Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander Holland

Eine OpenAPI-Entwicklungsumgebung zum Testen
von HTTP-basierten Microservices

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Holland

Eine OpenAPI-Entwicklungsumgebung zum Testen
von HTTP-basierten Microservices

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Stefan Sarstedt
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am: 23. Juni 2020

Alexander Holland

Thema der Bachelorarbeit
Eine OpenAPI-Entwicklungsumgebung zum Testen von HTTP-basierten Microservices

Stichworte
OpenAPI, Swagger, Testen, HTTP, REST, Microservices, JavaScript, Electron

Kurzzusammenfassung
Es wird eine OpenAPI-Entwicklungsumgebung geplant und umgesetzt. Die grafische Benutzeroberfläche wird in drei Spalten unterteilt (Anfrage, Formular, Ergebnis) und ermöglicht eine Bearbeitung der OpenAPI-Definition mit einem Editor, wodurch eine Echtzeitvorschau in der Anfrage- und Formular-Spalte erzeugt wird. Für die Erstellung von verketteten und automatisierten Tests wird eine TestAPI entwickelt, die mit dem Formular oder Editor bearbeitet werden kann.

Alexander Holland

Title of the paper
An OpenAPI development environment for testing HTTP-based microservices

Keywords
OpenAPI, Swagger, Testing, HTTP, REST, Microservices, JavaScript, Electron

Abstract
An OpenAPI development environment is planned and implemented. The graphical user interface is divided into three columns (request, form, result) and enables editing of the OpenAPI definition with an editor, which creates a real-time preview in the request and form column. For the creation of chained and automated tests a TestAPI is developed, which can be edited with the form or editor.

Inhaltsverzeichnis

1 Einleitung ... 11
1.1 Motivation ... 11
1.2 Zielsetzung ... 11
1.3 Aufbau der Arbeit ... 12
2 Grundlagen ... 13
2.1 Allgemein ... 13
2.1.1 HTTP ... 13
2.1.2 JSON ... 15
2.1.3 YAML ... 16
2.1.4 JSON Schema ... 16
2.1.5 OpenAPI und Swagger ... 17
2.1.6 REST ... 17
2.2 Umsetzung ... 17
2.2.1 JavaScript ... 17
2.2.2 Node.js und npm ... 17
2.2.3 HTML und DOM ... 18
2.2.4 CSS und Less ... 19
2.2.5 jQuery ... 20
2.2.6 Electron ... 20
2.3 Fazit ... 20

3 Analyse 21
3.1 OpenAPI-Struktur 21
3.1.1 Parameters 21
3.1.2 RequestBody 22
3.1.3 Components und $ref 22
3.1.4 Example(s) 23
3.2 Vergleichbare Projekte 25
3.2.1 Swagger UI 25
3.2.2 Swagger Editor 28
3.2.3 Postman und Insomnia 30
3.3 Anforderungen 35
3.4 Fazit 37
4 Spezifikation 38
4.1 Szenarien 38
4.1.1 OpenAPI bearbeiten 39
4.1.2 Anfrage senden 40
4.1.3 Test erstellen 41
4.1.4 Anfragen verketten 42
4.2 Benutzeroberfläche 43
4.2.1 Aufbau 43
4.2.2 Output 44
4.2.3 API-Ansicht 45
4.2.4 Formular 46
4.2.5 Test-Ansicht 50
4.2.6 Kontextmenüs 51
4.2.7 Pop-ups 54
4.3 Fazit 57

5 Architektur 58
5.1 Lösungsstrategie 58
5.2 Sichten 59
5.2.1 Bausteinsicht 59
5.2.2 Innensicht 60
5.2.3 Laufzeitsicht 64
5.3 Entwurfsentscheidungen 71
5.4 Fazit 71
6 Umsetzung 72
6.1 Verwendete npm-Pakete 72
6.1.1 Implementierung 72
6.1.2 Testumgebung 77
6.2 UI 78
6.2.1 Elementerzeugung 78
6.2.2 Elementzugriff 80
6.2.3 Button-Manager 80
6.2.4 Theme 81
6.3 Common 82
6.3.1 Config 82
6.3.2 HTTP-Request 82
6.3.3 Object-Wrapper 83
6.3.4 YAML-Wrapper 84
6.4 OpenAPI 87
6.4.1 Validierung 87
6.4.2 Refactoring 88
6.4.3 Resolve 89

6.5 TestAPI....91
6.5.1 Validierung....91
6.5.2 Datenverkettung....92
6.5.3 Testausführung....94
6.5.4 Testausgabe....94
6.6 Unit-Tests....96
6.6.1 Mocha-Tests....96
6.6.2 Data-Driven Unit-Tests....97
6.7 Fazit....97
7 Fazit....98
7.1 Zusammenfassung....98
7.2 Evaluierung....99
7.2.1 OpenAPI-IDE....99
7.2.2 Data-Driven Unit-Tests....100
7.2.3 Fazit....101
7.3 Ausblick....102

Abbildungsverzeichnis

2.1 HTTP-Kommunikation ... 13
3.1 Path Parameters (vgl. SmartBear Software 2019e) ... 21
3.2 Describing Request Body (vgl. SmartBear Software 2019f) ... 22
3.3 Reusable Bodies (vgl. SmartBear Software 2019g) ... 22
3.4 Example in Object ... 23
3.5 Example in Property ... 23
3.6 Request Body Examples (vgl. SmartBear Software 2019k) ... 24
3.7 Parameter Examples (vgl. SmartBear Software 2019l) ... 24
3.8 Swagger UI: Schnittstellen-Darstellung ... 25
3.9 Swagger UI: Dokumentationsansicht ... 26
3.10 Swagger UI: "Content-Typ"-Auswahl ... 26
3.11 Swagger Editor: Darstellung ... 28
3.12 Swagger UI: Jump to definition (Schnittstelle) ... 28
3.13 Swagger UI: Jump to definition (Schema) ... 28
3.14 Refaktorisierung vorher und nachher ... 29
3.15 Postman: Grafische Benutzeroberfläche ... 30
3.16 Insomnia: Grafische Benutzeroberfläche ... 31
3.17 Postman: Test scripts (vgl. Postman inc. 2020b) ... 32
3.18 Postman: Globale Variable setzen ... 32
3.19 Postman: Anfrage und Verkettung ... 33
3.20 Insomnia: Verkettung aktivieren und Pop-up öffnen ... 34
3.21 Insomnia: Pop-up zum Bearbeiten der Verkettung ... 34
4.1 OpenAPI-IDE: Drei Spalten Übersicht ... 43
4.2 OpenAPI-IDE: Schnittstellen-Bereich ... 43
4.3 OpenAPI-IDE: Output-Spalte der API-Ansicht ... 44
4.4 OpenAPI-IDE: Output-Spalte der Test-Ansicht ... 44
4.5 OpenAPI-IDE: Fehlerbehandlung ... 45
4.6 OpenAPI-IDE: API-Ansicht ... 45
4.7 OpenAPI-IDE: Formular ... 46
4.8 Element für Number ... 47

4.9 Element für String ... 47
4.10 Element für Boolean ... 47
4.11 Element für Object ... 48
4.12 Element für Array ... 48
4.13 Element für Enum (Einfachauswahl) ... 49
4.14 Element für Enum (Mehrfachauswahl) ... 49
4.15 OpenAPI-IDE: Test-Ansicht ... 50
4.16 OpenAPI-IDE: Kontextmenü für die Schnittstellen ... 51
4.17 OpenAPI-IDE: Kontextmenü für die Tests ... 51
4.18 OpenAPI-IDE: Kontextmenü für die Editor-Zeilen ... 52
4.19 OpenAPI-IDE: NULL-Element ... 53
4.20 OpenAPI-IDE: Kontextmenü für die Elemente ... 53
4.21 OpenAPI-IDE: Pop-up für die Einstellungen ... 54
4.22 OpenAPI-IDE: Pop-up für die Umstrukturierung ... 55
4.23 OpenAPI-IDE: Pop-up zum Speichern eines Beispiels ... 55
4.24 OpenAPI-IDE: Pop-up zum Laden eines Beispiels ... 56
4.25 OpenAPI-IDE: Pop-up zum Speichern eines Tests ... 56
4.26 OpenAPI-IDE: Pop-up zum Bearbeiten eines Tests ... 57
4.27 OpenAPI-IDE: Pop-up zum Schließen der Applikation ... 57
5.1 Bausteinsicht ... 59
5.2 GUI-Innensicht ... 60
5.3 Core-Innensicht ... 61
5.4 Das request-Objekt ... 62
5.5 Das response-Objekt ... 62
5.6 TestAPI-Struktur ... 63
5.7 Beispiel für den Aufbau eines Sequenzdiagramms ... 64
5.8 Sequenzdiagramm: OpenAPI-Definition laden ... 65
5.9 Sequenzdiagramm: Sidebar initialisieren ... 66
5.10 Sequenzdiagramm: Formular initialisieren Teil 1 ... 67
5.11 Sequenzdiagramm: Formular initialisieren Teil 2 ... 68
5.12 Sequenzdiagramm: Formular bearbeiten ... 69
5.13 Sequenzdiagramm: Anfrage senden ... 70
6.1 Electron: Inter-Process-Communication ... 73
6.2 Konfiguration für die Erstellung der Stand-Alone-Dateien ... 74
6.3 Kommandozeilenparameter ... 75
6.4 Ace-Editor im Node-Projekt ... 76
6.5 Umwandlung der parameters zu JSON Schema ... 77
6.6 Formular: PropertyContainer-Klasse ... 78
6.7 Formular: PropertyContainer-HTML ... 78
6.8 UI: Template-Klasse ... 79
6.9 UI: ButtonManager-Klasse ... 80
6.10 Dark- und Light-Theme ... 81

6.11 Common: Config-Klasse ... 82
6.12 Common: HTTPRequest-Klasse ... 82
6.13 Common: ObjectWrap-Klasse ... 83
6.14 ObjectWrap: getValue-Funktion ... 83
6.15 Common: YamlWrap-Klasse ... 84
6.16 Beispiel: AST für YAML-Text ... 85
6.17 YamlWrap: getPositionToPath-Funktion ... 86
6.18 OpenAPI: Validate-Klasse ... 87
6.19 Validate: orderAndValidateCustomObject-Beispiel ... 88
6.20 OpenAPI: Refactor-Klasse ... 88
6.21 Refaktorisierungs-Kategorien ... 88
6.22 OpenAPI: Resolve-Klasse ... 89
6.23 Resolve: getParameter-Funktion ... 90
6.24 TestAPI: validateTestAPI-Funktion ... 91
6.25 TestAPI: getTestIds-Funktion ... 92
6.26 TestAPI: getTestIdPaths-Funktion ... 93
6.27 TestAPI: Abfolge der Testausführung ... 94
6.28 Output: Copy $ref ... 95
6.29 YamlWrap: addUniqueRootId-Funktion ... 95
6.30 YamlWrap: removeUniqueRootIdFromRef-Funktion ... 95
6.31 Mocha-Tests ... 96
6.32 Data-Driven Tests: Aufbau und Erzeugung ... 97
6.33 Data-Driven Tests: Ausschnitt der Testergebnisse ... 97
7.1 OpenAPI-IDE: Mouse-Hover Formularattribut ... 99
7.2 Vorschau der OpenAPI-IDE ... 101

1 Einleitung

1.1 Motivation

Die Entwicklung mittels einer Microservices-Architektur ist heutzutage eine beliebte Herangehensweise für die Umsetzung von modularen Systemen. Ein Microservice folgt der Unix-Philosophie „Mache eine Sache und mache sie gut“ (vgl. McIlroy 1978). Die Umsetzung von Microservices erfolgt oft mit einer API-Definition. Diese beschreibt, welche Datenstruktur abhängig von der angesprochenen Schnittstelle geliefert wird. Die API sollte möglichst vollständig sein, da eine Anpassung der veröffentlichten Schnittstelle Einfluss auf alle Benutzer hätte. Bei der Definierung einer API ist daher frühes Testen wichtig, welches im Rahmen dieser Arbeit mithilfe der OpenAPI umgesetzt wird.

1.2 Zielsetzung

Ziel der Arbeit ist die Erstellung eines Werkzeugs, zum Testen einer API, bei dem die Funktionalitäten des Swagger Editors auf Basis einer eigenen Umsetzung (OpenAPI-IDE) verbessert und erweitert werden. Zunächst wird die grafische Benutzeroberfläche aus Gründen der Übersicht in drei Spalten geteilt: Anfrage, Formular und Ergebnis. Für die Erzeugung der Testdaten wird anhand der OpenAPI-Definition ein Formular generiert, welches vom Benutzer ausgefüllt werden kann. Das Formular wird bereits bei der Eingabe der Daten auf Unstimmigkeiten anhand der OpenAPI geprüft. Außerdem werden manuelle Anfragen und automatisierte Tests mit optionaler Verkettung von Daten ermöglicht.

Abgrenzung

Die OpenAPI hat umfangreich Funktionalitäten, die nicht alle im Rahmen dieser Arbeit umgesetzt werden können. Das Ziel ist eine möglichst breite Abdeckung der Beispiel API "Swagger Petstore" (vgl. SmartBear Software 2019a).

Der Petstore nutzt überwiegend den Content-Typ "application/json", deshalb wird nur dieser vollständig unterstützt. Die Authentifizierung und Autorisierung (security) wird nicht beachtet (vgl. SmartBear Software 2019b). Außerdem wird es nicht möglich sein, die Schema mit "oneOf, anyOf, allOf, not" zu kombinieren (vgl. SmartBear Software 2019c).

1.3 Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel gegliedert. In Kapitel 1 „Einleitung“ wird die Motivation und Zielsetzung der Arbeit erläutert. Das „Grundlagen“ Kapitel beschreibt zunächst Protokolle, die zum Nachvollziehen dieser Arbeit empfohlen werden. Im Anschluss werden die Werkzeuge für die Umsetzung des Systems vorgestellt. Kapitel 3 „Analyse“ untersucht die OpenAPI-Struktur und vergleicht einige Projekte, die häufig zum Testen von HTTP-Schnittstellen zum Einsatz kommen. Anschließend werden die Anforderungen an das Zielsystem definiert. In Kapitel 4 „Spezifikation“ werden die Anwendungsszenarien und der Aufbau der grafischen Benutzeroberfläche festgelegt. Das „Architektur“ Kapitel zeigt die Lösungsstrategie, die verschiedenen Sichten und die Entwurfsentscheidungen. Kapitel 6 „Umsetzung“ beschreibt die eingesetzten Pakete, die Komponenten und den Aufbau der Unit-Tests für das umgesetzte System. In Kapitel 7 „Fazit“ wird eine Zusammenfassung der Arbeit, die Evaluierung der Umsetzung und ein Ausblick auf die Weiterentwicklung des Systems geliefert.

2 Grundlagen

Dieses Kapitel beschreibt im allgemeinen Teil, die Grundlagen zu HTTP-basierten Microservices und anschließend die Werkzeuge für die Umsetzung des Systems.

2.1 Allgemein

In diesem Abschnitt werden Grundlagen erläutert, die zum Nachvollziehen dieser Arbeit empfohlen werden. Dazu gehören folgende Protokolle: HTTP, JSON, YAML, JSON Schema, OpenAPI und REST.

2.1.1 HTTP

Hypertext Transfer Protocol (HTTP) ist ein Client/Server-Protokoll zum Austauschen von Informationen anhand einer Anfrage und Antwort (Abbildung 2.1). In diesem Unterabschnitt werden die im Rahmen der Bachelorarbeit benötigten Kernfunktionalitäten von HTTP1/1 beschrieben (vgl. Fielding u. a. 1999).

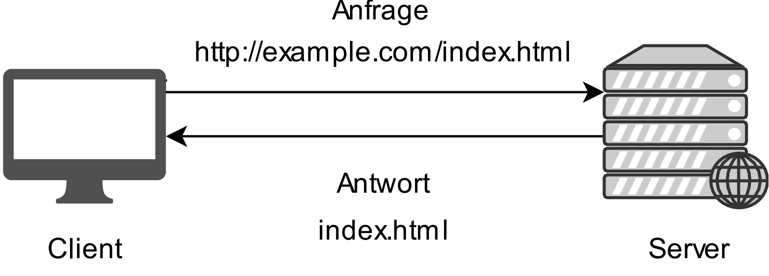


Abbildung 2.1: HTTP-Kommunikation

Anfrage (Request)
Die folgenden Punkte beschreiben eine Anfrage (vgl. Fielding u. a. 1999, Kapitel 5).

- Ein Uniform Resource Locator (URL) ist eine Wegbeschreibung, die zu einer Ressource auf einem Server führt (vgl. Fielding u. a. 1999, Kapitel 3.2.2).

  Beispiel:

URL: https://petstore.swagger.io/v2/pet/findByStatus?status=pending&more=params	
scheme	Beschreibt das eingesetzte Protokoll (http/https, ftp, file, …).
host	Adresse des Servers.
path	Der Pfad zur Ressource auf dem Server.
query	Anfrageparameter, zum Beispiel für das Filtern von Ressourcen.

- Die Methode vermittelt dem Server, welche Operation auf die gefundene Ressource angewendet werden soll (vgl. Fielding u. a. 1999, Kapitel 5.1.1).

  Beispiel:

Methode	
PUT	Erstellt oder ersetzt die Ressource.
GET	Ruft die Ressource ab.
POST	Ersetzt die Ressource.
DELETE	Löscht die Ressource.

- Der Header enthält Metadaten, die zum Beispiel den Typ und die Länge vom Body beschreiben (vgl. Fielding u. a. 1999, Kapitel 5.3, 7.1).

  Beispiel:

Header

```
Content-Type: application/json
Content-Length: 42
```

- Der Body enthält eine Ressource, die an den Server geschickt wird. Zum Beispiel ein Textdokument oder Bild (vgl. Fielding u. a. 1999, Kapitel 7.2).

  Beispiel:

Body

```
{
   "summary": "Simple JSON Body Example"
}
```

Antwort (Response)
Die Antwort ist der Anfrage strukturell ähnlich, ein besonderer Unterschied ist der Anfragezustand, welcher in der "Status-Line" mit dem "Status-Code" und der zugehörigen Abkürzung angegeben wird (vgl. Fielding u. a. 1999, Kapitel 6).

Beispiel:

Status-Line	HTTP/1.1 200 OK	
Status-Code	**Abkürzung**	**Beschreibung**
…		
200	OK	Die Anfrage wurde erfolgreich ausgeführt.
201	Created	Die Ressource wurde erstellt.
…		
400	Bad Request	Anfrage enthält Fehler.
404	Not Found	Ressource nicht vorhanden.
…		

2.1.2 JSON

JavaScript Object Notation (JSON) ist ein Klartext-Format, welches von Menschen und Computern verarbeitet werden kann. Es ist Programmiersprachen unabhängig und wird häufig zum Austauschen von Ressourcen im Internet eingesetzt.
Eine Eigenschaft (Paar) im JSON-Objekt hat folgenden Aufbau: "Schlüssel": Wert, wobei die Paare mit einem Komma getrennt werden. Der Schlüssel ist ein beliebiger String und wird im Text dieser Arbeit blau hervorgehoben. Der Wert kann Folgendes enthalten: object, array, number, string, true, false oder null (vgl. ECMA 2017). Für eine Steigerung der Lesbarkeit kann ein JSON-Text mittels "Pretty Print" formatiert werden.

Beispiel: JSON-Text

Mit Pretty Print

```
 1 {
 2   "string": "Mein Text",
 3   "number": 1.0,
 4   "array": [
 5     42,
 6     "Text"
 7   ],
 8   "object": {
 9     "text": "Hallo"
10   },
11   "boolean": true
12 }
```

Ohne Pretty Print (Abgekürzt)

```
{"string":"Mein Text","number":1.0}
```

2.1.3 YAML

YAML Ain’t Markup Language (YAML) ist mit JSON vergleichbar, es wird überwiegend für Konfigurationsdateien verwendet, da Kommentare möglich sind und weniger sichtbare Zeichen benötigt werden (Tabs anstatt Klammern). Jeder JSON-Text lässt sich in einen äquivalenten YAML-Text umwandeln, jedoch kann die umgekehrte Transformation zu einem Verlust von Informationen führen (vgl. Ben-Kiki u. a. 2009).

Beispiel: YAML-Text

```
1 # Kommentar
2 array:
3   - 42
4   - Text
5 boolean: true
6 number: 1.0
7 object:
8   text: Hallo
9 string: "Mein Text"
```

2.1.4 JSON Schema

JSON Schema ist eine Organisation die "drafts" spezifiziert, welche zum Beispiel eingesetzt werden, um JSON-Objekte anhand eines Schemas zu validieren und Daten wie Programmcode, Formulare, … zu generieren (vgl. JSON Schema 2019a).

Beispiel: Login-Formular generieren (vgl. rjsf-team 2020)

```
 1 {
 2   "type": "object", "required": ["email", "password"],
 3   "properties": {
 4     "email": {
 5       "type": "string", "title": "Email"
 6       "format": "email",
 7     },
 8     "password": {
 9       "type": "string", "title": "Password",
10       "minLength": 3
11     }
12   }
13 }
```

2.1.5 OpenAPI und Swagger

Die OpenAPI spezifiziert eine Struktur, womit die Erstellung einer HTTP-basierten API ermöglicht wird, welche im JSON- oder YAML-Format dargestellt werden kann. Bei der Beschreibung der Ressourcen (Schema) orientiert sich die OpenAPI überwiegend an der Spezifikation von JSON Schema (vgl. The Linux Foundation u. a. 2018).
Swagger ist eine beliebte Werkzeugsammlung zum Arbeiten mit der OpenAPI 3.0 und Swagger 2.0 (vgl. Pinkham 2017).

2.1.6 REST

Representational State Transfer (REST) ist ein Architekturstil, der verschiedene Bedingungen festlegt, die erfüllt werden müssen, um die Vorteile von REST zu erhalten (vgl. Fielding 2000). Das HTTP-Protokoll und REST sind eng verwandt und werden dadurch gerne verwechselt, die OpenAPI wirbt mit REST, obwohl HTTP gemeint ist (vgl. Miller 2016).

2.2 Umsetzung

In diesem Abschnitt werden die relevanten Werkzeuge für die Umsetzung des Systems vorgestellt, dazu gehört die Programmiersprache, Laufzeitumgebung, der Paketverwalter und einige Pakete.

2.2.1 JavaScript

> „JavaScript (JS) ist eine leichtgewichtige, interpretierte oder JIT-übersetzte Sprache mit First-Class-Funktion. Bekannt ist sie hauptsächlich als Skriptsprache für Webseiten geworden, […] Man sollte JavaScript nicht mit der Programmiersprache Java verwechseln. […] Die beiden Programmiersprachen haben eine sehr unterschiedliche Syntax, Semantik und Verwendung.“ (Mozilla.org u. a. 2019)

2.2.2 Node.js und npm

Node.js (Node) ist eine serverseitige JavaScript-Laufzeitumgebung, welche zum Beispiel mit PHP und ASP.NET vergleichbar ist. Einen Einblick in die Laufzeitumgebung gab es auf der JavaScript-Konferenz 2014 (vgl. Roberts 2014).

> „Als asynchrone, Event-basierte Laufzeitumgebung wurde Node speziell für die Entwicklung von skalierbaren Netzwerkanwendungen entworfen. […] Dies steht im Gegensatz zu den heutzutage üblichen Modellen für Nebenläufigkeit, bei denen Threads des Betriebssystems genutzt werden. Thread-basiertes Networking ist vergleichsweise ineffizient und sehr schwer umzusetzen. Zudem müssen sich Node-Nutzer nicht um Deadlocks im Prozess sorgen, da es keine Blockierung gibt. Fast keine Funktion in Node führt direkt I/O-Operationen aus, daher wird der Prozess nie blockiert. Da nichts blockiert, können mit Node sinnvoll skalierbare Systeme entwickelt werden.“ (Node.js u. a. 2019)

Der Node package manager (npm) ist ein Paketverwalter, der die Entwicklung von Node-basierten Anwendungen unterstützt. Pakete können über ein Command Line Interface (CLI) und der im Projekt angelegten Datei "package.json" verwaltet werden (vgl. npm, Inc. 2020).

2.2.3 HTML und DOM

Hypertext Markup Language (HTML) ist eine Auszeichnungssprache, welche die Struktur und den Inhalt einer Webseite mittels HTML-Tags beschreibt (vgl. Refsnes Data 2019a).
Document Object Model (DOM) ist eine Schnittstelle, die zum Beispiel von JavaScript zur Verfügung gestellt wird, sie ermöglicht das Abrufen und Manipulieren von HTML-Strukturen (vgl. WHATWG 2020).

Das folgende Beispiel zeigt ein HTML-Dokument mit einem Eingabefeld, dem mittels DOM der Wert "My Name" zugewiesen wird.

Beispiel: Index.html

```
1  <!DOCTYPE html>
2  <html>
3
4    <head>
5      <link rel="stylesheet" type="text/css" href="style.css">
6      <title>Name Text Input</title>
7    </head>
8
9    <body>
10     Name <input id="name-textfield" type="text">
11   </body>
12
13   <script>
14     document.getElementById("name-textfield").value = "My Name"
15   </script>
16
17 </html>
```

2.2.4 CSS und Less

Cascading Style Sheets (CSS) beschreibt das Aussehen der HTML-Elemente, dazu gehört zum Beispiel die Breite, Höhe, Farbe und Schriftart (vgl. Refsnes Data 2019b).
Um eine monolithische Entwicklung zu meiden, kann die CSS-Definition in einer separaten CSS-Datei verwaltet und in der HTML-Datei verlinkt werden (siehe S.18 index.html Zeile 5).
Das folgende Beispiel setzt die Hintergrundfarbe des HTML-Bereichs und der Eingabefelder.

Beispiel: style.css

```
1 html {
2   background: #90caf9;
3 }
4
5 input {
6   background: #90caf9;
7 }
```

Leaner Style Sheets (Less) erweitert die Möglichkeiten von CSS mit zum Beispiel vereinfachter Variablen-Handhabung und Mixins. Abhängig vom Anwendungsfall kann Less während der Kompilierung oder zur Laufzeit in das CSS-Format gewandelt werden (vgl. Sellier u. a. 2019).
Im folgenden Beispiel wird eine Variable für die Farbe "#90caf9" erzeugt, wodurch eine Anpassung an einer Stelle ermöglicht wird, welche zur Laufzeit oder statisch in der Datei "style.less" durchgeführt werden kann.

Beispiel: style.less

```
1  @bgColor: #90caf9;
2
3  .bgColor {
4    background: @bgColor;
5  }
6
7  html {
8    .bgColor;
9  }
10
11 input {
12   .bgColor;
13 }
```

2.2.5 jQuery

jQuery ist eine Bibliothek für JavaScript, sie vereinfacht das Arbeiten mit dem DOM durch eine kompakte Schreibweise und vieler Hilfsfunktionen (vgl. The jQuery Foundation 2019a).

Beispiel:

	JavaScript	jQuery
1	`// Get Elem.`	
2	`var btn = document.getElementById('btn')`	`$('#btn')`
3	`// Hide Elem.`	
4	`btn.style.display = 'none'`	`$('#btn').hide()`

2.2.6 Electron

> „Electron ist eine Open-Source-Bibliothek, die von GitHub für das Erstellen von plattformübergreifenden Desktop-Anwendungen mit HTML, CSS und JavaScript, entwickelt wurde. Electron vollbringt dies durch das Kombinieren von Chromium und Node.js in einem einfachen Schritt, zum Erstellen von Apps für Mac, Windows und Linux.“ (GitHub Inc. 2019a)

2.3 Fazit

Dieses Kapitel hat die Grundlagen vorgestellt, welche zum Erreichen der Zielsetzung eingesetzt werden. Der Allgemeine Teil wird zum Nachvollziehen der Bachelorarbeit empfohlen, die Werkzeuge für die Umsetzung kommen in Kapitel 6 zum Einsatz.

3 Analyse

In diesem Kapitel wird zunächst die OpenAPI-Struktur analysiert, wodurch zeitintensive Szenarien und das Definieren von Beispieldaten verdeutlicht wird. Anschließend werden Vor- und Nachteile von Swagger Editor, Postman und Insomnia betrachtet. Zum Schluss werden die Anforderungen an das Zielsystem definiert und das Kapitel mit einem Fazit zusammengefasst.

3.1 OpenAPI-Struktur

In diesem Abschnitt werden Strukturen der OpenAPI beschrieben, die für das Definieren von Schnittstellen benötigt werden, wobei der Fokus besonders auf $ref und example(s) liegt.

3.1.1 Parameters

Der Parametertyp wird mit der in-Eigenschaft festgelegt und kann folgende Werte annehmen: query, header, path und cookie (vgl. SmartBear Software 2019d). In Abbildung 3.1 wird ein Beispiel für einen Parameter vom Typ "path" gezeigt. Bei diesem ist besonders wichtig, dass der name (Zeile 6) mit der {id} übereinstimmt (Zeile 2).

```
1  paths:
2    /users/{id}:
3      get:
4        parameters:
5          - in: path
6            name: id # Note the name is the same as in the path
7            required: true
8            schema:
9              type: integer
10           description: The user ID
```

Abbildung 3.1: Path Parameters (vgl. SmartBear Software 2019e)

3.1.2 RequestBody

Unter content wird die Definierung von verschiedenen Content-Typen ermöglicht. Diese können das gleiche oder verschiedene Schema enthalten (Abbildung 3.2 Zeile 7, 13).

```
 1 paths:
 2   /pets:
 3     post:
 4       requestBody:
 5         content:
 6           application/json: # Content-Type
 7             schema:
 8               type: object
 9               properties:
10                 name:
11                   type: string
12           application/xml:
13             schema:
14               …# Enthält gleiches Schema wie application/json Zeile 7
```

Abbildung 3.2: Describing Request Body (vgl. SmartBear Software 2019f)

3.1.3 Components und $ref

Bei der Definierung von Content-Typen mit dem gleichen Schema entsteht doppelter Code, welcher mit JSON Reference ($ref-Pfad) minimiert werden kann (vgl. Bryan und Zyp 2012). Das folgende Beispiel zeigt verschiedene Content-Typen, welche jeweils auf dasselbe Schema referenzieren (Abbildung 3.3 Zeile 9, 12).

```
 1 paths:
 2   /pets:
 3     post:
 4       summary: Add a new pet
 5       requestBody:
 6         content:
 7           application/json:
 8             schema:
 9               $ref: '#/components/schemas/Pet' # Reference to an object
10           application/xml:
11             schema:
12               $ref: '#/components/schemas/Pet'
```

Abbildung 3.3: Reusable Bodies (vgl. SmartBear Software 2019g)

Mit $ref können nicht nur Schemas verlinkt werden, sondern zum Beispiel auch parameters oder ganze requestBodies (vgl. SmartBear Software 2019h).
Die Verlinkung hat den Nachteil, dass viel gescrollt werden muss, um die Objekte zu finden. Außerdem ist die Wahrscheinlichkeit hoch, dass der Benutzer kurz darauf wieder die Schnittstelle betrachten möchte. Im Moment ist $ref noch ein zeitraubender Faktor, der mit den Funktionen "Jump to definition" und "Jump back" optimiert werden kann.

3.1.4 Example(s)

Der Benutzer kann Beispieldaten in der OpenAPI definieren, die ein Client automatisch einsetzen kann, um eine Anfrage mit Testdaten zu füllen. Das example kann auf Objekt-Ebene für ausgewählte properties definiert werden (Abbildung 3.4 Zeile 10) oder im jeweiligen Attribut (Abbildung 3.5 Zeile 9, 12).

```
 1 components:
 2   schemas:
 3     User:       # Schema name
 4       type: object
 5       properties:
 6         id:
 7           type: integer
 8         name:
 9           type: string
10       example: # Object-level example
11         id: 1
12         name: Jessica Smith
```

Abbildung 3.4: Example in Object

```
 1 components:
 2   schemas:
 3     User:             # Schema name
 4       type: object
 5       properties:
 6         id:
 7           type: integer
 8           format: int64
 9           example: 1 # Property example
10         name:
11           type: string
12           example: New order
```

Abbildung 3.5: Example in Property

(vgl. SmartBear Software 2019i)

Der Petstore setzt oft default in den Schemadefinitionen ein, welches nicht mit example verwechselt werden sollte, da default vom Server eingesetzt wird, wenn der Client einen optionalen Wert nicht angegeben hat (vgl. SmartBear Software 2019j).

Durch manuelles Testen hat sich herausgestellt, dass bei der OpenAPI-Validierung keine Überprüfung der example(s) durchgeführt wird. Zum Beispiel würde eine Umbenennung der id oder eine type-Änderung (Abbildung 3.4 Zeile 6, 7) zu einem fehlerhaften example ohne Hinweis führen (Abbildung 3.4 Zeile 11).

Die examples sind Werte, die der Benutzer manuell wählen kann, um eine Anfrage mit Testdaten zu füllen. Das folgende Beispiel zeigt anhand eines requestBody, wie examples definiert werden können (Abbildung 3.6 Zeile 9).

```
 1 paths:
 2   /users:
 3     post:
 4       requestBody:
 5         content:
 6           application/json:
 7             schema:
 8               $ref: '#/components/schemas/User'
 9             examples:
10               Jessica: # Example 1
11                 value:
12                   id: 10
13               Ron:     # Example 2
14                 value:
15                   id: 11
```

Abbildung 3.6: Request Body Examples (vgl. SmartBear Software 2019k)

Ein weiteres Beispiel sind examples für Parameter (Abbildung 3.7). Die Bindung der examples an einen Parameter scheint zunächst feingranular, dies ist jedoch auf die Definierung der parameters zurückzuführen. Die Wahrscheinlichkeit ist hoch, dass ein Parameter in weiteren Schnittstellen benötigt wird, aber unwahrscheinlich, dass die Kombination der Parameter auch dieselbe ist, deshalb ist ein Schema für die jeweiligen Typen nicht vorteilhaft.

```
16       parameters:
17         - in: query
18           name: limit
19           schema:
20             type: integer
21           examples:       # Multiple examples
22             zero:         # Distinct name
23               value: 0    # Example value
24               summary: A sample limit value # Optional description
25             max:
26               value: 50
```

Abbildung 3.7: Parameter Examples (vgl. SmartBear Software 2019l)

3.2 Vergleichbare Projekte

In diesem Abschnitt werden drei beliebte Umsetzungen vorgestellt, die zum Arbeiten mit HTTP-Schnittstellen eingesetzt werden. Es wird auf Vor- und Nachteile eingegangen, um anhand dieser, mögliche Alternativen zu entwickeln. Zunächst wird die Swagger UI und der Swagger Editor gezeigt, wobei die grafische Benutzeroberfläche, das Formular und die OpenAPI spezifischen Funktionalitäten betrachtet werden. Im Anschluss folgt eine Analyse von Postman und Insomnia, hier liegt das Testen von Schnittstellen im Fokus.

3.2.1 Swagger UI

Die Swagger UI ist eine grafische Benutzeroberfläche, die mithilfe der OpenAPI-Definition eine Dokumentation generiert, die gleichzeitig als Client zum Testen der Schnittstellen eingesetzt werden kann (vgl. SmartBear Software 2019m). Leider sind die Funktionalitäten begrenzt und die Steuerung mühsam, welches in diesem Unterabschnitt verdeutlicht wird.

Darstellung

Die Schnittstellen in der Swagger UI werden als Liste von aufklappbaren Elementen dargestellt (Abbildung 3.8).

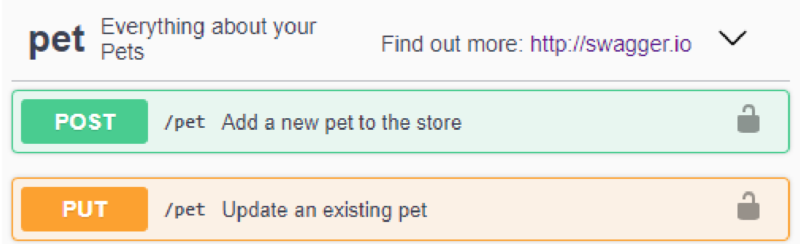


Abbildung 3.8: Swagger UI: Schnittstellen-Darstellung

Durch das Aufklappen wird zunächst die Dokumentationsansicht angezeigt (Abbildung 3.9).

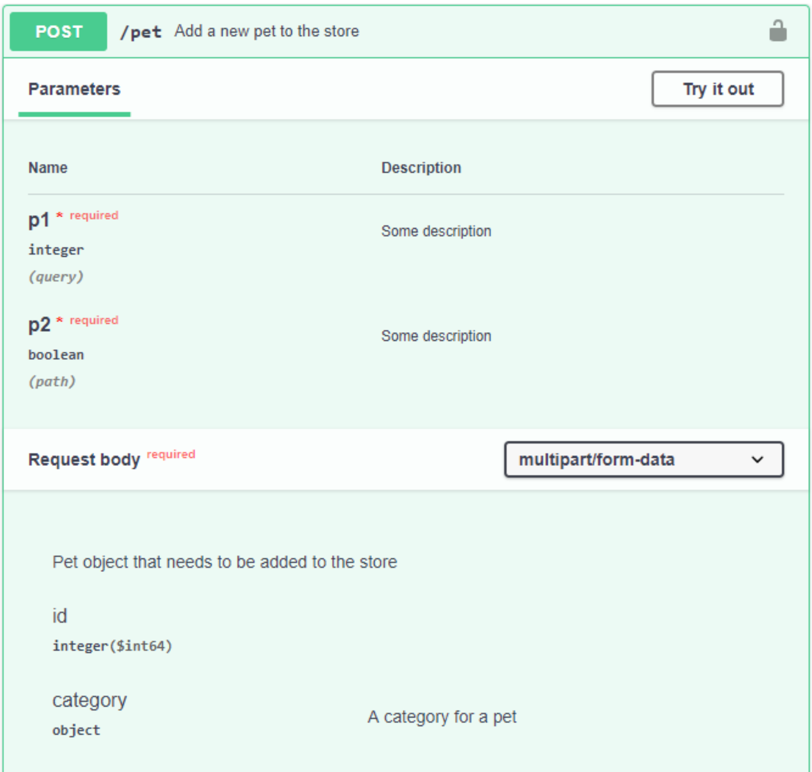


Abbildung 3.9: Swagger UI: Dokumentationsansicht

Der "Request body" hat eine Auswahlbox, womit ein Content-Typ gewählt werden kann, dadurch wird die Darstellung vom Body entsprechend angepasst (Abbildung 3.10).
Die "Parameters" haben immer dieselbe Darstellung, wie ein "Request body" mit dem Content-Typ "multipart/form-data".


```
{
  "id": 0,
  "category": {
    "name": "name",
    "id": 6
  }
}
```

Abbildung 3.10: Swagger UI: "Content-Typ"-Auswahl

Mit dem "Try it out"-Knopf (Abbildung 3.9) kann der Client-Modus aktiviert werden, wodurch die Eingabe der Anfragedaten ermöglicht wird.

Probleme:

1. **Formular:**
    - Der Typ "object" wird als einfaches Texteingabefeld ohne Syntaxprüfung angezeigt. Das Objekt könnte zusätzlich als Formular (multipart/form-data) angeboten werden.
    - Es gibt keine Möglichkeit den Body als Formular darzustellen und gleichzeitig im "application/json" Format zu senden, da der gewählte Content-Typ entscheidet wie der Body dargestellt wird.
      Eine mögliche Lösung, wäre eine einheitliche Darstellung aller Daten als Formular und eine separate Anzeige der Daten im ausgewählten Content-Typ, welcher zum Senden eingesetzt wird.
    - Der Typ "array" nimmt in einem Formular bei der Erzeugung eines neuen Elements vertikalen Platz ein, wodurch alle folgenden Elemente verschoben werden. Mit einem Scrollbereich würde der vertikale Platz unverändert bleiben, solange der Benutzer keine Vergrößerung des Elements vornimmt.
2. **Semantikprüfung:**
    - Eine Überprüfung der eingegebenen Daten (Parameters/Request body) anhand der OpenAPI-Definition findet kaum statt. Beim Senden der Anfrage wird nur sichergestellt, dass alle *required-Felder einen Wert enthalten.
    - Bei der Antwort wird lediglich geprüft, ob der gelieferte Status-Code in der Schnittstelle definiert ist.
3. **Allgemeine Darstellung:**
    - Die Information "description" und "type" wird für jedes Attribut dargestellt. Diese könnten mit einem Tooltip minimiert werden, welches alle Attribut-Informationen enthält.
    - Der *required-Text könnte durch ein * ersetzt werden.
    - Der Parametertyp ist eine doppelte Information, die durch eine Trennung in verschiedene Kategorien gelöst werden könnte.
    - Die grafische Benutzeroberfläche nutzt den vorhandenen horizontalen Platz aus, wodurch große Abstände zwischen den Elementen entstehen.
    - Die Interaktion mit der Swagger UI verändert die Struktur, dadurch muss sich der Benutzer immer wieder neu orientieren.
4. **Beispieldaten:**
    - Jedes Element mit definierten examples erhält eine Dropdown-Liste (Dropdown), welche viel Platz einnimmt und keine Vorschau der Daten als Tooltip anzeigt.
5. **Verkettung:**
    - Es gibt keine Unterstützung für die Anfragen- und Datenverkettung.

3.2.2 Swagger Editor

Der Swagger Editor hat zwei Spalten (vgl. SmartBear Software 2019n), auf der rechten Seite befindet sich die Swagger UI, welche eine Echtzeitvorschau der definierten OpenAPI erzeugt. Die linke Spalte enthält einen Texteditor (vgl. Ajax.org B.V. 2020, Ace), der eine Bearbeitung der OpenAPI ermöglicht. Er liefert unter anderem folgende Funktionalitäten:

- Tastenkürzel für das Suchen, Kopieren, Ausschneiden und Einfügen von Text.
- Farblich Hervorhebung für Kontextabhängige Inhalte wie YAML, JSON, …, welches die Unterscheidung von Schlüssel, Zeichenketten, Zahlen, … vereinfacht.

Außerdem lassen sich mehrzeilige Paare einklappen, wodurch die Lesbarkeit gesteigert wird (Abbildung 3.11 Zeile 37). Die Texteinfügemarke (Cursor) ist am Ende von Zeile 34 zu sehen.

```
29 paths:
30   /pet:
31     post:
32       tags:
33         - pet
34       summary: Add a new pet
35       requestBody:
36         $ref: '#/components/requestBodies/Pet'
37       responses:
```

Abbildung 3.11: Swagger Editor: Darstellung

Probleme:

1. **Jump to definition**:
   - Die Suche nach einem verlinkten $ref-Objekt (Abbildung 3.11 Zeile 36) ist in den meisten Fällen mit viel Scrollen oder der Suchfunktion (Strg+F) verbunden. Dieser Vorgang könnte mit einem Kontextmenü-Eintrag im Editor vereinfacht werden.
   - Die Funktionalität für den Sprung zu einer Schnittstellen-Definition ist leicht zu übersehen, da der Knopf erst angezeigt wird, wenn der Mauszeiger über dem Schnittstellen-Element ist (Abbildung 3.12). Abweichend davon wird bei einem Schema der Knopf dauerhaft angezeigt (Abbildung 3.13). Die Funktionalität könnte einheitlich und dauerhaft angezeigt werden.

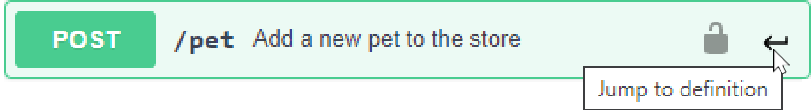


Abbildung 3.12: Swagger UI: Jump to definition (Schnittstelle)

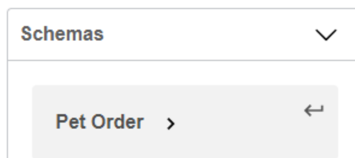


Abbildung 3.13: Swagger UI: Jump to definition (Schema)

2. **Speichern der API:**
   - Da der Swagger Editor eine Webapplikation ist, kann die API aus Sicherheitsgründen nicht automatisch in der geöffneten Datei gespeichert werden (Strg+S).
3. **Refaktorisierung**:
   - Eine manuelle Umstrukturierung eines Schemas in die components-Eigenschaft ist ein zeitraubender Vorgang. Hier könnte eine Funktionalität für das automatische Verschieben hilfreich sein (Abbildung 3.14 links vorher, rechts nachher).

	Vorher	Nachher
1	`paths:`	
2	`  /pet:`	
3	`    put:`	
4	`      requestBody:`	
5	`        content:`	
6	`          application/x-www-form-urlencoded:`	
7	`            schema:`	`schema:`
8	`              type: object`	`  $ref: '#/components/schemas/Pet'`
9	`              properties:`	
10	`                name:`	
11	`                  type: string`	
12	`      responses:`	
13	`        '405':`	
14	`          description: Invalid input`	
15	`components:`	`components:`
16	`  schemas: {}`	`  schemas:`
17		`    Pet:`
18		`      type: object`
19		`      properties:`
20		`        name:`
21		`          type: string`

Abbildung 3.14: Refaktorisierung vorher und nachher

3.2.3 Postman und Insomnia

Postman und Insomnia sind API-Entwicklungsumgebungen die sich ähneln, sie waren als Chrome-App entstanden und sind später zu einer regulären Webapplikation gewechselt, da Chrome-Apps nicht mehr unterstützt werden (vgl. Roy-Chowdhury 2016). Beide ermöglichen das Definieren von Schnittstellen und Senden der Anfragen. Postman hat umfangreiche Funktionalitäten, wie zum Beispiel das automatisierte Ausführen von Anfragen mit dem "Collection Runner" und die Testvalidierung (vgl. Postman inc. 2020a).
Insomnia hingegen konzentriert sich auf eine angenehme grafische Benutzeroberfläche mit einfacher Handhabung (vgl. Insomnia REST Client 2019a).
Im Folgenden wird auf die Benutzeroberfläche, die Testvalidierung und die Verkettung eingegangen.

Theme

Manche Applikationen bieten beim ersten Start eine Theme-Auswahl, andere ermöglichen diese Änderung nur über die Einstellungen. Postman hat als Standard das Light-Theme (Abbildung 3.15) und Insomnia das Dark-Theme (Abbildung 3.16). Beide ermöglichen eine Theme-Anpassung in den Einstellungen und haben keine Auswahl beim ersten Start.

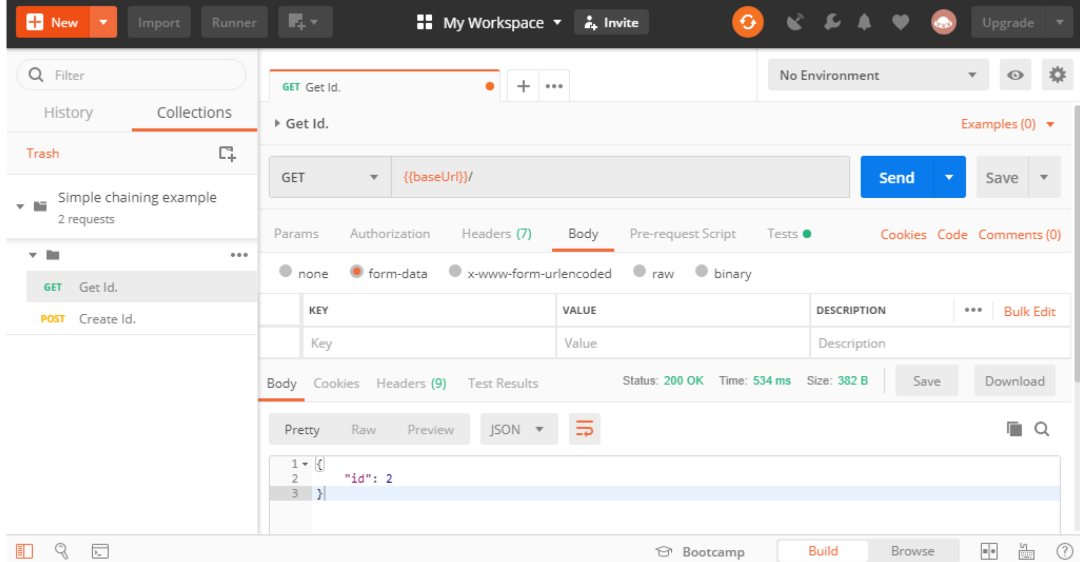


Abbildung 3.15: Postman: Grafische Benutzeroberfläche

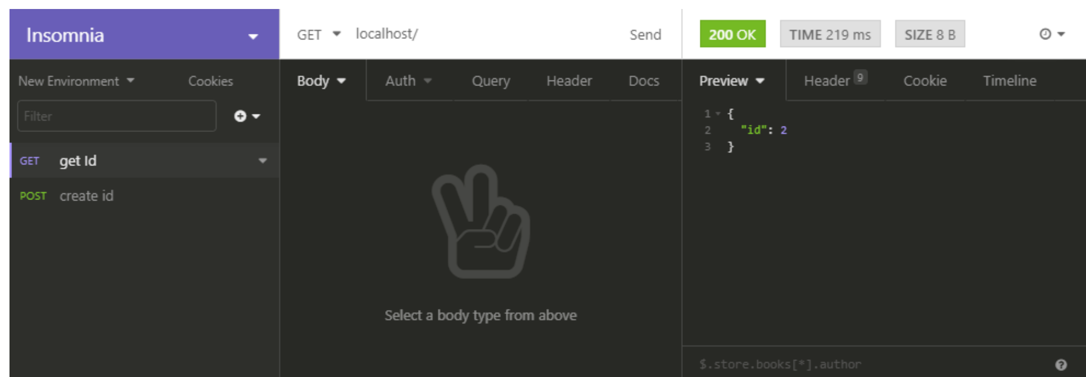


Abbildung 3.16: Insomnia: Grafische Benutzeroberfläche

Struktur

Die Standard-Struktur von Postman (Abbildung 3.15) hat zwei Spalten, links sind die Schnittstellen, rechts im oberen Teil sind die Anfragedaten und der untere Teil zeigt die Antwort. Unten rechts sind drei Symbole zu sehen, wobei "Two pane view" einen Wechsel zwischen der horizontalen und vertikalen Ansicht für die Anfrage und Antwort ermöglicht. Dies vereinfacht zum Beispiel das Arbeiten mit einer Antwort, die viel vertikalen Platz benötigt.

In Abbildung 3.16 wird die Standard-Struktur von Insomnia angezeigt, welche drei Spalten hat und in den Einstellungen geändert werden kann.

Die Unterschiede verdeutlichen die Grundidee der beiden Benutzeroberflächen, Insomnia versteckt selten benötigte Einstellungen, wohingegen Postman so viele Einstellungen wie möglich zeigt, damit eine kurzfristige Anpassung ermöglicht wird.

OpenAPI

Postman und Insomnia unterstützen das Importieren einer OpenAPI-Definition, jedoch gibt es keine Validierung der Anfrage und Antwort anhand der OpenAPI. Außerdem kann die OpenAPI-Definition nicht bearbeitet oder exportiert werden (Stand 03.05.2019).

Testvalidierung

Die Testumgebung von Postman nutzt eine JavaScript-Umgebung zum Definieren der Tests und ermöglicht dadurch eine Validierung der Antworten (vergleichbar mit Unit-Tests). Insomnia hingegen hat keine Funktionalität für die Testvalidierung.

Beispiel: Postman-Testvalidierung
Abbildung 3.17 zeigt eine bedingte Validierung, die einen Status-Code 200 erwartet. Auf der rechten Seite ist eine Hilfe in Form von Code-Schnipsel (SNIPPETS) zu sehen.

```
1 pm.test("Status code is 200", function () {
2     pm.response.to.have.status(200);
3 });
```

Test scripts are written in JavaScript, and are run after the response is received.
Learn more about tests scripts

SNIPPETS
Status code: Code is 200
Response body: Contains string
Response body: JSON value check
Response body: Is equal to a string

Abbildung 3.17: Postman: Test scripts (vgl. Postman inc. 2020b)

Problem:

- Die Testskripte werden pro Schnittstelle definiert. Dies unterstützt ein ausführliches Testen einer Schnittstelle, jedoch wird dadurch das Nachvollziehen der Abhängigkeiten und die Bearbeitung einer Variable erschwert. Hier wäre eine Übersicht aller Testskripte vorteilhaft.

Verkettung
Das Verketten von Anfrage- und Antwortdaten funktioniert bei Postman und Insomnia ähnlich. Beide nutzen einen Bereich, in dem Variablen abgelegt und anschließend mit einer speziellen Syntax abgerufen werden können.
Das folgende Beispiel führt zunächst eine GET-Schnittstelle aus, welche einen Antwort-Body mit einer "id" liefert, diese wird im Anfrage-Body der POST-Schnittstelle eingefügt.

Beispiel: Postman-Verkettung
Zunächst wird ein Skript geschrieben, dass die gewünschte Variable nach dem Aufruf der GET-Schnittstelle global ablegt (Abbildung 3.18):

- Zeile 1 wandelt den Antwort-Body in ein JavaScript-Objekt um.
- Zeile 2 legt die "id"-Eigenschaft global unter dem Namen "id" ab.

Das Skript wird ausgeführt, sobald die Antwort der GET-Schnittstelle vorhanden ist.

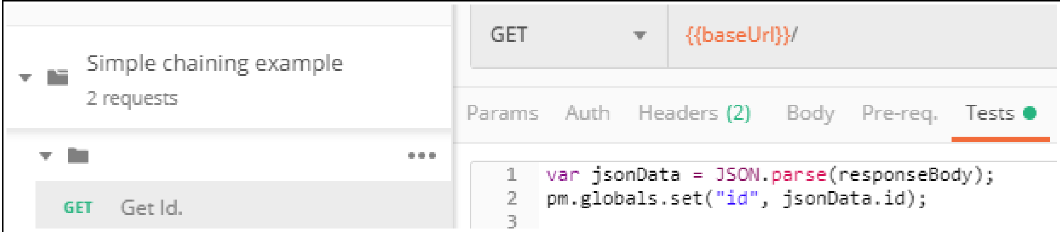


Abbildung 3.18: Postman: Globale Variable setzen

Die globalen Variablen können in den Einstellungen unter "Globals" eingesehen werden. Die Variable "id" kann mit {{id}} abgerufen werden (siehe Abbildung 3.19 Zeile 2). Eine Anfrage erfolgt mit dem Send-Knopf Send, welcher für jede Anfrage manuell gedrückt werden muss. Alternativ können Anfragen einer Sammlung automatisiert mit dem "Collection Runner" ausgeführt werden, dieser ist über den Runner-Knopf erreichbar (vgl. Asthana 2014).

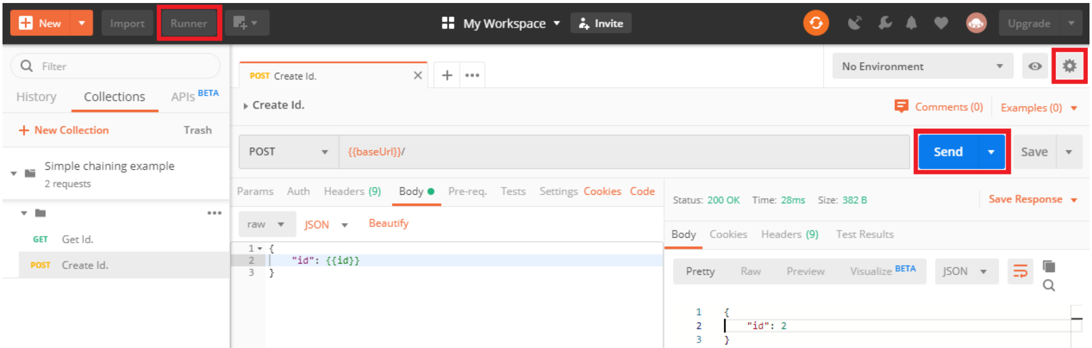


Abbildung 3.19: Postman: Anfrage und Verkettung

Probleme:

- Falls die "id" nicht vorhanden ist, wird die Anfrage ohne einen Wert gesendet. Der Body sieht dann wie folgt aus { id: } und ist kein gültiger JSON. Bei mehreren Werten müsste der fehlende Wert manuell gesucht werden. In diesem Fall könnte ein Hinweis auf den fehlenden Wert hilfreich sein.
- Der "Collection Runner" führt die Anfragen synchron aus, es gibt keine Auswahl, um bestimmt Anfragen asynchron auszuführen.

Beispiel: Insomnia-Verkettung
Die verketteten Daten können an der benötigten Stelle abgerufen werden, indem zum Beispiel "response" im JSON-Bereich geschrieben wird (Abbildung 3.20, links). Dadurch wird ein Kontextmenü angezeigt, bei dem "Response = Body Attribute" gewählt werden kann.
Durch einen Linksklick auf den roten Bereich (Abbildung 3.20, rechts) wird ein Dialogfenster (Pop-up) geöffnet (Abbildung 3.21).

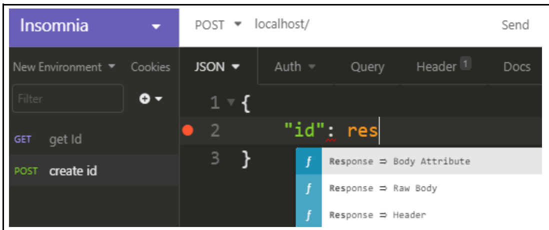


Abbildung 3.20: Insomnia: Verkettung aktivieren und Pop-up öffnen

In Abbildung 3.21 wird zunächst das "Body Attribute" aus dem "Attribute"-Feld gewählt und anschließend die GET-Schnittstelle im "Request"-Feld.
Das Feld "Trigger Behavior" beschreibt, ob die "Request" erneut gesendet werden soll, sobald die "id" benötigt wird. Dadurch muss nur die POST-Schnittstelle ausgeführt werden.
Alternativ ist eine manuelle Ausführung der GET-Schnittstelle möglich, wodurch die "id" in der Historie abgelegt wird (vgl. Insomnia REST Client 2019b).

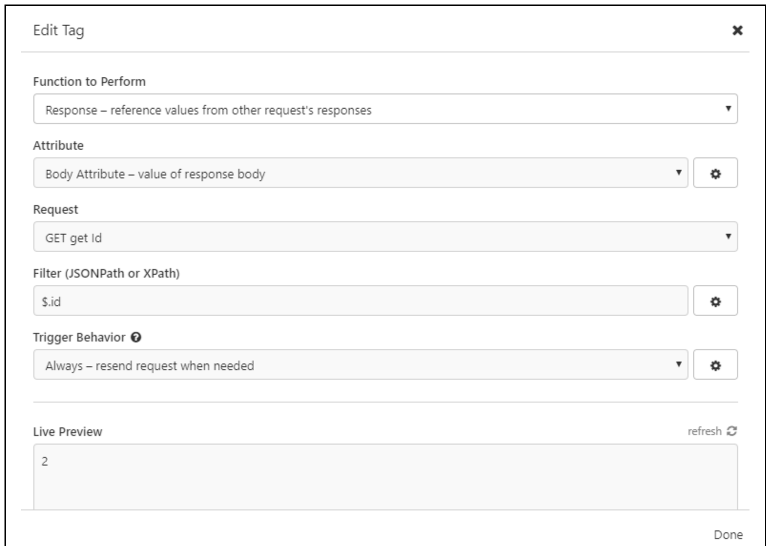


Abbildung 3.21: Insomnia: Pop-up zum Bearbeiten der Verkettung

Probleme:

- Automatisches Ausführen ist nur möglich, wenn eine Datenverkettung vorhanden ist.
- Keine asynchrone Ausführung möglich.
- Keine Möglichkeit, die Tests mit Bedingungen zu validieren.

3.3 Anforderungen

Der folgende Abschnitt listet Funktionalitäten, die vom System unterstützt werden sollen.

Funktionale Anforderungen

- **OpenAPI**

  Das System kann:

  1. Schnittstellen als Liste von wählbaren Elementen anzeigen.
  2. Eine YAML-Repräsentation der OpenAPI-Definition im Editor anzeigen.
  3. Swagger-Definitionen (Version <3) als OpenAPI (Version >=3) importieren.

  Der Benutzer kann:

  4. Anhand eines Schnittstellenelements zur Definition im Editor springen.
  5. Anhand eines Schnittstellenelements zur parameters-Definition im Editor springen.
  6. Anhand eines Schnittstellenelements zum requestBody im Editor springen.
  7. Auf Knopfdruck die ausgewählte Schnittstelle ausführen.
  8. Auf Knopfdruck einen durch $ref ersetzbaren Definitionsabschnitt verschieben.

- **TestAPI**

  Das System kann:

  9. Testfälle als Baumstruktur von wählbaren Elementen anzeigen.
  10. Eine YAML-Repräsentation der TestAPI-Definition im Editor anzeigen.
  11. Bei einer Test-Duplizierung, die betroffenen $ref-Pfade aktualisieren.
  12. Testfälle synchron ausführen.
  13. Testfälle asynchron ausführen.
  14. Testfälle automatisiert ausführen.
  15. Den Verlauf der Ausführung anzeigen.
  16. Im Verlauf den Status anzeigen (running/ok/fail).
  17. Im Verlauf die gesendete Anfrage anzeigen.
  18. Im Verlauf die erhaltene Antwort anzeigen.
  19. Eine Liste der Anfrage-Abweichungen zur OpenAPI-Definition anzeigen.
  20. Eine Liste der Antwort-Abweichungen zur OpenAPI-Definition anzeigen.
  21. Prüfen ob die Antwort, den expected-Werten des Tests entspricht.

  Der Benutzer kann einen Test:

  22. Anhand eines Formulars erstellen.
  23. Anhand eines Formulars bearbeiten.
  24. Anhand einer Baumstruktur verschieben.
  25. Auf Knopfdruck duplizieren.
  26. Auf Knopfdruck löschen.
  27. Auf Knopfdruck im TestAPI-Editor einsehen (Sprung zur Definition).
  28. Auf Knopfdruck ausführen.
  29. Mit Daten aus der TestAPI-Definition oder einer Test-Antwort mithilfe von $ref verketten.

- **Formular**
  Das System kann:
  30. Folgende Typen darstellen: integer, number, string, boolean, object, array, enum.
  31. Eine YAML-Repräsentation des Formulars im Editor anzeigen.
  32. Bei Änderungen des Formulars eine Echtzeitvorschau im Editor anzeigen.
  33. Bei Änderungen im Editor eine Echtzeitvorschau im Formular anzeigen.
  34. Bei Änderung eines Formularwerts anhand der OpenAPI auf Gültigkeit prüfen.
  35. Formularfelder mit dem example-Wert initialisieren.
  36. Bei einem leeren Feld den default-Wert vorschlagen.

  Der Benutzer kann:
  37. Die Werte eines Formularfelds ändern.
  38. Elemente vom Typ "object" in einem separaten Formular bearbeiten.
  39. Auf Knopfdruck alle Elemente eines Arrays löschen.
  40. Auf Knopfdruck ein Element eines Arrays löschen.
  41. Die Reihenfolge der Elemente eines Arrays ändern.
  42. Auf Knopfdruck die Array-Elemente in umgekehrter Reihenfolge anordnen.
  43. Felder auf den example-Wert zurücksetzen.
  44. Werte auf "null" setzen.
  45. Felder deaktivieren/aktivieren.
  46. Wertänderungen rückgängig machen und wiederherstellen.
  47. Werte ausschneiden, kopieren und einfügen.
  48. Beispieldaten in ein Formular aus der OpenAPI-Definition laden.
  49. Beispieldaten anhand eines Formulars in der OpenAPI-Definition speichern.
- **Editor**
  Das System kann:
  50. Den Editor-Inhalt validieren und auf Fehler hinweisen.
  51. YAML-Syntax farblich hervorheben.
  52. Bei einer Änderung des Editorinhalts, nach einer festgelegten Zeitverzögerung, die Validierung auslösen, wenn die Cursor-Position nicht geändert wird.
  53. Bei Formularelement Mouse-Hover, den Cursor auf die Schlüssel-Position setzen.

  Der Benutzer kann:
  54. YAML-Text bearbeiten.
  55. Textbereiche auswählen und verschieben.
  56. Textänderungen rückgängig machen und wiederherstellen.
  57. Eine API aus der Zwischenablage in den aktiven Editor laden.
  58. Eine API durch Ziehen aus dem Dateiverzeichnis laden.
  59. Eine API über das Öffnen-Dialogfenster des Betriebssystems laden.
  60. Eine API auf Knopfdruck in der aktuellen Datei speichern.
  61. Eine API in einer neuen Datei speichern.
  62. Auf Knopfdruck eine Validierung auslösen.
  63. Auf Knopfdruck den $ref-Pfad zur Cursor-Position in die Zwischenablage kopieren.
  64. Die Schriftgröße des Editors und der OpenAPI-IDE separat ändern.

- **Anwendung**
  Das System kann:
  65. Beim Schließen einen Hinweis auf ungesicherte Änderungen liefern.
  66. Beim Beenden alle Einstellungen automatisiert speichern.
  67. Beim Starten alle Einstellungen automatisiert laden.
  68. Beim Starten über die Kommandozeile die angegebene OpenAPI und TestAPI laden.
  69. Beim Starten über die Kommandozeile alle Tests ausführen.
  70. Sich die Fensterposition und Größe merken.
  71. Sich die Spaltenorientierung merken.
  72. Sich die Positionen der Schiebeelemente merken.
  73. Sich die Editor-Schriftgröße merken.
  74. Sich die Schriftgröße der Oberfläche merken.
  75. Sich merken, ob die summary einer Schnittstelle angezeigt werden soll.
  76. Sich merken, ob der Pfad einer Schnittstelle angezeigt werden soll.
  77. Sich die Anzahl der bevorzugten YAML-Indents merken.
  78. Sich die Zeitverzögerung für das automatisierte Auslösen der Validierung merken.
  79. Auf folgenden Plattformen ausgeführt werden: Windows 10, Ubuntu >18, OS X 10.

3.4 Fazit

Die Analyse hat gezeigt, dass der Swagger Editor verbessert werden kann und alternative Werkzeuge als Inspiration dienen können. Jedes der vorgestellten Werkzeuge hat seine Stärken und ist in bestimmten Anwendungsfällen passender für die Lösung eines Problems als andere. Die folgende Tabelle zeigt eine Zusammenfassung der OpenAPI-Funktionalitäten.

Funktionalität	Swagger Editor	Postman	Insomnia	OpenAPI-IDE
Formular	o	-	-	+
Semantikprüfung	o	-	-	+
Syntaxprüfung	o	+	+	+
Beispieldaten	+	o	o	+
Speichern (als OpenAPI)	o	-	-	+
Datenverkettung	-	+	+	+
Automatisiertes Ausführen	-	+	o	+
Automatisiertes Validieren	-	+	-	+
OpenAPI-Definition bearbeiten	+	-	-	+
Jump to *	o	-	-	+
Umstrukturierung	-	-	-	+
	4.5 P	4.5 P	3 P	11P

4 Spezifikation

Dieses Kapitel beschreibt zunächst Szenarien, die mit dem fertigen System umgesetzt werden können. Im Anschluss wird eine grafische Benutzeroberfläche vorgestellt, die alle Szenarien und Anforderungen unterstützt.

4.1 Szenarien

Im Folgenden wird zunächst das Bearbeiten einer OpenAPI-Definition und das Senden einer Anfrage beschrieben. Anschließend werden einfache und verkettete Testfälle definiert und ausgeführt.

Beispiel: Aufbau eines Szenarios

Ziel:
Der Benutzer führt eine Aktion aus und das System verarbeitet diese.

Erfolgsszenario:
1. Der Benutzer führt eine Aktion aus.
2. Das System verarbeitet die Aktion.

Fehlerfall:
1.1. Erster möglicher Fehler der Benutzeraktion.
1.2. Zweiter möglicher Fehler der Benutzeraktion.
2.1. Erster möglicher Fehler bei der Verarbeitung durch das System.
…

Optional:
1.a. Der Benutzer führt eine alternative Aktion aus, um Schritt eins durchzuführen.
1.b. Der Benutzer hat eine weitere Möglichkeit für den ersten Schritt.

4.1.1 OpenAPI bearbeiten

Ziel:

Der Benutzer importiert eine Swagger-Definition als OpenAPI-Definition und fügt ein neues Attribut in einem der Schemas hinzu. Zum Schluss wird die bearbeitete OpenAPI in einer neuen Datei gespeichert.

Erfolgsszenario:

1. Der Benutzer startet das System mit einem Doppelklick.
2. Der Benutzer öffnet eine Swagger-Definition über das "Öffnen"-Pop-up (4.2.2).
3. Das System prüft, ob die Datei eine Swagger-Definition enthält und wandelt diese in eine OpenAPI-Definition um.
4. Das System fügt alle vorhandenen Schnittstellen der Sidebar-Liste hinzu und füllt das Server-Auswahlmenü mit den definierten Servern (4.2.3).
5. Der Benutzer öffnet das Kontextmenü einer Schnittstelle, welche das gesuchte Schema enthält und wählt "Jump to requestBody" (Kontextmenüs 1).
6. Das System öffnet den OpenAPI-Editor und zeigt den requestBody an, welcher in diesem Fall ein $ref enthält.
7. Der Benutzer öffnet in der $ref-Zeile das Kontextmenü und wählt "Jump to definition" (Kontextmenüs 3).
8. Das System zeigt im OpenAPI-Editor die content-Definition an.
9. Der Benutzer sucht die schema-Eigenschaft.
10. Der Benutzer definiert in properties das neue Attribut.
11. Der Benutzer wählt im Output-Bereich "Speichern" (4.2.2).
12. Das System zeigt ein Pop-up für die Wahl der Zieldatei, da es sich um eine importierte Swagger-Definition handelt.
13. Der Benutzer wählt die Zieldatei.
14. Das System speichert die OpenAPI-Definition im Dateisystem.

Fehlerfall:

3.1. Falls die Swagger-Definition fehlerhaft ist, liefert das System eine Fehlerbeschreibung.
4.1. Es sind keine Schnittstellen definiert, das Senden der Anfrage wird deaktiviert.
4.2. Es sind keine Server definiert, das Senden der Anfrage wird deaktiviert.

Optional:

1.a. Der Benutzer startet das System über die Kommandozeile und gibt dabei die zu öffnende Swagger-Definition an (Sprung zu Schritt 3).
2.a. Der Benutzer fügt die Swagger-Definition über die Zwischenablage in den OpenAPI-Editor ein.
2.b. Der Benutzer zieht die Swagger-Definition aus dem Dateiverzeichnis in das System.
5.a. Der Benutzer sucht das Schema in der OpenAPI-Definition (Sprung zu Schritt 10).

4.1.2 Anfrage senden

Ziel:
Der Benutzer möchte ein Anfrageformular als Beispiel speichern, die Anfrage ausführen und die Antwort einsehen.

Erfolgsszenario:
1. Der Benutzer wählt eine Schnittstelle aus der Sidebar (Linksklick).
2. Das System erstellt und zeigt das zugehörige Formular an.
3. Der Benutzer füllt das Formular aus und drückt auf "Beispiel speichern" (4.2.4).
4. Das System zeigt das Pop-up "Beispiel speichern" (Pop-ups 3).
5. Der Benutzer gibt die "Id" und "Beschreibung" ein.
6. Der Benutzer bestätigt das "Beispiel speichern"-Pop-up.
7. Das System fügt das Beispiel in die OpenAPI-Definition ein.
8. Der Benutzer wählt den Zielserver für die Anfrage (4.2.3).
9. Der Benutzer führt die Anfrage mit dem Run-Knopf aus (4.2.1).
10. Das System sendet die Anfrage und wartet auf eine Antwort.
11. Der Benutzer wechselt zum Debug-Editor und betrachtet die response (4.2.3).

Fehlerfall:
10.1. Das System kann den Server nicht erreichen oder der Server antwortet nicht innerhalb des erwarteten Zeitraums. Die Antwort zeigt einen status mit der Zahl Null an.

Optional:
3.a. Der Benutzer speichert das Formular nicht, da die Werte nur einmalig benötigt werden (Sprung zu Schritt 8).
5.a. Der Benutzer ändert Werte im Vorschaubereich.

4.1.3 Test erstellen

Ziel:

Der Benutzer erstellt einen Test mithilfe eines geladenen Beispiels und definiert Werte in der expected-Eigenschaft, welche vom System bei der Ausführung automatisiert mit der Antwort verglichen werden.

Erfolgsszenario:

1. Der Benutzer wählt eine Schnittstelle aus der Sidebar.
2. Das System erstellt und zeigt das zugehörige Formular an.
3. Der Benutzer drückt auf "Beispiel laden" (4.2.4).
4. Das System zeigt das Pop-up "Beispiel laden" (Pop-ups 4).
5. Der Benutzer wählt ein Beispiel und bestätigt das Pop-up.
6. Das System füllt das Formular mit den gewählten Beispieldaten.
7. Der Benutzer drückt im Formularbereich auf den "Test speichern"-Knopf (4.2.3).
8. Das System zeigt das Pop-up "Test speichern" (Pop-ups 5).
9. Der Benutzer gibt die "Id" ein und wählt "Root" als übergeordneten Test.
10. Der Benutzer drückt den Run-Knopf, um einen Testlauf auszuführen.
11. Das System fügt die Antwort in die expected-Eigenschaft ein.
12. Der Benutzer entfernt Paare aus expected, welche nicht geprüft werden sollen.
13. Der Benutzer bestätigt das "Test speichern"-Pop-up.
14. Das System fügt den Test in die TestAPI-Definition ein.
15. Der Benutzer wechselt zur Test-Ansicht (4.2.5).
16. Der Benutzer wählt den erstellten Test aus und drückt auf den Run-Knopf.
17. Das System sendet die Anfrage und wartet auf eine Antwort.
18. Das System vergleicht die Antwort mit den expected-Werten und liefert einen Erfolgshinweis für die Testausführung.

Fehlerfall:

9.1. Das System verbietet die Bestätigung, da eine belegte "Id" eingegeben wurde.

18.1. Die expected-Werte stimmen nicht überein, es wird ein Fehlerhinweis angezeigt und die Ausführung von weiteren Tests wird abgebrochen.

Optional:

3.a. Der Benutzer füllt das Formular mit den gewünschten Daten (Sprung zu Schritt 7).

5.a. Der Benutzer ändert Werte im Vorschau-Editor.

10.a. Der Benutzer definiert keine expected-Werte (Sprung zu Schritt 13).

10.b. Der Benutzer gibt die expected-Werte manuell ein (Sprung zu Schritt 13).

4.1.4 Anfragen verketten

Ziel:
Der Benutzer definiert eine Verkettung, welche aus zwei Tests besteht, wobei der zweite Test auf Daten des ersten Tests zugreift.

Erfolgsszenario:
1. Der Benutzer erstellt einen synchronen Test (4.1.3).
2. Der Benutzer erstellt einen zweiten Test, diesen fügt er dem bereits erstellten Test als Kind hinzu.
3. Der Benutzer wechselt zur Test-Ansicht (4.2.5).
4. Der Benutzer führt den ersten Test aus, um die Antwort-Struktur zu erhalten.
5. Das System zeigt die Anfrage und Antwort im Ausgabe-Editor an (Nur-Lese-Modus).
6. Der Benutzer sucht im Editor den Schlüssel, welcher in der nächsten Anfrage eingesetzt werden soll.
7. Der Benutzer öffnet in dieser Zeile das Kontextmenü und wählt "Copy $ref" (Kontextmenüs 3).
8. Das System kopiert den $ref-Pfad zum Schlüssel in die Zwischenablage.
9. Der Benutzer öffnet das Kontextmenü des zweiten Tests und wählt "Jump to definition" (Kontextmenüs 2).
10. Der Benutzer fügt den $ref-Pfad aus der Zwischenablage als Wert des Zielschlüssels ein.
11. Der Benutzer führt den ersten Test aus.
12. Das System sendet die Anfrage und wartet auf eine Antwort.
13. Das System markiert den ersten Test als erfolgreich und führt den zweiten Test aus.
14. Das System ersetzt alle $ref-Pfade. Der Wert aus der Antwort des ersten Tests, wird in die Anfrage des zweiten Tests eingesetzt.
15. Das System sendet die Anfrage des zweiten Tests.
16. Das System erhält die Antwort und markiert den Test als erfolgreich.

Fehlerfall:
13.1. Die Anfrage ist fehlgeschlagen, das System liefert einen Hinweis und die Ausführung von weiteren Tests wird abgebrochen.
14.1. Das System kann den Schlüssel anhand des $ref-Pfads nicht finden, es wird ein Hinweis geliefert und die Ausführung von weiteren Tests wird abgebrochen.

Optional:
4.a. Der Benutzer kennt bereits den $ref-Pfad für die Verkettung und gibt diesen manuell im Editor ein (Sprung zu Schritt 11).
9.a. Der Benutzer sucht die Test-Definition im TestAPI-Editor.

4.2 Benutzeroberfläche

Dieser Abschnitt beschreibt die grafische Benutzeroberfläche im Detail. Dazu gehören Aufbau, Ansichten, Kontextmenüs und Pop-ups.

4.2.1 Aufbau

In Abbildung 4.1 sind drei Spalten zu sehen (Sidebar, Form und Output), die durch verschiebbare Elemente getrennt werden (rot markiert). Die Output-Spalte hat ein vertikales Schiebeelement (grün markiert), welches mit einem Doppelklick durch ein horizontales Element ausgetauscht werden kann.

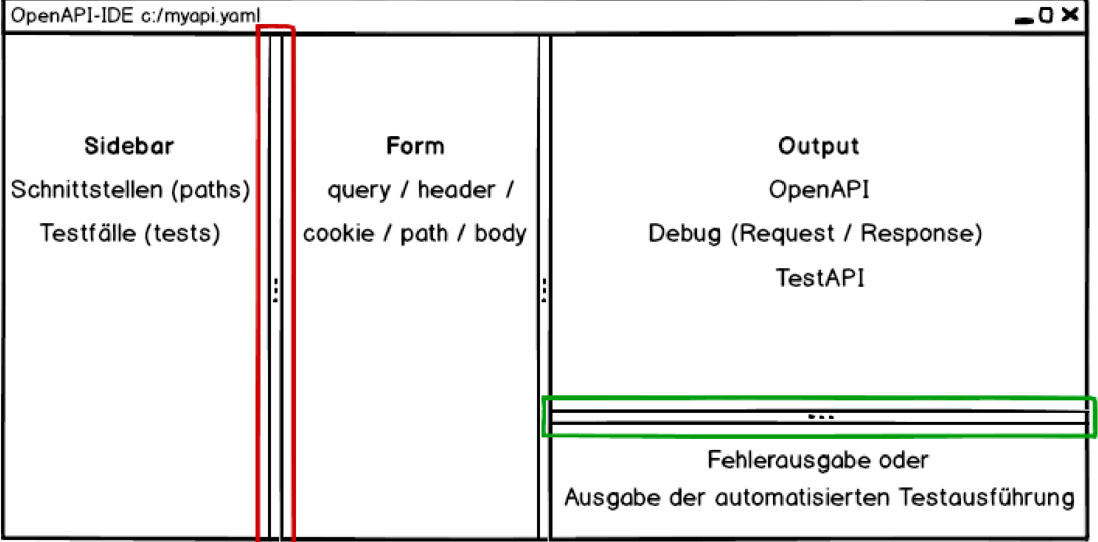


Abbildung 4.1: OpenAPI-IDE: Drei Spalten Übersicht

Die Tabs API Test im oberen Bereich der Sidebar ermöglichen das Wechseln zwischen der API- und Test-Ansicht (Abbildung 4.2). Im unteren Bereich der Sidebar stehen abhängig von der aktiven Ansicht entweder Schnittstellen oder Testfälle zur Auswahl, diese können mit dem Run-Knopf ▶ (Tastenkürzel F4) ausgeführt werden. Bei laufender Anfrage wird "Run" in einen Stop-Knopf ■ gewandelt, welcher zum Abbrechen der Anfrage dient.

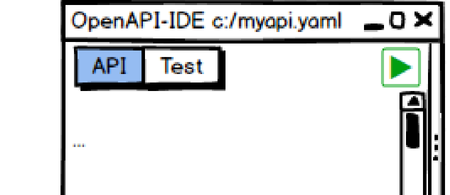


Abbildung 4.2: OpenAPI-IDE: Schnittstellen-Bereich

4.2.2 Output

Im oberen Bereich der Output-Spalte sind Steuerelemente, die sich abhängig von der aktiven Ansicht leicht unterscheiden.

- Die Tabs OpenAPI Debug in der API-Ansicht wechseln den Editor.
  - Der OpenAPI-Tab zeigt die OpenAPI-Definition (Abbildung 4.3).
  - Der Debug-Tab zeigt zunächst die vollständige Anfrage und die vorhandenen Formular-Fehler an. Nach der Ausführung mit dem Run-Knopf, wird zusätzlich die Antwort angezeigt (Abbildung 4.6).
- Bei der Test-Ansicht wird der Editor für die TestAPI angezeigt (Abbildung 4.4).

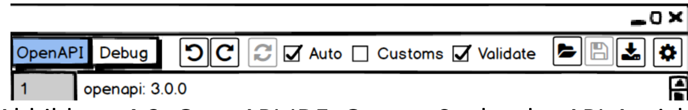


Abbildung 4.3: OpenAPI-IDE: Output-Spalte der API-Ansicht

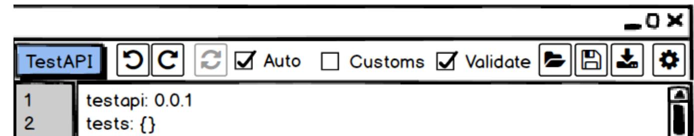


Abbildung 4.4: OpenAPI-IDE: Output-Spalte der Test-Ansicht

- Die Knöpfe "Rückgängig" (Strg+Z) und "Wiederherstellen" (Strg+Umschalt+Z), machen Textänderungen rückgängig oder stellen sie wieder her.
- Die Auto-Checkbox ☑ Auto und der Reload-Knopf (F5) sind für das Auslösen der Validierung zuständig. Nach einer Änderung des Editorinhalts kann entweder manuell der Reload-Knopf gedrückt werden oder beim aktiven Auto-Modus auf die automatisierte Aktualisierung gewartet werden, welche nach einer festgelegten Zeitverzögerung ausgelöst wird, falls die Cursor-Position nicht geändert wird.
- Die Checkbox "Customs" und "Validate" unterstützt die Bearbeitung der request im Debug-Bereich (siehe Abbildung 6.19).
- Der Öffnen-Vorgang (Strg+O) aktiviert abhängig von der geladenen Datei (OpenAPI/TestAPI), die entsprechende Ansicht und platziert den Inhalt im Editor.
- Mit "Speichern" (Strg+S) wird die aktive Datei (OpenAPI/TestAPI) gespeichert.
  Der "Speichern unter"-Knopf (Strg+Umschalt+S) zeigt ein Pop-up zum Wählen einer Zieldatei.
  Das "Speichern" ist nur möglich, wenn ungesicherte Änderungen vorhanden sind. Falls keine Datei ausgewählt wurde, wird beim "Speichern" der Vorgang "Speichern unter" ausgelöst.
- Der Knopf "Einstellungen" öffnet das Pop-up 1.

Fehlerbehandlung
Bei zum Beispiel einem YAML-Syntaxfehler (Abbildung 4.5), wird in der Fehlerzeile mit ✖ ein Hinweis angezeigt und das Schiebeelement im Output-Bereich farblich hervorgehoben. Durch das Bewegen des Schiebeelements wird ein Editor sichtbar, der die vollständige Fehlermeldung anzeigt. Zur Vermeidung von Folgefehlern wird nur der erste Fehler angezeigt, zusätzlich werden Bereiche deaktiviert, die durch den Fehler beeinflusst werden.

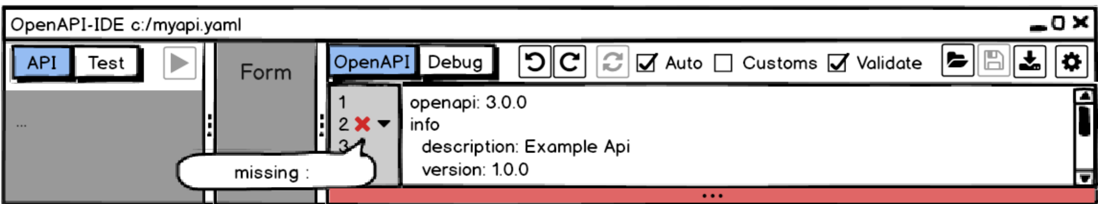


Abbildung 4.5: OpenAPI-IDE: Fehlerbehandlung

4.2.3 API-Ansicht

Die API-Ansicht ermöglicht das Bearbeiten der OpenAPI-Definition, Senden einzelner Anfragen und die Erstellung eines Tests anhand des Formulars (Abbildung 4.6).

- Die Elemente in der Sidebar zeigen die einzelnen Schnittstellen der OpenAPI-Definition.
- Durch das Wählen einer Schnittstelle mit einem Linksklick, wird das zugehörige Formular generiert, welches in der mittleren Spalte in Abbildung 4.6 zu sehen ist.
- Im oberen Teil des Formulars kann über ein Dropdown der Server und der Content-Typ gewählt werden.
- Der "Test speichern"-Vorgang kann mit 🖫 ausgelöst werden (Pop-ups 5).
- Der "Edit Mode" ist für die Bearbeitung eines Tests und kann über ein Kontextmenü aktiviert werden (Kontextmenüs 2). Der Modus wird automatisch deaktiviert, sobald eine abweichende Aktion ausgeführt wird oder der Test mit 🖫 aktualisiert wurde (Pop-ups 6).

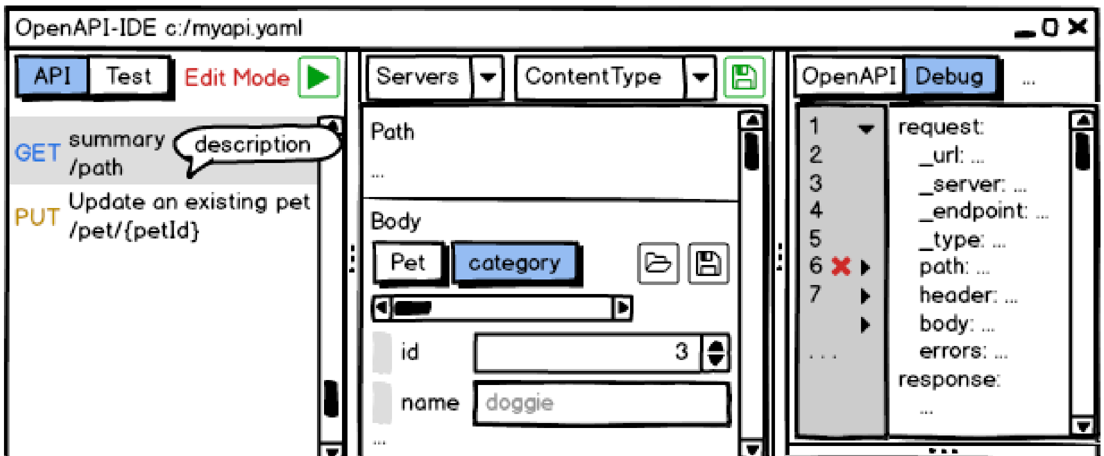


Abbildung 4.6: OpenAPI-IDE: API-Ansicht

4.2.4 Formular

Das Formular kann folgenden Kategorien enthalten: Query, Header, Path, Cookie und Body (Abbildung 4.7). Im Vergleich zu anderen Kategorien unterstützt der Body nicht nur primitive Datentypen wie Zahl, Text, Wahrheitswert oder eine geordnete Sammlung, sondern auch verschachtelte Objekte. Die verschachtelten Objekte werden mittels Tabs umgesetzt, wobei eine Tab-Aktivierung das zugehörige Formular erzeugt. Diese Umsetzung wird gewählt, um das Aufklappen von Objekten zu meiden, welches vertikale Veränderungen und unübersichtliche Verschachtelungen verursachen würde. Der Name des ersten Tabs ist der Schema-Name aus den components oder der xml.name, falls diese nicht vorhanden sind, lautet der Name "Object". Alle weiteren Tabs setzen den Attributnamen ein, wobei Array-Elemente zusätzlich den Index anzeigen. Eine Objekt Zusammenfassung liefert ein Tooltip des jeweiligen Tabs. Die aktuellen Formularwerte können mit "Beispiel speichern" (Pop-ups 3) in der OpenAPI-Definition abgelegt werden und mit "Beispiel laden" (Pop-ups 4) kann das aktuelle Formular mit Beispieldaten gefüllt werden.

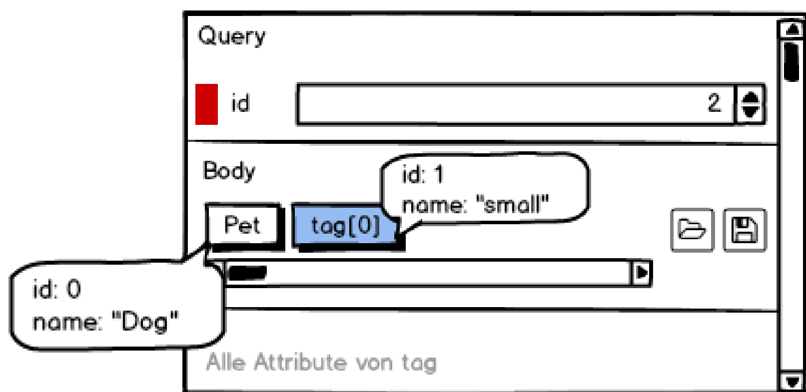


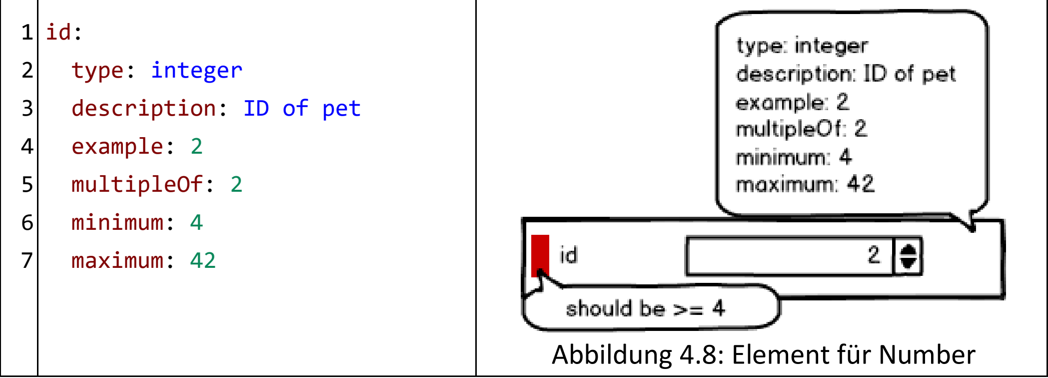
Abbildung 4.7: OpenAPI-IDE: Formular

Darstellung der Attribute

Im folgenden Teil werden die unterstützten Datentypen und deren Elemente vorgestellt, welche mittels type unterschieden werden (vgl. SmartBear Software 2019o). Die OpenAPI bietet viele Möglichkeiten die Grenzen eines Attributs zu beschreiben. Wenn der Benutzer einen Wert abweichend zur Definition eingibt, wird auf der linken Seite des Elements ein rotes Rechteck angezeigt, welches ein Tooltip mit weiteren Details liefert.
Die vollständige Attribut-Definition wird in einem Tooltip angezeigt und kann im ganzen Element-Behälter mittels Mouse-Hover aktiviert werden (Abbildung 4.8), diese Aktion setzt die Cursor-Position im Debug-Editor auf das entsprechende Attribut.

- Ein Zahlenelement kann mit type "integer" (Ganzzahl) oder "number" (Kommazahl) erzeugt werden. Das Eingabefeld wird mit dem example-Wert initialisiert, falls dieser nicht vorhanden ist, wird stattdessen die Zahl Null eingesetzt (Abbildung 4.8).
  Auf der rechten Seite sind zwei Knöpfe, mit denen der Zahlenwert vergrößert oder verkleinert werden kann. Falls ein multipleOf-Wert definiert ist, wird dieser als Schrittweite eingesetzt. Die Grenzen können mit minimum/maximum angegeben werden.

Beispiel: Number

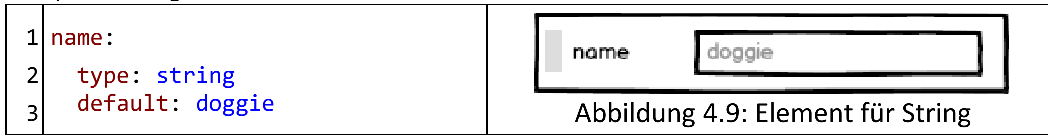


```
1 id:
2   type: integer
3   description: ID of pet
4   example: 2
5   multipleOf: 2
6   minimum: 4
7   maximum: 42
```


Abbildung 4.8: Element für Number

- Ein einzeiliges Textelement kann mit type "string" erzeugt werden. Als Platzhalter wird bei einem leeren Feld der default-Wert vorgeschlagen (Abbildung 4.9).

Beispiel: String

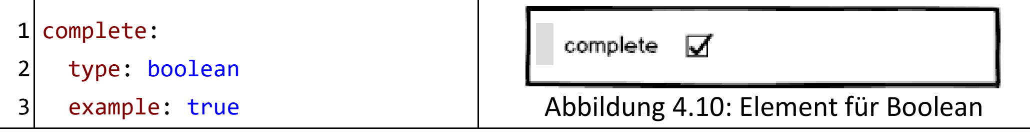


```
1 name:
2   type: string
3   default: doggie
```


Abbildung 4.9: Element für String

- Ein Wahrheitswert wird als Checkbox dargestellt und kann mit type "boolean" erstellt werden (Abbildung 4.10).

Beispiel: Boolean



```
1 complete:
2   type: boolean
3   example: true
```


Abbildung 4.10: Element für Boolean

- Ein Objekt wird als Dropdown mit folgender Auswahl dargestellt (Abbildung 4.11):
  - Der "Current"-Eintrag wird für die Darstellung von nicht gespeicherten Objekten benötigt. Bei der Aktivierung wird das aktuelle Objekt in einem neuen Formular-Tab angezeigt. Dies kann alternativ mit dem Edit-Knopf erreicht werden.
  - Mit "{}" kann dem Wert ein leeres Objekt zugewiesen werden.
  - "New" erstellt ein neues Objekt und öffnet das zugehörige Formular-Tab.
  - Alle weiteren Einträge ("Category1") sind gespeicherte examples (3.1.4).

Beispiel: Object

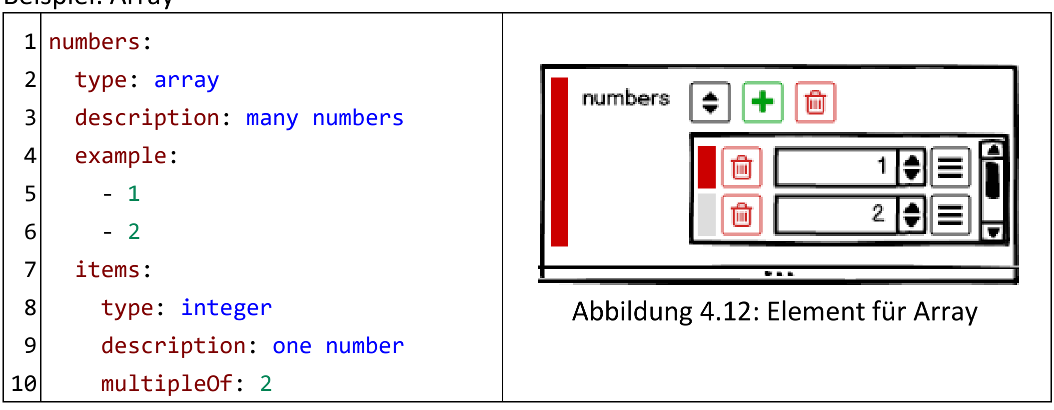
```
1 category:
2   type: object
3   required:
4     - id
5   properties:
6     id:
7       type: integer
```

Abbildung 4.11: Element für Object

- Ein Array zeigt Elemente an, welche mit items beschrieben werden (Abbildung 4.12). Die erste Zeile hat drei Knöpfe:
  - Mit "Reverse" werden alle Elemente in umgekehrter Reihenfolge angeordnet.
  - Der Add-Knopf erzeugt ein neues Element am Ende der Liste.
  - "Löschen" entfernt alle Elemente aus der Liste.

  Die zweite Zeile zeigt die Liste. Jeder Eintrag hat einen Knopf zum "Löschen" des Elements und einen weiteren Knopf zum "Ändern" der Position mit Mouse-Drag. Der Element-Behälter hat außerdem ein vertikales Schiebeelement zum Vergrößern der Listendarstellung.

Beispiel: Array



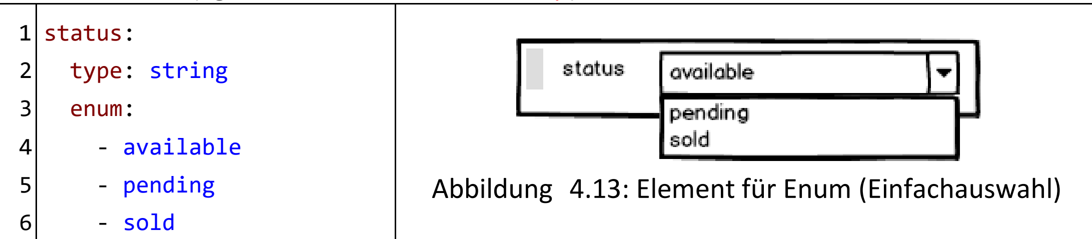
```
1 numbers:
2   type: array
3   description: many numbers
4   example:
5     - 1
6     - 2
7   items:
8     type: integer
9     description: one number
10    multipleOf: 2
```


Abbildung 4.12: Element für Array

- Ein Enum liefert eine Liste von wählbaren Strings (Abbildung 4.13, 4.14).

Beispiel: Enum

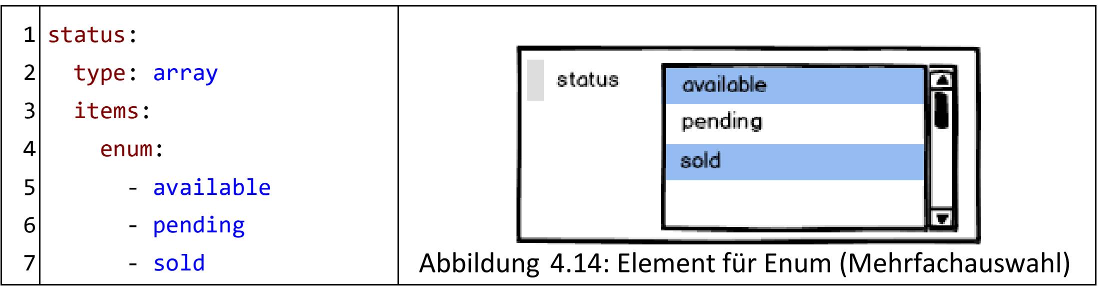
Einfachauswahl (vgl. SmartBear Software 2019p)



```yaml
status:
  type: string
  enum:
    - available
    - pending
    - sold
```


Abbildung 4.13: Element für Enum (Einfachauswahl)

Mehrfachauswahl



```yaml
status:
  type: array
  items:
    enum:
      - available
      - pending
      - sold
```


Abbildung 4.14: Element für Enum (Mehrfachauswahl)

4.2.5 Test-Ansicht

In Abbildung 4.15 werden in der Sidebar die einzelnen Tests als Baumstruktur dargestellt. Ein Test-Element hat folgenden Aufbau:

- Ein Test mit mindestens einem Kind hat einen Fold-Knopf ☐ zum Verstecken der Kinder.
- Der async-Zustand wird mit Async oder Sync dargestellt.
- Die Test-ID darf im ganzen Baum nur einmal vorkommen.
- Die description wird mit einem Bindestrich von der Test-ID getrennt.
- Der Status der Testausführung wird mit verschiedenen Farben dargestellt:
  - Grün (erfolgreich)
  - Orange (laufend)
  - Rot (fehlgeschlagen)

  Der Tooltip des Status-Bereichs zeigt bei einer laufenden Anfrage die request und bei einem abgeschlossenen Test die response. Diese Informationen entsprechen den Testergebnissen, welche sich in der Abbildung auf der rechten Seite im Editor befinden.
- Der Tooltip eines Testelements zeigt die request- und expected-Werte an.

Um die Reihenfolge der Ausführung zu ändern, kann ein Element mit Mouse-Drag verschoben werden. Ein ausgewählter Test dient als Startpunkt für die Ausführung eines Testzweigs. Damit die Testergebnisse und TestAPI-Definition gleichzeitig einsehbar sind, wird in der Test-Ansicht auf einen weiteren Tab verzichtet. Der rechte Editor zeigt entweder einen TestAPI-Fehler oder die Testergebnisse.

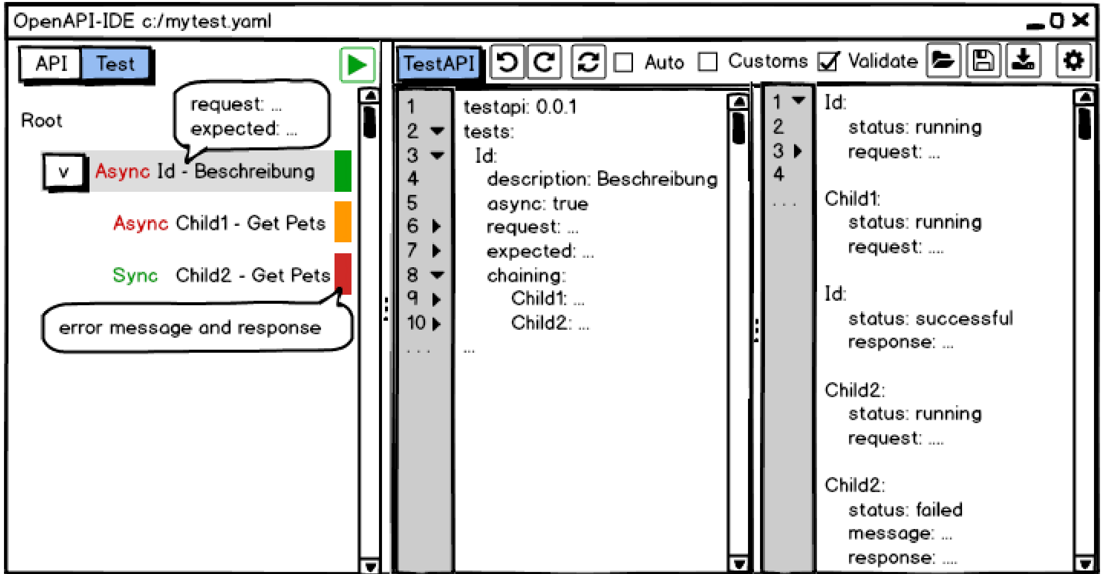


Abbildung 4.15: OpenAPI-IDE: Test-Ansicht

4.2.6 Kontextmenüs

Im Folgenden werden die Aktivierungsbereiche und Optionen der Kontextmenüs beschrieben.

1. Schnittstellenelement

Ein Rechtsklick auf eines der Schnittstellenelemente in der Sidebar öffnet das Kontextmenü (Abbildung 4.16). Es hat folgende Optionen:

- **Jump to definition** springt im OpenAPI-Editor zur Definitionsstelle.
- **Jump to parameters** springt im OpenAPI-Editor zur parameters-Definitionsstelle. Die Option wird nur angezeigt, wenn eine Definition vorhanden ist.
- **Jump to requestBody** verhält sich wie "Jump to parameters" (springt zum requestBody).
- **Delete** entfernt die ausgewählte Schnittstelle aus der OpenAPI-Definition und Sidebar.

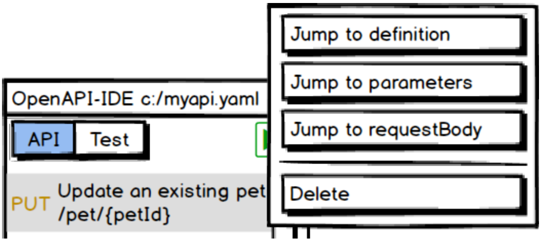


Abbildung 4.16: OpenAPI-IDE: Kontextmenü für die Schnittstellen

2. Testelement

In Abbildung 4.17 ist das Kontextmenü für die Testelemente zu sehen. Es hat folgende Optionen:

- **Jump to definition** setzt die Cursor-Position im TestAPI-Editor auf die Definitionsstelle.
- **Toggle Async** aktiviert oder deaktiviert die asynchrone Ausführung für den Test.
- **Edit** öffnet die API-Ansicht und füllt das Formular mit den Testdaten.
- **Delete** entfernt den Test aus der TestAPI-Definition und Sidebar.
- **Duplicate** kopiert den ausgewählten Test und fügt ihn auf gleicher Höhe ein.
- **Jump to API** wechselt in die API-Ansicht und wählt die Schnittstelle, welche für die Testerzeugung eingesetzt wurde.

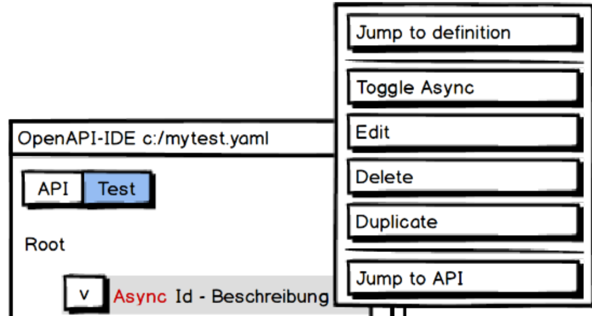


Abbildung 4.17: OpenAPI-IDE: Kontextmenü für die Tests

3. Editor

Das Kontextmenü im Texteditor hat abhängig vom aktiven Editor und der ausgewählten Zeile andere Optionen. In Abbildung 4.18 werden diese aus Platzgründen getrennt dargestellt:

- Links werden Editor und Zeilenabhängigen Optionen angezeigt, auch wenn semantische Fehler vorhanden sind, aber nicht bei Syntaxfehlern.
- Rechts werden Funktionalitäten angezeigt, die immer vorhanden sind.

- **Open in tab** wird nur im Debug-Editor angezeigt, wenn es sich bei der Zeile um ein Objekt der body-Eigenschaft handelt. Bei Aktivierung wird das Objekt im Formular angezeigt.
- **Select in sidebar** wird nur im OpenAPI-Editor angezeigt, wenn es sich bei dem Schlüssel um eine Schnittstelle handelt. Es wählt das Schnittstellenelement in der Sidebar aus.
- **Refactor** wird nur im OpenAPI-Editor angezeigt und ermöglicht die Umstrukturierung von parameters, requestBody, responses und schema.
- **Jump to definition** wird nur angezeigt, wenn die Zeile eine $ref-Verlinkung enthält. Bei der Aktivierung wird das verlinkte Objekt im Editor angezeigt.
- **Jump back** baut bei der Ausführung von "Jump to definition" einen Stapel auf, welcher beim Ausführen der Aktion, die oberste Position vom Stapel nimmt und anzeigt. Diese Option wird nur angezeigt, wenn der Stapel nicht leer ist.
- **Copy $ref** ermittelt den vollständigen $ref-Pfad zum Schlüssel und kopiert diesen in die Zwischenablage (Clipboard).
- **Copy $ref index** verhält sich wie das normale "Copy $ref" und wird in der ersten Zeile bei jedem Array-Elemente angezeigt, wenn es sich um ein Array vom type "object" handelt. Es kopiert den $ref-Pfad bis zum Array-Index, anstatt des Schlüssels zur Objekt-Eigenschaft.
- **Cut, Copy, Paste** ermöglicht das Ausschneiden, Kopieren und Einfügen von Text über die Zwischenablage.
- **Undo, Redo** stellt Textänderungen wieder her oder macht sie rückgängig.
- **Unfold/Fold all** ermöglicht das Ein- und Ausklappen aller Schlüssel/Wert-Paare.

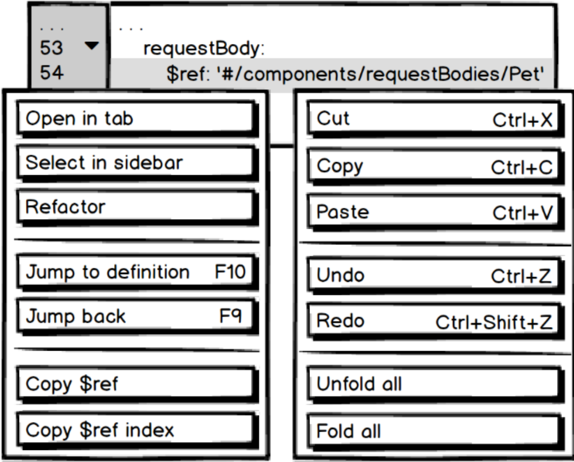


Abbildung 4.18: OpenAPI-IDE: Kontextmenü für die Editor-Zeilen

4. Formular

Die Optionen des Kontextmenüs im Formular unterscheiden sich abhängig von der Kategorie. In Abbildung 4.20 werden diese beispielhaft aus Platzgründen getrennt dargestellt. Der rote Rahmen in der Abbildung zeigt den Aktivierungsbereich für das Kontextmenü.

- **Load/Save example** wird nur bei Parameter-Elementen angezeigt und öffnet bei der Aktivierung das "Load/Save"-Pop-up (Pop-ups 3, 4).
- **Set to example** wird bei jedem Element angezeigt und überschreibt den aktuellen Wert mit dem example-Wert.
- **Set to null** wird bei jedem Element angezeigt. Es setzt den Wert auf "null" und tauscht das Element durch einen Knopf mit dem Text "NULL" aus (Abbildung 4.19).
  Bei einem Klick auf den "NULL"-Knopf, wird das vorherige Element wiederhergestellt und auf den example-Wert gesetzt. Wenn der Wert ungültig ist, wird stattdessen ein typspezifischer Neutralwert zugewiesen.

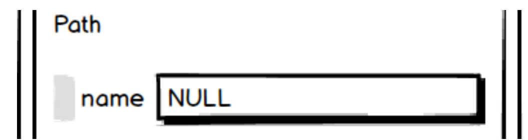


Abbildung 4.19: OpenAPI-IDE: NULL-Element

- **Disable field** wird bei allen Elementen angezeigt, ausgeschlossen sind die einzelnen Elemente eines Arrays. Es entfernt das Attribut aus der Anfrage, tauscht das Element durch einen Knopf mit dem Text "Disabled" und verhält sich wie der "Null"-Knopf.
- **Cut, Copy, Paste, Undo, Redo** verhält sich wie im Editor (Kontextmenüs 3) und wird bei jedem Element angezeigt.

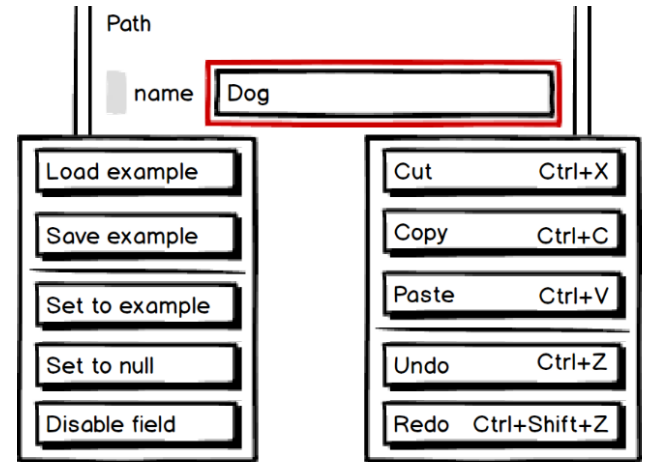


Abbildung 4.20: OpenAPI-IDE: Kontextmenü für die Elemente

4.2.7 Pop-ups

Im folgenden Unterabschnitt werden die Dialogfenster (Pop-ups) vorgestellt. Sie werden bei Vorgängen eingesetzt, die eine Benutzereingabe benötigen. Die meisten Pop-ups haben einen Knopf zum "Bestätigen" ✔ und "Abbrechen" ✖ des Vorgangs, jedoch gibt es Ausnahmen, die im jeweiligen Pop-up erläutert werden. Die Knöpfe werden oben rechts in der Pop-up-Leiste angezeigt, um Platz zu sparen. Ein Pop-up kann nur bestätigt werden, wenn alle Pflichtfelder (*) ausgefüllt sind. Ein Pop-up verhindert die Interaktion mit Elementen im Hintergrund, dies ermöglicht zum Beispiel ein Fokussieren der Pop-up-Elemente mit der Tab-Taste.

1. Einstellungen

Eine Änderung in den Einstellungen wird direkt angewendet, um eine Vorschau zu liefern, welches zum Beispiel bei einer Schriftgrößenänderung hilfreich ist (Abbildung 4.21). Der Reset-Knopf ⟲ setzt die Einstellungen auf den Installationszustand zurück.

- **Editor Font Size**: Die Schriftgröße für den Editor-Inhalt (Strg+Umschalt+Mausrad).
- **Primary Font Size**: Die Schriftgröße für alle Elemente außer dem Editor (Strg+Mausrad).
- **Show Summary**: Wenn aktiv, wird in der Sidebar die Schnittstellen-summary angezeigt.
- **Show Path**: Wenn aktiv, wird in der Sidebar der Schnittstellen-path angezeigt.
- **Indent Size**: Die Anzahl der Leerzeichen für die Einrückung der YAML-Texte.
- **Reload Delay In Milliseconds**: Die Zeitverzögerung für die automatische Validierung.
- **Request Timeout In Milliseconds**: Die Dauer, bis das System eine Anfrage abbricht.

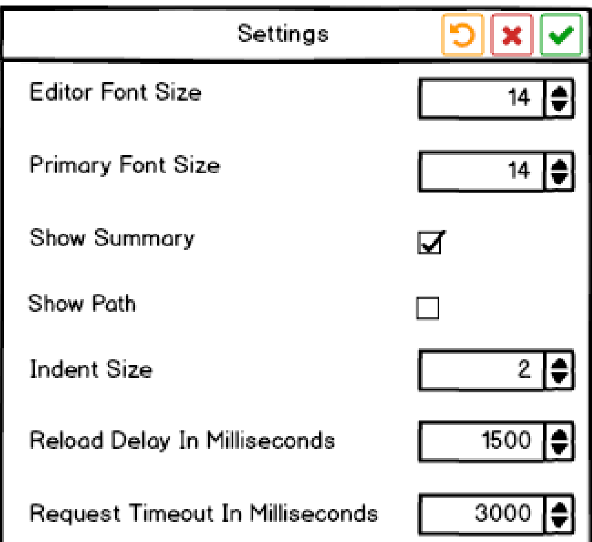


Abbildung 4.21: OpenAPI-IDE: Pop-up für die Einstellungen

2. Umstrukturierung (Refactoring)
Abbildung 4.22 zeigt das Pop-up für die Umstrukturierung. Das Pop-up erwartet eine "Id", welche für die eindeutige Zuordnung des Objekts in der components-Eigenschaft eingesetzt wird (Swagger Editor 3). Das rote Rechteck wird angezeigt, wenn die "Id" bereits belegt ist. Der zugehörige Tooltip beschreibt, ob der enthaltene Wert mit der "Preview" übereinstimmt. Ein Objekt wird überschrieben, wenn das Pop-up mit einer bereits vorhandenen "Id" bestätigt wird. Die Preview ist ein Editor im Nur-Lese-Modus, welcher das zu bewegende Objekt anzeigt.

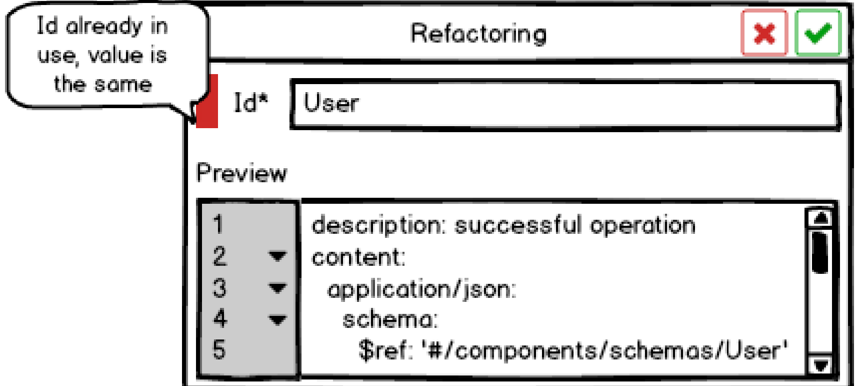


Abbildung 4.22: OpenAPI-IDE: Pop-up für die Umstrukturierung

3. Beispiel speichern
Abbildung 4.23 zeigt das Pop-up zum Ablegen von Beispieldaten in der OpenAPI-Definition. Der "Preview"-Tooltip zeigt eine Liste aller vorhandenen Fehler der value-Eigenschaft an.

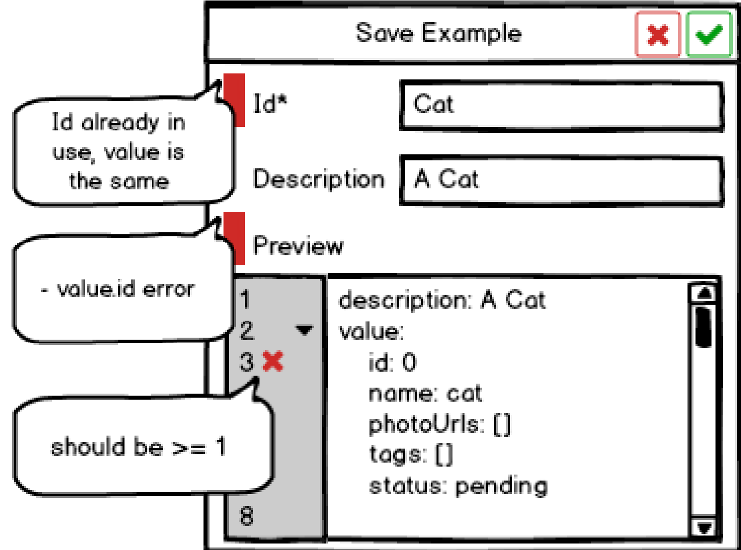


Abbildung 4.23: OpenAPI-IDE: Pop-up zum Speichern eines Beispiels

4. Beispiel laden
Beim Laden eines Beispiels ist das "Id"-Feld ein aufklappbares Auswahlmenü mit allen vorhandenen examples. Der Tooltip eines "Id"-Eintrags zeigt eine Vorschau der Beispieldaten an. Wenn ein Beispiel gewählt wurde, ermöglicht der "Preview"-Editor nochmals das Bearbeiten und Validieren der Beispieldaten, welche mit dem Bestätigen-Knopf ☑ in das Formular geladen werden können (Abbildung 4.24).

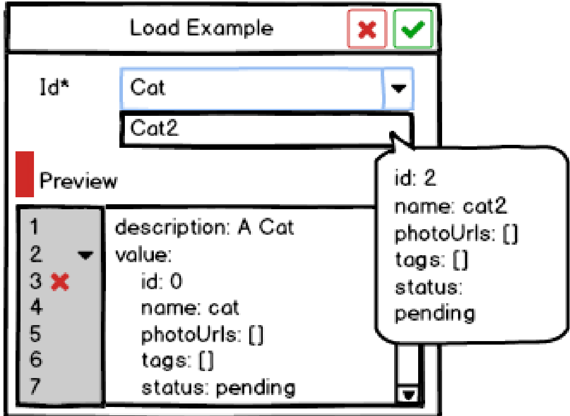


Abbildung 4.24: OpenAPI-IDE: Pop-up zum Laden eines Beispiels

5. Test speichern
Das Pop-up zum Speichern eines Tests hat zwei Spalten, die durch ein horizontales Schiebeelement getrennt werden. Die linke Seite enthält die Testdaten, wobei das Async-Element entscheidet, ob der Test im Hintergrund ausgeführt werden soll. Der Run-Knopf ▶ im "Preview"-Feld sendet die Anfrage und legt die Antwort in der expected-Eigenschaft ab, wodurch die Erstellung einer automatisierten Validierung vereinfacht wird. Die rechte Seite zeigt den aktuellen Testbaum, der eine Verkettung der Tests ermöglicht. Das Pop-up kann nicht bestätigt werden, wenn die "Id" bereits im Testbaum vorhanden ist (Abbildung 4.25).

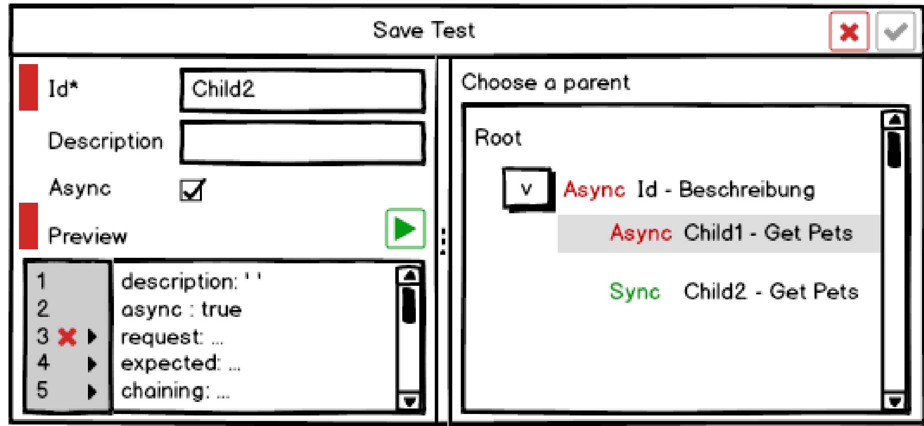


Abbildung 4.25: OpenAPI-IDE: Pop-up zum Speichern eines Tests

6. Test bearbeiten

Bei der Bearbeitung eines Tests wird nur die linke Spalte von "Test speichern" (Pop-ups 5) angezeigt. Eine Bearbeitung der "Id" ist in diesem Pop-up nicht möglich (Abbildung 4.26).

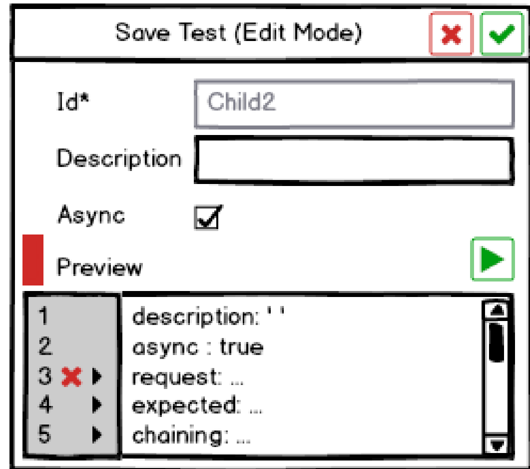


Abbildung 4.26: OpenAPI-IDE: Pop-up zum Bearbeiten eines Tests

7. OpenAPI-IDE schließen

Bei nicht gespeicherten Änderungen der OpenAPI- oder TestAPI-Definition, wird beim Schließen der Anwendung ein Hinweis angezeigt. Mit "Speichern" wird derselbe Vorgang ausgelöst, wie das "Speichern" im Output-Bereich (4.2.2). Der Verwerfen-Knopf beendet die Anwendung ohne "Speichern" (Abbildung 4.27).

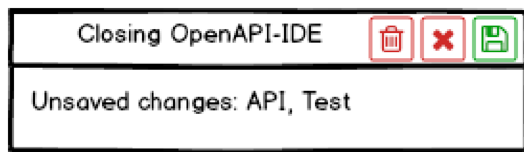


Abbildung 4.27: OpenAPI-IDE: Pop-up zum Schließen der Applikation

4.3 Fazit

In diesem Kapitel wurden die grundlegenden Funktionalitäten zum Bearbeiten der OpenAPI- und TestAPI-Definition spezifiziert.

Es wird eine Test-Erzeugung anhand des Formulars ermöglicht, welche bei Bedarf verkettet ausgeführt und automatisiert validiert werden. Die Beispieldaten können über ein Pop-up in der OpenAPI-Definition abgelegt oder in das Formular geladen werden.

Es gibt die Möglichkeit einer Umstrukturierung im OpenAPI-Bereich und das Anzeigen von Definitionsstellen auf Knopfdruck. Außerdem wird ein einheitliches Formular für das Erstellen von Anfragen zur Verfügung gestellt. Des Weiteren wird im ganzen System auf Abweichungen zur OpenAPI-Definition geprüft und hingewiesen.

5 Architektur

Im folgenden Kapitel wird die Architektur vorgestellt. Zunächst wird die Lösungsstrategie aufgelistet, anschließend werden die verschiedenen Sichten gezeigt und zum Schluss die Entwurfsentscheidungen erläutert.
Als Vorlage für dieses Kapitel wurde arc42 eingesetzt (vgl. Starke u. a. 2019).

5.1 Lösungsstrategie

Der folgende Abschnitt zeigt die Qualitätsziele der Architektur.

Qualitätsziel	**Einsatzgebiet und Lösung**
Erweiterbarkeit	Das System bietet Funktionalitäten verschiedenster Bereiche, welche als Vorlage für die Erweiterung des Systems eingesetzt werden können.
Attraktivität	• Ein minimalistischer Aufbau der Benutzeroberfläche. • Überschaubare Funktionalitäten. • Der Benutzer benötigt keine Programmierkenntnisse, sondern nur OpenAPI/YAML-Erfahrung. • Bearbeiten und Austauschen des Farbschemas über eine Konfigurationsdatei.
Wiederverwertbarkeit	Der Systemkern ist modular aufgebaut und kann zum Beispiel in einem System für YAML-basierte Anwendungsfälle eingesetzt werden.
Effizienz	Die Formular-Generierung kann bei großen Schemas zu Wartezeiten führen. Die Generierung wird auf ein Minimum beschränkt und nur für aktualisierte Bereiche ausgeführt.
Installierbarkeit	Es wird keine Installation benötigt, da eine Stand-Alone-Datei angeboten wird.

5.2 Sichten

Die Sichten beschreiben die Bestandteile und die Kommunikationswege des Systems. Zunächst wird die Bausteinsicht vorgestellt, dann die Innensicht und zum Schluss die Laufzeitsicht.

5.2.1 Bausteinsicht

Die Bausteinsicht besteht aus den Bereichen: Main, GUI und Core (Abbildung 5.1):

- **Main** ist zuständig für die Erzeugung des Electron-Fensters, Verwaltung der Kommandozeilenparameter und die Initialisierung aller Komponenten.
- **GUI** füllt das Electron-Fenster mit grafischen Elementen und definiert ihr Verhalten.
- **Core** enthält den Logikkern des Systems. Er liefert beispielsweise Hilfsfunktionen zum Arbeiten mit den API-Definitionen und für die Erzeugung der grafischen Elemente.

Zyklische Abhängigkeiten gibt es zwischen Sidebar, Form und Output, diese werden in der Abbildung mit Pfeilen angedeutet. Eine Auflösung der Abhängigkeiten ist nicht nötig, da die Komponenten durch den Verzicht auf Dependency Injection unabhängig voneinander initialisiert werden können (siehe 5.3 Entwurfsentscheidungen).

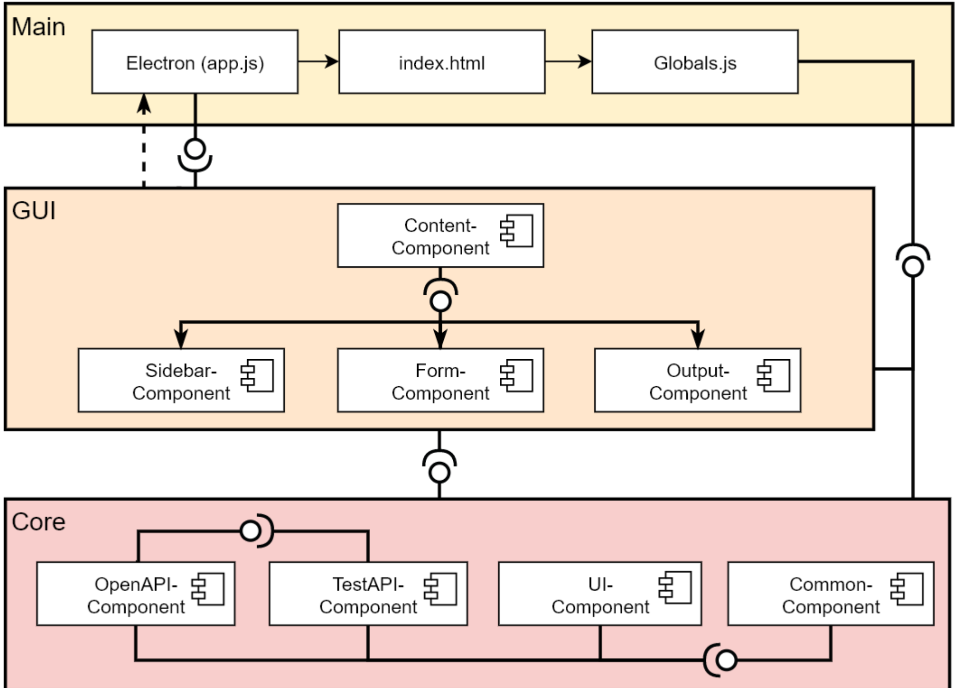


Abbildung 5.1: Bausteinsicht

5.2.2 Innensicht

Die Innensicht beschreibt den Aufgabenbereich und zeigt die einzelnen Klassen einer Komponente, welche in zwei Schichten aufgeteilt werden:

- Die Zugriffsschicht enthält die Fassade, sie ist für den Zugriff auf die Komponente zuständig und leitet Funktionensaufrufe an die verantwortlichen Klassen in der Logikschicht weiter.
- Die Logikschicht enthält Klassen, welche Experten für einen Aufgabenbereich sind. Eine isolierte Klasse vereinfacht die Code-Verwaltung, dadurch müssen keine Beziehungen gepflegt werden und das Nachvollziehen einer Funktion geschieht unabhängig von anderen Klassen. In manchen Situationen wird eine Beziehung mit anderen Komponenten oder Klassen eingegangen, um ein Kopieren von Code zu meiden, da dies zu einem größeren Verwaltungsaufwand führen würde.

GUI (Abbildung 5.2)

- **Content** unterteilt die Komponenten: Sidebar, Form und Output in drei Spalten und verwaltet die Tastenkürzel des Systems.
- **Sidebar** stellt die API- und Test-Ansicht zur Verfügung. Die API-Ansicht liefert eine Liste mit den Schnittstellen der OpenAPI-Definition und die Test-Ansicht enthält eine Baumstruktur mit den Tests der TestAPI-Definition.
- **Form** ist für die Formular-Generierung und Erzeugung der request zuständig. Außerdem stellt sie die Pop-ups für die Testerstellung und Verwaltung der examples zur Verfügung.
- **Output** verwaltet das Laden und Speichern der OpenAPI/TestAPI. Die Komponente ermöglicht das Bearbeiten der request, OpenAPI und TestAPI in einem Editor. Zusätzlich stellt sie Pop-ups zur Verfügung für die System-Einstellungen, das OpenAPI-Refactoring und das Schließen der Applikation.

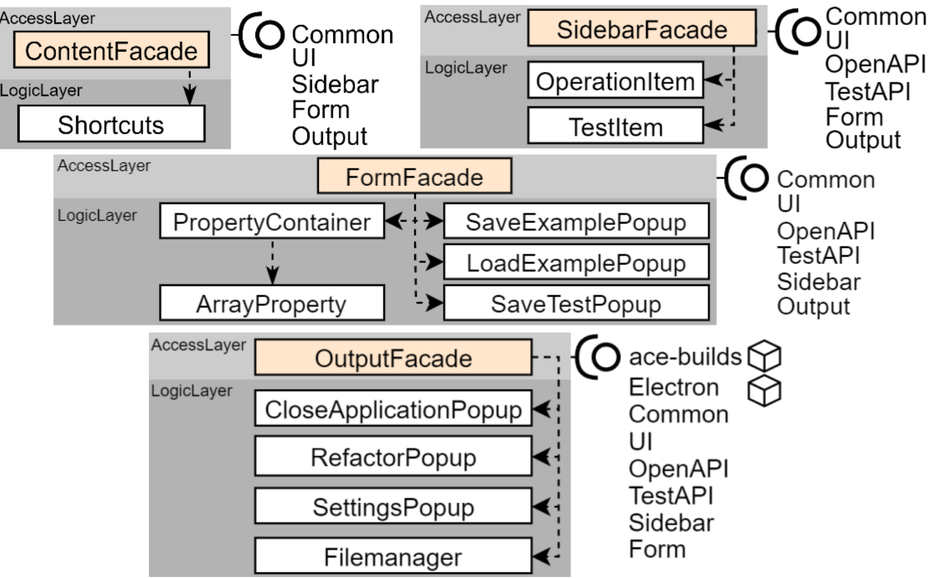


Abbildung 5.2: GUI-Innensicht

Core (Abbildung 5.3)

- **OpenAPI** liefert Funktionalitäten zum Validieren einer OpenAPI-Definition und vereinfacht den Zugriff auf die Schnittstellen, Examples und Schemas.
- **TestAPI** liefert Funktionalitäten zum Validieren einer TestAPI-Definition und ermöglicht das Hinzufügen und Ausführen von Tests.
- **UI** hilft bei der Erzeugung von grafischen Elementen und vereinfacht das Arbeiten mit den Less-Variablen.
- **Common** liefert Hilfs-Funktionen für diverse Anwendungsbereiche:
  - **ObjectWrap** kann zum Beispiel einen Wert durch Angabe eines $ref-Pfads liefern.
  - **YamlWrap** kann zum Beispiel anhand einer Cursor-Position den $ref-Pfad zum Objekt und umgekehrt liefern.
  - **HTTPRequest** hilft bei der URL-Erzeugung und sendet die HTTP-Anfragen.
  - **Config** ermöglicht das Arbeiten mit Konfigurationsdateien, welche auf der Festplatte abgelegt werden.
  - **CommonFacade** initialisiert die Common-Klassen global. Die Fassade stellt keine API zur Verfügung, da die Komponente kein Experte für einen Bereich ist und die Verwaltung der Delegations-Funktionen keinen Mehrwert liefern würde.

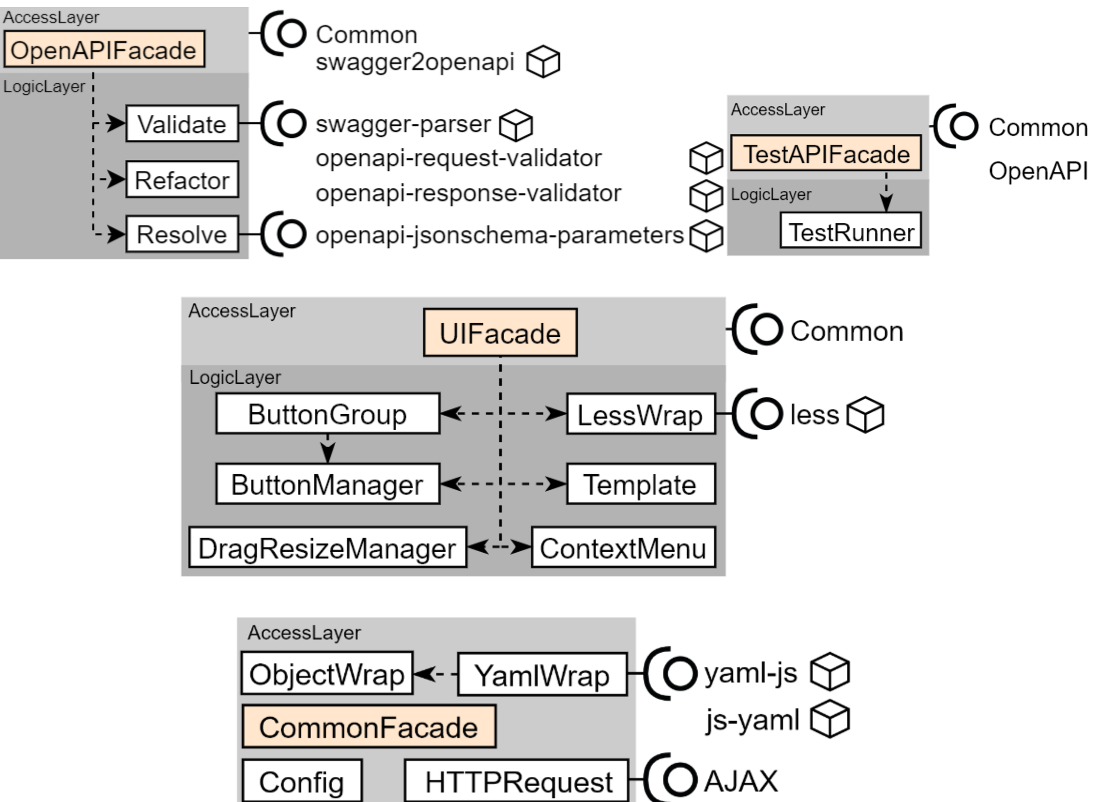


Abbildung 5.3: Core-Innensicht

Strukturen

Im Folgenden werden die Strukturen von Debug, TestAPI und Test erläutert. Sie wurden für dieses System entwickelt und werden in den Komponenten Sidebar, Form, Output, OpenAPI und TestAPI eingesetzt.

1. Debug-Struktur

Das Debug-Objekt besteht aus den Eigenschaften request und response.

Die request enthält alle Informationen zum Senden einer HTTP-Anfrage und kann über die Form-Fassade erzeugt werden (Abbildung 5.4). Die Schlüssel mit einem Unterstrich an erster Stelle dürfen nicht manuell über den Editor bearbeitet werden, da es zum Beispiel keine Möglichkeit gibt die _url in _server und _endpoint zu trennen. Das Anpassen von _server, _endpoint oder _type ist theoretisch möglich, jedoch ist der Aufwand im Rahmen der Bachelorarbeit zu groß, da der Benutzer diese Änderungen über die UI vornehmen kann. Die _url zeigt den zusammengebauten Wert, welcher aus _server, _endpoint und query besteht. Die errors-Eigenschaft enthält alle Abweichungen des Formulars zur OpenAPI-Definition.

```
1  request:
2     _url: 'http://petstore.swagger.io/v2/pet/dog?id=1'
3     _server: 'http://petstore.swagger.io/v2'
4     _endpoint: /pet/{name}
5     _type: POST
6     cookie: ...
7     query: ...
8     path: ...
9     header: ...
10    body: ...
11    errors: ...
```

Abbildung 5.4: Das request-Objekt

Die response wird anhand der HTTP-Antwort in der OpenAPI-Fassade erzeugt und hat folgende Struktur (Abbildung 5.5).

```
12 response:
13    status: 200
14    time: 0.042s
15    header: ...
16    body: ...
17    errors: ...
```

Abbildung 5.5: Das response-Objekt

2. TestAPI-Struktur

Die TestAPI wurde für dieses System entwickelt, um die Anforderungen aus Abschnitt 3.3 zu erfüllen.
Eine TestAPI-Definition besteht aus einer Version (testapi) und den tests (Abbildung 5.6).

Die request eines Test-Objekts (Zeile 6) entspricht der Debug-Struktur mit folgenden Abweichungen:

- Die _url ist nicht vorhanden, da die Anpassung eines path- oder query-Werts eine Aktualisierung der _url benötigen würde. Außerdem nimmt dieser Wert viel Platz ein und kann über den "Edit Mode" betrachtet werden.
- Die Anfrage-Fehler werden erst bei der Testausführung angezeigt. Dies ist aus Platzgründen vorteilhaft und vereinfacht die Umsetzung, da keine Aktualisierung der errors nach der Bearbeitung eines Tests durchgeführt werden muss.

In expected können folgende Eigenschaften angegeben werden: status, header, body, errors und errorsCount, welche automatisch und unabhängig voneinander über eine strikte Gleichheit oder deepEqual-Funktion mit der gelieferten Antwort verglichen werden.
Die automatisierte Validierung sieht fehlende expected-Eigenschaften und Tests mit einem leeren expected-Objekt als erfolgreiche an, wodurch die Anfragen verkettet und ohne Validierung ausgeführt werden können.

Mit chaining können Tests verschachtelt werden, dadurch entsteht eine Baumstruktur, die das Ausführen eines Zweigs oder des ganzen Baums ermöglicht.

```yaml
1  testapi: 0.0.1
2  tests:
3    Test1:
4      description: ``
5      async: true
6      request: ...
7      expected:
8        status: 200
9      chaining:
10       Test2: ...
11       Test3: ...
```

Abbildung 5.6: TestAPI-Struktur

5.2.3 Laufzeitsicht

Der folgende Unterabschnitt zeigt Sequenzdiagramme, welche den Kernzyklus des Systems wiederspiegeln. Die folgenden Szenarien beziehen sich alle auf die OpenAPI, da die TestAPI vom Aufbau sehr ähnlich ist.

Aus Gründen der Übersichtlichkeit werden die Funktionsargumente in den folgenden Diagrammen teilweise abgekürzt und einige Operationen weggelassen, wie zum Beispiel das Aktualisieren oder Erstellen einzelner GUI-Elemente. Ein Beispiel für den Aufbau der Sequenzdiagramme befindet sich in Abbildung 5.7.

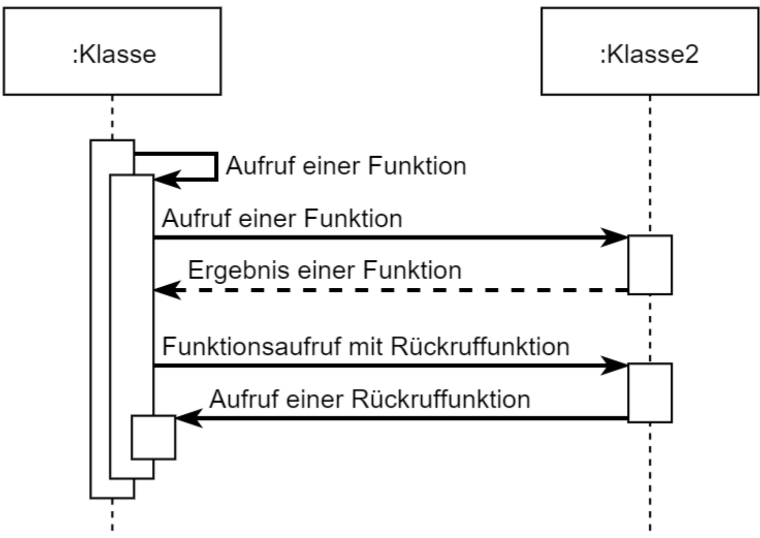


Abbildung 5.7: Beispiel für den Aufbau eines Sequenzdiagramms

1. OpenAPI-Definition laden
Der Benutzer möchte eine OpenAPI laden (Abbildung 5.8).

- (1, 2) Bevor das "Öffnen"-Pop-up angezeigt wird, prüft das System, ob die aktuelle API-Datei ungesicherte Änderungen enthält und ermöglicht dem Benutzer diese zu speichern.
- (3) Nachdem die OpenAPI-Datei ausgewählt wurde, wird der aktive Dateipfad für die Speichern-Funktionalität aktualisiert. Im Anschluss wird geprüft, welche API vorliegt, wodurch der entsprechende editorMode (API/Test) aktiviert wird.
- (4-8) Als nächstes wird die geöffnete Textdatei in das System geladen, da es sich um eine neue Datei handelt (initialer Ladevorgang), wird die Undo-Historie des Editors zurückgesetzt, wodurch das versehentliche Überspeichern einer API verringert wird.
  Der Ladevorgang geschieht in zwei Schritten, diese werden asynchron ausgeführt, da die eingesetzten Node-Pakete keine synchronen Funktionen zur Verfügung stellen.
  - (5-6) Wenn es sich um eine Swagger-Definition handelt, wird eine Umwandlung in das OpenAPI-Format durchgeführt.
  - (7-8) Wenn die OpenAPI-Definition keine Fehler enthält, liefert die Funktion das openAPIObj und die refs (6.4.3).
- (9) Der API-Text wird auch im Fehlerfall zum Editor hinzugefügt, damit der Benutzer die Möglichkeit hat, vorhandene Fehler zu beseitigen.

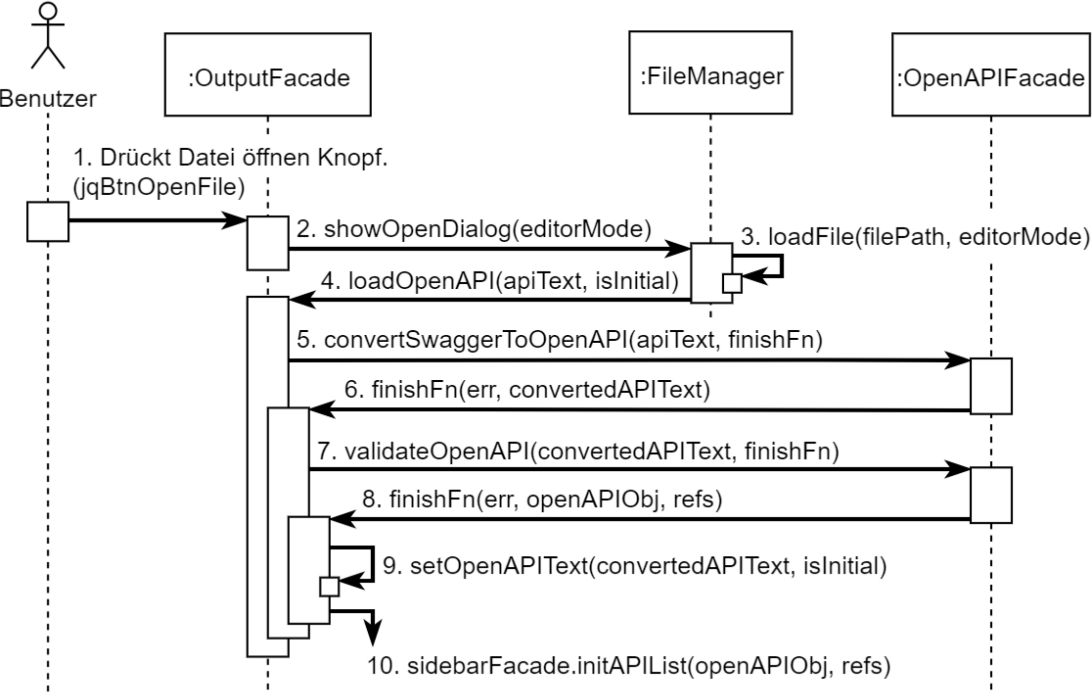


Abbildung 5.8: Sequenzdiagramm: OpenAPI-Definition laden

2. Sidebar initialisieren

Das folgende Diagramm beschreibt die Erzeugung der Schnittstellenliste in der Sidebar (Abbildung 5.9).

- (1) Folgende Variablen werden in der OpenAPI-Fassade zwischengespeichert:
  (3) openAPIObj, refs
  (6) endpoint, type
  (7) contentType
  Sie werden bei undefinierten Argumenten eines Funktionsaufrufs in der OpenAPI-Fassade eingesetzt, um die Anzahl der benötigten Variablen zu verringern.
- (2) Die aktuelle request wird in sidebarFacade.customData zwischengespeichert. Sie wird zum Wiederherstellen der Formulardaten bei Änderungen der OpenAPI-Definition eingesetzt.
- (4) Erzeugt alle Schnittstelleelemente.
- (5) Falls möglich, wird die vorherige Auswahl wiederhergestellt:
  - Wenn die ausgewählte Schnittstelle entfernt wurde, wird die erste Schnittstelle ausgewählt.
  - Falls keine Schnittstelle vorhanden ist, wird Schritt 7 nicht ausgeführt.

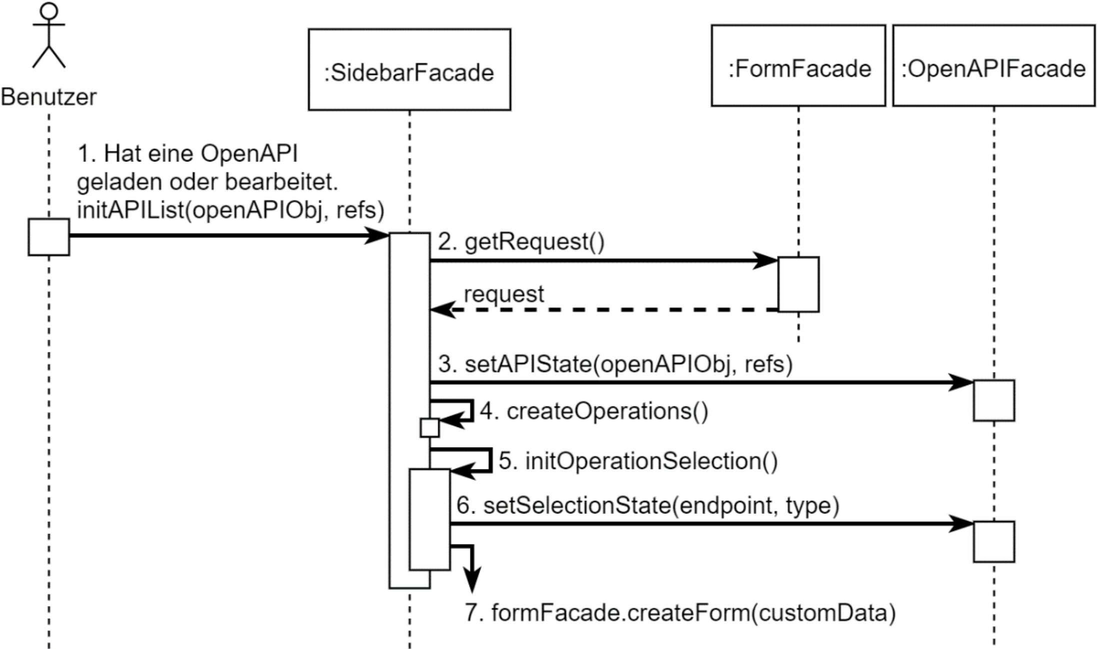


Abbildung 5.9: Sequenzdiagramm: Sidebar initialisieren

3. Formular initialisieren

Der Vorgang in Abbildung 5.10-5.11 wird nach der Auswahl einer Schnittstelle ausgelöst.

- (2-3) Das Formular wird in einen leeren Zustand versetzt und abhängig vom aktiven Tab initialisiert:
    - Im Falle einer Schnittstellenwahl, wird das Root-Tab erzeugt (pathToProperty='body').
    - Bei einer OpenAPI- oder Debug-Bearbeitung werden die vorherigen Tabs wiederhergestellt und das aktuelle Formular neu generiert, welches durch eine Ausführung von "openSubForm" umgesetzt wird (ohne "createForm").
- (4) Die Body-Informationen werden abgerufen und für Folgendes eingesetzt:
    - Das schema wird zwischengespeichert (formFacade.treeData) und in Schritt 13 für die Formular-Validierung eingesetzt.
    - Die zuständige Dropdown-Liste wird mit den contentTypes gefüllt. Anschließend erfolgt eine Wiederherstellung der vorherigen Auswahl mithilfe der customData.
    - Der Boolean isRootArray wird ebenfalls zwischengespeichert. Er wird benötigt, um die Handhabung eines Spezialfalls zu vereinfachen, bei dem sich im requestBody ein Schema vom Typ "array" anstatt "object" befindet. Das Array customData.body wird in diesem Fall zu einem Objekt mit einem Attribut vom Typ "array" umgewandelt (customData.body.array).
- (5-6) Die parameters-Schema werden zwischengespeichert. Außerdem werden hier bereits die UI-Elemente für die parameters erzeugt, wie in Schritt 11 für den requestBody.

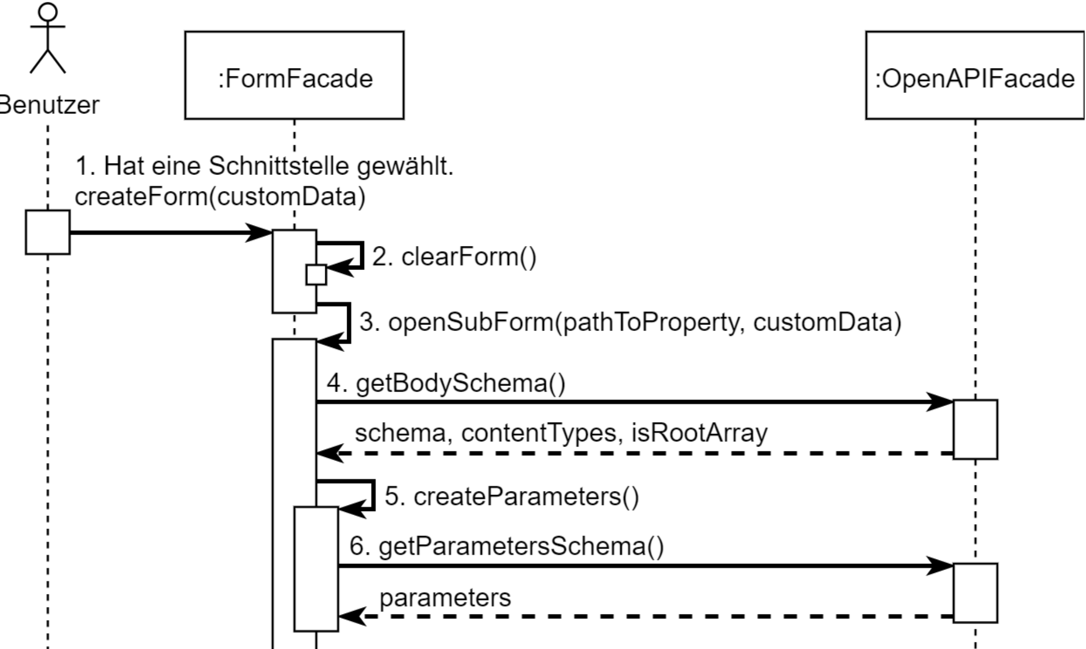


Abbildung 5.10: Sequenzdiagramm: Formular initialisieren Teil 1

- (7) Erzeugt das Body-Formular für das Objekt, welches sich in customData an Position pathToProperty befindet.
- (8) Prüft anhand der pathToProperty, ob das Objekt zum aktiven Tab in customData vorhanden ist und liefert den Best-Effort-Pfad (validTabPath).
- (9) Ruft das Schema ab, welches sich im Best-Effort-Pfad befindet.
- (10) Erzeugt die einzelnen Tab-Knöpfe bis zum aktuellen Schema.
- (11) Die Erzeugung der einzelnen UI-Elemente geschieht anhand des aktuellen Schemas, dabei werden Attributdaten gesammelt, wie zum Beispiel pId, example und der type. Bei einem Attribut vom Typ "array" wird entschieden, ob es sich um ein Enum-Feld handelt, da es keinen type "enum" gibt.
- (12) Die gesammelten Informationen werden der Klasse "PropertyContainer" übergeben, welche das UI-Element erzeugt und dem jqContainer hinzufügt.
- (13-14) Prüft alle Kategorien anhand ihres Schemas, ob der customData-Wert dem definierten Typ entspricht. Bei einer bestehenden Abweichung wird eine Umwandlung durchgeführt. Wenn diese nicht möglich ist, wird der example-Wert eingesetzt, falls keiner definiert wurde, wird ein typspezifischer Neutralwert zugewiesen. Zusätzlich wird sichergestellt, dass die Ordnung der customData-Eigenschaften mit der Reihenfolge des Schemas übereinstimmt.
- (15) Erzeugt die request und fügt sie in den Debug-Editor ein (Laufzeitsicht 4 Schritt 4).

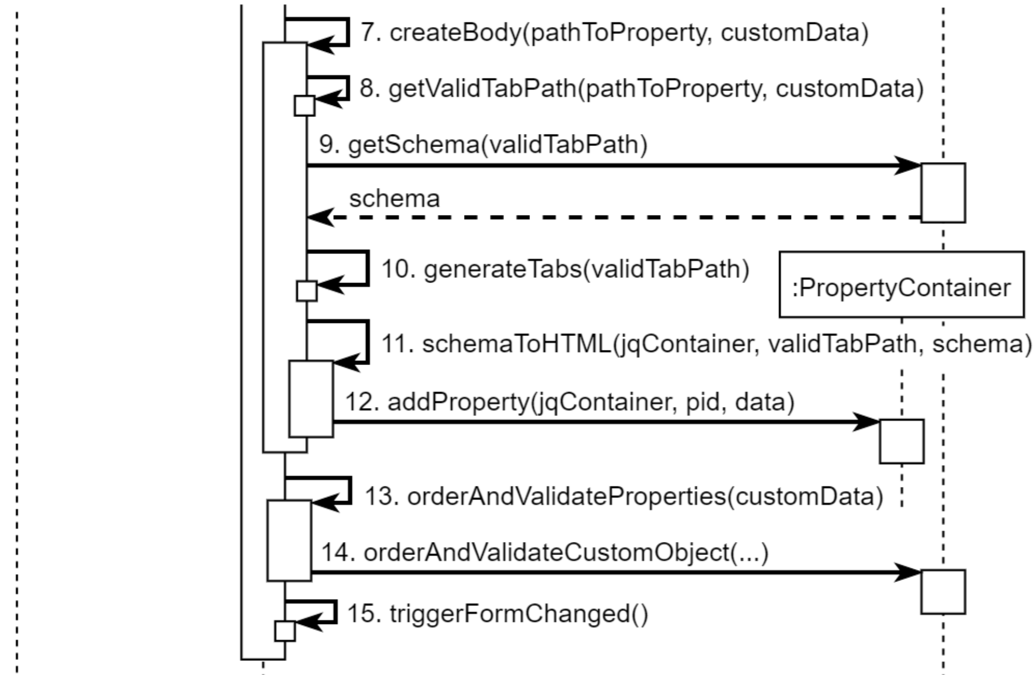


Abbildung 5.11: Sequenzdiagramm: Formular initialisieren Teil 2

4. Formular bearbeiten

Der Benutzer hat im Formular einen Attributwert geändert, dadurch wird ein "change"-Event ausgelöst (Abbildung 5.12).

- (1) Vor der Ausführung von "valueChanged", wird der eingegebene Wert bei zum Beispiel einem Zahlenelement in einen number-Typ gewandelt, da JavaScript unabhängig vom Typ des Eingabeelements einen String liefert.
- (2) "updateNullDisabledState" prüft, ob der value einen null- oder undefined-Wert enthält, wodurch das Eingabeelement in einen Null- oder Disabled-Zustand versetzt wird.
- (3) Der value wird mithilfe der Element-id (pathToProperty) in formFacade.treeData abgelegt.
- (4-8) Aktualisiert das Formular und den Debug-Editor:
  - (5) Die request wird anhand der formFacade.treeData erzeugt.
  - (6) Die errors sind in der request enthalten und werden für die Fehlerhinweise der einzelnen Formular-Elemente eingesetzt.
  - (7) Die Tab-Tooltips werden aktualisiert (Abbildung 4.7).
  - (8) Setzt den Anfragetext im Debug-Editor.

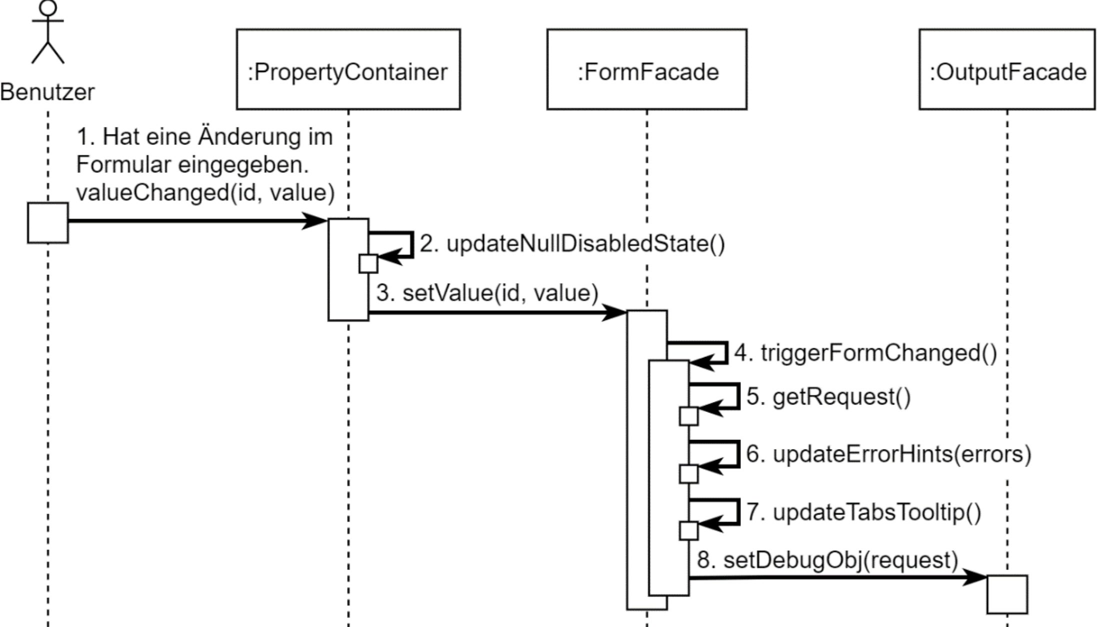


Abbildung 5.12: Sequenzdiagramm: Formular bearbeiten

5. Anfrage senden
Der Benutzer wählt eine Schnittstelle aus der Sidebar und drückt auf den Senden-Knopf (Abbildung 5.13).

- (2-3) Die Form-Fassade liefert die request, welche alle Informationen zum Senden der Anfrage enthält.
- (4) Um ein Blockieren der UI zu verhindern, werden alle HTTP-Anfragen asynchron ausgeführt (finishFn). Das Senden der Anfrage erwartet ein Array, welches zum Abbrechen der laufenden Anfrage(n) eingesetzt werden kann.
- (5) Ruft die Antwortdaten aus dem event ab und erzeugt das response-Objekt.
- (6-7) Das debugObj enthält die request und response (siehe Strukturen 1) und wird zum Aktualisieren des Debug-Editors eingesetzt.

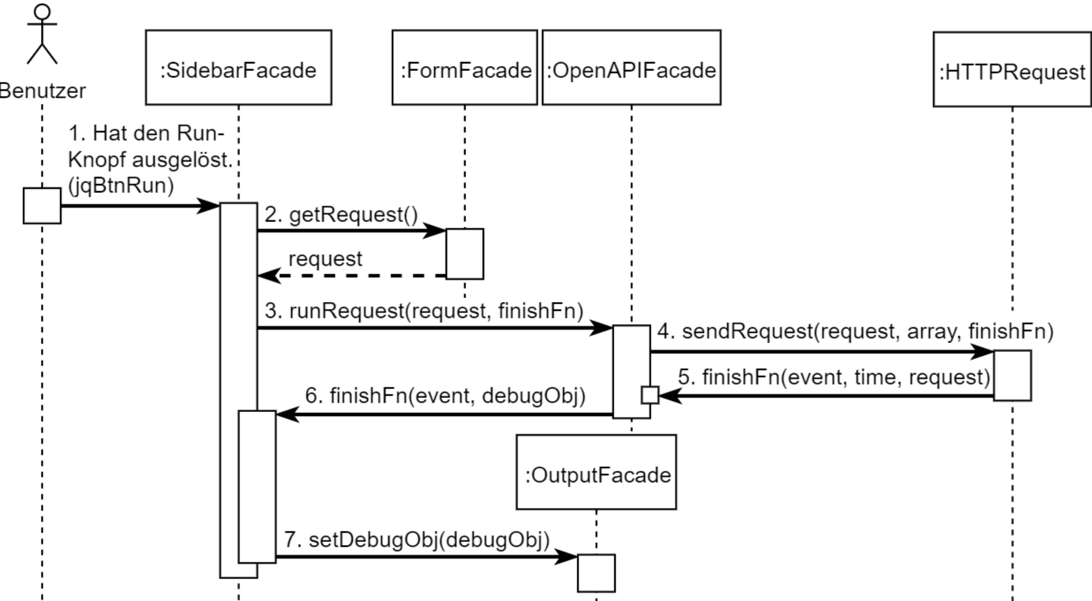


Abbildung 5.13: Sequenzdiagramm: Anfrage senden

5.3 Entwurfsentscheidungen

Die vorgestellte Architektur folgt Prinzipien, die sich in der Softwareentwicklung bewährt haben und zur Robustheit des Systems beitragen (austauschbar, wartbar, testbar, lesbar, …). Ein Prinzip lässt sich nicht nur einem Vorteil zuordnen, sondern hat Einfluss auf viele Eigenschaften der Architektur. Im Rahmen der Bachelorarbeit steht die Umsetzung im Vordergrund, deshalb wird großen Wert auf Prinzipien gesetzt, die zu einem lesbaren und testbaren Programmcode führen, welches mit einem modularen System erreicht werden kann. Als Vorlage dient die Quasar sd&m Standardarchitektur (vgl. Siedersleben 2003). Sie definiert unter anderem verschiedene Schnittstellenkategorien, die für eine Modularisierung des Systems eingesetzt werden. JavaScript hat keine Typsicherheit-, Interfaces- oder Public/Private-Konstrukte, deshalb ist der Einsatz von Komponenten abhängigen Sichten anhand der Schnittstellenkategorien 0, T, A_F, A, A_x kaum möglich, die Folge ist eine starke Abhängigkeit zwischen den Klassen, wodurch nicht mehr von Modularisierung gesprochen werden kann (vgl. Clean Code Developer 2015a, Law of Demeter). Eine Lösung für dieses Problem sind die Fassaden. Auch wenn JavaScript keine Schnittstellensyntax zur Verfügung stellt, kann in der Implementierung eine Konvention eingehalten werden, bei der ein Zugriff auf ein Modul (Komponente) nur über die Fassade geschieht. Diese delegiert Funktionsaufrufe an die zuständigen Klassen und verhindert dadurch, die direkte Kommunikation zwischen den Logikklassen, wodurch die Kopplung im gesamten System verringert wird (vgl. Clean Code Developer 2015b, Information Hiding Principle). Eine Fassade umschließt einen funktionalen Bereich, dadurch entsteht eine Komponente und die Lesbarkeit steigt (vgl. Clean Code Developer 2015c, Separation of Concerns). Eine Komponente kann mehrere Klassen enthalten, da es verschiedene Aufgabenbereiche innerhalb eines funktionalen Bereichs geben kann, wie zum Beispiel die Logik und die Persistenz (vgl. Clean Code Developer 2015d, Single Responsibility Principle). Bei dem umgesetzten System handelt es sich um eine Schichtenarchitektur, da Klassen und Komponenten in zusammengehörige Schichten gruppiert werden.

Umsetzung

- Die Komponenten werden einmalig global initialisiert und sind im gesamten Renderer-Prozess erreichbar (Electron 6.1.1). Auf Dependency Injection wird verzichtet, weil das Weiterreichen von Komponenten zu einem größeren Verwaltungsaufwand führen würde (vgl. Clean Code Developer 2015e, Keep it simple, stupid).
- Funktionen, die in keiner Relation zu einer weiteren Funktion der Komponente stehen, werden in der Fassade definiert, um die Erstellung einer weiteren Logikklasse mit nur einer Funktion zu vermeiden.

5.4 Fazit

Durch den Einsatz von Architekturprinzipien wurde eine robuste Architektur entwickelt, welche modular aufgebaut ist.

6 Umsetzung

Das folgende Kapitel stellt die Umsetzung vor. Zunächst werden die verwendeten npm-Pakete vorgestellt und anschließend die Funktionsweise der Komponenten: UI, Common, OpenAPI und TestAPI beschrieben. Zum Schluss werden die Unit-Tests erläutert und das Kapitel mit einem Fazit zusammengefasst.

6.1 Verwendete npm-Pakete

Die Paketwahl für die Lösung eines Anwendungsfalls ist nicht immer leicht. Sie wird in einigen Fällen durch eine geringe Auswahl erleichtert, wie zum Beispiel bei dem Parser für die OpenAPI (swagger-parser). In anderen Fällen gibt es Pakete, die trotz einer großen Auswahl nicht alle Anforderungen erfüllen. Hier kann eine eigene Implementierung produktiver sein (Beispiel Config 6.3.1).
Ein Paket sollte über eine aktive Community verfügen, welches anhand der Paket-Statistik auf npmjs.com oder dem zugehörigen Repository (überwiegend auf GitHub) nachvollzogen werden kann. Eine aktive Community ermöglicht vorhandene Probleme mittels einer Pull-Request oder dem Öffnen eines Tickets zu lösen. In Unterabschnitt 6.1.1 werden die Pakete für die Implementierung gelistet und 6.1.2 zeigt die Pakete zum Testen des Systems.

6.1.1 Implementierung

Es folgt eine Beschreibung der verwendeten npm-Pakete für die Implementierung. Einige Pakete wurden bereits im Grundlagen Kapitel erläutert (Less 2.2.4, jQuery 2.2.5) oder sind irrelevant für die Implementierung, da sie zum Beispiel nur für den Code-Style zuständig sind (stylelint, eslint).

Electron (electron)
Mit Electron lässt sich ein Desktop-Fenster erzeugen, welches webbasierte Inhalte anzeigen kann (2.2.6). Da es sich um eine webbasierte Applikation handelt, wird der Eindruck erzeugt, dass die Implementierung auch im Webbrowser (Renderer) ausgeführt werden kann, leider ist dies nicht der Fall. Bei einem Electron-Projekt sollte möglichst früh entschieden werden, ob eine Webversion geplant ist, damit die Architektur entsprechend gestaltet werden kann (Beispiel "Speichern einer Datei").
Electron besteht aus einem Main- und Renderer-Prozess (Node-Server, Chromium) und ermöglicht die Nutzung von "require" im Renderer, wodurch die Implementation nicht im Webbrowser ausgeführt werden kann.
Die Kommunikation zwischen dem Main- und Renderer-Prozess geschieht in Electron über eine asynchrone Schnittstelle "Inter-Process-Communication (IPC)". Ein direkter Zugriff ist aus Sicherheitsgründen nicht möglich, da eine Website im Renderer-Prozess auf System-Operationen über den Node-Server zugreifen könnte (vgl. GitHub Inc. 2019b).

Beispiel:

- Im oberen Teil der Abbildung 6.1 wird der Zugriff auf eine Variable im Main-Prozess gezeigt. Ein direkter Zugriff mittels `remote.global.path` ist nicht möglich.
- Der untere Teil zeigt das Setzen einer Variable, welches über ein Signal umgesetzt wird.

	Main	Renderer
1	`global.path = './test'`	`const remote = require('electron').remote`
2		`var path = remote.getGlobal('path')`
1	`const ipcM = require('electron').ipcMain`	`const ipcR = require('electron').ipcRenderer`
2	`ipcM.on('isClosing', (event, isClosing) => {`	`ipcR.send('isClosing', true)`
3	`  global.isClosing = isClosing`	
4	`})`	

Abbildung 6.1: Electron: Inter-Process-Communication

Electron Window State (electron-window-state)
Dieses Paket speichert den Fensterzustand, welcher beim Starten der Applikation automatische wiederhergestellt wird.

Electron Builder (electron-builder)
Das Paket ermöglicht die Erzeugung der Stand-Alone-Dateien für die Plattformen: Windows, Mac und Linux. Mit dem Befehl `npm run dist` wird die Stand-Alone-Datei für das laufende Betriebssystem erzeugt. Die Einstellungen können in der Datei "package.json" durchgeführt werden (Abbildung 6.2).

Beispiel: package.json

```
 1 ...
 2 "build": {
 3   "appId": "de.alexanderholland.tool.openapi-ide",
 4   "win": {
 5     "artifactName": "${productName}v${version}.exe",
 6     "target": ["portable"],
 7     "icon": "src/icon/icon.ico"
 8   },
 9   "mac": {
10     "artifactName": "${productName}v${version}.dmg",
11     "category": "de.alexanderholland.tool.openapi-ide",
12     "icon": "src/icon/icon.icns"
13   },
14   "linux": {
15     "artifactName": "${productName}v${version}",
16     "target": ["AppImage"],
17     "icon": "src/icon/icon.png",
18     "category": "Utility"
19   }
20 },
21 "scripts": {
22   "dist": "electron-builder"
23 }
24 ...
```

Abbildung 6.2: Konfiguration für die Erstellung der Stand-Alone-Dateien

Die Icons werden abhängig vom System in verschiedenen Formaten erwartet. Diese können zum Beispiel in Linux über ein Shell-Skript mittels icnsutils, imagemagick und librsvg2-bin anhand einer SVG-Datei (Scalable Vector Graphics) automatisch generiert werden.

Electron Args (electron-args)
Dieses Paket vereinfacht die Befehlserstellung für die Eingabeaufforderung, wodurch das Öffnen einer API und die Testausführung über die Konsole ermöglicht wird (Abbildung 6.3).

Beispiel: app.js

```
1  // Read command line arguments
2  const parseArgs = require('electron-args')
3  const cli = parseArgs(`
4    openapi-ide
5
6    Usage
7      $ openapi-ide [option] [path]
8
9    Options
10     -help      Show Help
11     -api       OpenAPI File
12     -test      TestAPI File
13     -run       Run Tests on Start
14
15   Examples
16     $ openapi-ide -a ./api.yaml
17     $ openapi-ide -a ./api.yaml -t ./test.yaml
18     $ openapi-ide -r -a ./api.yaml -t ./test.yaml
19   `,
20 {
21   alias: {
22     h: 'help',
23     a: 'api',
24     t: 'test',
25     r: 'run'
26   }
27 })
28
29 global.apiPath = cli.flags.api
30 global.testPath = cli.flags.test
31 global.run = cli.flags.run
```

Abbildung 6.3: Kommandozeilenparameter

Ace-Editor (ace-builds)
Der Ace-Editor hat eine umfangreiche Dokumentation, welche sehr verstreut wirkt. Es gibt viele Beispiele für die Integration in einen Browser (Renderer), jedoch keine Hinweise auf ein npm-Paket oder Beispiele für das Einbinden in ein Node-Projekt. In Abbildung 6.4 wird das Paket geladen und die pfadabhängigen Einstellungen gesetzt. Die Vorteile eines npm-Pakets wurden in Unterabschnitt 2.2.2 beschrieben.

```
1 ...
2 const Ace = require('ace-builds/src/ace.js')
3 Ace.config.set('basePath', path.join(projectRootPath, 'node_modules/ace-builds/src/'))
4 ...
5 editor.setTheme('ace/theme/twilight')
6 editor.session.setMode('ace/mode/yaml')
```

Abbildung 6.4: Ace-Editor im Node-Projekt

Swagger Parser (swagger-parser)
Der Swagger Parser stellt Funktionalitäten für das Arbeiten mit der OpenAPI-Definition zur Verfügung und wird für verschiedene Programmiersprachen unterstützt, wie zum Beispiel Java oder JavaScript. Der Parser kann eine OpenAPI-Definition validieren und bei Unstimmigkeiten zum Standard einen Hinweis liefern.
Eine weitere Funktionalität hilft bei der Auflösung von Referenzen, dabei erfolgt eine Ersetzung aller $ref-Pfade durch die verlinkten Objekte. Alternativ kann der Parser ein Verzeichnis liefern, bei dem die referenzierten Objekte anhand des $ref-Pfads abgerufen werden können, welches zum Lösen von zyklischen Abhängigkeiten eingesetzt werden kann.

Swagger zu OpenAPI (swagger2openapi)
Dieses Paket wandelt eine Swagger-Definition in eine OpenAPI-Definition um.

YAML-JS (yaml-js)
Das Paket ist eine PyYAML-Portierung und kann für einen YAML-Text den zugehörigen Abstract Syntax Tree (AST) erzeugen, welcher für "Copy $ref" und "Jump to Definition" benötigt wird. Leider erzeugt die stringify-Funktion des Pakets eine unübersichtliche Repräsentation und wird nicht mehr weiterentwickelt.

JS-YAML (js-yaml)
Dieses Paket ist ebenfalls eine PyYAML-Portierung, welche aktiv weiterentwickelt wird und über umfangreiche stringify-Funktionalitäten verfügt, jedoch fehlt hier die AST-Funktion.

Request/Response Validator (openapi-request-validator) (openapi-response-validator)
Das Paket führt eine Validierung der Anfrage- und Antwortdaten anhand der OpenAPI-Definition durch und liefert eine Liste der Abweichungen.

OpenAPI-Parameters zu JSON Schema (openapi-jsonschema-parameters)
Dieses Paket wandelt die parameters einer Schnittstelle in eine "JSON Schema"-Struktur um, wodurch eine Gruppierung der Typen entsteht, welches die Handhabung der parameters bei der Implementierung vereinfacht (Abbildung 6.5).

	Eingabe	Ausgabe
1	`parameters:`	`parameters:`
2	`  - in: 'path'`	`  path:`
3	`    name: 'boo'`	`    properties:`
4	`    type: 'string'`	`      boo:`
5	`  - in: 'query'`	`        type: 'string'`
6	`    name: 'foo'`	`  query:`
7	`    type: 'string'`	`    properties:`
8	`    required: true`	`      foo:`
9		`        type: 'string'`
10		`    required: ['foo']`

Abbildung 6.5: Umwandlung der parameters zu JSON Schema

6.1.2 Testumgebung

Es folgt eine Beschreibung der verwendeten npm-Pakete für die Testumgebung.

Mocha (mocha)
Mocha ist eine beliebte Testumgebung für das automatisierte Testen von JavaScript-Code.

JS Dom (jsdom)
Dieses Paket emuliert einen Teil vom DOM und wird eingesetzt, um AJAX-Anfragen in der Testumgebung mittels jQuery zu ermöglichen, da die Testumgebung ohne Electron ausgeführt wird.

Express (express)
Dieses Paket startet einen Webserver auf der lokalen Maschine, welcher zum Testen des Vorgangs "Anfrage senden" eingesetzt wird (Laufzeitsicht 5).

Body-Parser (body-parser)
Der Body-Parser wird für den Express-Webserver benötigt. Er wandelt den body einer eingehenden Anfrage um, wodurch der Zugriff auf den body ermöglicht wird.

6.2 UI

Dieser Abschnitt beschreibt zunächst die Handhabung der UI-Elemente. Anschließend wird die Schnittstelle für die Verwaltung der Knöpfe gezeigt und die Theme-Umsetzung erläutert.

6.2.1 Elementerzeugung

HTML-Elemente können mit JavaScript erzeugt und manipuliert werden. Die Implementierung versucht möglichst wenig HTML-Code innerhalb der JS-Dateien zu definieren, um eine Trennung von Darstellung und Logik zu schaffen, wodurch die Lesbarkeit gesteigert wird. Einzeiliger HTML-Code wird in JS-Dateien definiert (Abbildung 6.6) und mehrzeilige Elemente in HTML-Dateien (Abbildung 6.7). Das folgende Beispiel zeigt anhand eines Formularelements, wie eine HTML-Datei in das DOM geladen und mit dynamischen Informationen gefüllt wird (Abbildung 6.6-6.8).

Beispiel:

```
 1 ...
 2 this.jq = uiFacade.getTemplate(__filename, {
 3   id: id,
 4   name: name,
 5   dragDiv: data.isArray ? `<div class="dragDiv" id="drag-${id}"></div>` : '',
 6   required: (required !== '') ? ' *' : '', // show * on required properties
 7   description: description
 8 })
 9 jQCategoryContainer.append(this.jq)
10 ...
```

Abbildung 6.6: Formular: PropertyContainer-Klasse

```
 1 <div class="item-container" id="item-container-${id}" title="${description}">
 2   <div class="flex-column fill-available-width fit-content-height">
 3     <div class="test-status" id="${id}-error-hint"></div>
 4     <div class="margin-left id-column margin-right" id="${id}-col0">
 5       <div style="margin-bottom: 5px; user-select: text;">
 6         ${name}${required}
 7       </div>
 8     </div>
 9     <div class="input-column flex-row" id="${id}-col1"></div>
10   </div>
11   ${dragDiv}
12 </div>
```

Abbildung 6.7: Formular: PropertyContainer-HTML

```
getTemplate (filePath, data) {
  /* Replace .js with .html at the end of 'filePath'
     in code-behind cases where '_filename' is used. */
  if (filePath.endsWith('.js')) {
    filePath = filePath.substring(0, filePath.length - 3) + '.html'
  }

  // Is cached?
  if (!this.templates.hasOwnProperty(filePath)) {
    // Load file
    let file = fs.readFileSync(filePath, 'utf8')
    // Cache file
    this.templates[filePath] = file
  }

  // Inject id's, title, ...
  return $(this.injectDataIntoHTML(this.templates[filePath], data))
}

injectDataIntoHTML (text, data) {
  // 'text' invalid, throw error
  if (typeof text !== 'string') {
    throw new Error(`Parameter 'text' has to be 'string'`)
  }
  // 'data' invalid, throw error
  if (data == null) throw new Error(`Parameter 'data' has to be 'Object'`)

  // Solves an issue where tooltips > 1024 char bleed into other divs
  if (data.hasOwnProperty('tooltip')) {
    data.tooltip = data.tooltip.substring(0, 1024)
  }

  // Replaces each ${*} in text with 'data' value
  return text.replace(/\${\s?(\w*)\s?}/g, (match, found) => {
    return (data.hasOwnProperty(found) && data[found] != null) ? data[found] : ''
  })
}
```

Abbildung 6.8: UI: Template-Klasse

6.2.2 Elementzugriff

In manchen Fällen wird das jQuery-Objekt erst spät zum DOM hinzugefügt, um trotzdem Anpassungen an Elementen des jQuery-Objekts durchführen zu können, wie zum Beispiel das Hinzufügen eines Event-Listeners, wird anstatt des einfachen Elementzugriffs mit `$(`#${id}`)` eine find-Funktion eingesetzt `this.jq.find(`#${id}`)`. Damit bei der Suche nach einer ID keine Verwechslung mit einem CSS-Selektor entsteht, wird jedes Sonderzeichen in der ID mit einem Escape-Zeichen `\\` versehen (vgl. The jQuery Foundation 2019b).

6.2.3 Button-Manager

Knöpfe werden in den verschiedensten Bereichen der grafischen Oberfläche eingesetzt, zum Beispiel im ContextMenu, den Pop-ups oder dem Formular. Der Button-Manager zentralisiert die Verwaltung der Knöpfe und liefert folgende Funktionalitäten (Abbildung 6.9):

```
1  getButton (id, text, title, hasBorder = true, hasHover = true)
2
3  getContextButton (id, text, shortcutText, title)
4  getTextButton (id, text, title, hasBorder, hasHover)
5  getIconButton (id, iconName, title, hasBorder, hasHover)
6
7  setIcon (button, iconName)
8  setIconColor (button, styleClass)
9
10 selectButton (button)
11 deselectButton (button)
12
13 enableButton (button)
14 disableButton (button)
```

Abbildung 6.9: UI: ButtonManager-Klasse

Die Icons für die Knöpfe werden mittels Google-Material-Icons umgesetzt, welche für Webanwendungen optimiert sind (vgl. Google 2019). Sie haben nur eine Farbe und werden als SVG-Datei zur Verfügung gestellt, welches eine einfache Anpassung der Farbe und Größe zur Laufzeit ermöglicht.

6.2.4 Theme

Das System unterstützt einen Dark- und Light-Theme, welcher abhängig von den Betriebssystem-Einstellungen automatisch gewählt wird. Die Abbildung 6.10 zeigt, wie mithilfe von "Media queries" die bevorzugte Hintergrund-Farbe gewählt wird:

Beispiel: style.less

```
1  @backgroundColorDark: #141414;
2  @backgroundColorLight: #ffffff;
3
4  .backgroundColor {
5    @media (prefers-color-scheme: dark) {
6      background-color: @backgroundColorDark;
7    }
8
9    @media (prefers-color-scheme: light) {
10     background-color: @backgroundColorLight;
11   }
12 }
13
14 html,
15 body {
16   .backgroundColor;
17   ...
18 }
```

Abbildung 6.10: Dark- und Light-Theme

Die Einstellung für das bevorzugte Theme kann zum Beispiel unter Windows 10 im folgenden Menü vorgenommen werden:

Designs und verwandte Einstellungen → Farben → Farbe auswählen → Hell/Dunkel

Die Farben der beiden Themes lassen sich in der Datei "theme.json" anpassen:

Windows: C:\Users\<username>\AppData\Roaming\OpenAPI-IDE\theme.json

6.3 Common

In diesem Abschnitt wird die Common-Komponente beschrieben. Sie enthält folgende Klassen: Config, HTTPRequest, ObjectWrap und YamlWrap.

6.3.1 Config

Bei der Erzeugung einer Config-Instanz, wird der Dateipfad für die Konfigurationsdatei erwartet, welcher zum Laden und Speichern der Daten dient. Der Konstruktor registriert einen Event-Listener "beforeunload", welcher beim Beenden der Applikation ausgelöst wird, wodurch das store-Objekt in einen JSON-String gewandelt und im angegebenen Dateipfad gespeichert wird. Die Klasse liefert folgende Funktionalitäten (Abbildung 6.11):

```
1  getValue (key, defaultValue)
2  setValue (key, value)
3  delete (key)
4  getKeys ()
5  has (key)
6  getStore ()
7  clear ()
8  load ()
9  save ()
10 setFilePath (filePath)
```

Abbildung 6.11: Common: Config-Klasse

6.3.2 HTTP-Request

Die HTTPRequest-Klasse liefert Hilfsfunktionen zum Senden von HTTP-Anfragen und hat folgende Funktionalitäten (Abbildung 6.12):

```
1 getHeadersObject (headersString)
2 getEndpointString (endpointString, pathData)
3 getQueryString (queryData)
4 sendRequest (request, runningRequests, finishFn)
```

Abbildung 6.12: Common: HTTPRequest-Klasse

Beispiel:

Funktion	Eingabe	Ausgabe
getHeadersObject	`('test:1\r\nconnection: close\r\n')`	`{test:1, connection:'close'}`
getEndpointString	`('/pet/{petId}', { petId: 1 })`	`'/pet/1'`
getQueryString	`({ stat: 'ok', tags: ['Dog', 'Cat'] })`	`'?stat=pending&tags=Dog,Cat'`

6.3.3 Object-Wrapper

Die ObjectWrap-Klasse vereinfacht das Arbeiten mit Objekten und $ref (Abbildung 6.13). Sie übernimmt zum Beispiel das Finden eines Objekts anhand eines $ref-Pfads und führt im Anschluss die gewählte Operation aus (Beispiel siehe Abbildung 6.14).

```
 1 // object helper
 2 copy (obj)
 3 deepEqual (obj1, obj2)
 4 // operations
 5 clear (obj, path)
 6 createPath (obj, path)
 7 deleteEntry (obj, path)
 8 getValue (obj, path)
 9 setValue (obj, path, newValue)
10 // path helper
11 decodeKey (key)
12 encodeKey (key)
13 encodePath (path)
14 decodePath (path)
15 getAllPaths (obj)
16 getUniqueKey (key, excludeKeys)
17 toArray (path)
```

Abbildung 6.13: Common: ObjectWrap-Klasse

	Eingabe	Funktion
1	`let obj = {`	`getValue (obj, path) {`
2	`  pet: {`	`  if (!obj || (typeof obj !== 'object')) {`
3	`    id: 0`	``    throw Error(`Invalid 'obj'`) }``
4	`  }`	`  if (!path || !Array.isArray(path)) {`
5	`}`	``    throw Error(`Invalid 'path'`) }``
6	`let path = ['pet', 'id']`	`  let value = obj`
7	`let id = objectWrap.getValue(obj, path)`	`  for (let key of path) {`
8	`// id = 0`	`    if (value && (typeof value === 'object') &&`
9		`    value.hasOwnProperty(key)) {`
10		`      value = value[key]`
11		`    } else { return undefined }`
12		`  }`
13		`  return value`
14		`}`

Abbildung 6.14: ObjectWrap: getValue-Funktion

6.3.4 YAML-Wrapper

Die YamlWrap-Klasse liefert YAML bezogene Funktionalitäten (Abbildung 6.15). Die Klasse wird in den verschiedensten Bereichen eingesetzt, zum Beispiel wird die Funktion "removeUniqueRootIdFromRef" und "addUniqueRootId" nur im Bereich der TestAPI benötigt (siehe 6.5), wohingegen "stringify" in vielen Bereichen des Systems benutzt wird.

```
1  addUniqueRootId (yamlText)
2  applyIndent (yamlText, indentSize)
3  ast (yamlText, useCacheOnError)
4  convertToPositionMode (position, mode)
5  getPathToPosition (yamlText, position, isIndex, useCacheOnError)
6  getPositionToPath (yamlText, path, useCacheOnError)
7  getRefValueInPosition (yamlText, position)
8  isLineARef (yamlText, position)
9  isLineItemWithObject (yamlText, position)
10 isPositionWithLine (obj)
11 isPositionWithRow (obj)
12 parse (yamlText)
13 removeUniqueRootIdFromRef (path)
14 stringify (obj, replacer, options)
15 yamlErrorToSyntaxError (err)
```

Abbildung 6.15: Common: YamlWrap-Klasse

Die Funktionen "getPathToPosition" und "getPositionToPath" werden für die Umsetzung von "Copy $ref" und "Jump to Definition" benötigt. Der Swagger Editor dient als Vorlage für die Implementierung dieser Funktionen (vgl. SmartBear Software 2019q).

Das folgende Beispiel zeigt die Umsetzung der getPositionToPath-Funktion. Sie generiert zunächst mit yamlJS.compose einen AST, welcher für jede "Node" einen "Tag" enthält, der den value-Typ angibt (map, seq, str, int). Außerdem wird eine Markierung angegeben (Start/End), welche die Position für den Schlüssel oder Wert im YAML-Text beschreibt (Abbildung 6.16).

Beispiel: Vereinfachter AST für YAML-Text: myKey: 1

```
1  Tag: tag:yaml.org,2002:map
2  value:
3    -
4      - tag: tag:yaml.org,2002:str
5        value: myKey
6        start_mark:
7          line: 0
8          column: 0
9        end_mark:
10         line: 0
11         column: 5
12     - tag: tag:yaml.org,2002:int
13       value: '1'
14       start_mark:
15         line: 0
16         column: 7
17       end_mark:
18         line: 0
19         column: 8
20 start_mark:
21   line: 0
22   column: 0
23 end_mark:
24   line: 0
25   column: 8
```

Abbildung 6.16: Beispiel: AST für YAML-Text

In Abbildung 6.17 wird der übergebene $ref-Pfad rekursiv abgearbeitet, dabei wird der aktuelle Schlüssel eingesetzt um die nächste "Node" im AST zu finden. Das Ergebnis enthält die Start-Markierung der zuletzt gefundenen "Node".

```
getPositionToPath (yamlText, path, useCacheOnError) {
  if (typeof yamlText !== 'string') { throw Error('Invalid text') }

  let find = (currentAst, subPath, last) => {
    let pathKey = objectWrap.decodeKey(subPath.shift())

    if (!pathKey && currentAst) { // Path end, found result
      return currentAst.start_mark
    } else if (currentAst.tag === TAG_MAP) { // MAP
      for (let [key, value] of currentAst.value) { // Search 'pathKey'
        // If node key is the same as current 'pathKey'
        if (key.value === pathKey) {
          if (subPath[0]) { // Peek, if not path end, keep going
            return find(value, subPath, key)
          } else { // Path end, return key start_mark
            return key.start_mark
          }
        }
      }
      // Current 'pathKey' not found return best effort
      return last ? last.start_mark : currentAst.start_mark
    } else if (currentAst.tag === TAG_SEQ) { // SEQUENCE
      // if has index 'pathKey', keep going
      if (pathKey && currentAst.value.hasOwnProperty(pathKey)) {
        return find(currentAst.value[pathKey], subPath, currentAst.value[pathKey])
      } else { // Current 'pathKey' not found return best effort
        return currentAst.start_mark
      }
    } else { // Current 'pathKey' not found return best effort
      return currentAst.start_mark
    }
  }

  let ast = this.ast(yamlText, useCacheOnError)
  let pathCopy = objectWrap.toArray(path)
  let rs = find(ast, pathCopy)
  return this.convertToPositionMode(rs, this.positionMode)
}
```

Abbildung 6.17: YamlWrap: getPositionToPath-Funktion

6.4 OpenAPI

In diesem Abschnitt wird die OpenAPI-Komponente vorgestellt. Sie liefert eine Schnittstelle, um das Arbeiten mit der OpenAPI zu vereinfachen. Dazu gehören die OpenAPI-Validierung, das Refactoring und der Zugriff auf OpenAPI-Definitionsbereiche mittels Resolve.

6.4.1 Validierung

Die Validate-Klasse liefert Funktionalitäten zum Prüfen folgender Objekte: Anfrage, Antwort, und OpenAPI-Definition (Abbildung 6.18).

```
1  convertErrorPathToFormPath (errors, tabId)
2  getExampleOrEmpty (openApiExample, type)
3  getRequestErrors (request, openAPIObj, refs)
4  getResponseErrors (response, openAPIObj, refs, endpoint, type)
5  orderAndValidateCustomArray (schema, customs, isKeepingCustoms, isValidatingValues,
6  refs, pathToSchema, valueChangedFn)
7  orderAndValidateCustomObject (schema, customs, isKeepingCustoms, isValidatingValues,
8  refs, pathToSchema, valueChangedFn)
9  validateCustomValue (value, schema)
10 validateOpenAPI (openAPIText, finishFn)
```

Abbildung 6.18: OpenAPI: Validate-Klasse

Eine Validierung von Daten geschieht im System in verschiedenen Szenarien:

- Bei **Änderungen der OpenAPI** wird die Funktion "validateOpenAPI" ausgeführt, sie nutzt den swagger-parser, welcher die Validierung der OpenAPI-Definition übernimmt (Laufzeitsicht 1).
- Bei **Änderungen der Formulardaten** wird "getRequestErrors" ausgeführt, welche für die Markierung der ungültigen Daten im Formular eingesetzt wird (Laufzeitsicht 4 Schritt 6).
- Beim **Empfangen einer Antwort** wird "getResponseErrors" ausgeführt, diese liefert eine Liste der Abweichungen zur OpenAPI-Definition (Laufzeitsicht 5 Schritt 5).

Mit der Funktion "orderAndValidateCustomObject" werden Abweichungen zur OpenAPI-Definition erkennt und bei Bedarf behoben (siehe Beispiel nächste Seite).

Das Beispiel in Abbildung 6.19 zeigt ein "Eingabe"-Objekt, welches vom Entwickler über den Debug-Editor mit einer Custom-Variable (category) erweitert wurde. Die Attributreihenfolge wird immer anhand des Schemas sortiert, wodurch die Suche für den Entwickler vereinfacht wird. Die Custom-Variablen werden am Ende des "Ausgabe"-Objekts angehängt.

	Schema	Eingabe	Ausgabe		
			[ ] Customs [ ] Validate	[x] Customs [ ] Validate	[x] Customs [x] Validate
1	`type: object`	`id: '41'`	`id: '41'`	`id: '41'`	`id: 41`
2	`properties:`	`category: {}`	`type: []`	`type: []`	`type: {}`
3	`  id:`	`type: []`		`category: {}`	`category: {}`
4	`    type: integer`				
5	`  type:`				
6	`    type: object`				
7	`    properties:`				
8	`      id:`				
9	`        type: integer`				

Abbildung 6.19: Validate: orderAndValidateCustomObject-Beispiel

6.4.2 Refactoring

Die Refactor-Klasse bietet folgende Funktionalitäten (Abbildung 6.20).

```
1 getComponentCategory (openAPIObj, path)
2 isRefactor (openAPIObj, path)
3 refactorToComponent (openAPIObj, path, id)
```

Abbildung 6.20: OpenAPI: Refactor-Klasse

Die Funktion "refactorToComponent" verschiebt ein Objekt in die components-Eigenschaft und setzt an der leeren Stelle einen $ref-Pfad zum Objekt. Die Liste in Abbildung 6.21 zeigt, welche Objekte verschoben werden können, da die verschiebbaren Objekte durch die OpenAPI-Spezifikation eingeschränkt werden. Mit der Funktion "getComponentCategory" kann der Zielbereich für das ausgewählte Objekt ermittelt werden. Die Funktion "isRefactor" liefert eine undefinierte Kategorie, wenn die Verschiebung nicht möglich.

	Quellpfad	Zielpfad
1	`paths/.../parameters/'index'`	`components/parameters/'name'`
2	`paths/.../responses/'status'`	`components/responses/'name'`
3	`paths/.../requestBody`	`components/requestBodies/'name'`
4	`paths/.../schema`	`components/schemas/'name'`
5	`components/.../schema`	

Abbildung 6.21: Refaktorisierungs-Kategorien

6.4.3 Resolve

Die Resolve-Klasse vereinfacht den Zugriff auf Bereiche der OpenAPI-Definition, sie löst die $ref-Pfade auf und liefert die geforderten Objekte. Die Schnittstelle hat folgenden Aufbau (Abbildung 6.22):

```
 1 convertArraySchemaToObject (requestBodyObject, schemaWithArrayInRoot)
 2 getBodySchema (operation, refs, contentType)
 3 getExamples (pathToProperty, openAPIObj, refs, endpoint, type, contentType)
 4 getExamplesPath (pathToProperty, openAPIObj, refs, endpoint, type, contentType)
 5 getOperation (openAPIObj, endpoint, type)
 6 getParameter (pathToProperty, openAPIObj, refs, endpoint, type)
 7 getParametersSchema (operation, refs)
 8 getPaths (openAPIObj)
 9 getSchema (pathToProperty, openAPIObj, refs, endpoint, type, contentType)
10 getSchemaNameWithPath (pathToProperty, openAPIObj, refs, endpoint, type, contentType)
11 getSchemaNameWithRef (requestBodyRef, requestBody)
12 isExamplesInComponents (pathToProperty, openAPIObj, refs, endpoint, type)
13 resolveRef (obj, refs)
```

Abbildung 6.22: OpenAPI: Resolve-Klasse

- Für das Bearbeiten und Abrufen der examples werden die Funktionen "getExamples", "getExamplesPath" und "isExamplesInComponents" eingesetzt.
- Die Funktion "convertArraySchemaToObject" wandelt ein Array in ein Objekt um, wenn das übergebene Schema vom Typ "array" ist (siehe Laufzeitsicht 3 Schritt 4).
- Der Name eines Schemas kann mit der Funktion "getSchemaNameWithPath" oder "getSchemaNameWithRef" abgerufen werden, er wird über xml.name oder dem letzten Eintrag im $ref ermittelt. Der Name wird zum Beispiel im Root-Tab des Formulars angezeigt oder zum Ermitteln der Parameter-ID eingesetzt, welches das Laden und Speichern eines examples vereinfacht.
- Der Zugriff auf die parameters wird durch die Funktionen "getParameter" und "getParametersSchema" unterstützt. Das beschriebene Problem in Abbildung 3.7 wird mit der Funktion "getParametersSchema" gelöst, sie führt die Umwandlung der abgerufenen Parameters durch (Abbildung 6.5) und ermöglicht eine einheitliche Handhabung von parameters und requestBody.
- Die Funktion "getBodySchema" ruft das Schema vom requestBody ab, welches sich in der übergebenen `operation` und dem `contentType` befindet. Die Funktion liefert folgende Daten: schemaName, schema, contenTypes, isRootArray und die examples.
- Die getSchema-Funktion ruft das gesuchte Schema anhand der übergebenen Variable `pathToProperty` ab, welche auf ein Schema-Property, wie zum Beispiel 'body-tags-id' zeigen kann. Dies wird durch eine Kombination von "getBodySchema" und "getParametersSchema" erreicht.

- Die getParameter-Funktion stellt zunächst in Abbildung 6.23 Zeile 2-11 sicher, dass die Variable pathToProperty das erwartete Format und eine der möglichen Kategorien enthält (Beispiel: 'query-petId'). In Zeile 13-14 werden alle Parameter abgerufen, diese befinden sich im Pfad "paths/endpoint/type/parameters" und werden zum Suchen des Zielparameters eingesetzt (Zeile 16). Ein Parameter kann mit $ref verlinkt werden und muss zunächst aufgelöst werden (Zeile 18). Dies geschieht mit dem refs-Objekt, welches vom swagger-parser bei der Validierung der OpenAPI-Definition geliefert wird.
  In Zeile 19 wird geprüft, ob es sich um den gesuchten Parameter handelt. Der Rückgabewert der getParameter-Funktion enthält folgende Werte: der gesuchte Parameter, die Parameter-Position im parameters-Array und ein Boolean der auf true gesetzt wird, wenn es sich um einen referenzierten Parameter handelt.

```
 1 getParameter (pathToProperty, openAPIObj, refs, endpoint, type) {
 2   let pathToPropertyArray = pathToProperty.split('-')
 3   if (pathToPropertyArray.length !== 2) {
 4     throw Error(`Path to Property has wrong length: ${pathToProperty}`)
 5   }
 6 
 7   let paramCategory = pathToPropertyArray[0]
 8   let possibleCategories = openAPIFacade.categories.slice(0, -1) // Remove body
 9   if (!possibleCategories.includes(paramCategory)) {
10     throw Error(`Path to Property has invalid category: ${paramCategory}`)
11   }
12 
13   let operation = openAPIFacade.getOperation(openAPIObj, endpoint, type)
14   let parameters = operation.parameters
15   let paramName = pathToPropertyArray[1]
16   for (let index in parameters) {
17     let parameterRef = parameters[index]
18     let parameter = this.resolveRef(parameterRef, refs)
19     if (paramCategory === parameter.in && paramName === parameter.name) {
20       return {
21         parameter: parameter,
22         index: Number(index),
23         isInComponents: parameter !== parameterRef
24       }
25     }
26   }
27   throw Error(`Parameter not found: ${pathToProperty}`)
28 }
```

Abbildung 6.23: Resolve: getParameter-Funktion

6.5 TestAPI

Dieser Abschnitt beschreibt die TestAPI-Validierung, die Verkettung von Daten, die Abarbeitung der Tests und die Testausgabe.

6.5.1 Validierung

Die TestAPI-Validierung prüft in Abbildung 6.24 Zeile 5, ob die version angegeben wurde, das tests-Objekt vorhanden ist, jeder Test die erwartete Struktur enthält (Strukturen 2) und alle Test-IDs mit einem Buchstaben beginnen (Details in Abbildung 6.26). Eine Test-ID, die nur aus Zahlen besteht, würde eine Sortierung beim Einfügen in das tests-Objekt auslösen und dadurch zu einer abweichenden Testreihenfolge führen.
In Zeile 7-14 wird sichergestellt, dass jede Test-ID eindeutig ist. Dies ermöglicht eine Datenverkettung anhand der Test-ID und vereinfacht das Nachvollziehen der Testausgabe.

TestAPIFacade.js

```
 1 validateTestAPI (testAPIText) {
 2   let testAPIObj = yamlWrap.parse(testAPIText)
 3 
 4   // Check for expected structure
 5   this.getTestIdPaths(testAPIObj)
 6 
 7   // Check if contains duplicate test id
 8   let testIds = this.getTestIds(testAPIObj.tests)
 9   while (testIds.length > 0) {
10     let id = testIds.pop()
11     if (testIds.includes(id)) {
12       throw Error(`Test id already in use: ${id}`)
13     }
14   }
15 
16   return testAPIObj // Valid TestAPI
17 }
```

Abbildung 6.24: TestAPI: validateTestAPI-Funktion

Die getTestIds-Funktion erwartet ein Test-Root-Objekt (test.chaining oder tests) über welches rekursiv iteriert wird. Die Funktion liefert eine Liste aller Test-IDs (Abbildung 6.25).

TestAPIFacade.js

```
 1 getTestIds (testObj) {
 2   let rs = []
 3   if (testsObj) {
 4     for (let testId in testsObj) {
 5       let property = testsObj[testId]
 6       if (property && typeof property === 'object' &&
 7       property.hasOwnProperty('chaining')) {
 8         rs.push(testId)
 9         rs = rs.concat(this.getTestIds(property.chaining))
10       }
11     }
12   }
13   return rs
14 }
```

Abbildung 6.25: TestAPI: getTestIds-Funktion

6.5.2 Datenverkettung

Für die Datenverkettung werden $ref-Pfade eingesetzt. Im TestAPI-Bereich wird vom "JSON Reference"-Standard abgewichen, weil die Verschachtelung von Tests bereits bei einer geringen Tiefe zu unübersichtlichen $ref-Pfaden führt. Die "Custom Version" funktioniert nur mit eindeutigen Test-IDs. Es werden außerdem nur lokale Referenzen unterstützt, deshalb wird im $ref-Pfad auf # verzichtet.

Beispiel:

JSON Reference	`$ref: '#/tests/Test1/chaining/Test2/response/body/id'`
Custom Version	`$ref: 'Test2/response/body/id'`

Die getTestIdPaths-Funktion (Abbildung 6.26) liefert ein Objekt, wodurch die kürzere Schreibweise ermöglicht wird (siehe Beispiel). Zusätzlich wird bei der Iteration sichergestellt, dass alle Tests die erwarteten Bereiche enthalten (async, request/…, expected und chaining).

Beispiel:

Id	JSON Reference
Test2	'#/tests/Test1/chaining/Test2'

TestAPIFacade.js

```
getTestIdPaths (testAPIObj) {
  let rs = { [this.ROOT_ID]: ['tests'] } // Add root 'id:path' to result

  // Invalid TestAPI version
  if (!testAPIObj) {
    throw Error(`Invalid testAPIObj: ${testAPIObj}`)
  }
  if (!testAPIObj.hasOwnProperty('testapi') || testAPIObj.testapi !== this.VERSION) {
    throw Error(`Line 1 missing testapi: ${this.VERSION}`)
  }

  // Check tests key
  this.checkProperty(testAPIObj, 'tests', 'root', 'object')

  let onEachTestFn = (testId, path) => {
    rs[testId] = path // Add current path to result

    // Check for invalid Id
    if (!this.isValidTestId(testId)) {
      throw Error(`Invalid test id: ${testId}`)
    }
  }
  // Check for 'async', 'request', 'expected', 'chaining' keys
  this.validateTests(['tests'], testAPIObj.tests, onEachTestFn)

  return rs
}
```

Abbildung 6.26: TestAPI: getTestIdPaths-Funktion

6.5.3 Testausführung

Die TestAPI kann einen Test synchron oder asynchron ausführen. Bei der asynchronen Testausführung kann es zu verschiedenen Ergebnissen kommen, da nicht auf die Antwort des Servers gewartet wird, sondern der nächste Test auf derselben Ebene ausgeführt wird. Im synchronen Fall wird der nächste Test erst nach Erhalt der Antwort ausgeführt, wenn Kinder vorhanden sind, werden diese zunächst abgearbeitet und danach folgen die Tests der aktuellen Ebene.

Beispiel (Abbildung 6.27):
Der asynchrone "Logout" wird vom Server als ein langandauernder Vorgang simuliert, wodurch die Antwort von "GetUsers" garantiert vor dem "Logout" geliefert wird. Im realen Fall kann es passieren, dass die Antwort direkt geliefert wird, wodurch die Ausführung von "GetUsers" und "DeleteFail" vertauscht wären. Die "Logout"-Antwort löst die Ausführung von "DeleteFail" aus. Hier wird erwartet, dass der Löschvorgang fehlschlägt, da der Benutzer bereits ausgeloggt ist. Die Abfolge der Testausführung in Abbildung 6.27 sieht dadurch wie folgt aus: ReCreateUser, Login, DeleteUser, AddUser, Logout, GetUsers, DeleteFail.

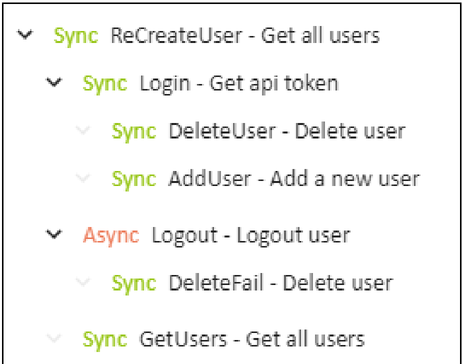


Abbildung 6.27: TestAPI: Abfolge der Testausführung

6.5.4 Testausgabe

Die Testergebnisse sind detailliert und ermöglichen ein Nachvollziehen der Testabfolge. Wenn ein Test fehlschlägt, wird die Ausführung abgebrochen, um die Fehlersuche zu minimieren.
Die Ausgabe unterstützt die "Copy $ref"-Funktionalität, welche in der Test-Definition für die Verkettung von Daten eingesetzt werden kann. Da die Testausgabe für jeden Test zwei Einträge im Editor erzeugt (Anfrage/Antwort), entsteht durch die doppelten Schlüssel auf der Root-Ebene ein ungültiger YAML-Text, welches die Nutzung von "Copy $ref" zunächst verhindert.

Die folgende Umsetzung löst das "Copy $ref" Problem durch eine interne Nummerierung aller Root-Schlüssel (Abbildung 6.28-6.30).

OutputFacade.js: "Copy $ref" für die Testausgabe

```
1  ...
2  let yamlText = yamlWrap.addUniqueRootId(text)
3  let refPath = yamlWrap.getPathToPosition(yamlText, position, false)
4  let ref = `$ref: '${yamlWrap.removeUniqueRootIdFromRef(refPath)}'`
5  uiFacade.setClipboard(ref)
```

Abbildung 6.28: Output: Copy $ref

YamlWrap.js: Nummerierung im YAML-Text hinzufügen

```
 1 addUniqueRootId (yamlText) {
 2   let rs = ''
 3   let counter = 0
 4   for (let line of yamlText.split('\n')) {
 5     // First char in line has no tab/space and the line is not empty
 6     if (line[0] !== ' ' && line[0] !== '\t' && line !== '') {
 7       let withCounter = counter + '_' + line
 8       counter += 1
 9       rs += withCounter + '\n'
10     } else {
11       rs += line + '\n'
12     }
13   }
14   return rs.substring(0, rs.length - 1)
15 }
```

Abbildung 6.29: YamlWrap: addUniqueRootId-Funktion

YamlWrap.js: Nummerierung im $ref-Pfad entfernen

```
1 removeUniqueRootIdFromRef (path) {
2   path = objectWrap.toArray(path)
3   let subArray = path[0].split('_')
4   if (!isNaN(subArray[0])) { // has number?
5     subArray.shift() // remove number
6   }
7   path[0] = subArray.join('_') // need to join back, it could containt more _
8   return path.join('/')
9 }
```

Abbildung 6.30: YamlWrap: removeUniqueRootIdFromRef-Funktion

6.6 Unit-Tests

In diesem Abschnitt wird die im Rahmen dieser Arbeit entstandene Data-Driven Lösung vorgestellt. Anhand der Common-Komponente wird der Unterschied zwischen der einfachen Definierung von Tests (Abbildung 6.31) und dem Data-Driven Ansatz (Abbildung 6.32) beschrieben.

6.6.1 Mocha-Tests

Die Testdateien werden überwiegend im Hauptverzeichnis des Projekts unter "test" abgelegt. Der Name und Speicherort einer Testdatei ist frei wählbar, auch die enthaltenen Testfälle können zum Beispiel für alle Klassen innerhalb einer Datei definiert werden, wodurch das Finden eines Testfalls erschwert wird. Der Entwickler kann auch entscheiden, ob describe für die Trennung von Funktionen und Klassen eingesetzt wird (Abbildung 6.31).

```
 1 describe('YamlWrap', () => {
 2   describe('convertPositionRowToLine', () => {
 3     it('1-returns-position-copy', () => {
 4       let position = yamlWrap.convertPositionRowToLine({ line: 1, column: 2 })
 5       assert.deepStrictEqual(position, { line: 1, column: 2 })
 6     })
 7
 8     it('2-returns-line-instead-of-row-position', () => {
 9       let position = yamlWrap.convertPositionRowToLine({ row: 1, column: 0 })
10       assert.deepStrictEqual(position, { line: 1, column: 0 })
11     })
12
13     it('3-throws-error', () => {
14       try {
15         yamlWrap.convertPositionRowToLine({ row: 0 })
16       } catch (e) {
17         assert.deepStrictEqual(e.message, `Convert Position Failed: \n{"row":0}`)
18       }
19     })
20   })
21 })
```

Abbildung 6.31: Mocha-Tests

6.6.2 Data-Driven Unit-Tests

Ein Data-Driven Ansatz lädt die Eingabe- und Ausgabedaten eines Testfalls aus einer externen Datei (Datenbank, JavaScript, ...) und setzt diese in die Testlogik ein (Mocha-Test), wodurch eine Trennung entsteht, die eine Steigerung der Wartbarkeit zur Folge hat.
Die Umsetzung erkennt nur Tests, die in der vorgegebenen Ordnerstruktur definiert sind "Komponente/test/data-driven/Klasse/Funktion/Testfall.js" (siehe Abbildung 6.32).
Die Datei "data-driven.js" initialisiert die Testumgebung und ermöglicht das Laden von zusätzlichen Komponenten und Testdaten. Die Tests können mit `npm run test` ausgeführt werden. Einen Ausschnitt der Testergebnisse zeigt Abbildung 6.33.

```
common
  test
    data-driven
      HTTPRequest
      ObjectWrap
      YamlWrap
        applyIndent
        convertPositionRowToLine
          1-returns-position-copy.js
          2-returns-line-instead-of-row-position.js
          3-throws-error.js
        getPathToPosition
        getPositionToPath
    testdata
    data-driven.js
  CommonFacade.js
  Config.js
  HTTPRequest.js
  ObjectWrap.js
  YamlWrap.js
```

1-returns-position-copy.js

```
1 module.exports = {
2   params: [{ line: 1, column: 2 }],
3   expected: { line: 1, column: 2 }
4 }
```

2-returns-line-instead-of-row-position.js

```
1 module.exports = {
2   params: [{ row: 1, column: 2 }],
3   expected: { line: 1, column: 2 }
4 }
```

3-throws-error.js

```
1 module.exports = {
2   params: [{ row: 0 }],
3   expected: `Convert Position Failed:
4  {"row":0}`
5 }
```

Abbildung 6.32: Data-Driven Tests: Aufbau und Erzeugung

```
YamlWrap
  convertPositionRowToLine
    √ 1 returns position copy
    √ 2 returns line instead of row position
    √ 3 throws error
```

Abbildung 6.33: Data-Driven Tests: Ausschnitt der Testergebnisse

6.7 Fazit

Die Implementierung bietet die Core- und GUI-Schicht, welche das Hinzufügen von weiteren OpenAPI bezogenen Funktionalitäten vereinfachen. Der Vorgang kann anhand dieses Kapitels nachvollzogen werden.

7 Fazit

In diesem Kapitel wird die Bachelorarbeit zunächst zusammengefasst, dann folgt eine Evaluierung des umgesetzten Systems und zum Schluss werden mögliche Erweiterungen beschrieben.

7.1 Zusammenfassung

In dieser Arbeit wurde ein System entwickelt "OpenAPI-IDE", welches das Testen von HTTP-Schnittstellen ermöglicht (2.1.1) und das Arbeiten mit der OpenAPI (2.1.5) unterstützt.

Im Analyse Kapitel (3) wurden bereits vorhandene Systeme zum Testen von HTTP-Schnittstellen verglichen. Anschließend wurde anhand der gesammelten Informationen ein System Spezifiziert (4), welches möglichst viele Vorteile kombiniert.
Durch die Analyse hat sich herausgestellt, dass die Suche nach einer Definitionsstelle in der OpenAPI-Definition viel Zeit in Anspruch nehmen kann (3.1.3). Deshalb sind Funktionalitäten für einen Sprung zur Definitionsstelle entstanden (Kontextmenüs 1). Dieser Vorgang kann auch an bestimmten Stellen eingesetzt werden, um UI-Element anhand einer Definitionsstelle zu fokussieren (Kontextmenüs 3).
Das System liefert einen Editor für die Bearbeitung der OpenAPI (4.2.2) und eine Refaktorisierungs-Funktionalität, wodurch die Verschiebung eines Objekts vereinfacht wird (Pop-ups 2 und 6.4.2).
Damit die Formulardaten nicht bei jeder Anfrage manuell eingegeben werden müssen, können diese in der OpenAPI-Definition gespeichert und zu einem späteren Zeitpunkt wiederhergestellt werden (Swagger UI 4 und Pop-ups 3, 4).
Die Darstellung von verschachtelten Schemas wurde im Formular mittels Tabs umgesetzt, um eine möglichst überschaubare Struktur zu erhalten (Swagger UI 3 und 4.2.4).
Es wurde eine TestAPI entwickelt, diese ermöglicht ein automatisiertes Ausführen der Schnittstellen, die Validierung der Antworten und eine Verkettung der Anfrage- und Antwortdaten (Strukturen 2 und 6.5).

7.2 Evaluierung

In diesem Abschnitt wird auf Vor- und Nachteile des umgesetzten Systems "OpenAPI-IDE" und dem experimentellen Data-Driven Ansatz eingegangen.

7.2.1 OpenAPI-IDE

1. Vergleich des Swagger Editors mit der OpenAPI-IDE

- In Situationen, bei denen kein Wert angegeben wurde, zum Beispiel `example: ` zeigt der Swagger Editor keine Reaktion, wohingegen die OpenAPI-IDE bei der Validierung einen "null" Wert einsetzt, welches bei der automatischen Auslösung der Code-Validierung nachteilig ist, da ein unerwarteter Wert eingefügt wird. Beide Lösungen sind nicht optimal und sollten einen Hinweis liefern.
- Der Swagger Editor zeigt eine Liste möglicher Schlüssel für die aktuelle Cursor-Position, welche mit dem Tastenkürzel "Strg+Leerzeichen" aktiviert werden kann. Diese Liste fehlt bei der OpenAPI-IDE.

2. Vergleich der gängigen Formular-Generatoren mit der OpenAPI-IDE

Bei der Tab-basierten Umsetzung der OpenAPI-IDE wird die aktuelle Sicht beim Hinzufügen oder Entfernen von Elementen nur minimal verändert, indem Objekte als Behälter und eigenständiges Formulare angeboten werden, wohingegen die gängigen Formular-Generatoren das komplette Objekt innerhalb eines Formulars darstellen (vgl. JSON Schema 2019b). Beim Mouse-Hover eines Formularattributs wird die Cursor-Position auf den zugehörigen Schlüssel im Debug-Editor gesetzt, wodurch die Position des Attributwerts innerhalb des Objekts verdeutlicht wird, welches besonders bei verschachtelten Objekten vorteilhaft ist (Abbildung 7.1, Zeile 12).

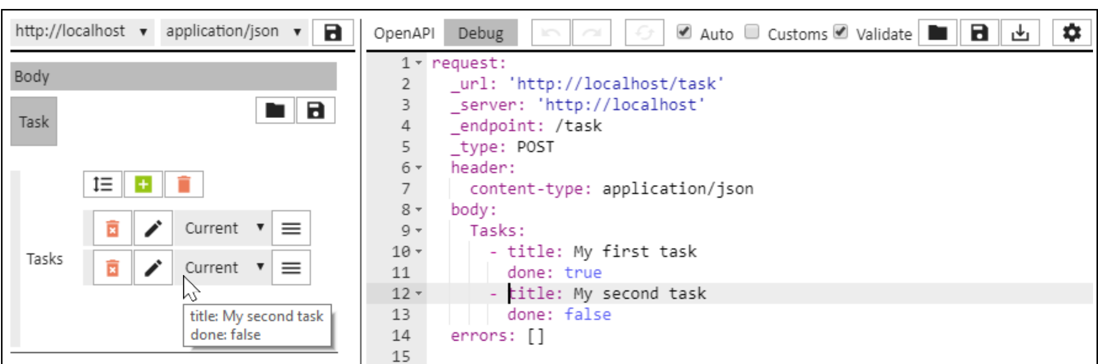


Abbildung 7.1: OpenAPI-IDE: Mouse-Hover Formularattribut

3. TestAPI

Durch Änderungen an der OpenAPI-Definition entstehen Abweichung in der TestAPI-Definition, die zu folgenden Problemen führen:

- Das Erkennen dieser Abweichungen wird vom System nicht direkt unterstützt, sondern erst bei einer Ausführung der betroffenen Testfälle angezeigt.
- Der "Edit Mode" (4.2.3) ermöglicht eine Testaktualisierung über das Formular, welches nur funktioniert, wenn keine Änderung in der OpenAPI-Definition an path und type durchgeführt wurde.
- Das Formular hat kein Element für die Darstellung von "JSON Reference"-Objekten, wodurch diese bei einer Testaktualisierung zurückgesetzt werden.
- Das Anpassen abweichender Testfälle ist bei einer größeren Anzahl nicht mehr verwaltbar und sollte durch einen automatisierten Vorgang unterstützt werden.

7.2.2 Data-Driven Unit-Tests

Im folgenden Unterkapitel werden einige Vor- und Nachteile der umgesetzten Data-Driven Lösung (6.6.2) aufgelistet.

Vorteile

- Tests funktionieren nur, wenn die korrekte Ordnerstruktur eingehalten wird, wodurch eine sorgfältige Erzeugung der Tests erzwungen wird.
- Durch die vorgegebene Ordnerstruktur sind Tests leicht zu finden und können als Nachschlagewerk eingesetzt werden.
- Bei der Testerzeugung gibts es keine Ablenkung durch die Testumgebung (Mocha).

Nachteile

- Rückruffunktionen (Callbacks) können nicht getestet werden.
- Die Ein- und Ausgabedaten müssen mit einem vorgegebenen JavaScript-Objekt definiert werden.
- Die Umbenennung einer Klasse oder Funktion mithilfe eines Werkzeugs (Textersetzung) benötigt einen manuellen Schritt für die Umbenennung des zugehörigen Ordners.

7.2.3 Fazit

Das umgesetzte System (Abbildung 7.2) setzt die Zielsetzung aus Abschnitt 1.2 um und unterstützt alle Anforderungen der Analyse 3.3.

Das System kann als Alternative für den Swagger Editor eingesetzt werden und liefert zusätzliche Funktionalitäten, wie zum Beispiel das Verketten von Daten oder die automatisierte Ausführung und Validierung von Schnittstellen.

Wenn die im Rahmen dieser Arbeit entstandene TestAPI nicht verwendet werden soll, kann das System für die Entwicklung einer OpenAPI-Definition eingesetzt werden, welches durch folgende Funktionalitäten erleichtert wird: Laden und Speichern von Beispieldaten, Refaktorisierung, diverse Sprungbefehle ("Jump to Definition", "Open in tab", ...) und das Senden einzelner Anfragen anhand eines einheitlichen Formulars, welches bereits bei der Eingabe auf Abweichungen zur OpenAPI-Definition hinweist.

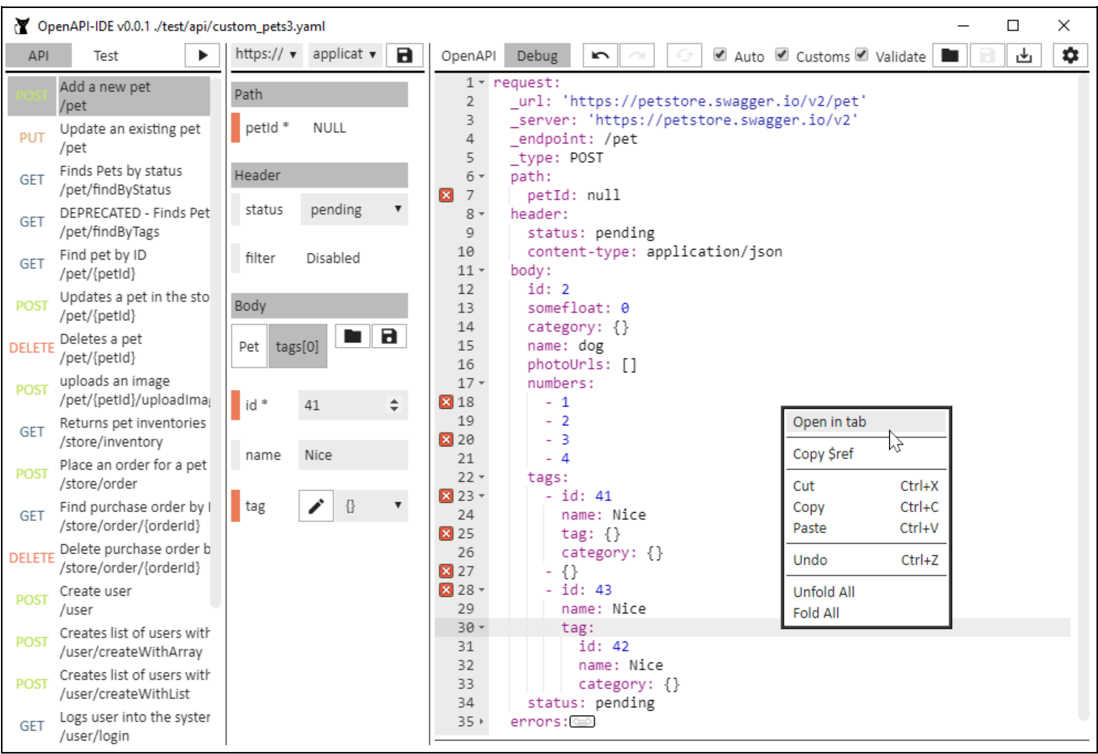


Abbildung 7.2: Vorschau der OpenAPI-IDE

7.3 Ausblick

Die OpenAPI-IDE hat noch keine Unterstützung für die Client/Server-Generierung. Dies könnte mit dem OpenAPI-Generator ermöglicht werden (vgl. OpenAPI-Generator u. a. 2020).
Die grafische Benutzeroberfläche sollte weitere Pop-ups anbieten, wodurch zum Beispiel das Erstellen und Bearbeiten von Schnittstellen, Schema oder dem info-Objekt vereinfacht wird (vgl. SmartBear Software 2019r).
Da die Tastatur oft bevorzugt wird, könnte eine vollständige Unterstützung zum Navigieren der OpenAPI-IDE und ein Menü für die Anpassung der Tastenkürzel sinnvoll sein.
Es sollten weitere Sprungbefehle angeboten werden, damit die manuelle Suche verringert wird, zum Beispiel ein "Jump to Property" im Formular für die einzelnen Attribute (Kontextmenüs 4) und ein "Jump to Servers" bei der Serverauswahl (4.2.3). Dem Formular fehlen außerdem weitere Attributtypen (4.2.4), beispielsweise für die Bearbeitung von $ref, binary, date oder color.
Die Funktionalitäten aus der Abgrenzung 1.2 sollten zur Verfügung stehen, da zum Beispiel die security-Eigenschaft zum Testen von gesicherten Schnittstellen benötigt wird.
Beim Einfügen eines JSON-Objekts in beispielsweise den Debug-Bereich, könnte es eine Option zum Umwandeln von JSON zu YAML geben.
Die automatisierte Testvalidierung bietet für die einzelnen Objekt-Kategorien nur einen exakten Vergleich und könnte zusätzlich eine partielle Überprüfung ermöglichen.
Das System könnte im "Continuous Integration"-Vorgang eingesetzt werden, welches bereits größtenteils implementiert ist aber noch nicht getestet wurde.
Um eine Übersicht der OpenAPI-Definition zu liefern, könnte beispielsweise mit Graphviz, ein UML-Diagramm für die definierten Schnittstellen und Schemas erzeugt werden.
Das System könnte als Vorlage für die Entwicklung eines neuen Systems dienen, das nicht mit der OpenAPI, sondern direkt mit dem JSON Schema (2.1.4) arbeitet. Das generierte Formular könnte Personen bei der Befüllung einer Datenbank unterstützen.

Literaturverzeichnis

[ace-builds] Ajax.org B.V.; Contributors: *Packaged version of Ace code editor* - Version: 14.01.2020 - URL: https://github.com/ajaxorg/ace-builds/releases/tag/v1.4.8 - Abrufdatum: 22.01.2020

[Ajax.org B.V. 2020] Ajax.org B.V.: *About Ace* - Version: 2020 - URL: https://ace.c9.io/#nav=about - Abrufdatum: 15.02.2020

[Asthana 2014] Asthana, Abhinav: *Extracting data from responses and chaining requests* - Version: 27.01.2014 - URL: https://blog.getpostman.com/2014/01/27/extracting-data-from-responses-and-chaining-requests/ - Abrufdatum: 08.05.2019

[Ben-Kiki u. a. 2009] Ben-Kiki, Oren; Evans, Clark; Net, Ingy döt: *YAML Ain't Markup Language (YAML™) Version1.2* - Version: 2009 - URL: https://yaml.org/spec/1.2/spec.pdf - Abrufdatum: 20.01.2019

[body-parser] Ong, Jonathan; Wilson, Douglas Christopher; Contributors: *Node.js body parsing middleware* - Version: 26.04.2019 - URL: https://github.com/expressjs/body-parser/releases/tag/1.19.0 - Abrufdatum: 30.10.2019

[Bryan und Zyp 2012] Bryan, Paul C.; Zyp, Kris: *JSON Reference* - Version: 16.09.2012 - URL: https://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03 - Abrufdatum: 08.02.2020

[Clean Code Developer 2015a] Clean Code Developer: *Law of Demeter* - Version: 2015 - URL: https://clean-code-developer.de/die-grade/gruener-grad/#Law_of_Demeter - Abrufdatum: 04.10.2019

[Clean Code Developer 2015b] Clean Code Developer: *Information Hiding Principle* - Version: 2015 - URL: https://clean-code-developer.de/die-grade/gelber-grad/#Information_Hiding_Principle - Abrufdatum: 04.10.2019

[Clean Code Developer 2015c] Clean Code Developer: *Separation of Concerns (SoC)* - Version: 2015 - URL: https://clean-code-developer.de/die-grade/orangener-grad/#Separation_of_Concerns_SoC - Abrufdatum: 16.09.2019

[Clean Code Developer 2015d] Clean Code Developer: *Single Responsibility Principle (SRP)* - Version: 2015 - URL: https://clean-code-developer.de/die-grade/orangener-grad/#Single_Responsibility_Principle_SRP - Abrufdatum: 16.09.2019

[Clean Code Developer 2015e] Clean Code Developer: *Keep it simple, stupid (KISS)* - Version: 2015 - URL: https://clean-code-developer.de/die-grade/roter-grad/#Keep_it_simple_stupid_KISS - Abrufdatum: 04.10.2019

[configstore] Google; Contributors: *Easily load and persist config without having to think about where and how* - Version: 11.06.2019 - URL: https://github.com/yeoman/configstore/releases/tag/v5.0.0 - Abrufdatum: 20.12.2019

[ECMA 2017] Ecma International: *ECMA-404: The JSON Data Interchange Standard* - Version: 2017 - URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf - Abrufdatum: 20.01.2019

[electron] GitHub Inc.; Contributors: *Build cross-platform desktop apps with JavaScript, HTML, and CSS* - Version: 25.10.2019 - URL: https://github.com/electron/electron/releases/tag/v6.1.2 - Abrufdatum: 30.10.2019

[electron-args] akameco: *CLI helper for electron* - Version: 23.08.2016 - URL: https://github.com/akameco/electron-args/releases/tag/v0.1.0 - Abrufdatum: 30.10.2019

[electron-builder] Loopline Systems; Contributors: *A complete solution to package and build a ready for distribution Electron app with "auto update" support out of the box* - Version: 31.07.2019 - URL: https://github.com/electron-userland/electron-builder/releases/tag/v21.2.0 - Abrufdatum: 30.10.2019

[electron-window-state] Wiehle, Marcel; Contributors: *A library to store and restore window sizes and positions for your Electron app* - Version: 25.11.2018 - URL: https://github.com/mawie81/electron-window-state/releases/tag/v5.0.3 - Abrufdatum: 30.10.2019

[express] Holowaychuk, TJ; Shtylman, Roman; Wilson, Douglas Christopher; Contributors: *Fast, unopinionated, minimalist web framework for Node.js* - Version: 26.05.2019 - URL: https://github.com/expressjs/express/releases/tag/4.17.1 - Abrufdatum: 30.10.2019

[Fielding 2000] Fielding, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures* - Version: 2000 - URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm - Abrufdatum: 24.01.2019

[Fielding u. a. 1999] Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T.: *Hypertext Transfer Protocol - HTTP/1.1, RFC2616* - Version: 1999 - URL: https://tools.ietf.org/html/rfc2616 - Abrufdatum: 19.01.2019

[GitHub Inc. 2019a] GitHub Inc.: *Electron-Dokumentation "Über Electron"* - Version: 25.10.2019 - URL: https://github.com/electron/electron/blob/dabaa7557a165cc107f89fd7c3e3c3210f9b5758/docs/tutorial/about.md - Abrufdatum: 30.10.2019

[GitHub Inc. 2019b] GitHub Inc.: *Electron Application Architecture* - Version: 26.07.2018 - URL: https://github.com/electron/electron/blob/v6.1.2/docs/tutorial/application-architecture.md - Abrufdatum: 30.10.2019

[Google 2019] Google: *Material icons are delightful, beautifully crafted symbols for common actions and items.* - Version: 2019 - URL: https://material.io/resources/icons/?style=sharp - Abrufdatum: 30.10.2019

[Insomnia REST Client 2019a] Insomnia REST Client: *Debug APIs like a human, not a robot* - Version: 2019 - URL: https://insomnia.rest/ - Abrufdatum: 08.05.2019

[Insomnia REST Client 2019b] Insomnia REST Client: *Chaining Requests* - Version: 2019 - URL: https://support.insomnia.rest/article/43-chaining-requests - Abrufdatum: 08.05.2019

[jsdom] Insua, Elijah; Contributors: *A JavaScript implementation of various web standards, for use with Node.js* - Version: 14.10.2019 - URL: https://github.com/jsdom/jsdom/releases/tag/15.2.0 - Abrufdatum: 30.10.2019

[JSON Schema 2019a] JSON Schema: *This page lists implementations with (or actively working towards) support for draft-06 or later.* - Version: 16.09.2019 - URL: http://json-schema.org/implementations.html - Abrufdatum: 30.01.2019

[JSON Schema 2019b] JSON Schema: *Various levels of support for UI generation primarily from the validation vocabulary or combined with UI specific definition.* - Version: 23.04.2020 - URL: http://json-schema.org/implementations.html#web-ui-generation - Abrufdatum: 23.04.2020

[js-yaml] Puzrin, Vitaly; Contributors: *JavaScript YAML parser and dumper. Very fast.* - Version: 05.04.2019 - URL: https://github.com/nodeca/js-yaml/releases/tag/3.13.1 - Abrufdatum: 30.10.2019

[McIlroy 1978] McIlroy, Malcolm Douglas: *UNIX Time-Sharing System* - Version: 1978 - URL: https://archive.org/details/bstj57-6-1899/page/n3 - Abrufdatum: 24.01.2019

[Miller 2016] Miller, Darrel: *OpenAPI's use of the term REST* - Version: 03.09.2016 - URL: https://github.com/OAI/OpenAPI-Specification/issues/777#issuecomment-244566444 - Abrufdatum: 29.01.2019

[mocha] OpenJS Foundation; Contributors: *Simple, flexible, fun JavaScript test framework for Node.js & The Browser* - Version: 18.10.2019 - URL: https://github.com/mochajs/mocha/releases/tag/v6.2.2 - Abrufdatum: 30.10.2019

[Mozilla.org u. a. 2019] Mozilla; Contributors: *Webtechnologien für Entwickler "JavaScript"* - Version: 27.10.2018 - URL: https://developer.mozilla.org/de/docs/Web/JavaScript - Abrufdatum: 20.01.2019

[Node.js u. a. 2019] Node.js; Contributors: *Über Node.js* - Version: 2019 - URL: https://nodejs.org/de/about/ - Abrufdatum: 23.01.2019

[npm, Inc. 2020] npm, Inc.: *npm Documentation* - Version: 09.04.2020 - URL: https://docs.npmjs.com/ - Abrufdatum: 09.04.2020

[OpenAPI-Generator u. a. 2020] OpenAPI-Generator; Contributors; SmartBear Software: *OpenAPI Generator allows generation of API client libraries (SDK generation), server stubs, documentation and configuration automatically given an OpenAPI Spec (v2, v3)* - Version: 2020 - URL: https://github.com/OpenAPITools/openapi-generator - Abrufdatum: 18.04.2020

[openapi-jsonschema-parameters] Spencer, Joe; Contributors: *Converts openapi parameters to a jsonschema format.* - Version: 23.10.2019 - URL: https://www.npmjs.com/package/openapi-jsonschema-parameters/v/1.2.0 - Abrufdatum: 20.11.2019

[openapi-request-validator] Spencer, Joe; Contributors: *Validate request properties against an OpenAPI spec.* - Version: 09.12.2019 - URL: https://www.npmjs.com/package/openapi-request-validator/v/4.2.0 - Abrufdatum: 09.12.2019

[openapi-response-validator] Spencer, Joe; Contributors: *Validate a response according to an openapi schema.* - Version: 23.10.2019 - URL: https://www.npmjs.com/package/openapi-response-validator/v/4.0.0 - Abrufdatum: 09.12.2019

[Pinkham 2017] Pinkham, Ryan: *What Is the Difference Between Swagger and OpenAPI?* - Version: 31.10.2017 - URL: https://smartbear.com/blog/develop/what-is-the-difference-between-swagger-and-openapi/ - Abrufdatum: 24.01.2019

[Postman inc. 2020a] Postman inc.: *Introduction* - Version: 14.01.2020 - URL: https://learning.postman.com/docs/postman/launching-postman/introduction/ - Abrufdatum: 14.01.2020

[Postman inc. 2020b] Postman inc.: *Test scripts* - Version: 03.01.2020 - URL: https://learning.getpostman.com/docs/postman/scripts/test_scripts/ - Abrufdatum: 14.01.2020

[Refsnes Data 2019a] Refsnes Data: *HTML Introduction* - Version: 2019 - URL: https://www.w3schools.com/html/html_intro.asp - Abrufdatum: 20.01.2019

[Refsnes Data 2019b] Refsnes Data: *CSS Tutorial* - Version: 2019 - URL: https://www.w3schools.com/css/ - Abrufdatum: 20.01.2019

[rjsf-team 2020] rjsf-team: *A React component for building Web forms from JSON Schema.* - Version: 2020 - URL: https://rjsf-team.github.io/react-jsonschema-form/ - Abrufdatum: 19.02.2020

[Roberts 2014] Roberts, Philip: *What the heck is the event loop anyway?* - Version: 2014 - URL: https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html - Abrufdatum: 23.01.2019

[Roy-Chowdhury 2016] Roy-Chowdhury, Rahul: *From Chrome Apps to the Web* - Version: 19.08.2016 - URL: https://blog.chromium.org/2016/08/from-chrome-apps-to-web.html - Abrufdatum: 06.05.2019

[Sellier u. a. 2019] Sellier, Alexis; The Core Less Team: *Overview* - Version: 2019 - URL: http://lesscss.org/# - Abrufdatum: 20.01.2019

[Siedersleben 2003] Siedersleben, Johannes: *Quasar: Die sd&m Standard-architektur* - Version: 2003 - URL: https://docplayer.org/19346553-Quasar-quasar-die-sd-m-standardarchitektur-teil-1-johannes-siedersleben-hrsg.html - Abrufdatum: 19.02.2020

[SmartBear Software 2019a] SmartBear Software: *Swagger Petstore 1.0.0* - Version: 2019 - URL: https://petstore.swagger.io/v2/swagger.json - Abrufdatum: 30.01.2019

[SmartBear Software 2019b] SmartBear Software: *Authentication and Authorization* - Version: 2019 - URL: https://swagger.io/docs/specification/authentication/ - Abrufdatum: 30.01.2019

[SmartBear Software 2019c] SmartBear Software: *oneOf, anyOf, allOf, not* - Version: 2019 - URL: https://swagger.io/docs/specification/data-models/oneof-anyof-allof-not/ - Abrufdatum: 30.01.2019

[SmartBear Software 2019d] SmartBear Software: *Parameter Types* - Version: 2019 - URL: https://swagger.io/docs/specification/describing-parameters#types - Abrufdatum: 23.04.2019

[SmartBear Software 2019e] SmartBear Software: *Path Parameters* - Version: 2019 - URL: https://swagger.io/docs/specification/describing-parameters#path-parameters - Abrufdatum: 23.04.2019

[SmartBear Software 2019f] SmartBear Software: *Describing Request Body* - Version: 2019 - URL: https://swagger.io/docs/specification/describing-request-body/ - Abrufdatum: 23.04.2019

[SmartBear Software 2019g] SmartBear Software: *Reusable Bodies* - Version: 2019 - URL: https://swagger.io/docs/specification/describing-request-body/ - Abrufdatum: 23.04.2019

[SmartBear Software 2019h] SmartBear Software: *Places Where $ref Can Be Used* - Version: 2019 - URL: https://swagger.io/docs/specification/using-ref#allowed-places - Abrufdatum: 23.04.2019

[SmartBear Software 2019i] SmartBear Software: *Object and Property Examples* - Version: 2019 - URL: https://swagger.io/docs/specification/adding-examples#schemas - Abrufdatum: 16.04.2019

[SmartBear Software 2019j] SmartBear Software: *Common Mistakes* - Version: 2020 - URL: https://swagger.io/docs/specification/describing-parameters#mistakes - Abrufdatum: 04.02.2020

[SmartBear Software 2019k] SmartBear Software: *Request and Response Body Examples, Multiple Examples* - Version: 2019 - URL: https://swagger.io/docs/specification/adding-examples#request-response-bodies - Abrufdatum: 16.04.2019

[SmartBear Software 2019l] SmartBear Software: *Parameter Examples* - Version: 2019 - URL: https://swagger.io/docs/specification/adding-examples#parameters - Abrufdatum: 16.04.2019

[SmartBear Software 2019m] SmartBear Software: *Swagger UI* - Version: 2019 - URL: https://swagger.io/tools/swagger-ui/ - Abrufdatum: 16.04.2019

[SmartBear Software 2019n] SmartBear Software: *Swagger Editor* - Version: 2019 - URL: https://swagger.io/tools/swagger-editor/ - Abrufdatum: 16.04.2019

[SmartBear Software 2019o] SmartBear Software: *Data Types* - Version: 2019 - URL: https://swagger.io/docs/specification/data-models/data-types/ - Abrufdatum: 02.06.2019

[SmartBear Software 2019p] SmartBear Software: *Enums* - Version: 2019 - URL: https://swagger.io/docs/specification/data-models/enums/ - Abrufdatum: 02.06.2019

[SmartBear Software 2019q] SmartBear Software: *Funktionen "getLineNumberForPath" und "pathForPosition" in ast.js* - Version: 23.05.2019 - URL: https://github.com/swagger-api/swagger-editor/blob/8e09dfe7ac9ab33afb157d4668915bb0047c404e/src/plugins/ast/ast.js - Abrufdatum: 15.11.2019

[SmartBear Software 2019r] SmartBear Software: *API General Info* - Version: 2019 - URL: https://swagger.io/docs/specification/api-general-info/ - Abrufdatum: 25.02.2020

[Starke u. a. 2019] Starke, Gernot; Hruschka, Peter; Müller, Ralf D.: *arc42 helps to develop, communicate and improve software architectures.* - Version: 2019 - URL: https://arc42.org/ - Abrufdatum: 10.04.2019

[swagger2openapi] Mermade Software; Contributors: *Convert Swagger 2.0 definitions to OpenAPI 3.0 and resolve/validate/lint* - Version: 19.07.2019 - URL: https://github.com/Mermade/oas-kit/releases/tag/swagger2openapi%405.3.1 - Abrufdatum: 30.10.2019

[swagger-parser] Messinger, James; Contributors: *Swagger 2.0 and OpenAPI 3.0 parser/validator* - Version: 24.10.2019 - URL: https://github.com/APIDevTools/swagger-parser/releases/tag/v8.0.3 - Abrufdatum: 30.10.2019

[The jQuery Foundation 2019a] The jQuery Foundation: *What is jQuery?* - Version: 2019 - URL: https://jquery.com/ - Abrufdatum: 30.01.2019

[The jQuery Foundation 2019b] The jQuery Foundation:
Escapes any character that has a special meaning in a CSS selector. - Version: 2019 - URL: https://api.jquery.com/jQuery.escapeSelector/ - Abrufdatum: 03.11.2019

[The Linux Foundation u. a. 2018] The Linux Foundation; Contributors:
OpenAPI Specification - Version: 06.10.2018 - URL: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md - Abrufdatum: 20.01.2019

[WHATWG 2020] WHATWG (Apple, Google, Mozilla, Microsoft):
Introduction to "The DOM" - Version: 06.04.2020 -
URL: https://dom.spec.whatwg.org/#introduction-to-the-dom - Abrufdatum: 09.04.2020

[yaml-js] Connelly, Chris; Contributors: *A port of PyYAML to CommonJS* -
Version: 18.12.2017 - URL: https://github.com/connec/yaml-js/releases/tag/0.3.0 -
Abrufdatum: 30.10.2019

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 23. Juni 2020

Alexander Holland

Anhang

A1: JavaScript-Programmcode der umgesetzten OpenAPI-IDE.

Der Anhang A1 ist auf der beiliegenden CD.