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

In der heutigen Softwareentwicklung gewinnen strukturierte Schnittstellen zwischen Systemkomponenten zunehmend an Bedeutung. Insbesondere Web-APIs bilden die Essenz vieler verteilter Anwendungen, bei denen unterschiedliche Systeme, etwa Frontend- und Backend-Komponenten, zuverlässig miteinander kommunizieren müssen. Ein zentrales Element solcher APIs ist die präzise Beschreibung der übermittelten Daten. Auch wenn die Typisierung hierbei eine essenzielle Rolle spielt, um Fehler zu vermeiden, Entwicklungsprozesse zu standardisieren und die Wartbarkeit langfristig zu sichern, ist die Definition der Schnittstellen nicht Trivial. In der folgenden Arbeit soll aufgezeigt werden, wie Schnittstellen strukturiert definiert werden können und welche Tools existieren um diese Definition konsistent über Systeme hinweg durchzuführen.

Diese Arbeit richtet sich an Leser*innen mit technischem Hintergrund im Bereich der Softwareentwicklung, insbesondere an Softwareentwickler*innen und -architekt*innen, die mit der Definition, Analyse oder Nutzung von Web-APIs befasst sind. Darüber hinaus ist sie auch für Informatikstudent*innen geeignet, die bereits Grundkenntnisse im Bereich Typensysteme besitzen und diese auf in der Praxis in Bezug auf Schnittstellen anwenden möchten, um die Wartbarkeit von ihren APIs zu erhöhen.

1.1 Grundlagen

Für das Verständnis der Arbeit wird angenommen, dass Leser*innen ein Grundlegendes Wissen über Webanwendungen sowie REST-APIs besitzen. Auch wenn kein tieferes Wissen diesbezüglich erforderlich ist, hilft es dennoch wenn Begrifflichkeiten und Grundlegende Konzepte wie Endpunkte, HTTP-Methoden und Anfrage/Antwortkörper bekannt sind.

Es sollten Grundlegende Kenntnisse über die Struktur von JSON vorlegen, da dies als Beispiel sowie für die OpenAPI Schemata verwendet wird.

Kenntnisse über TypeScript sind hilfreich, aber nicht erforderlich. Die Typentheoretischen Begriffe und Konzepte werden in der Arbeit erläutert. Hierfür ist ein Basiswissen über die Mengenlehre förderlich.

Es ist nicht nötig, dass Leser*innen bereits über ein Wissen zu OpenAPI oder API Schemata verfügen, da dies das Thema der Arbeit ist.

1.2 Problemstellung

In der Praxis besteht häufig eine Diskrepanz zwischen der dokumentierten API und ihrer tatsächlichen Implementierung. Diese Abweichungen können zu Laufzeitfehlern, Missverständnissen in der Entwicklung und erhöhtem Wartungsaufwand führen, insbesondere dann, wenn keine oder nur unzureichende Typisierung verwendet wird. Dynamisch typisierte Systeme sind in dieser Hinsicht besonders anfällig, da Fehler oft erst spät, beispielsweise zur Laufzeit oder beim Testen, erkannt werden. Dies ist der Grund, warum Sprachen wie TypeS-

cript existieren, welche versuchen existierende Sprachen um Typsysteme anzureichern.

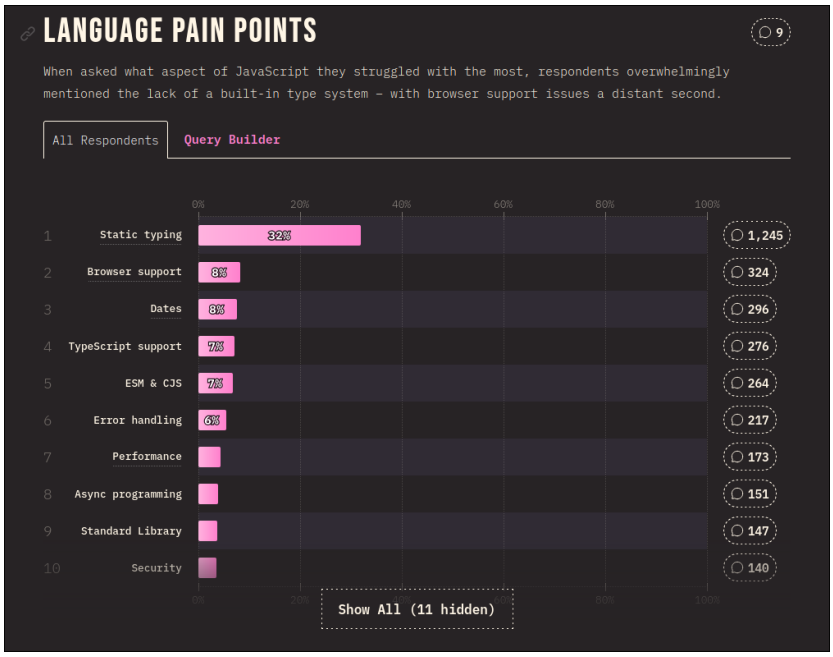

Abbildung 1: 32% der Befragten in der „State of JavaScript 2024“ Umfrage gaben an das (fehlende) statische Typisierung einer ihrer größten Probleme mit der Sprache sei [GB24]

Selbst bei Verwendung von statisch typisierten Sprachen bleibt die manuelle Synchronisierung zwischen API-Dokumentation und Code eine fehleranfällige Aufgabe. Dies betrifft sowohl Backend-Entwickler*innen, die Schnittstellen bereitstellen, als auch Frontend-Entwickler*innen, die auf diese Schnittstellen zugreifen und korrekte Datenannahmen treffen müssen. Bei Diskrepanzen zwischen den Definitionen kann es zu Problemen führen, bei denen Nutzer der Website unerwartete Fehler erhalten was ein Betriebsrisiko für das Softwareprodukt bedeutet. Um dies zu vermeiden sind eventuell End-To-End tests nötig, welche kostenaufwendig und schwer zu pflegen sind.

1.3 Ziel der Arbeit

Ziel dieser Arbeit ist es, die Relevanz und den Nutzen von Typensystemen bei der Modellierung und Beschreibung moderner Web-APIs aufzuzeigen. Im Mittelpunkt steht dabei die Analyse, wie typentheoretische Konzepte, insbesondere Produkt- und Summentypen, auf API-Schnittstellen übertragen und durch Standards wie OpenAPI formalisiert werden können.

Dabei wird exemplarisch untersucht, wie sich aus API-Schemata typisierte Datenstrukturen generieren lassen, um sowohl Konsistenz als auch Typsicherheit für das Softwareprodukt zu gewährleisten und die Wartbarkeit und Fehlerresistenz zu erhöhen. Die Arbeit veranschaulicht diese Zusammenhänge anhand konkreter Beispiele und Anwendungsfälle.

Insgesamt soll die Arbeit Architekten von Webanwendungen dabei helfen API-Schnittstellen eindeutig zu definieren und die in der Arbeit erwähnten Tools sinnvoll einzusetzen, um konsistente, klar definierte und typsichere Endpunkte zu schreiben. Die Arbeit soll primär als Startpunkt gelten und die Tools nicht im Detail beschreiben oder Anweisungen geben, wie diese zu benutzen sind. Es soll ausschließlich dargestellt werden, was für ein Lösungsansatz für das gegebene Problem existiert.

2 Systematische Literaturrecherche

2.1 Rechercheziel

Das Ziel der Quellensuche soll es sein eine Lösung zu dem Zuvor erläuterten Problem zu finden. Da Grundlegende Kenntnisse über TypeScript sowie Wissen über die Existenz von OpenAPI existieren, sollen diese Informationen verifiziert und formalisiert werden, um Leser*innen ein Neutrales Bild über diese Tools und ihre Anwendung zu präsentieren.

TODO

2.2 Relevante Quellen für die Literatur

Zur Quellensuche wurde vor allem Google Scholar herrangezogen, welche die Suche nach wissenschaftlichen Quellen vereinfacht. Auf Basis der Suchen wurden die folgenden Quellen / Veröffentlichungsorte von wissenschaftlichen Papieren aufgrund ihrer Relevanz auf Bezug zu dem Rechercheziel identifiziert.

2.2.1 ACM Digital Library

Die ACM Digital Library (ACMDL) ist eine geeignete Quelle für wissenschaftliche Arbeiten zu Themen aus der Informatik, da sie peer-reviewte Publikationen aus diesem Bereich bereitstellt. Sie bietet Zugang zu aktueller Forschung und anerkannten Konferenzen, die regelmäßig Beiträge zu relevanten Aspekten des Themas veröffentlichen. Die Inhalte sind wissenschaftlich fundiert und werden von Fachleuten der Community erstellt. [Com]

2.2.2 IEEE Xplore Digital Library

Die IEEE Xplore Digital Library bietet eine verlässliche Grundlage für wissenschaftliche Arbeiten in diesem Bereich. Sie umfasst eine große Auswahl an Fachartikeln, Konferenzbeiträgen und technischen Standards mit Schwerpunkt auf Informatik und verwandten Disziplinen. Besonders relevant ist sie aufgrund

ihres Fokus auf praxisnahe und anwendungsorientierte Forschung, die aktuelle Entwicklungen im Themenfeld umfassend abbildet. Die Inhalte stammen aus etablierten Fachkreisen und unterliegen strengen Qualitätsstandards. [EE]

2.2.3 Blogartikel

Blogposts bieten eine wichtige Ressource zur schnellen Informationsgewinnung und praktischen Anwendung. Entwicklerblogs und technische Beiträge von Unternehmen oder Fachleuten liefern häufig praxisnahe Anleitungen, Beispielcodes und Erfahrungsberichte, die in der wissenschaftlichen Literatur oft fehlen oder nur stark abstrahiert vorhanden sind. Dadurch ermöglichen Blogposts einen unmittelbaren Zugang zu neuen Technologien, Best Practices und aktuellen Entwicklungen. Gleichzeitig sind jedoch Risiken für die wissenschaftliche Arbeit erkenntlich: Die Inhalte unterliegen meist keiner formalen Qualitätskontrolle, sind nicht immer langfristig verfügbar und reflektieren oft subjektive Sichtweisen einzelner Entwickler*innen. Für eine fundierte wissenschaftliche Nutzung müssen solche Quellen daher kritisch bewertet und durch etablierte Literatur oder offizielle Dokumentationen ergänzt werden.

2.2.4 Offizielle Dokumentation

Die offizielle Dokumentation stellt in der Informatik eine der zuverlässigsten Quellen für die Nutzung und Implementierung von Technologien dar. Sie wird in der Regel von den Entwickler*innen der jeweiligen Software oder Plattformen erstellt und bietet detaillierte, präzise Informationen zu Funktionen, Parametern und Anwendungsmöglichkeiten. Da offizielle Dokumentationen oft direkt aus dem Quellcode und den Designentscheidungen der Entwickler*innen abgeleitet werden, gelten sie als die autoritativste und genaueste Informationsquelle. Zudem bieten sie eine stabile und langfristig verfügbare Referenz, die regelmäßig aktualisiert wird, um mit neuen Versionen und Änderungen Schritt zu halten. Ein Risiko lässt sich allerdings darin sehen, dass eine voreingenommene Haltung gegenüber der präsentierten Software existiert. Deshalb muss auch hier mit Hilfe von unabhängigen Quellen belegt werden wie Beispielsweise Effektiv eine Software ist.

2.3 Relevante Suchbegriffe

Auch wenn die Arbeit im Deutschen Formuliert ist, wird ein Großteil der Recherche im Englischen durchgeführt, da in der Informatik die meiste Literatur Englischsprachig ist, und es so warscheinlicher ist, dass Quellen gefunden werden.

2.3.1 foo

English
- Schnittstellendefinition - API Schema - OpenAPI - Algebräischer Datentyp

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 Vermeidung von Fehlern zur Laufzeit eines Programmes. Typischerweise wird hierfür ein Algorithmus eingesetzt welcher überprüft, ob alle Typen, die innerhalb eines Programmtextes verwendet werden, konsistent 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 viele Anwendungsfälle abzudecken.

Ein Typsystem beschreibt die Regeln einer Programmiersprache zur Bildung, Kombination und Überprüfung von diesen Typen.

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

Produkte (auch `Kartesisches Produkt`) sind ein Zusammenschluss von mehreren Typen. Als Notation für Produkte wird entweder $\times$ oder & benutzt.

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

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 & ...`

Es ist möglich das eine Schnittstelle optionale Felder besitzt, welche mit einer Summe des unären Typen (bespielsweise `undefined`) dargestellt wird. [2].

Beispielsweise könnte eine Referenz auf eine assoziierte Merge Request existieren, dies muss aber nicht der Fall sein:

Eine Vereinfachte Darstellung, des Oben genannten Beispiels mit Optionalem Feld in einem JSON-ähnlichen Format könnte wiefolgt aussehen:

```
{
  "file_name": string,
  "size": number,
  "related_merge_request": number | undefined
  ...
```

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

```
}
```

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:

```
{
  "type": "object",
  "properties": {
    "file_name": {"type": "string" },
    "size": {"type": "integer" },
    "last_commit_id": {"type": "string" },
    "execute_filemode": {"type": "boolean" }
  },
  "required": ["file_name", "size"]
}
```

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.

[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:

```
1 interface FileInfo {
2   file_name: string;
3   size: number;
4   last_commit_id: string | undefined;
5   execute_filemode: boolean | undefined;
6 }
```

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

Neben der Generierung einer Typdefinition aus einem Schema ermöglichen OpenAPI-Tools auch die Erstellung von Schemata aus bestehenden APIs. Zum Beispiel kann im Spring Framework für Java das Tool springdoc-openapi ein Schema aus einem RestController generieren. Wenn das generierte Schema vom Frontend zur Erstellung seiner Typdefinition verwendet wird ergibt sich auch hier eine klare Definition, ohne das das Schema händisch erstellt werden muss.

3.2.3 Limitationen in Bezug auf Typen

Obwohl OpenAPI, wie aufgezeigt die Möglichkeit gibt Typen aus definitierten Schnittstellen zu bilden, gibt es auch einige wichtige Limitationen bei der Übersetzung.

Ein zentrales Problem ist, dass OpenAPI über das Attribut `format` zusätzliche semantische Einschränkungen auf primitive Typen spezifizieren kann, die über die reine Typinformation hinausgehen. Beispielsweise kann ein Feld vom Typ `string` das Format `date-time` tragen, was bedeutet, dass nur Zeichenketten im ISO 8601 Zeitformat gültig sind. Auch für `integer` oder `number` existieren Formate wie `int32`, `int64` oder `float`. Diese Formate definieren eingeschränkte Wertebereiche oder bestimmte Darstellungsformen.

Das Problem hierbei ist, dass viele Programmiersprachen, insbesondere solche mit struktureller Typisierung wie TypeScript, diese Formatinformationen nicht direkt in ihre Typensysteme übersetzen können. Während das Tooling den Typ korrekt als `string` oder `number` übernimmt, geht die mit dem Format verbundene Einschränkung dabei verloren. Der erzeugte Typ vermittelt also nur einen Teil der in OpenAPI spezifizierten Bedeutung, was zu einem Verlust an Präzision führen kann.

Ein weiteres Beispiel betrifft Wertebereiche, die über die `minimum`- und `maximum`-Eigenschaften eines Schemas definiert werden. Auch hier kann OpenAPI präzise

definieren, dass zum Beispiel ein Zahlenwert zwischen 0 und 100 liegen muss. In vielen Zielsprachen lässt sich dies jedoch nicht direkt typisieren, da es keine eingebauten Mechanismen für bereichsbegrenzte Typen gibt. Entsprechend müssen solche Einschränkungen entweder in Validierungslogik separat abgebildet oder manuell dokumentiert werden, was wiederum potenzielle Inkonsistenzen schafft.

3.2.4 Herausforderungen und Risiken automatischer Typgenerierung

Trotz der offensichtlichen Vorteile automatisierter Typgenerierung, wie der Reduktion manueller Fehler und der Sicherstellung von Konsistenz zwischen Dokumentation und Implementierung, bringt dieser Ansatz auch einige Herausforderungen mit sich. Ein zentrales Problem stellt der sogenannte Schema-Drift dar: Wenn sich das OpenAPI-Schema ändert, aber die generierten Typen im Frontend nicht aktualisiert werden, kann dies zu schwer auffindbaren Laufzeitfehlern führen. Zudem besteht bei generierten Typen die Gefahr einer Fehlinferenz, etwa wenn das Schema nicht präzise genug formuliert ist oder vereinfachte Annahmen über optionale Felder getroffen werden. In größeren Projekten kann es außerdem zu Konflikten zwischen manuell gepflegten und automatisch erzeugten Typdefinitionen kommen, insbesondere wenn Entwickler*innen lokal Anpassungen vornehmen, die beim nächsten Generierungslauf überschrieben werden. In solchen Fällen ist eine klare Tooling-Strategie sowie ein bewusstes Versions- und Änderungsmanagement unerlässlich, um Konsistenz und Wartbarkeit langfristig sicherzustellen.

3.3 Persönliche Reflexion

In meiner bisherigen Arbeit als Softwareentwickler bin ich immer wieder auf die Herausforderungen gestoßen, die inkonsistente Schnittstellen mit sich bringen. Diese Probleme reichen von Missverständnissen in der Kommunikation zwischen verschiedenen Systemen, bis hin zu erhöhtem Wartungsaufwand und einer schlechteren Benutzererfahrung. Der Umgang mit solchen Schnittstellen kann durchaus mühsam sein, da kleine Änderungen oder Fehler schwerwiegende Auswirkungen auf die gesamte Systemarchitektur haben können.

In diesem Kontext habe ich OpenAPI als eine interessante Möglichkeit kennengelernt, um Schnittstellen klarer und standardisierter zu gestalten. OpenAPI ermöglicht es, Schnittstellen strukturiert und maschinenlesbar zu definieren, was viele Vorteile mit sich bringt. Durch die Dokumentation von APIs in einer klaren, standardisierten Form können potenzielle Inkonsistenzen frühzeitig erkannt und behoben werden, was den Entwicklungsprozess insgesamt effizienter und weniger fehleranfällig macht.

Trotz der vielen Vorteile, die OpenAPI mit sich bringt, bleibt ein gewisser kritischer Blick bestehen. Die Definition von APIs erfordert eine sorgfältige Planung und kann bei komplexen Systemen schnell unübersichtlich werden. Zudem besteht die Gefahr, dass bei einer rein automatisierten Generierung der Schnittstellen die Qualität und Flexibilität leidet, wenn Entwickler nicht ausreichend in den Prozess eingebunden sind. Es bleibt also eine Herausforderung, das Po-

tenzial von OpenAPI voll auszuschöpfen, ohne die Kontrolle über die Feinheiten der Schnittstellen und deren Anpassungsfähigkeit zu verlieren.

Insgesamt sehe ich OpenAPI als eine vielversprechende Methode zur Verbesserung der Konsistenz und Wartbarkeit von Schnittstellen, auch wenn ich die Notwendigkeit für eine kritische und bewusste Implementierung nicht aus den Augen verlieren möchte. Die Balance zwischen Automatisierung und manuellem Feingefühl bleibt dabei entscheidend, um langfristig stabile und leistungsfähige Systeme zu entwickeln.

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

4.2 Kritische Reflektion

Im Nachhinein lässt sich einiges an Verbesserungspotential sehen. Einer der größten Faktoren ist die Zeitaufteilung, welche nicht Ideal abgelaufen ist. Einerseits habe ich zu spät angefangen und somit einen Großteil der Arbeit gegen Ende der geplanten Phase geschrieben, andererseits hätte ich mir konkrete Zwischenziele setzen, und so strukturierter arbeiten können.

Bei der Recherche hätte ich mir mehr Zeit lassen können und erst Stichpunkte herrausarbeiten können anstatt, dass ich einfach drauf los schreibe. Dies ist auch darauf Zurückzuführen, dass ich ein persönliches Interesse an dem bearbeiteten Thema hatte, und so eventuell voreingenommen in die Arbeit gegangen bin. Hier könnte es Hilfreich sein die Recherche von einer ünwissenderen"Perspektive durchzuführen um neutralere Ergebnisse zu erhalten.

4.3 Ausblick

Basierend auf der Arbeit wäre es möglich verschiedene weitergehende Themen zu behandeln. Einerseits kann untersucht werden wie effektiv Typsysteme für den Anwendungsfall der APIs sind, anderseits können aber auch Thematisch tiefere Themen beleuchtet werden. Beispielsweise könnten weitere Generierungsmöglichkeiten aus OpenAPI Schemas beleuchtet werden (Automatisierte API Zugriffe durch KI Agenten, Maschinelles Testen, Dokumentation, etc.).

Auch in Bezug auf Typsysteme und die Typentheorie lassen sich weitere Themenkomplexe finden, wie beispielsweise Summentypen für Systemnahe Programmierung verwendet werden können oder die Einsatzmöglichkeiten von Typen als Richtlinie für Generative KI Systeme in der Programmierung.

5 Anhang

Literatur

[Com] Association for Computing Machinery. URL: https://www.acm.org/publications/about-publications.

[EE] Institute of Electrical und Electronics Engineers. URL: https://journals.ieeeauthorcenter.ieee.org/when-your-article-is-published/about-the-ieee-xplore-digital-library/.

[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