-files-dts. – Eingesehen am 18.12.2020

[11] Parsy, Valentin: Typescript : class vs interface. (2018). – URL https://medium.com/front-end-weekly/typescript-class-vs-interface-99c0ae1c2136. – Eingesehen am 20.02.2021

[12] PPI AG: *Website von Cysmo®*. – URL https://www.cysmo.de/. – Eingesehen am 01.02.2021

[13] R. Fielding, Julian F. R.: RFC 7231, HTTP/1.1 Semantics and Content. (2014). – URL https://tools.ietf.org/pdf/rfc7231.txt

[14] Rumbaugh, James E. ; Jacobson, Ivar ; Booch, Grady: *The Unified Modelling Language Reference Manual*. Addison-Wesley-Longman, 1999. – URL https://www.researchgate.net/publication/220694093_The_Unified_Modelling_Language_Reference_Manual. – ISBN 978-0-201-30998-0

[15] SmartBear Software: *Docker-Image für OpenAPI*. – URL https://github.com/OpenAPITools/openapi-generator#16---docker. – Eingesehen am 01.02.2021

[16] SmartBear Software: oneOf, anyOf, allOf, not. . – URL https://swagger.io/docs/specification/data-models/oneof-anyof-allof-not/. – Eingesehen am 15.01.2021

[17] SmartBear Software: *openapi-generator auf Github*. 2020. – URL https://github.com/OpenAPITools/openapi-generator. – Eingesehen am 21.12.2020

[18] Steen, Maarten van ; Tanenbaum, A. S.: *Distributed Systems*. Maarten van Steen, 2018. – URL https://www.distributed-systems.net/index.php/books/ds3/. – ISBN 978-90-815406-2-9

[19] Unbekannt: *class-validator für Typescript auf Github*. 2020. – URL https://github.com/typestack/class-validator. – Eingesehen am 22.12.2020

Glossar

Angular Ein Frontend-Framework in JavaScript/Typescript, welches von der *PPI AG* in modernen Projekten eingesetzt wird.

Apache License 2.0 Es handelt sich um eine anerkannte *Freie Software Lizenz*, die es erlaubt, die lizenzierte Software zu verändern, verwenden und zu verteilen.

Application Programming Interface Eine API ist eine definierte Schnittstelle, die von Programmen benutzt werden kann. Verteile Systeme (insbesondere Webseiten) benutzen oft *REST*-Schnittstellen.

Backend Ein Backend ist nicht sichtbar für den Endnutzer und bietet *APIs* für *Frontends* an. Backendanwendungen laufen auf Servern und werden beispielsweise in PHP, Java, C# oder C++ programmiert.

Continuous Integration / Continuous Development Hierbei handelt es sich um Grundsätze. CI sagt aus, dass Code noch während der Entwicklung automatisch integriert werden soll (u.a. mit sog. Pipelines, die den Code testen). CD hingegen sagt aus, dass Umgebungen automatisch ausgerollt werden sollen (z. B. bei einem Commit in den master-Branch wird automatisch ein neues Release für die produktive Umgebung veröffentlicht).

Cysmo® Das Tool Cysmo® ist ein Produkt der *PPI AG* zur automatisierten Risikobewertung von Webseiten.

Datentransferobjekt Ein Datentransferobjekt stellt i.d.R. eine Klasse dar, die mehrere Informationen bündelt.

Fat Application Eine Fat Application beschreibt eine Software, welche völlig unabhängig von einem Zentralserver agieren kann. Die Anwendung enthält alle nötigen Softwarekomponenten.

Fork Bei einem Fork handelt es sich um eine Kopie eines Github-Repositories, welche unter einem anderen Namen weitergeführt wird. Mit einem *Pull Request* ist es möglich, dass Änderungen eines Fork zurück ins ursprüngliche Repository gebracht werden.

Frontend Ein Frontend ist der sichtbare Teil für Endnutzer. Alle Webseiten, Apps und die meisten Programmen fallen in diese Kategorie. Frontends werden auf Endgeräten ausgeführt. Webseiten werden mit HTML, CSS und Javascript programmiert.

Gitlab Gitlab ist eine bekannte Open Source Software die Quellcodeverwaltung via Git ermöglicht. Mithilfe einer Webseite lassen sich Änderungen und ganze Prozesse visualisieren und optimieren.

HTTP-Interceptor Ein HTTP-Interceptor unterbricht einen HTTP-Request und kann den ursprünglichen Aufruf bearbeiten. Dies wird genutzt um z. B. Authentifzierungsmerkmale zu einem Aufruf hinzufügen, ohne das die Entwickler dies im eigentlichen Aufruf bedenken müssen.

Interface Definition Language Mithilfe einer IDL lässt sich ein Interface beschreiben. Es beinhaltet Informationen über angebotende Methoden und deren Parameter. Je nach Sprache sind auch weiterführende Informationen (wie z. B. Regeln für Validierung oder Beispieldaten) möglich.

Java Virtual Maschine Programme, die in Java implementiert sind, kompilieren zu einem Bytecode. Anders als andere Programme ist dieser Bytecode unabhängig vom Betriebssystem. Damit Programme in Java funktionieren, laufen diese einer virtuellen Maschine, der sog. JVM.

Kotlin Kotlin ist eine junge Programmiersprache von JetBrains die nach Java-Bytecode kompiliert werden kann und somit in der JVM nutzbar ist. Sie bietet gegenüber Java einige Vorteile indem moderne Konstrukte wie Datenklassen und Lambda Ausdrücke nativ nutzbar sind.

Merge Request Ein *Merge Request* (oder auch *Pull Request*) bezeichnet eine Anfrage für das Zusammenführeren verschiedener Branches aus Git. Ein *Merge Request*

kann zwar von jedem gestellt werden, aber nur bestimmte Personen dürfen diesen annehmen.

Mustache Mustache ist eine Template-Engine, die es ermöglicht Codevorlagen zu erstellen. In einer Codevorlage können Platzhalter definiert werden, die später durch Parameter ersetzt werden.

NPM NPM ist ein Paket-Manager für Javascript und Typescript. Mithilfe von NPM lassen sich Abhängigkeiten definieren, die dann automatisch installiert werden können.

OpenAPI Specification Die OpenAPI Specification (ehemals Swagger) ist eine populäre *IDL*. Es ist eine standardirisierte Form um Schnittstellen zu beschreiben. Zusätzlich kann eine in OpenAPI Specification definierte Schnittstelle in die meisten Programmiersprachen exportiert werden.

PPI AG Die PPI AG ist ein Softwareunternehmen, welche diese Bachelorarbeit unterstützt.

Representational State Transfer REST ist ein Softwareparadigma, welches vorgibt wie die Kommunikation zwischen verschiedenen Systemen aussehen soll. Der große Vorteil an REST ist, dass das Web als zugrundeliegende Struktur verwendet wird. Da fast jedes System in der Lage ist das HTTP-Protokoll zu verwenden, können diese Systeme auch via REST kommunizieren.

Round Trip Time Gibt die Zeit an, die ein Datenpaket benötigt, um in einen verteilten Softwaresystem zwischen zwei Punkten hin und wieder zurück zu reisen. Wenn ein System ein Datenpaket an ein anderes schickt, dann vergeht etwas Zeit, bis es dort ankommt. Und es vergeht noch mehr Zeit, bis eine Antwort beim ursprünglichen System zurück kommt. Die gesamte Laufzeit ist die Round Trip Time.

Single Point of Truth Ein Ausdruck in der Informatik, der besagt, dass die "Wahrheit" nur an einer Stelle zu finden. Es gibt somit keine Redundanzen der "Wahrheit". Die "Wahrheit" steht stellvertretend für verschiedene Daten. Hiermit kann Programmcode, Datensätze oder Definitionen gemeint sein.

Spring Eine Javabibliothek die viele grundlegende Funktionen wie REST-APIs oder *Dependency Injection* bietet. Spring ist sehr weit verbreitet und wird häufig in *Backends* verwendet.

Time-to-Value Time-to-Value ist die Zeit, die benötigt wird, bis eine Änderung am Produkt beim Endnutzer ankommt. Der Worst-Case tritt ein, wenn eine Änderung am Code kurz nach einem Release vorgenommen wurde. Dann liegt diese Änderung solange brach, bis es ein neues Release gibt, was u.U. Monate dauern könnte.

Typescript Typescript erweitert JavaScript um ein wichtiges Konzept: Typisierung.
In JavaScript gibt es keine Datentypen, welche jedoch in Typescript vorhanden sind. Entwicklungen in Typescript haben somit weniger Fehlerpotential, weil der Compiler die Typen kennt und somit Entwickler unterstützen kann.

Verteiltes System Ein verteiltes System besteht aus mehreren autonomen Recheneinheiten, welche dem Softwarenutzer als ein einzelnes kohärentes System erscheint.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ____________________________

Vorname: ____________________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Minimalredundante Validierung von Benutzereingaben in Front- und Backend eines interaktiven Softwaresystems

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

________________	________________	____________________________
Ort	Datum	Unterschrift im Original