Inhaltsverzeichnis

1 Konzeption des Prototyps

1.1 Datenmodellierung

Die zentrale Datenstruktur der Applikation ist die Beschreibung der API selbst, welche als Grundlage für alle Operationen verwendet wird. Da sich der Prototyp beim Eingabeformat auf das OpenAPI Spezifikationsformat beschränkt, wurde zunächst ein OpenAPI Metamodell erstellt, welches in Abbildung 1 dargestellt ist. Dieses Modell wird von der Applikation als Basis-Datenstruktur verwendet. Zur Unterstützung mehrerer Eingabeformate müssten in einer späteren Iteration auch andere Spezifikationsformate untersucht werden, um Gemeinsamkeiten abzuleiten und ein Datenmodell zu erstellen, in das alle Formate überführt werden können.

Es wurden ebenfalls zwei weitere Datenmodelle erstellt, die für die im Prototyp festgelegten Ziele benötigt werden. Zunächst wurde ein Modell der JSON-Schema Spezifikation erstellt, welche sich nur in wenigen Divergenzen von der OpenAPI Schema Definition unterscheidet:

1. Einige im OpenAPI Schema vorhandene Attribute wie `nullable` oder `deprecated` werden nicht unterstützt.
2. Die `type` Angabe kann sowohl ein String wie auch ein Array von Strings sein.
3. in `$schema` wird zusätzlich ein Link zur verwendeten JSON-Schema Version angegeben.

Diese Unterschiede müssen bei einer Umwandlung rekursiv aufgelöst werden, da ein Schema weitere verschaltete Schema-Defitionen enthalten kann. Das Modell einer JSON-Schema Definition ist in Abbildung 2 dargestellt.

Das Testsuite-Metamodell (Abbildung 3) stellt den Aufbau einer Testsuite dar, welche aus mehreren Testcases besteht. In jedem Testcase wird eine Anfrage mit entsprechenden Parametern ausgeführt, sowie eine Reihe von Assertions:

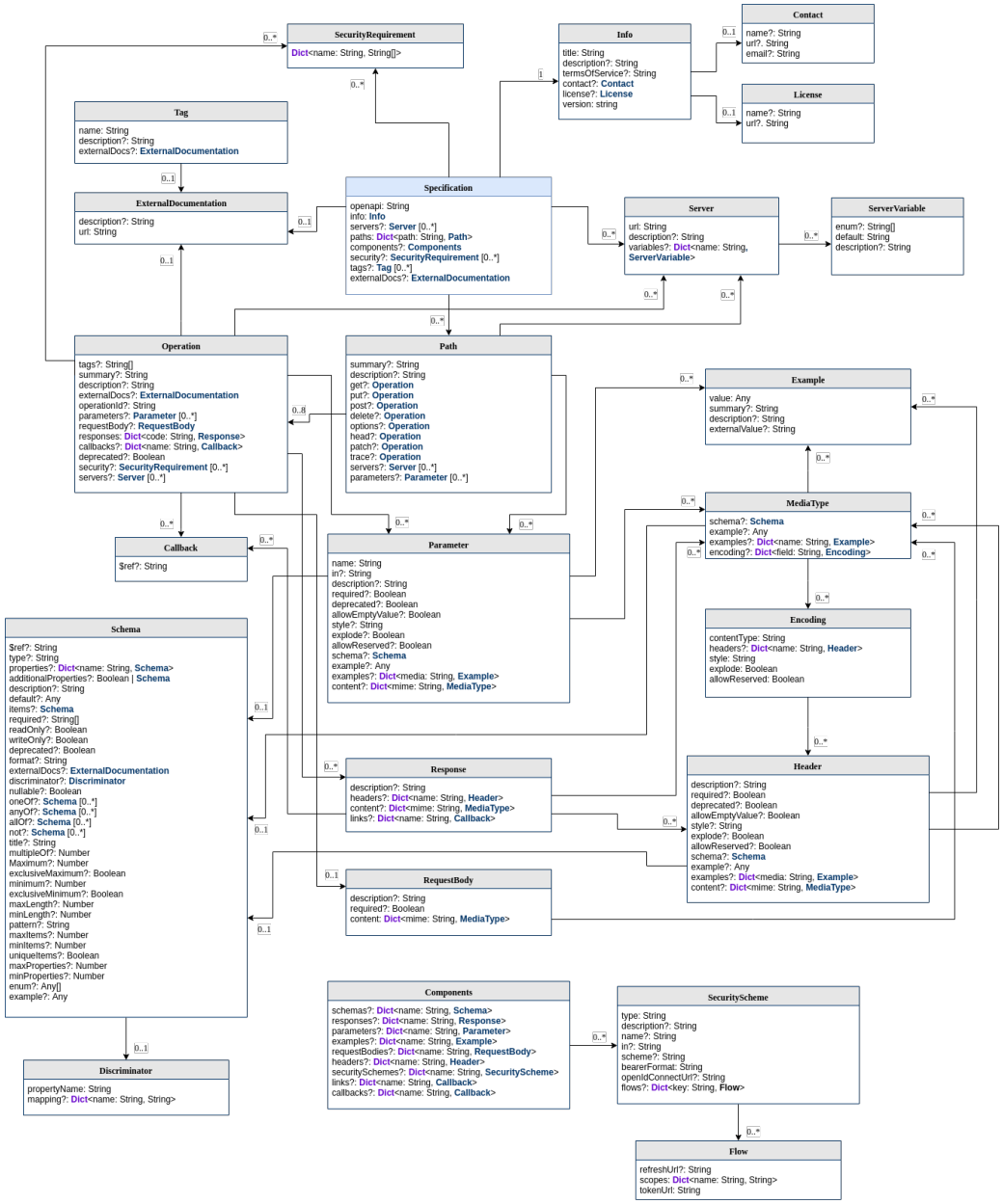

Abbildung 1: OpenAPI Metamodell

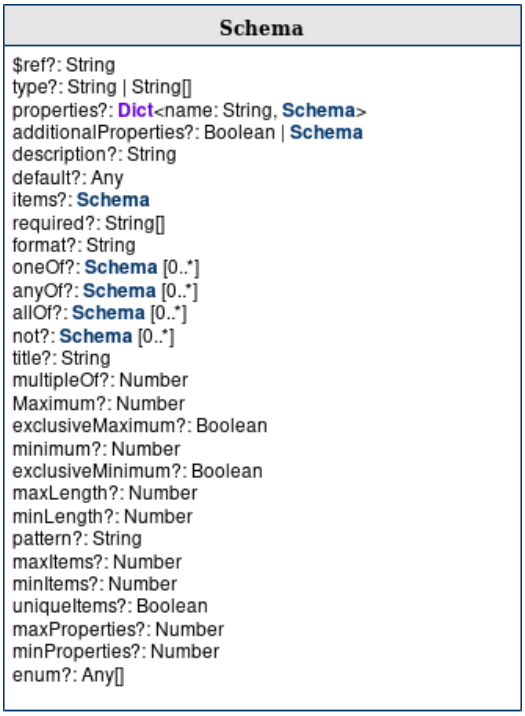

Abbildung 2: JSON-Schema Metamodell

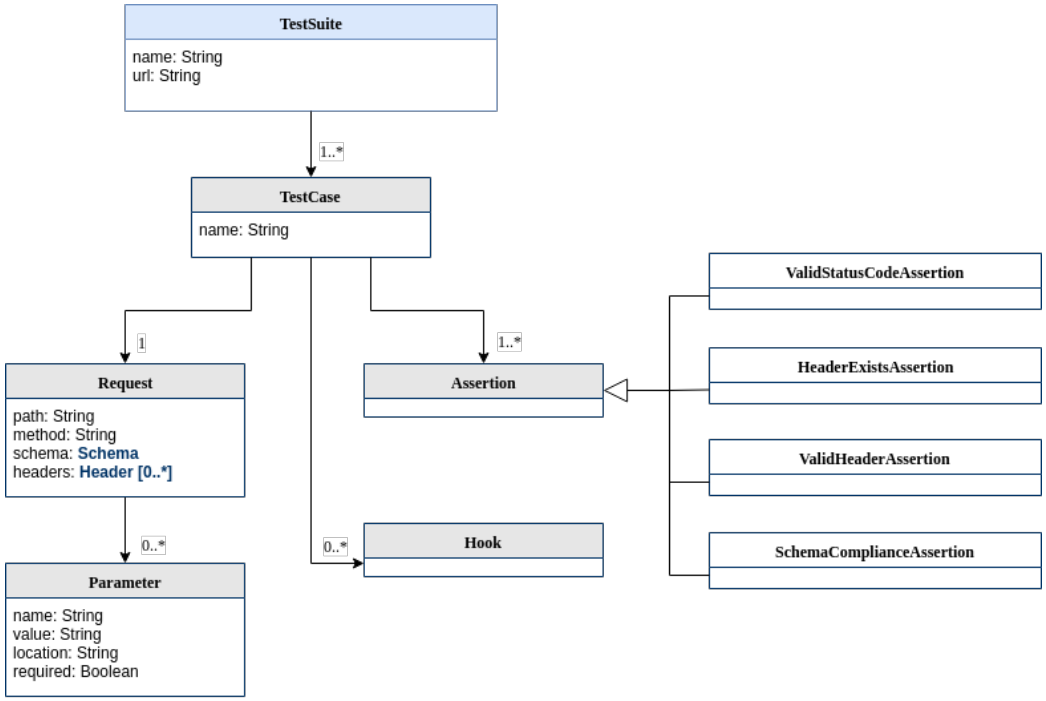

Abbildung 3: Testsuite Metamodell

1. `ValidStatusCodeAssertion` prüft, ob der HTTP-Statuscode der erhaltenen Antwort valide ist. Bei nominalen Testcases wird ein erfolgreicher Statuscode (2xx) erwartet, bei fehlerhaften Testcases ein Client-Fehler (4xx).

2. `HeaderExistsAssertion` prüft, ob alle in der Spezifikation definierten HTTP-Header vorhanden sind.

3. `ValidHeaderAssertion` prüft, ob alle HTTP-Header valide Werte haben.

4. `SchemaComplianceAssertion` validiert die vom Server erhaltene Antwort mit dem passenden JSON-Schema.

Ein Testcase führt ebenfalls mehrere Hooks aus, die von dem Entwickler angepasst werden können. Dadurch kann zum Beispiel Authentifizierung ermöglicht werden.

1.2 Architektur

Unter Berücksichtigung der beiden Ziele **Modularität** und **einfache Einbindung in Projekte** wird für die Umsetzung eine Javascript CLI-Applikation entwickelt. In den meisten Projekten sind bereits Javascript-Abhängigkeiten installiert, sodass sich das Tool leicht einbinden lässt. Durch NPM und die Aufteilung der Applikation in einzelne Pakete mit dynamischen Imports lässt sich ebenfalls die Modularität erreichen.

Die CLI-Applikation besteht aus einer Menge an Befehlen, welche alle den gleichen Aufbau haben, schematisch dargestellt in Abbildung 4. Jeder Befehl besteht aus einem *Parser*, der die API Spezifikation einliest, einem oder mehreren *Decorators*, welche die geparste Spezifikation mit befehlspezifischer Logik und Informationen anreichern, sowie einer oder mehreren *Factories*, die einen Output generieren.

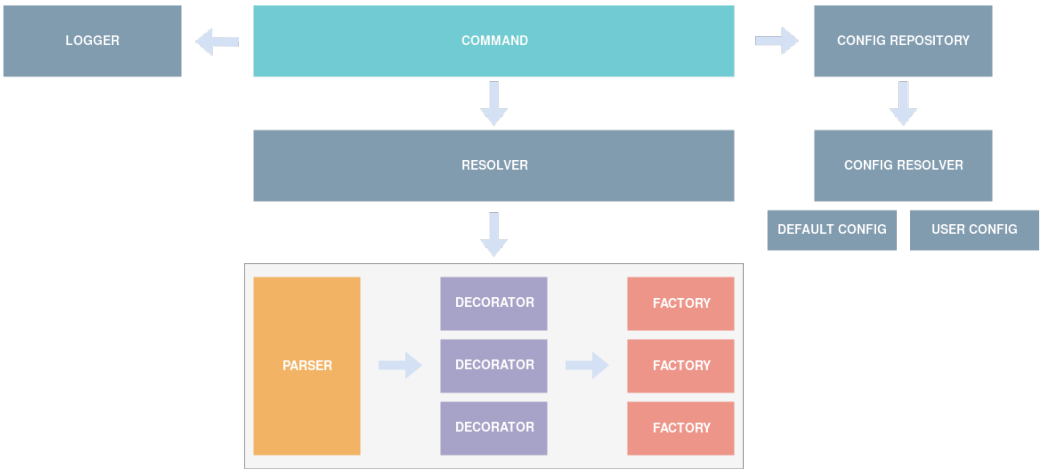

Abbildung 4: Architektur

Diese Komponenten sind beliebig austauschbar und konfigurierbar. Dadurch wird ermöglicht, dass bei Änderungen am Input- oder Output-Format die restliche Logik nicht verändert werden muss. So können z.B. mithilfe von 2 Factories Tests für zwei unterschiedliche Sprachen oder Frameworks gene-

riert werden, ohne dass etwas an dem Parsing oder Anreichern der Daten (im Falle der Tests z.B. das Ableiten von Parameter-Werten) verändert werden muss. Durch dynamische Imports mithilfe eines *Resolvers* werden die einzelnen Pakete zur Laufzeit eingebunden. Somit können Entwickler auch eigene Pakete erstellen, die dann von der Applikation verwendet werden können. Da Javascript keine Interfaces unterstützt, wird stattdessen Typescript für die Implementierung verwendet. Damit kann sichergestellt werden, dass Entwickler auf fest definierte Interfaces und Typdefinitionen zurückgreifen können, an die sich ihre Pakete halten müssen.

1.3 Vorgehensmodell

In folgendem Abschnitt wird kurz auf die Vorgehensweise der zwei Befehle `make:schemas` und `make:tests` eingegangen, welche im Prototyp der Applikation umgesetzt werden sollen.

1.3.1 Generierung von JSON Schemas

Zur Generierung der JSON Schemas werden ein *Decorator* und eine *Factory* implementiert:

- **Decorator**: der Decorator iteriert über alle Endpunkte und darin definierte Request- und Response-Bodies, und wandelt, falls vorhanden, die jeweiligen Open-API Schema Definitionen in valide JSON Schemas um. Dafür werden die bereits in Abschnitt 1.1[3.1] beschriebenen Unterschiede berücksichtigt. Anschließend werden die erstellten JSON Schemas der API Spezifikation hinzugefügt.
- **Factory**: die Factory schreibt die generierten JSON Schemas in `json`-Dateien in den konfigurierten Output-Ordner, und verwendet dabei einfach verständliche Dateinamen. So wird das JSON Schema der Anfrage eines HTTP `PATCH` Aufrufs auf die Ressource `/books` z.B. in `requests/books-update.json` gespeichert.

1.3.2 Generierung von Testcases

Zur Generierung von Testcases werden ebenfalls ein neuer *Decorator* und eine *Factory* implementiert. Der Befehl benutzt außerdem den bereits für JSON Schemas definierten *Decorator*, da in den Tests die Antworten vom Server mithilfe von JSON Schemas validiert werden.

- **Decorator**: der Decorator der Decorator iteriert über alle Endpunkte und erstellt das in Abbildung 3 definierte Testsuite Modell. Für jede Anfrage werden mithilfe von verschiedenen Heuristiken Parameterwerte abgeleitet, die in den Testanfragen benutzt werden können. Die hierfür verwendeten Heuristiken basieren auf den bereits in Ed-douibi, Izquierdo und Cabot (2018) definierten Heuristiken, wurden jedoch für eine erhöhte Genauigkeit noch angepasst bzw. erweitert und an die aktuelle Syntax der OpenAPI v3.0.2. Spezifikation angepasst:

  1. Der Wert eines Parameters `p` kann abgeleitet werden aus: Beispielwerten (`p.example`, `p.examples`, `p.schema.example`, `p.content.example`, `p.content.examples` oder `p.content.schema.example`), Standardwerten (`p.schema.default`, `p.schema.items.default` wenn `p` vom Typ `array` ist, `p.content.schema.default` oder `p.content.schema.items.default`) oder Enums (`p.schema.enum`, `p.items.enum` wenn `p` vom Typ `array` ist, `p.content.schema.enum` oder `p.content.schema.items.enum`).
  2. Der Wert eines Parameters kann abgeleitet werden, wenn eine Anfrage mit einem automatisch generierten Wert (bspw. ein String wenn der Parameter vom Typ `string` ist) erfolgreich ist.
  3. Der Wert eines Parameters kann abgeleitet werden, wenn für die selbe Ressource in einer der vorhanden Operationen ein Attribut gleichen Namens im Schema der Antwort definiert ist.

  Für fehlerhafte Testcases müssen fehlerhafte Parameter bzw. Attribut-Werte abgeleitet werden, was mithilfe verschiedener Regeln basierend auf dem Typ und den im Schema definierten Einschränkungen möglich ist, aufgelistet in den Tabellen 1 und 2.

Tabelle 1: Datentypen

Typ	**Regel**
`object`	Ein Objekt, welches das Schema verletzt
`integer` `number`	Zufälliger String der keine Zahlen enthält
`boolean`	Zufälliger String der nicht `true` oder `false` ist

- **Factory**: die Factory generiert die einzelnen Testcases in einer bestimmten Zielsprache. Dafür werden die in den Decorators angelegten Parameterwerte und JSON Schemas verwendet. Für jede API Operation wird, wenn für alle erfoderlichen Parameter (`p.required = true`) Werte abgeleitet werden konnten, für jede mögliche Parameterkombination ein separates Testcase erstellt, in dem eine Anfrage an den Server mit den abgeleiteten Parameterwerten geschickt wird. Anschließend wird geprüft, dass die Antwort einen validen Statuscode hat (2xx bei nominalen Testcases bzw. 4xx bei fehlerhaften Testcases), und mit den generierten JSON Schemas validiert werden kann. Über *Hooks*, die bei jedem Testcase vor und nach dem Ausführen der Anfrage aufgerufen werden, kann der Entwickler Anpassungen vornehmen. So kann z.B. Authentifizierung ermöglicht werden.

Tabelle 2: Einschränkungen

Einschränkung	**Regel**
`enum`	String oder Zahl, die nicht im Enum enthalten ist
`maximum / exclusiveMaximum`	Integer, der größer als das Maximum ist
`minimum / eclusiveMinimum`	Integer, der kleiner als das Minimum ist
`minLength`	String, der kürzer als `minLength` ist
`maxLength`	String, der länger als `maxLength` ist
`maxItems`	Array mit mehr Items als `maxItems`
`minItems`	Array mit weniger Items als `minItems`
`format`*`:date-time`* `format`*`:email`* `format`*`:hostname`* `format`*`:ipv4`* `format`*`:ipv6`* `format`*`:uri`*	Zufälliger String aus `[A-Za-z]`

Literatur

Ed-douibi, Hamza, Javier Luis Cánovas Izquierdo und Jordi Cabot (2018). „Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach“. In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, S. 181–190.