Seminararbeit im Seminar Trends der Softwaretechnik
Sommersemester 2025

Typsichere Schnittstellendefinition von API-Endpunkten

Nils Derenthal (7217566)
nils.derenthal002@stud.fh-dortmund.de
Informatik Dual

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Grundlagen 2
1.2 Problemstellung 2
1.3 Ziel der Arbeit 3

2 Systematische Literaturrecherche **4**

3 Hauptteil **4**
3.1 Begriffserläuterungen 4
3.1.1 Typen und Typsystem 4
3.1.2 Algebräische Typen 4
3.2 Schnittstellendefinition von REST APIs 5
3.2.1 API Schemas & OpenAPI 6
3.2.2 Typgenerierung 7
3.3 Persönliche Reflexion 7

4 Zusammenfassung und Ausblick **7**
4.1 Zusammenfassung 7
4.2 Kritische Reflektion 7
4.3 Ausblick 7

5 Anhang **7**

1 Einleitung

Typensicherheit ist ein oft diskutiertes Thema in der Softwareentwicklung. Viele Sprachen

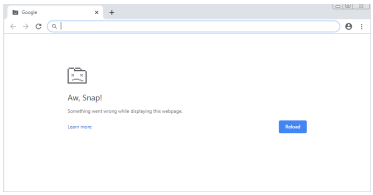
Die folgende Arbeit richtet sich an angehende Softwarearchitekten und soll dabei helfen API-Design sowie Implementation mithilfe von Typsystemen zu erläutern. Das Thema wird im folgenden durch anschauliche Beispiele im Kontext von reelen Szenarien praxisnah dargestellt.

1.1 Grundlagen

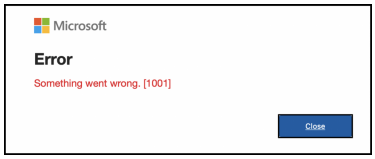
Für das Verständnis der Arbeit wird angenommen, dass Leser*innen ein Grundlegendes Wissen über Webanwendungen sowie REST-APIs besitzen. Es ist desweiteren hilfreich, wenn Kenntnisse über eine Sprache mit typisierung vorhanden sind (Beispielsweise Klassen und Objekttypen in Java, Types in TypeScript oder Enums in Rust). Die Typentheorie muss nicht bekannt sein und alles was in diesem Bezug relevant ist wird im Laufe der Arbeit erläutert. Hierfür wird ein Basiswissen über Mengen verlangt.

1.2 Problemstellung

Fehler in der Softwareentwicklung sind auch heutzutage noch sehr präsent. Ob in einem Studienprojekt, der Website einer Behörde, oder dem Betriebssystem eines Millardenkonzerns: Im Alltag trifft man nicht selten auf solche Fehler.



(a) Ein Crash im Chrome-Browser



(b) Ein Fehler während des Microsoft Logins

Abbildung 1: Fehler in alltäglichen Anwendungen

Ein Ansatz um Fehler zur Laufzeit von Software zu verhindern sind Typsysteme, wie Beispielsweise von TypeScript welches versucht JavaScript zu typisieren.

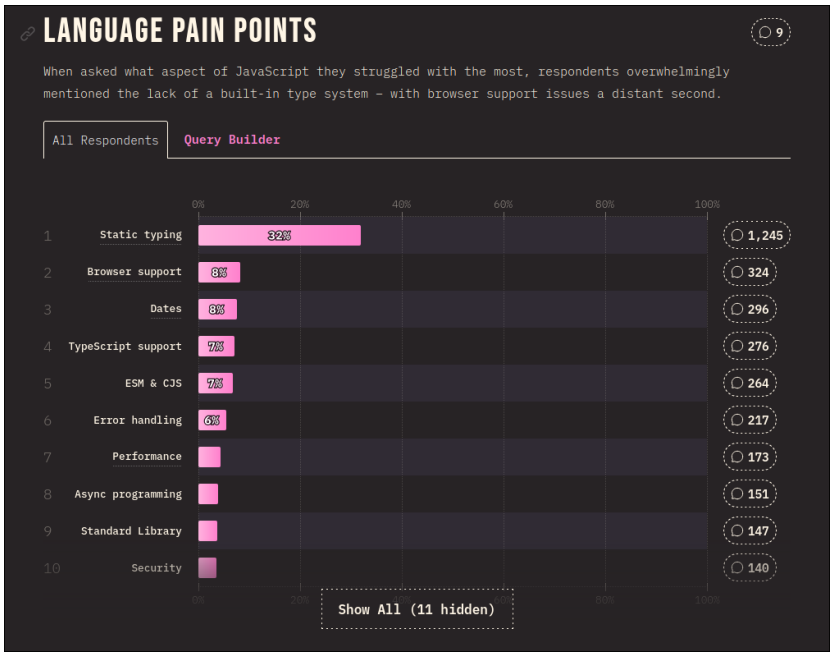


Abbildung 2: 32% der Befragten in der „State of JavaScript 2024“ Umfrage gaben an das (fehlende) statische Typisierung einer ihrer größten Probleme mit der Sprache sei [GB24]

In modernen Softwarearchitekturen kommunizieren Anwendungen häufig über APIs. Dabei werden die Datenstrukturen von Anfrage- und Antwort Daten oft mehrfach und in verschiedenen Systemen definiert - zum Beispiel getrennt im Front- und Backend. Diese Redundanz führt zu erhöhtem Wartungsaufwand und einem hohen Risiko von Fehlern, wenn sich Schnittstellen ändern, aber nicht von allen Konsumenten gleichermaßen angepasst werden.

1.3 Ziel der Arbeit

In der folgenden Ausarbeitung sollen zunächst Möglichkeiten aufgezeigt werden, wie Schnittstellen Typsicher definiert werden können. Daraufhin werden Tools präsentiert mit denen Schnittstellen über Systeme hinweg konsistent dargestellt werden können um hohe Wartungsaufwände und Fehler bei änderungen zu vermeiden.

Insgesamt soll die Arbeit Architekten von Webanwendungen dabei helfen API-Schnittstellen zu definieren und die in der Arbeit erwähnten Tools sinnvoll einzusetzen um konsistente, klar definierte und typsichere Endpunkte zu schreiben.

2 Systematische Literaturrecherche

3 Hauptteil

In diesem Abschnitt wird der zentrale Inhalt der Arbeit behandelt. Ausgehend von grundlegenden Konzepten der Typensysteme wird schrittweise erläutert, wie diese Konzepte auf die Struktur und Definition moderner Web-APIs übertragen werden können. Ziel ist es, ein Verständnis dafür zu schaffen, wie Typisierung zur Fehlervermeidung und zur strukturierten Entwicklung von Schnittstellen beiträgt.

3.1 Begriffserläuterungen

3.1.1 Typen und Typsystem

Die Grundsätzliche Idee eines Typsystems ist die verhinderung von Fehlern zur Laufzeit eines Programmes. Typischerweise wird hierfür ein Algorithmus eingesetzt welcher überprüft, dass alle Typen die innerhalb eines Programmtextes verwendet werden untereinander stimmig sind.

Typen können als eine Menge von möglichen Werten gesehen werden. So ist beispielsweise die Menge für den Typen von Ganzzahlen (oft `integer` oder `int`) im Fall von 32 bit $[-2^{31}, 2^{31} - 1]$. Einer der simpelsten Typen sind Wahrheitswerte (oft `boolean` oder `bool`) welche aus der Menge $\{\text{true}, \text{false}\}$ bestehen.

In vielen Sprachen gibt es primitive Typen dessen Wertemengen genau ein Element umfassen. In Java ist dies Beispielsweise `null` und in TypeScript existieren `null` und `undefined`. [1]. Diese Typen können in kombination mit Summen (siehe 3.1.2) die mögliche Abwesenheit eines Wertes darstellen. In Java Beispielsweise kann jeder nicht primitive Typ implizit den Wert `null` besitzen.

Oft stellen Sprachen Möglichkeiten bereit aus diesen, wie auch weiteren, als *primitiv* bezeichneten Typen, komplexere Strukturen zu bilden um kompliziertere Anwendungsfälle abzudecken.

Das Typesystem einer Sprache ist der Oberbegriff dafür wie Typen gebildet werden können und wie die richtigkeit dieser überprüft wird, falls das der Fall ist.

3.1.2 Algebräische Typen

In der Typentheorie ist oft von algebräischen Typen die Rede. Damit sind sogenannte Summen- und Produkttypen gemeint.

Summen (auch `Vereinigung` oder `Union`) stellen eine Auswahl zwischen mehreren Typen da. Beispielsweise kann eine Antwort einer API das tatsächliche Ergebnis einer Anfrage *oder* eine Fehlermeldung zurückgeben. In der Typentheorie wird für Summentypen das + Zeichen oder | verwendet. `Antwort = Fehler | TatsaechlicheAntwort` wäre eine Möglichkeit das Oben genannte Bei-

[1] Auch wenn `void` einen zusätzlichen unären Typen darstellt wird dieser hier ignoriert

spiel darzustellen. Mögliche Werte für `Antwort` kommen nun aus der *Vereinigung* der Mengen von den Typen `Fehler` und `TatsaechlicheAntwort`.

Produkte (auch **Schnittmengen**) sind ein Zusammenschluss von mehreren Typen. Als Notation für Produkte wird entweder × oder & benutzt. Wenn eine Anfrage Beispielsweise Informationen zu einem Nutzeraccount liefert könnten diese eine ID, einen Namen sowie eine E-mail Adresse Enthalten. Dies kann folgendermaßen dargestellt werden: `NutzerInfos = ID & Name & Email` In der Praxis, beispielsweise in Sprachen wie TypeScript, sind Produktypen oft benannt, sie sind also Schlüssel-Wert Paare.

Mit Hilfe dieser beiden Operationen lassen sich bis auf wenige Ausnahmen, auf die nicht weiter eingegangen wird, alle Typen darstellen.

3.2 Schnittstellendefinition von REST APIs

Schnittstellen werden oft durch sogenannte DTOs (Data Transfer Objects) definiert. Diese nehmen oft die Form von einfachen Produkttypen an. Um den Transfer von Daten über das Netzwerk möglich zu machen wird oft von Serialisierung gebrauch gemacht um die Daten in einem genormten Format zu senden. Eins der häufigsten Formate hierbei ist JSON (JavaScript Object Notation).

Im folgenden wird ein Beispiel aus der Gitlab REST API Betrachtet. Der Endpunkt `/projects/:id/repository/files/:filepath` erlaubt es Benutzer*innen eine Datei aus einem Repository abzufragen. Die Erhaltene Antwort von GitLab folgt immer dem selben Schema:

```
{
  "file_name": "file.hs",
  "size": 1476,
  ...
  "last_commit_id": "570e7b2abdd848b...",
  "execute_filemode": false
}
```

Dieses Schema lässt sich als ein Produkt der einzelnen Schlüssel-Wert Paare Darstellen:
`file_name=string & size=number & ...`
Eine Vereinfachte Darstellung, in einem JSON-ähnlichen Format könnte wiefolgt aussehen:

```
{
  "file_name": string,
  "size": number,
  ...
}
```

Es ist möglich das eine Schnittstelle optionale Felder besitzt, welche mit einer

Summe des unären Typen (bespielsweise `undefined`) dargestellt wird. [2].

TODO Beispiel

3.2.1 API Schemas & OpenAPI

Zur formalen Definition und Dokumentation von REST APIs haben sich sogenannte API-Schemas etabliert. Diese beschreiben die möglichen Anfragen und Antworten, die eine API erwartet bzw. zurückgibt. Ein weit verbreiteter Standard zur Beschreibung solcher Schnittstellen ist OpenAPI [3].

Ein OpenAPI-Dokument beschreibt typischerweise unteranderem:

- Pfade (Endpoints) der API
- HTTP-Methoden (GET, POST, etc.)
- Anfrageparameter und deren Typen
- Erwartete Antworttypen und HTTP-Statuscodes

im folgenden soll hierbei primär die Beschreibung der Afragen und Antworten betrachtet werden. Ein zentrales Konzept ist das sogenannte Schema, das die Struktur dieser Daten beschreibt. Die Schemata basieren in OpenAPI auf JSON.

Ein einfaches Beispiel für ein OpenAPI Schema, das dem bereits zuvor betrachteten GitLab Beispiel ähnelt, könnte wie folgt aussehen:

```
1  {
2    "type": "object",
3    "properties": {
4      "file_name": {"type": "string" },
5      "size": {"type": "integer" },
6      "last_commit_id": {"type": "string" },
7      "execute_filemode": {"type": "boolean" }
8    },
9    "required": ["file_name", "size"]
10 }
```

Hierbei ist ersichtlich, dass es sich um einen Produkttyp handelt, dessen Felder unterschiedliche primitive Typen besitzen. Felder, die nicht im `required`-Array gelistet sind, sind optional, was, wie zuvor beschrieben, typentheoretisch einer Summe mit einem unären Typen wie `undefined` entspricht.

Neben der Möglichkeit eine API so zu dokumentieren können Schemata wie diese auch dazu verwendet werden um diese Maschinell zu analysieren und zu transformieren.

[2] Auf den Unterschied zwischen optionalen Feldern und erforderlichen optionalen Feldern wird hier nicht weiter eingegangen.

[3] ehemals Swagger

3.2.2 Typgenerierung

Aus OpenAPI-Schemas lassen sich automatisch Typdefinitionen für Programmiersprachen generieren, was die Konsistenz zwischen API-Dokumentation und Implementierung sicherstellt. Dies ist möglich bei Projekten die eine eigene Backend- und Frontend-Struktur besitzen, sowie als auch bei Projekten die ausschließlich eine externe API benutzen welche ein OpenAPI Schema bereitstellt.

Für TypeScript ist das Tool `openapi-typescript` verfügbar. Aus dem zuvor gezeigten Schema wird folgenden Typ erzeugt:

```typescript
interface FileInfo {
  file_name: string;
  size: number;
  last_commit_id: string | undefined;
  execute_filemode: boolean | undefined;
}
```

Optionale Felder, also die, die nicht in dem `required` array existieren, werden als optional deklariert.

Besides generating a type definition from a schema, OpenAPI tools also allow generating schemas from existing APIs. For example, in the Spring Framework for Java, `springdoc-openapi` can generate a schema from a `RestController`. When the generated schema is used by the frontend to generate its type definition there is a clear

3.3 Persönliche Reflexion

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

4.2 Kritische Reflektion

4.3 Ausblick

5 Anhang

Literatur

[GB24] Sacha Greif und Eric Burel. *State of javascript 2024*. 2024. URL: https://2024.stateofjs.com/en-US/features/#language_pain_points.

Suchergebnisse	**Notizen**
N. H. Madhavji und I. R. Wilson. „Cray Pascal“. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. SIGPLAN '82. Boston, Massachusetts, USA: Association for Computing Machinery, 1982, S. 1–14. ISBN: 0897910745. DOI: 10.1145/800230.806975. URL: https://doi.org/10.1145/800230.806975	Notizen

Tabelle 1: Literaturliste zum Rechercheprotokoll