

E-Business Architekturen

Ergebnisse der Komplexübungen (Prüfungsleistung)

vorgelegt am
08.05.2023

an der
Hochschule für Wirtschaft und Recht Berlin
Fachbereich Duales Studium

von: Robert Neubert
Danny Neupauer
Hannes Roever
Fachrichtung: Wirtschaftsinformatik
Studienjahrgang: WI20C
Studienhalbjahr: Wintersemester 2022/23
Dozent: Prof. Dr. Andreas Schmietendorf

Inhaltsverzeichnis

1 Einleitung	1
2 Aufgabe 2: Serviceverzeichnisse	1
2.1 Analyse eines Verzeichnisdienstes	1
2.1.1 Vorgehensweise bei der Auswahl eines Verzeichnisdienstes	1
2.1.2 Allgemeines über den Verzeichnisdienst	2
2.1.3 Vergleich von API- und datenorientierten Schnittstellen	5
2.2 Analyse von Web-APIs	6
2.2.1 Erstellung des Bewertungsmodells	6
2.2.2 Einsatz des Bewertungsmodells	7
2.3 API - Spezifikationen	9
2.3.1 Spezifikationsanalyse & Analysetools	9
2.3.2 Einschränkungen und Alternativen	13
3 Übung 3b: Entwicklung eigener Service-Angebote	16
3.1 Möglichkeiten für Implementierung und Deployment	16
3.1.1 Analyse der Möglichkeiten	16
3.1.2 Analytischer Vergleich der Möglichkeiten	16
3.1.2.1 Implementierung	16
3.1.2.2 Deployment	21
3.2 Entwicklung	22
3.2.1 Rahmenbedingungen	22
3.2.1.1 Anforderungen	22
3.2.1.2 Verwendete Sprache(n)	23
3.2.1.3 Komponenten	23
3.2.1.3.1 REST API	23
3.2.1.3.2 Client	25
3.2.1.3.3 Datenbank	26
3.2.1.4 Eingesetzte Frameworks und Libraries	27
3.2.1.5 Konfiguration Entwicklungsumgebung	29
3.2.1.5.1 Datenbank	29
3.2.1.5.2 REST API	29
3.2.1.5.3 Client WebApp	30
3.2.1.6 Deployment	30
3.2.2 Umsetzung	30
3.2.2.1 Design	30
3.2.2.2 Implementierung	31
3.2.2.2.1 Client	31
3.2.2.2.2 REST API	33
3.2.2.2.3 Tests	35
3.2.2.3 Deployment	37
3.2.3 Anbindung Datenbank (Hannes)	39
4 Übung 4d: Sicherheit von Web APIs	40
4.1 Sicherheitsrisiken in Verbindung mit dem HTTP Protokoll	40
4.1.1 HTTPS und TLS	40
4.1.2 Authentifizierungsmöglichkeiten HTTP(S)	41
4.1.3 Cookies	41
4.1.3.1 Begriff	41
4.1.3.2 Sicherheitsrisiken	41
4.2 Möglichkeiten zur Risikominderung	42
4.2.1 OWASP	42

4.2.1.1	Ausgewählte Schwachstellen (5)	43
4.2.1.1.1	Broken Object Level Authorization	43
4.2.1.1.2	Excessive Data Exposure	44
4.2.1.1.3	Broken Function Level Authorization	45
4.2.1.1.4	Injection (hier SQL)	45
4.2.1.1.5	Insufficient Logging & Monitoring	46
4.2.1.2	Zwei ausgewählte Testmöglichkeiten (Hannes)	47
4.2.2	OAuth 2 und OIDC (Danny)	47
4.3	Praktische Anwendung von OAuth2	48
4.3.1	Testwerkzeuge	48
4.3.1.1	Authorization Code Flow	49
4.3.1.2	OIDC Flow	50
4.3.2	Implementierung	51

1 Einleitung

2 Aufgabe 2: Serviceverzeichnisse

2.1 Analyse eines Verzeichnisdienstes

In dieser Unteraufgabe wird sich mit der Nutzung und Analyse von Service-Verzeichnissen beschäftigt. Dazu wird sich zunächst für einen zu betrachtenden Verzeichnisdienst entschieden, welcher im weiteren Verlauf der Aufgabe auf seine Eigenschaften überprüft wird.

2.1.1 Vorgehensweise bei der Auswahl eines Verzeichnisdienstes

Um einen vollumfänglichen Überblick über den Verzeichnisdienst bieten zu können, wurde bei der Auswahl des Verzeichnisdienstes darauf geachtet, einen Verzeichnisdienst ohne Zugangsbeschränkungen mit öffentlich zugänglichen APIs zu wählen. Aufgrund dieser Vorgaben wir uns für das API-Verzeichnis des Bundes entschieden.

Die APIs, die unter der Webadresse <https://bund.dev/apis> (Abb.1) zusammengefasst sind, dienen dem Zweck, den Zugang zu verschiedenen Datensätzen und Verwaltungsverfahren der Bundesverwaltung zu erleichtern. Sie ermöglichen es Entwicklern und anderen interessierten Nutzern, auf eine standardisierte und dokumentierte Art und Weise auf diese Informationen zuzugreifen und sie in eigenen Anwendungen zu nutzen. Dabei können die APIs verschiedene Funktionalitäten bereitstellen, wie beispielsweise die Abfrage von Daten, die Bearbeitung von Anträgen oder die Einreichung von Dokumenten. Durch die Bereitstellung dieser APIs im Rahmen der Open Government Umsetzungsstrategie des Bundes wird eine transparentere und effizientere Zusammenarbeit zwischen Verwaltung und Bürgern angestrebt.



Schnittstellen für alle Lebens- und Geschäftslagen

Egal ob Sie als Privatperson eine Open-Source-Anwendung bauen oder als Unternehmen Open-Data in Ihre Produkte integrieren wollen. Unsere Schnittstellen stehen der gesamten Gesellschaft offen.

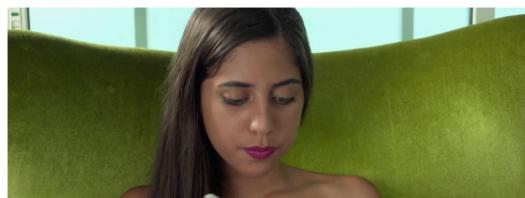


Abbildung 1: Ansicht der Startseite des Verzeichnisdienstes

2.1.2 Allgemeines über den Verzeichnisdienst

In ihrer Gesamtheit sind auf der Website insgesamt 47 diverse Web-APIs identifizierbar. Das Verzeichnis bietet keine Filter-, Sortier- oder Suchfunktionen und ist als einfache Liste dargestellt (Abb.2).

The screenshot shows a list of five APIs from the registry:

- regio iT: Abfallnavi API**
Zugriff auf die Termine der Müllabfuhr unterschiedlicher Kommunen.
[Dokumentation](#) [Github](#)
- Bundesagentur für Arbeit: Ausbildungssuche API**
Eine der größten Ausbildungsdatenbanken Deutschlands durchsuchen.
[Dokumentation](#) [Github](#)
- Autobahn GmbH: Autobahn App API**
Was passiert auf Deutschlands Bundesstraßen? API für aktuelle Verwaltungsdaten zu Baustellen, Staus und Ladestationen. Außerdem Zugang zu Verkehrsüberwachungskameras und vielen weiteren Datensätzen.
[Dokumentation](#) [Github](#)
- Bundesagentur für Arbeit: Berufssprachkursuche API**
Eine der größten Berufssprachkursdatenbanken Deutschlands durchsuchen.
[Dokumentation](#) [Github](#)
- Bundesagentur für Arbeit: Bewerberbörsse API**
API zur Bewerberbörsse der Bundesagentur für Arbeit.
[Dokumentation](#) [Github](#)
- Bundesministerium der Finanzen: Bundeshaushalt API**
API Beschreibung von Bundeshaushalt Digital.
[Github](#)

Abbildung 2: Ansicht der Startseite des Verzeichnisdienstes (runtergescrollt)

Alle APIs sind im REST Architekturstil umgesetzt, was quantitativ die Ergebnisse von Umfragen in der Industrie und unter Entwicklern abbildet (s. Abb.3). Die APIs können hauptsächlich verschiedenen Bundes- und nachgeordneten Behörden zugeordnet werden. Jedoch lassen sich vereinzelt auch APIs von Landesbehörden sowie von Anstalten des öffentlichen Rechts ausmachen.

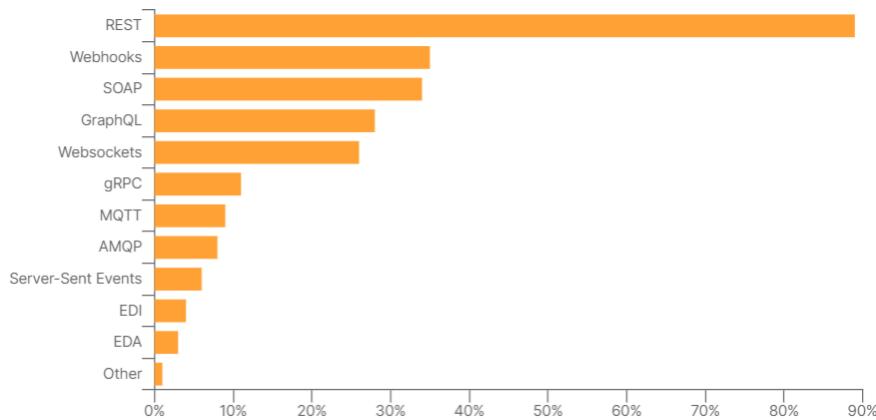


Abbildung 3: Architekturstil eingesetzter APIs¹

Die Qualität der bereitgestellten Dokumentationen variiert und erstreckt sich von minimalen Informationen auf Github (Abb.5) mit einem einzigen Beispiel (oder der Information, die API sei deaktiviert),

über mal schlechter, mal besser Nutzung von Open API (mit Swagger UI, Abb.4) bis hin zu umfangreichen und gut strukturierten Dokumentationen, teilweise mit eigener Webpräsenz (FIT-Connect, Abb.6).

The screenshot shows a GitHub repository page for 'Bundeshaushalt-API'. The repository has 1 branch and 0 tags. It contains several files like .github/workflows, python-client, CNAME, README.md, README_en.md, generator_config.yaml, index.html, openapi.yaml, and openapi_en.yaml. The README.md file contains links for [DE]/[EN] and a code block with a curl command. The right sidebar provides an API description for 'Bundeshaushalt Digital' with details like 1 star, 3 watching, and 0 forks. It also lists releases, packages, contributors (with 2 entries), and languages (Python 99.0%, Other 1.0%).

Abbildung 4: rudimentäre Dokumentation auf Github (Bundeshaushalt API)

The screenshot shows the 'Arbeitsagentur Ausbildungssuche API' documentation using the Swagger UI. The API is version 1.0.0 and follows OAS3 standards. It includes a 'Servers' dropdown set to 'https://rest.arbeitsagentur.de/infosyspub/absuche', an 'Authorize' button, and a 'default' section with two GET requests: '/pc/v1/ausbildungssangebot' (Ausbildungssuche) and '/pc/v1/ausbildungssangebot/{id}' (Ausbildungsdetails). Below these are 'Schemas', 'Response', and 'Details' sections.

Abbildung 5: gute, standardisierte Dokumentation mit Swagger UI (Arbeitsagentur API)

Self-Service API

Diese API-Spezifikation im Format OpenAPI herunterladen.

API Version anzeigen: 1.1.0 ▾

AUTHENTICATION

No API key applied

OAuth (OAuth2)

Authentifizierungsmethode, um den Zugriff nach RFC8725 zu autorisieren

CLIENT CREDENTIALS FLOW

Token URL: https://auth-testing.fit-connect.fitko.dev/token

Scopes:

- https://schema.fitko.de/fit-connect/oauth/scopes/self-service-api/create:destination - Erlaubt das Anlegen von Zustellpunkten
- https://schema.fitko.de/fit-connect/oauth/scopes/self-service-api/manage:destination:<id> - Erlaubt das Verändern und Löschen eines Zustellpunktes

client-id client-secret Authorization Header GET TOKEN

Expand all | Collapse all sections

Zustellpunktverwaltung

Endpunkte für die Verwaltung von Zustellpunkten.

<code>GET</code> /v1/destinations	Eigene Zustellpunkte auflisten
<code>POST</code> /v1/destinations	Zustellpunkt anlegen
<code>GET</code> /v1/destinations/{destinationId}	Zustellpunkt abfragen
<code>PUT</code> /v1/destinations/{destinationId}	Zustellpunkt vollständig aktualisieren
<code>DELETE</code> /v1/destinations/{destinationId}	Zustellpunkt löschen
<code>PATCH</code> /v1/destinations/{destinationId}	Zustellpunkt partiell aktualisieren
<code>POST</code> /v1/destinations/{destinationId}/keys	Fügt dem Zustellpunkt einen JWK hinzu.

Abbildung 6: nicht standardisierte, sehr umfangreiche Dokumentation in eigenen Format (FIT CONNECT, mehrere APIs)

Das Hochladen selbst entwickelter API's ist auf der BundDev Platform nicht vorgesehen. Es besteht aber die Möglichkeit über die GitHub Verbindung eigene Forks zu kreieren und diese zu verändern und anzupassen. Die Aktualität variiert, die letzten Aktualisierungen in den Repositories (oft 2 Jahre her) deuten jedoch auf eine im Vergleich zum Industriedurchschnitt deutlich geringere Deploymentfrequenz hin (s. auch Abb.7).

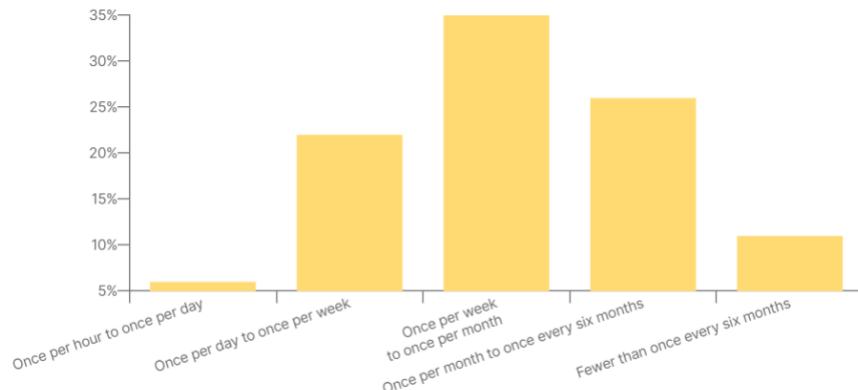


Abbildung 7: Deploymentfrequenz von APIs²

Da viele Datengrundlagen die von den API's genutzt werden öffentlich zugänglich sind besteht auch die

Möglichkeit mit Originaldaten zu arbeiten. Nachdem der Code angepasst oder verbessert wurde lässt sich dieser über einen Pull Request in den ursprünglichen Code integrieren. Das Melden von Problemen und Bugs ist in einigen Fällen nur über GitHub möglich (Abb.8). In wenigen Dokumentationen ist eine Kontaktmail hinterlegt. Eine standardisierte Vorlage für das Reporting von Schwierigkeiten besteht nicht.

Abbildung 8: Gitrepo der “populärsten” API (Autobahn API), Ansicht offener Issues. Oben sind auch die 4 offenen PRs, 10 Forks und 111 Sterne zu sehen (Stand: 7.5.2023)

Wie in (Abb.8) ersichtlich ist, können bei der Erstellung von Problemen verschiedene Tags zugeordnet werden, um eine Klassifizierung für den Entwickler zu erleichtern. Der Verzeichnisdienst ist ohne vorherige Anmeldung oder Registrierung zugänglich. Ebenso kann der API-Code ohne GIT-Registrierung oder Anmeldung heruntergeladen werden. Das Melden von Problemen über die GIT-Funktion erfordert jedoch eine vorherige Anmeldung und Registrierung. Analog dazu verhält es sich bei verfügbaren Pull Requests und ähnlichen Funktionalitäten.

2.1.3 Vergleich von API- und datenorientierten Schnittstellen

Die in der Aufgabe genannte Differenzierung in API und datenorientierten Schnittstelle ist aus unserer Sicht nicht sinnvoll. API steht für Application Programming Interface, zu deutsch laut Wikipedia Programmierschnittstelle oder Anwendungsschnittstelle. Diese sind wiederum in die in Tabelle 1 dargestellten Kategorien unterteilt.³ Datei- oder datenorientierte Schnittstellen sind dabei eine der Kategorien. REST APIs sind hingegen am ehesten den protokollorientierten Schnittstellen zuzuordnen, da sie architektonisch bedingt vom HTTP(S) Protokoll abhängen.⁴

³Vgl. Wikipedia (2023a)

⁴Vgl. Schnepf (2021)

Schnittstellen-Typ	Funktionsorientierte PS	Dateiorientierte PS	Objektorientierte PS	Protokollarientierte PS
Beschreibung	Stellt Funktionen zur Verfügung, die von anderen Anwendungen aufgerufen werden können.	Bietet die Möglichkeit, auf Daten in Dateien zuzugreifen und diese zu verarbeiten.	Basiert auf Objekten, die Funktionen und Eigenschaften enthalten und von anderen Anwendungen verwendet werden können.	Bietet eine strukturierte Art und Weise, um Daten zwischen Systemen auszutauschen.
Beispiele	DLL, Programm-APIs, Bibliotheken	CSV, XML, JSON	COM, CORBA, SOAP	HTTP, TCP/IP, FTP
Verwendung	Häufig in einfachen Anwendungen verwendet.	Nützlich für Anwendungen, die mit großen Datenmengen arbeiten.	Komplexere Anwendungen, die eine umfangreichere Struktur benötigen.	Weit verbreitet in verteilten Systemen und bei der Kommunikation zwischen verschiedenen Anwendungen.
Vorteile	Schnell und einfach zu implementieren.	Gut geeignet für die Arbeit mit großen Datenmengen.	Bietet eine flexible Art der Datenverarbeitung und -speicherung.	Bietet eine standardisierte Art der Kommunikation zwischen Systemen.
Nachteile	Kann bei komplexen Anwendungen unübersichtlich werden.	Begrenzte Funktionalität im Vergleich zu anderen Schnittstellentypen.	Komplex in der Implementierung und erfordert mehr Aufwand.	Kann weniger effizient als andere Schnittstellentypen sein.

Tabelle 1: Kategorisierung APIs

2.2 Analyse von Web-APIs

2.2.1 Erstellung des Bewertungsmodells

Wir haben uns dazu entschieden alle APIs aus dem BundDev Verzeichnis zu bewerten. So ist es möglich eine Übersicht der vom Bund bereitgestellten Schnittstellen zu gewinnen. Zu Beginn wurde das Bewertungsmodell in fünf Kategorien aufgeteilt. Diese lauten: Übersicht, Offenheit, Qualität, Dokumentation und Verfügbarkeit.

1. In diesem Teil wird nur eine Übersicht über die einzelnen API's dargestellt. Unter welchen URL lässt sich die Dokumentation finden? Wie viele get, post, put, delete und andere werden werden in der API verwendet? Hierbei wird keine Bewertung vorgenommen, sondern es wird nur aufgezählt.
2. Offenheit:
 - Quellcode: Die Offenlegung des Quellcodes einer API fördert Transparenz und Anpassungsfähigkeit, indem sie Entwicklern Einblicke in die Funktionsweise der API gewährt

und die Möglichkeit bietet, die API an individuelle Bedürfnisse anzupassen sowie Fehler zu beheben.

- Insofern ein Token zur Nutzung notwendig ist, sollte dies einfach, kostenlos und umgehend zur Verfügung gestellt werden (nach Registrierung)
- Zugänglichkeit: Dokumentation, Beispiele, Kontakt, verwendete Sprache (englisch, keine Fachbegriffe und Abkürzungen) sind nachvollziehbar und gut auffindbar
- Kontakt: Verantwortlicher und ggf. Entwickler können direkt erreicht werden

3. Qualität:

- Granularität: zu gering (wenige Ressourcen mit wenigen Routen geben sehr große Objekte zurück) bis zu hoch (für “normale” Usecases müssen für ein clientrelevantes Objekt mehrere Abfragen gestellt werden)
- TLS: ausschließlich oder Weiterleitung bei HTTP Aufruf (gut) über HTTP möglich (mittel) bis nur unter HTTP erreichbar (schlecht)
- Statuscode: Werden alle relevanten Statuscodes in den Dokumentationen genannt und ausreichen beschrieben?
- Ressourcen fachlich korrekt ausgewählt?
- Routen in angemessener Tiefe?

4. Dokumentation:

- Routen: beschrieben, Datentypen, Beispiele
- Ressourcen: benannt, Request und Response Modelle verfügbar (required? usw.)
- Parameter: benannt, Datentypen, Beispiele

5. Verfügbarkeit und Performance

2.2.2 Einsatz des Bewertungsmodells

Bewertungsteil	Beschreibung	Punkte
Token/Registrierung	Nicht notwendig ODER notwendig, aber leicht einzurichten. Wenn ja, mit Token sind alle Endpunkte verfügbar	2
	Notwendig, aber auch dann nicht alle Endpunkte verfügbar ODER notwendig, aber mit Hürden einzurichten	1
	Für alle Endpunkte nötig UND mit Hürden einzurichten ODER Token ungültig	0
Quellcode	Liegt vor und ist verlinkt	1
	Liegt nicht vor oder muss erst gesucht werden	0
Request Limit	Wenn nicht angegeben, stellen wir verteilt über 30 Minuten 1000 Anfragen. Wenn das klappt, gibt es zwei Punkte.	
	< 100	0
	< 1000	1
	< 10k	2
	> 10k	3
Kontakt	Ja (Link/E-Mail)	1
	nein	0

Tabelle 2: Bewertungsschema Offenheit

Bewertungsteil	Beschreibung	Punkte
Granularität	Ausgeglichen, sowohl Listen als auch Einschränkungen auf einzelne Objekte	1
	Sehr viele Use Cases mit einzelnen Endpunkten oder eine Anfrage erfordert verschiedene Ressourcen	0
	Zu geringe Granularität, sehr wenige Endpunkte mit zu umfangreichen Datenmodellen	0
Transportverschlüsselung	HTTPS und gegebenenfalls Weiterleitung von HTTP auf HTTPS	2
	HTTP und HTTPS, aber keine Weiterleitung von HTTP	1
	Nur HTTP	0
Routen	Jede Ressource hat einen Identifier und ist mit Nomen benannt. HTTP-Methoden sind korrekt eingesetzt. Maximale Tiefe beträgt 3.	2
	Mindestens 2 Bedingungen aus 2 sind erfüllt	1
	Eine oder keine Bedingung aus 2 erfüllt	0
Statuscodes	200, 201, 204, 400, 401, 403, 404, 409, 500, 503	2
	Mind. 200, 201, 400, 404, 500	1
	Keine	0
MIME Types	JSON und/ oder XML/ plain text	2
	Nur XML und/ oder plain text	1
	Keine/ plain	0
Versionierung	Verschiedene Majorversionen = 2	
	Nur latest = 1	
	Nicht angegeben = 0	

Tabelle 3: Bewertungsschema Qualität

Bewertungsteil	Beschreibung	Punkte
Routen	alle Routen mit Beispielen (Request/Response), wenn nicht sprechend mit Erläuterung, alle notwendigen und optionalen Parameter sind angegeben	2
	wie zwei, aber nur teilweise erfüllt	1
	Keine Doku	0
Ressourcen	alle Ressourcen dokumentiert mit Beispielen, Datentyp und ggfs. default Wert	2
	wie zwei, aber nur teilweise erfüllt	1
	Keine Doku	0
Parameter	alle Parameter mit Datentyp, erlaubtem Wertebereich, required und - wenn nicht sprechend - mit Beispiel, ggfs. Differenzierung der Rückgabebobjekte ist dokumentiert	2
	wie zwei, aber nur teilweise erfüllt	1
	Keine Doku	0
Statuscodes	Responsecodes mit Zahl, Rückgabestring und Datentyp, Mimetype oder Datenmodell dokumentiert	2
	wie zwei, aber nur teilweise erfüllt	1
	Keine Doku	0
Swagger	Swagger yaml oder json file vorhanden, sowie Swagger UI zum ausprobieren, alle Beispiele und ggfs Auth./Tokens in Swagger UI funktionieren	2
	Swagger yaml file oder json vorhanden, aber keine Swagger UI oder andere Möglichkeit zum ausprobieren	1
	Keine Doku/ Möglichkeit zum Testen	0

Tabelle 4: Bewertungsschema Dokumentation

2.3 API - Spezifikationen

2.3.1 Spezifikationsanalyse & Analysetools

Die gestellte Aufgabe werden wir in der gestellten Form nicht lösen. Begründung: Die Analyse der Struktur und Elemente einer WSDL, OpenAPI-Spezifikation oder eines GraphQL-Schemas kann evtl. akademisch interessant sein. Eine (wie hier geforderte) rein metrische Auswertung der Struktur oder der eingesetzten Elemente in einer REST API-Spezifikation ist in der Praxis aber nicht zielführend. Entwickler sind vielmehr daran interessiert, wie einfach es ist, mit der API zu arbeiten, welche Funktionen sie bereitstellt und welche Datenformate sie unterstützt.

Auswahlkriterien sind somit nicht aus Metriken generierte Zahlen, sondern dass die API gut dokumentiert ist, eine klare und konsistente Architektur aufweist und die verwendeten Datenformate den Anforderungen des Projekts entsprechen. Auch Aspekte wie Performance, Skalierbarkeit und Sicherheit spielen eine wichtige Rolle.

Statt einer metrischen Auswertung kann es in der Praxis sinnvoller sein, Tools wie Swagger Hub oder Postman zu verwenden, um eine REST API auszuprobieren und ihre Funktionalität zu testen. Diese Tools bieten eine interaktive Schnittstelle zum Testen von API-Endpunkten, zur Überprüfung von Parametern und zur Anzeige von Ergebnissen. Auf diese Weise können Entwickler schnell und einfach herausfinden, ob eine API für ihre Zwecke geeignet ist und wie sie mit ihr interagieren können.

Wir werden deshalb Screenshots der "Auswertungen", wie sie mit diesen Tools stattfinden, für Swagger Hub und Postman, bereitstellen, jedoch auf die zwangsläufige Redundanz beim "Vergleich" von 5 Services verzichten (5 Screenshots sind es trotzdem :-))⁵. Wir haben uns für REST

⁵Lachende Smileys lösen aufgrund des fröhlichen Gesichts das Feuern von Spiegelneuronen aus. Die in der Folge ausgeschütteten Hormone und Neurotransmitter erhöhen u.a. die Konzentration. Dies und nichts anderes wollen wir hier, wissenschaftlich fundiert, erreichen. Wikipedia (2023b)

entschieden, weil es von 89% der Entwickler verwendet wird, gefolgt von Webhooks (35%), SOAP (34%) und GraphQL (34%).⁶ Die Beispiele nutzen jeweils die Spezifikation der Autobahn API.

Abbildung 9: Ansicht in Swagger Hub nach Import einer Open API Spezifikation (json oder yaml). Ähnlich zur Swagger UI sind alle Routen, Parameter, Datenmodelle und Ressourcen gut visualisiert und können ausprobiert werden. Die API kann zudem durch NoCode Tools erweitert und verändert werden und anschließend sowohl die Dokumentation, als auch SourceCode für die meisten gängigen Sprachen exportiert werden.

Abbildung 10: Die in Abb.9 erwähnte Möglichkeit des Exports der Dokumentation, hier am Beispiel statischer HTML Files

⁶Vgl. Postman (2022)

```

1: {
2:   "roadworks": [
3:     {
4:       "extent": "10.728384054665147,54.00605746113356,10.775848767524598,54.09436740278899",
5:       "identifier": "UK9BRFDpUKtTX19tZG@uc2hfxYZMTU",
6:       "routeRecommendation": [],
7:       "coordinates": [
8:           {
9:               "lat": "54.0000857",
10:              "long": "10.729057"
11:            },
12:            {
13:                "icon": "123",
14:                "isLocked": "false",
15:                "description": [
16:                    "Beginn: 29.06.2021 09:00",
17:                    "Ende: 28.11.2021 17:00",
18:                    "Art der Maßnahme: Asphaltdeckerneuerung",
19:                    "Einschränkungen: Es steht nur 1 Fahrstreifen zur Verfügung.\n\nVollsperrung der AS Scharbeutz Ostseite vom 17.07.2021 - 15.09.2021",
20:                    ".\n\nVollsperrung der AS Scharbeutz Ostseite vom 16.09.2021 - 17.11.2021.",
21:                    "Maximale Durchfahrsbreite: 3.25\n"
22:                ],
23:                "title": "A1 | AS Pansdorf (17) - AS Neustadt-Mitte (14)",
24:                "point": "10.729057,54.0000857",
25:                "display_type": "ROADWORKS",
26:                "lorryParkFeatureIcons": [],
27:                "furthest": "true",
28:                "subtitle": "Lübeck Richtung Fehmarn",
29:                "startTimestamp": "2021-06-29T09:00:00+0200"
30:            }
1: ]
}

```

Abbildung 11: Ansicht in Mockoon (Client) nach Import der json/yaml Spezifikation. Die API kann hervorragend visuell analysiert werden. Natürlich ist auch eine Anpassung möglich und Rückgaben können auf vielerlei Weisen gemockt werden, so dass z.B. für die Cliententwicklung gegen diese spezifische API, schnell und ohne Zugriff auf produktive Ressourcen, getestet werden kann.

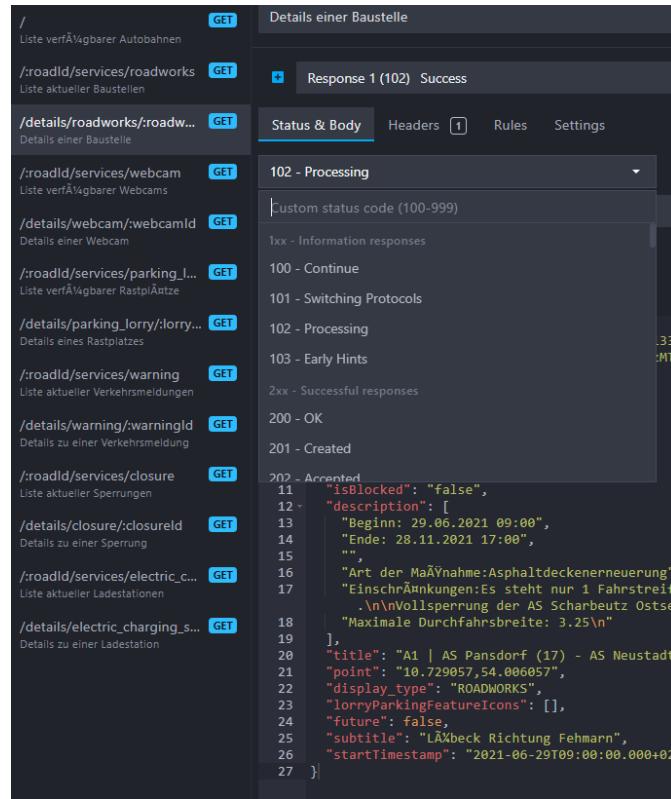


Abbildung 12: Beispiel der in der Autobahn API Spezifikation dokumentierten Rückgabecodes für die entsprechende Route. Diese können sofort genutzt werden, über logische Verknüpfungen dynamisch zurückgegeben, gelöscht und erweitert werden. Auch eine künstliche Verzögerung des Response ist möglich.

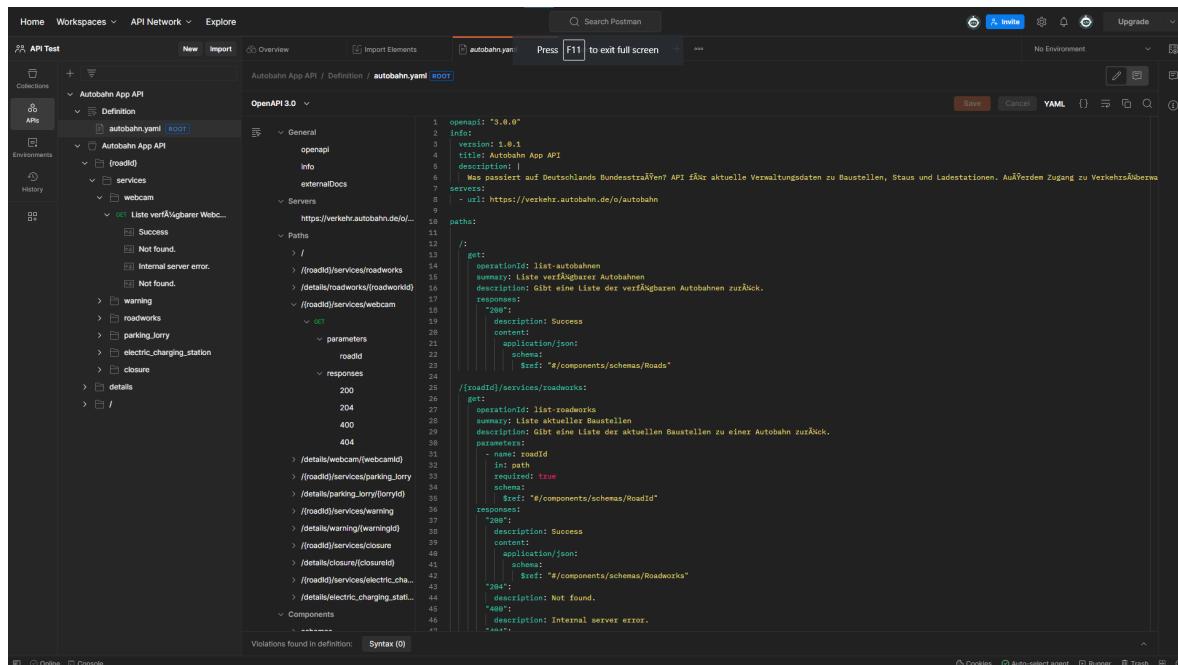


Abbildung 13: Analog zu Swagger Hub und Mockoon hier die Ansicht in Postman nach Import der Spezifikationsdatei. Die Ansicht ähnelt den beiden anderen, jedoch bietet Postman zusätzlich viele weitere Features zur Analyse, Dokumentation, Monitoring, Entwicklung und Deployment eigener APIs.

2.3.2 Einschränkungen und Alternativen

Die Fragen (Welche Informationen fehlen bei der gewählten Spezifikationen?, Recherchieren Sie nach alternativen Beschreibungsformen?) sind ähnlich praxisfern. Kein Mensch käme doch auf die Idee die verwendete REST-Schnittstelle von der vorhandenen Spezifikation abhängig zu machen. Verfügbarkeit, Preis, Skalierbarkeit, Performanz, Sicherheit etc. sind alles ausschlaggebende Kriterien - sowie natürlich, DASS die Schnittstelle gut dokumentiert ist (s. auch Abb.14). Womit ist dann eher zweitrangig.

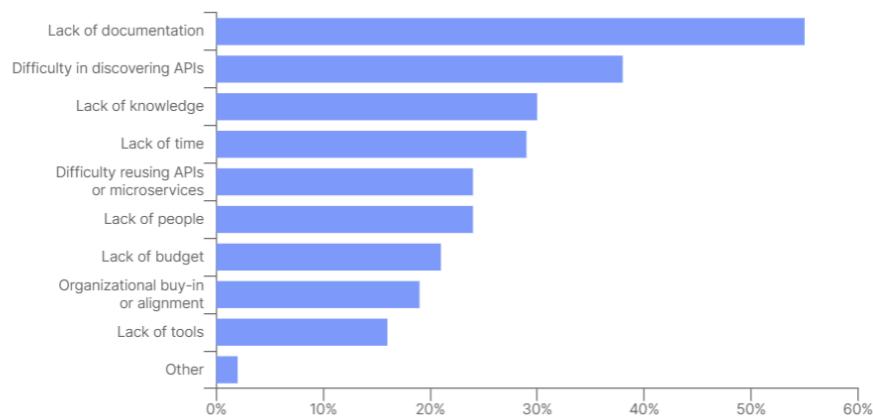


Abbildung 14: Häufigste Hindernisse bei der Verwendung von APIs⁷

Auch welche Informationen “fehlen” ist zwar nicht egal, aber insofern nicht zu beantworten, weil die notwendigen Informationen vom Usecase abhängen. Bezuglich der Funktionsweise lässt sich nur sagen: Spezifikation als Datei oder Link bereitstellen - ausprobieren - fertig. Die Tools unterscheiden sich hinsichtlich des Umfangs und der Komplexität, funktionieren aber in Grundzügen alle gleich: Sie visualisieren und ordnen die von der API bereitgestellten Funktionen. Schließlich ist im Bezug auf REST der Markt zwischen Json Schema und Swagger/Open API aufgeteilt (GraphQL: 34%, WSDL: 26%):⁸

“We also asked folks which API specifications they use and love. JSON Schema was by far the most popular choice, used by 72% of respondents. The next most popular were Swagger 2.0 (55%) and OpenAPI 3.x (39%)”

Deshalb erfolgt an dieser Stelle keine systematische Analyse, sondern lediglich ein Vergleich zwischen den beiden (Swagger und Open API haben fast einen identischen Funktionsumfang und werden hier zusammengefasst) und anschließend einige ergänzende Screenshots der am häufigsten eingesetzten Tools Swagger Hub und Postman (Abb.15) und ergänzend ein von uns gern eingesetztes Tool (Mockoon) - sowie ein Hinweis auf die Möglichkeit der automatisierten “Analyse” mit Postman Flows.

⁷Vgl. Postman (2022)

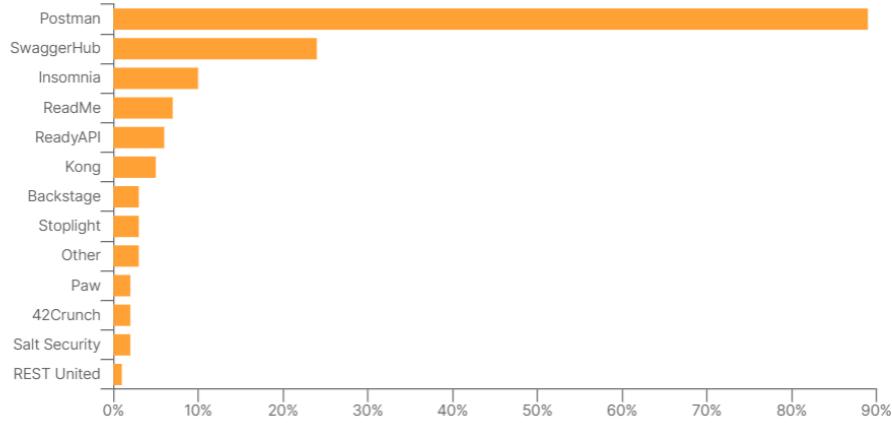


Abbildung 15: Beliebteste Tools und Plattformen für die Arbeit mit APIs⁹ (Disclaimer: Die Umfrage stammt von Postman selbst)

Funktion	JSON Schema	Swagger 2.0/OpenAPI 3.0
Schema-Definitionssprache	Ja	Ja
API-Beschreibung	Nein	Ja
API-Tests	Nein	Ja
API-Design	Nein	Ja
API-Visualisierung	Nein	Ja
API-Mocking	Nein	Ja
Beispielantworten	Nein	Ja
Beispielanfragen	Nein	Ja
Versionierung	Nein	Ja
Wiederverwendbare Komponenten	Nein	Ja
Sicherheitsdefinitionen	Nein	Ja
Unterstützung von Aufzählungen	Ja	Ja

Tabelle 5: Hauptunterschiede zwischen JSON Schema und Swagger 2.0/ OpenAPI 3.0

Listing 1: Testscript in Postman zum rekursiven Auslesen der zurückgegeben Objekte (distinct keys) die Ergebnisse können dann in Flows visualisiert werden s. Abb.16

```

1 pm.test("Status test", function() {
2     let keys = [];
3
4     function countKeys(obj) {
5         Object.keys(obj).forEach(key => {
6             if (!keys.includes(key)) {
7                 keys.push(key);
8             }
9             if (typeof obj[key] === 'object' && obj[key] !== null) {
10                 countKeys(obj[key]);
11             }
12         });
13     }
14     try {
15         const rootKey = Object.keys(pm.response.json())[0];
16         const root = pm.response.json()[rootKey][0];

```

```

17     if (root) {
18         countKeys(root);
19         console.log(keys.length);
20         pm.test(keys.length, function() {
21             pm.expect(true).to.eql(true);
22         });
23     } else {
24         console.log("Root element not found.");
25     }
26 } catch (error) {}
27
28 pm.expect(true).to.eql(true);
29
30 });

```

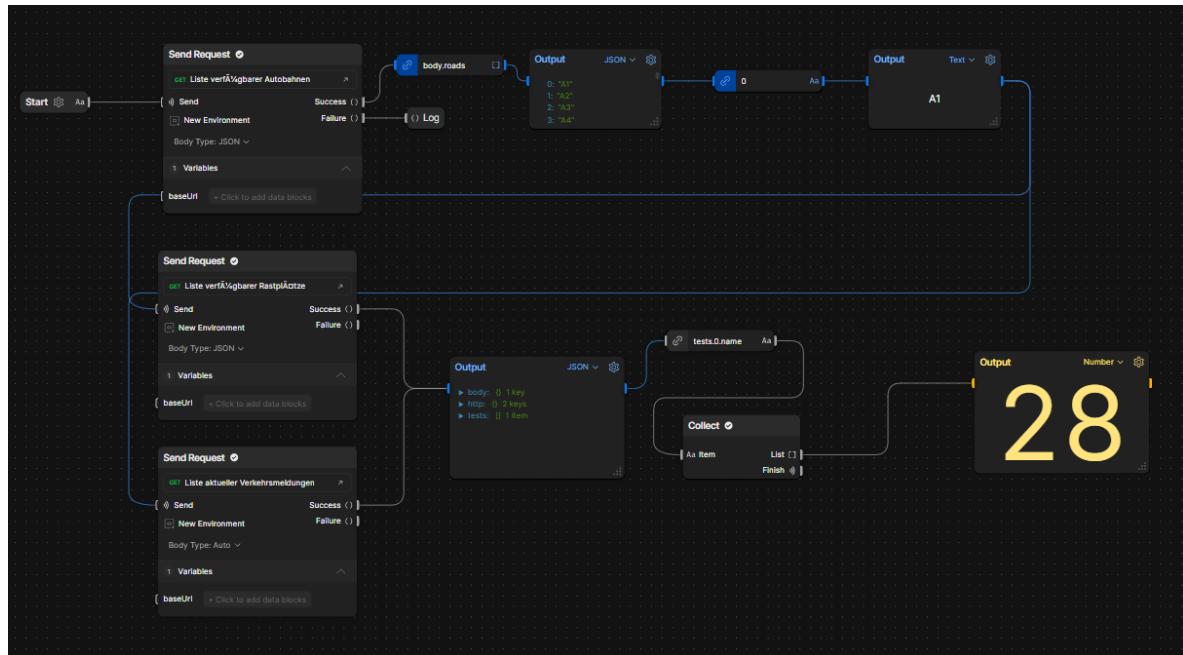


Abbildung 16: Postman LowCode Tool “Flows”, hier ein prototypisches Beispiel zur Nutzung für die Analyse: Der Abruf von 2 Resourcen der Autobahn API setzt einen Inputparameter voraus (Autobahnnummer), welcher im Response des ersten request enthalten ist. Der Aufruf triggert den o.g. Test und zählt rekursiv die Anzahl der eindeutigen Bezeichner. Natürlich könnten Informationen wie die Tiefe, Größe der Objekte etc. einfach hinzugefügt werden. Abschließend wird die Zahl für jeden Request ausgegeben. Auch hier wäre es durch integrierte Blöcke für statistische Auswertungen einfach umzusetzen, die Zahlen in einem Array festzuhalten und am Ende z.B. ein Balkendiagramm zu erzeugen. Da Low Code hier mit JS verknüpft werden kann und mehrere API Spezifikationen in einer “Collection” zusammengefasst werden können, wäre es zudem möglich iterativ alle Routen aller APIs abzufragen und automatisiert zu dokumentieren. Wir haben bereits erläutert warum wie dies nicht gemacht haben.

3 Übung 3b: Entwicklung eigener Service-Angebote

3.1 Möglichkeiten für Implementierung und Deployment

3.1.1 Analyse der Möglichkeiten

Nach eingehender Internetrecherche wurden einige Werkzeuge zur Implementierung und Deployment von APIs zusammengetragen. Zur besseren Einordnung dieser sind Eigenschaften wie Scope und Typ sowie falls vorhanden Laufzeitanforderungen aufgeführt.

Plattform / Service	Scope	Typ	Laufzeit
AWS Lamda mit AWS API Gateway	Implementierung, Deployment	SaaS	-
ApiGee	Implementierung, Deployment	SaaS	-
Postman Api Builder	Implementierung	SaaS	-
Firebase	Implementierung, Deployment	SaaS	-
Swagger Hub	Deployment	SaaS	-
Cloudflare Workers	Deployment	FaaS	-
Supabase	Implementierung, Deployment	PaaS	-
AWS Amplify	Deployment	PaaS	-
AWS ECS	Deployment	IaaS	-
Swagger Codegen	Implementierung	Executable	Java
Apicurio Studio	Implementierung	Executable	Java
ASP.NET	Implementierung	Framework	C#
express.js	Implementierung	Framework	JS
Flask	Implementierung	Framework	Python
Spring Boot	Implementierung	Framework	Java
Ruby on Rails	Implementierung	Framework	Ruby

Tabelle 6: Übersicht über verschiedene Plattformen und Services für die Implementierung, Deployment und Nutzung von APIs.

3.1.2 Analytischer Vergleich der Möglichkeiten

Im Folgenden werden ausgewählte der im vorigen Abschnitt identifizierten Implementierungs- und Deploymentmöglichkeiten mittels eines Bewertungsschemas verglichen. Die ausgewählten tools stehen exemplarisch für jeweils eine Implementierungs- oder Deploymentart.

3.1.2.1 Implementierung

Das der Bewertung verschiedener Implementierungsarten zugehörige Schema beinhaltet die Kriterien Komplexität der Implementierung, Komplexität der OpenApi-Spezifikationserstellung, Güte der Dokumentation, Popularität, Kosten und Geschwindigkeit. Zur Bewertung der Implementierungskomplexität werden die Implementierungen der gleichen Api verglichen. Dazu wird die künstliche Intelligenz ChatGPT genutzt. Diese bekommt pro Implementierungsart die gleiche Aufforderung, welche wie folgt aussieht:

Implement a rest api with one endpoint named /items. This endpoint should return all columns of the mysql database table item and should be able to response the http-codes 200, 404 and 500. Do so using the shortest possible way in Implementierungsart.

Die Ergebnisse wurden dann auf die Herkunft der nötigen libraries sowie die Anzahl der Funktionsaufrufe untersucht. Zur Bewertung der Komplexität der OpenApi Spezifikationserstellung wurde recherchiert, ob eine Spezifikationserstellung überhaupt möglich und wenn möglich ohne externe Hilfsmittel/Libraries möglich ist. Außerdem wurde berücksichtigt, ob die Erstellung automatisch oder manuell erfolgt. Zur effektiven Entwicklung von Software sind umfangreiche, verständliche und vor allem aktuelle Dokumentationen von großer Bedeutung. Aufgrund dessen ist auch die Dokumentationsgüte Teil des Bewertungsschemas. Hier fließen die Übersichtlichkeit, der Umfang, die Aktualität und die Verständlichkeit ein. Dabei gilt zu beachten, dass diese Bewertungen nicht objektiv messbar sind und daher subjektiv bewertet wurden. Außerdem wurde die Popularität mittels Google Trends bestimmt. Die Geschwindigkeit wurde anhand von Benchmarks gerankt. Ein weiteres sehr wichtiges Kriterium zu Auswahl der Entwicklungswerkzeuge ist die Preisstruktur dieser, weshalb diese ebenfalls aufgeführt ist.

Listing 2: Implementierung in Java

```

1 import java.sql.ResultSet;
2 import java.sql.SQLException;
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.jdbc.core.JdbcTemplate;
9 import org.springframework.jdbc.core.RowMapper;
10 import org.springframework.web.bind.annotation.GetMapping;
11 import org.springframework.web.bind.annotation.RequestMapping;
12 import org.springframework.web.bind.annotation.RestController;
13
14 @RestController
15 @RequestMapping("/items")
16 public class ItemController {
17
18     public static void main(String[] args) {
19         SpringApplication.run(Main.class, args);
20     }
21
22     @Autowired
23     private JdbcTemplate jdbcTemplate;
24
25     @GetMapping
26     public ResponseEntity<List<Item>> getAllItems() {
27         try {
28             List<Item> items = jdbcTemplate.query(
29                 "SELECT * FROM item",
30                 new RowMapper<Item>() {
31                     public Item mapRow(ResultSet rs, int rowNum) throws
32                         SQLException {
33                         Item item = new Item();
34                         item.setId(rs.getLong("id"));
35                         item.setName(rs.getString("name"));
36                         item.setDescription(rs.getString("description"));
37                         item.setPrice(rs.getDouble("price"));
38                         return item;
39                     }
40                 });
41             if (items.isEmpty()) {
42                 return new ResponseEntity<>(HttpStatus.NOT_FOUND);
43             }
44             return new ResponseEntity<>(items, HttpStatus.OK);
45         } catch (Exception e) {
46             return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
47         }
48     }
49
50     public static class Item {
51         private long id;
52         private String name;
53     }
54 }
```

```

52     private String description;
53     private double price;
54
55     public long getId() {
56         return id;
57     }
58
59     public void setId(long id) {
60         this.id = id;
61     }
62
63     public String getName() {
64         return name;
65     }
66
67     public void setName(String name) {
68         this.name = name;
69     }
70
71     public String getDescription() {
72         return description;
73     }
74
75     public void setDescription(String description) {
76         this.description = description;
77     }
78
79     public double getPrice() {
80         return price;
81     }
82
83     public void setPrice(double price) {
84         this.price = price;
85     }
86 }
87 }
```

Listing 3: Implementierung in python

```

1 from flask import Flask, jsonify
2 from flask_mysqldb import MySQL
3
4 app = Flask(__name__)
5 app.config['MYSQL_HOST'] = 'localhost'
6 app.config['MYSQL_USER'] = 'username'
7 app.config['MYSQL_PASSWORD'] = 'password'
8 app.config['MYSQL_DB'] = 'database'
9 mysql = MySQL(app)
10
11 @app.route('/items')
12 def get_items():
13     cur = mysql.connection.cursor()
14     cur.execute("SELECT * FROM item")
15     data = cur.fetchall()
16     if data:
17         return jsonify(data), 200
18     else:
19         return jsonify({"message": "No items found"}), 404
20
21 @app.errorhandler(500)
22 def internal_error(error):
23     return jsonify({"message": "Internal server error"}), 500
24
25 if __name__ == '__main__':
26     app.run(debug=True)
```

Listing 4: Implementierung in C#

```

1 using Microsoft.AspNetCore.Builder;
2 using Microsoft.AspNetCore.Http;
3 using Microsoft.Extensions.DependencyInjection;
4 using Dapper;
5 using MySql.Data.MySqlClient;
6 using System;
7 using System.Linq;
8
9 var builder = WebApplication.CreateBuilder(args);
10
11 builder.Services.AddSingleton<MySqlConnection>(sp =>
12     new MySqlConnection(builder.Configuration.GetConnectionString("DefaultConnectionString")));
13
14 var app = builder.Build();
15
16 app.MapGet("/items", async (HttpContext httpContext, MySqlConnection connection)
=>
17 {
18     try
19     {
20         var items = (await connection.QueryAsync<Item>("SELECT * FROM Items")).ToList();
21         if (items.Count == 0)
22         {
23             return Results.NotFound();
24         }
25         return Results.Ok(items);
26     }
27     catch (Exception ex)
28     {
29         Console.Error.WriteLine(ex);
30         return Results.StatusCode(StatusCodes.Status500InternalServerError);
31     }
32 });
33
34 app.Run();
35
36 public record Item(int Id, string Name, string Description, decimal Price,
        DateTime CreatedAt);

```

Kriterium	Unterkategorie	Bewertungen	Quellen
Komplexität der Implementierung	notwendige Hilfsmittel/Libraries	1: keine externen Hilfsmittel/Libraries nötig 0: externe Hilfsmittel/Libraries nötig	
	Anzahl der notwendigen Schritte/Funktionsaufrufe	0: mehr als der Durchschnitt der anderen 1: durchschnittlich 2: weniger als der Durchschnitt der anderen	
OpenAPI-Spezifikation	Integration	2: out of the box 1: Drittanbieter library 0: nicht möglich	Spring: ^a Flask: ^b .NET: ^c Postman: ^d
Implementierung		2: automatisch 1: manuell	
Dokumentation	Übersichtlichkeit (logische Untergliederung)	1: übersichtlich 0: unübersichtlich	Spring: ^e Flask: ^f .NET: ^g Postman: ^h
Umfang		1: vollumfänglich 0: unvollständig	
Aktualität		1: aktuell 0: (teils) veraltet	
Verständnis (ggf. mit Beispielen)		1: gut verständlich 0: nicht gut verständlich	
Popularität	-	Rangfolge entsprechend Google Trends (0-3)	ⁱ
Geschwindigkeit	-	Rangfolge entsprechend TechEmpower Benchmark (0-2)	^j
Skalierbarkeit	Replizierbarkeit	1: möglich 0: nicht möglich	Glassfish ^k
	Loadbalancing (Clustering)	1: möglich 0: nicht möglich	
	Effizienz	1: geringer Ressourcenverbrauch 0: hoher Ressourcenverbrauch	
Continuous Deployment	-	2: ohne Downtime möglich 1: mit Downtime möglich 0: nicht möglich	

^a<https://spring.io/>

^b<https://flask.palletsprojects.com/en/2.1.x/>

^c<https://dotnet.microsoft.com/>

^d<https://www.postman.com/>

^e<https://spring.io/>

^f<https://flask.palletsprojects.com/en/2.1.x/>

^g<https://dotnet.microsoft.com/>

^h<https://www.postman.com/>

ⁱ<https://trends.google.com/trends/explore?q=swagger%20ui,postman,insomnia&geo=US>

^j<https://www.techempower.com/benchmarks/>

^khttps://docs.oracle.com/cd/E19182-01/821-0915/jbi_cluster-create_t/index.html

Tabelle 7: Bewertungskriterien für API-Testtools

3.1.2.2 Deployment

Ähnlich dem Vergleich der Implementierungsmöglichkeiten wurden auch für den Deploymentmöglichkeiten-Vergleich repräsentative Vertreter für die drei verbreitetsten Arten Application Server, Cloud Server und Container gewählt. Diese wurden bezüglich der Skalierbarkeit und der Continuous-Deployment-Fähigkeit verglichen. Dabei wurde zur Bewertung der Skalierbarkeit die Replizierbarkeit und das Möglichkeit von Loadbalancing sowie die Effizienz herangezogen um so die Fähigkeit zum vertikalem Skalieren abzubilden. Die Continuous Deployment Fähigkeit wird damit bestimmt, ob dies grundsätzlich möglich ist und wenn ja, mit oder ohne Server Downtime.

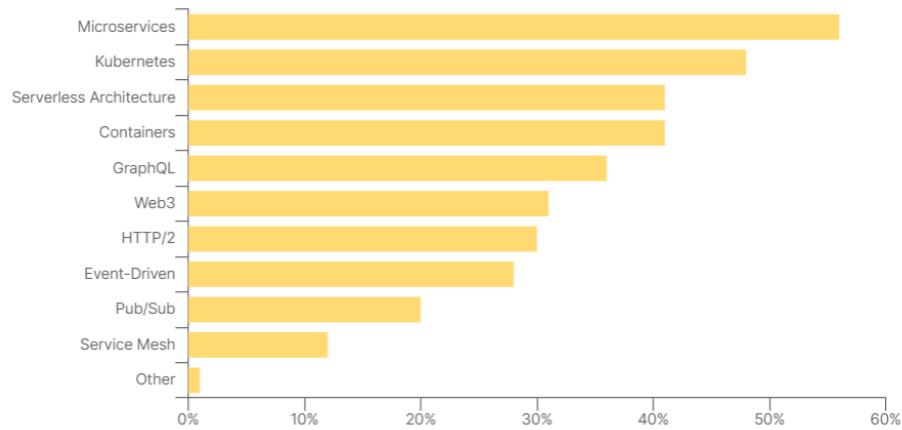


Abbildung 17: Häufigste Deploymentformen von REST APIs¹⁰

	Kriterium	Aws Api Gateway	Azure Api Management	Linode	AWS EC2	On Premise
Flexibilität Entwicklung	mehrere Sprachen	+	+	+	+	+
	vorgegebene Libraries/Frameworks	+	+	+	+	+
	Api-Dokumentation/-Spezifikation	+	+	+	+	+
Flexibilität Deployment	Integration in CD Pipelines	/	+	+	+	+
	Restriktionen	?	?	+	+	+
	verschiedene Umgebungen (Test/Production)	+	+	+	+	+
	parallele Versionen möglich	+	+	+	+	+
Skalierbarkeit Kosten	Containerisierung möglich	?	+	/	/	/
	initial	+	+	+	+	-
	laufend	-	-	/	/	/
Abhängigkeiten	vorgeschriebene Libraries/Frameworks	+	+	+	+	+
	Abhängigkeit von Anbieter selbst	-	-	-	-	+
	Laufzeitumgebung	/	/	-	/	+

3.2 Entwicklung

3.2.1 Rahmenbedingungen

Für eine nachvollziehbare Argumentation, warum der eingesetzte Toolstack verwendet wurde und welche Laufzeitumgebung und Art des Deployments als angebracht eingeschätzt wurde, sollen zunächst kurz die Anforderungen an die Anwendung dargestellt werden. Diese sind zwar “simuliert”, jedoch (in sehr oberflächlicher Form) an möglichen realen Anforderungen angelehnt. Da die nicht-funktionalen Anforderungen hier eher die Argumentationsgrundlage bilden, stehen diese im Fokus - funktionale Anforderungen sollten nur in Ausnahmefällen eine Determinante für Techstack und Deployment sein. Überlegungen, welche eine prototypische Umsetzung im gegebenen Rahmen sprengen würden, werden bewusst außer acht gelassen. Dazu gehören: ggfs. initial höhere Entwicklungskosten, verfügbare (Entwicklungs)ressourcen und Skillset der Beteiligten, architektonische Überlegungen und der Einsatz bestimmter Design Patterns, sowie das Thema Tests.

Des Weiteren halten wir eine Begründung, warum nun welche Entwicklungsumgebung eingesetzt wurde, nicht für sinnvoll. Welche IDE ein Entwickler verwendet, ob als Git nun Github, Gitlab oder Bitbucket verwendet wird und mit welchem Tool REST Endpunkte getestet werden ist entweder von den Vorlieben und Gewohnheiten des Einzelnen abhängig, oder durch Vorgaben des Arbeitgebers bestimmt (oder beides). Insofern beschränken wir uns bei diesen Punkten auf die Benennung der “Werkzeuge”, ohne das Warum weiter zu vertiefen. Stattdessen wollen wir die aus unserer Sicht viel wichtigere Frage beantworten, warum für den genannten Usecase eine bestimmte Sprache, Bibliotheken und Deploymentszenarien gewählt wurden.

3.2.1.1 Anforderungen

Funktionale Anforderungen:

- Anzeige von Basisinformationen zu Coderepositories (Autor, Sprache, Forks, Commits), welche über eine REST API abgerufen werden
- Löschen vorhandener, Hinzufügen neuer und Ändern vorhandener Repositories (Client)
- Persistierung der Änderungen in einer Datenbank
- Bereitstellung als Webapp

Nicht-funktionale Anforderungen:

- unterdurchschnittlich geringe TCO durch:
 - hohe Performanz und geringen Footprint bei der Hardwarenutzung
 - geringe Wartungskosten
 - einfache Verwaltung der Abhängigkeiten
 - einfaches Deployment
- gute Skalierbarkeit
- hohes Level an Sicherheit
- volle Flexibilität hinsichtlich der Laufzeitumgebung
- DB Typ möglichst offen

3.2.1.2 Verwendete Sprache(n)

Client und REST API sollen in Rust geschrieben werden, auch die verwendete Datenbank (Surreal DB) ist in Rust geschrieben. Rust ist eine multi-paradigmatische, noch recht junge (2015) Programmiersprache, die auf konzeptioneller Ebene einige Besonderheiten aufweist. Im Folgenden werden einige dieser Besonderheiten erläutert:

- Memory-Safety und Thread-Safety: Rust erreicht dies durch eine strenge Typisierung und durch Speicherzugriffsregeln, die sicherstellen, dass Speicher nur dann gelesen oder geschrieben werden kann, wenn es korrekt und sicher ist. Dies wird durch die Borrowing- und Ownership-Konzepte erreicht, die den Zugriff auf den Speicher in Rust stark reglementieren. Mit diesen Regeln ist es möglich, Memory-Safety-Garantien zu erzwingen, ohne dass ein Garbage-Collector erforderlich ist, aber auch ohne den in C und C++ verwendeten Ansatz der manuellen Speicherkontrolle.
- Laufzeitstabilität: Rust ist dafür bekannt, Laufzeitfehler quasi auszuschließen (von daher der Name - einmal ausgerollt kann die Anwendung vor sich hin rosten). Dies wird durch eine Kombination aus verschiedenen Techniken erreicht, darunter die bereits erwähnten Konzepte, den Verzicht auf nulls und einen in vielen Fällen funktionalen Programmierstil. Ausschlaggebend für die hohe Laufzeitstabilität ist zudem der tiefgreifende Compiler, der bereits bei der Übersetzung des Codes umfangreiche Fehlerprüfungen durchführt. Dadurch werden viele potenzielle Fehlerquellen bereits im Vorfeld erkannt und beseitigt.
- Gute Dokumentation: Die Gesamtdokumentation, insbesondere das Rust Book, aber auch die Dokumentation der einzelnen Bibliotheken, bietet sowohl Einsteigern als auch erfahrenen Entwicklern Hilfestellungen, um die Sprache zu erlernen und ihre Fähigkeiten zu verbessern.
- Management von Abhängigkeiten: das Management von Abhängigkeiten durch das Cargo-Build-System garantiert eine Kompatibilität der (transitiven) Abhängigkeiten und ein replizierbares Kompilat/Binary, sowie durch SemVer eine einfache Verwaltung der Abhängigkeiten
- Rust hat eine schnell wachsende Community und wird von immer mehr Unternehmen für die (Re)implementierung kritischer Komponenten eingesetzt (z.B. npm, Cloudflare und AWS Lambda). Teile des Android Kernels, sowie des Linuxkernels und neuerdings auch Systemkomponenten in Windows werden in Rust neu geschrieben. Diese Entwicklung deutet auf eine stabile Zukunft sowohl hinsichtlich technischem Support, also auch wachsender Entwicklerressourcen hin - ein wichtiges Argument bei der Businessentscheidung für eine Sprache.

3.2.1.3 Komponenten

3.2.1.3.1 REST API

Aus der Beschreibung in Verbindung mit den nicht funktionalen Anforderungen lässt sich die Entscheidung für Rust für die systemkritischen (REST API) Komponenten ableiten. Sicherheit, Stabilität, eine hohe Flexibilität der Laufzeitumgebungen, Performanz (s. auch Abb.?? sowie voraussichtlich geringe TOC sind bei einer Umsetzung mit Rust wahrscheinlicher als in den meisten anderen Sprachen. Microsoft führt beispielsweise einen großen Teil der Schwachstellen auf fehlerhafte Speicherverwaltung zurück, dieses Risiko wird durch die garantieerte Memory-Safety minimiert:

“Microsoft revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. Google also found a similar percentage of memory safety vulnerabilities over several years in Chrome.”¹¹

Für Rust im Backend spricht auch die vielfältige Auswahl von ausgereiften und hochperformanten Bibliotheken und Frameworks für die Backendentwicklung (Abb.18).

Composite Framework Scores							
Rnk	Framework	JSON	1-query	20-query	Fortunes	Updates	Plaintext
1	just	1,526,714	673,201	34,620	538,414	24,454	6,982,125 8,453 100.0%
2	may-minihhttp	1,546,221	642,348	34,493	520,976	24,192	7,023,484 8,334 98.6%
3	xitca-web	1,207,053	638,244	34,964	587,955	24,488	6,996,736 8,287 98.0%
4	drogon	1,086,998	622,274	29,935	616,607	21,877	5,969,800 7,801 92.3%
5	actix	1,498,561	512,830	29,198	512,422	20,635	7,017,232 7,667 90.7%
6	officefloor	1,374,439	558,932	33,331	432,309	23,691	6,383,827 7,492 88.6%
7	asp.net core	1,306,635	483,762	26,350	458,677	19,644	7,023,107 7,077 83.7%
8	salvo	1,082,630	631,785	33,700	542,547	23,733	1,928,951 7,061 83.5%
9	axum	847,891	612,714	34,880	498,541	24,324	3,780,458 6,982 82.6%
10	wizzardo-http	1,479,464	630,207	31,770	307,614	17,393	7,013,230 6,851 81.0%

Abbildung 18: Benchmark Backend Webframeworks

Ein Nachteil (der auch auf den Rest zutrifft, deshalb wird er nur einmal benannt) kann zudem sein, dass Rust noch eine sehr junge Sprache ist. Die Verfügbarkeit (insbesondere erfahrener) Entwickler ist sehr überschaubar und dürfte sich zudem in höheren Salären niederschlagen. Andererseits ist Rust seit sieben Jahren in Folge (Abb.19) die beliebteste Sprache unter Entwicklern, sowie diejenige, welche auf Platz 1 der Sprachen steht, die Entwickler neu lernen wollen (Abb.20). Das schlägt sich in einer schnell wachsenden Community nieder und dürfte langfristig auch die Verfügbarkeit von Entwicklern erhöhen. Dennoch birgt die Entscheidung im Gegensatz zu einer für eine “etabliertere Sprache” aus Personalsicht ein gewisses Risiko.

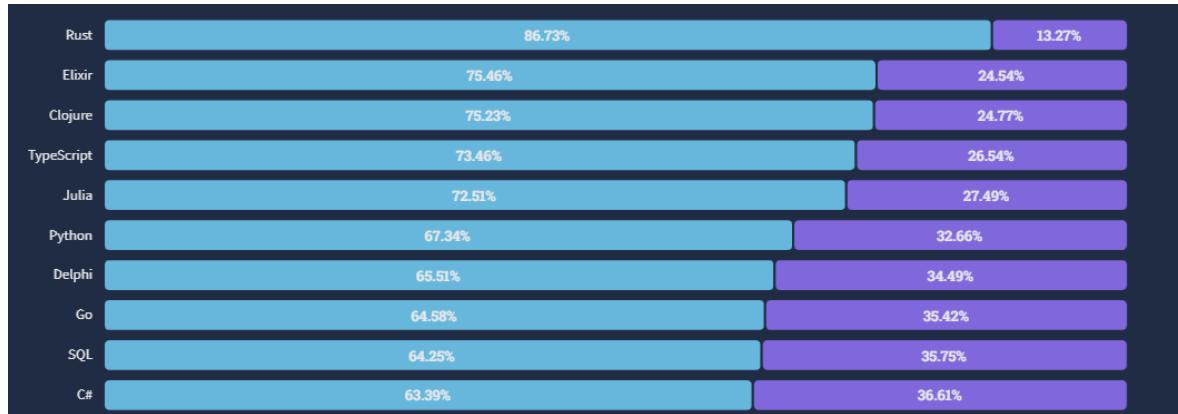


Abbildung 19: Top 10 Programming, scripting, and markup languages (Most loved and dreaded) im Stackoverflow Developersurvey 2022

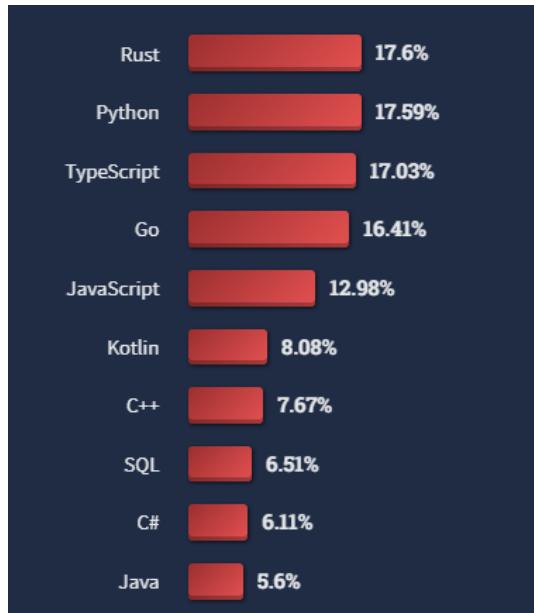


Abbildung 20: Top 10 Programming, scripting, and markup languages (Most wanted) im Stackoverflow Developersurvey 2022

3.2.1.3.2 Client

Die Entscheidung auch das Frontend in Rust zu implementieren war hingegen eher experimenteller Natur und würde - auch aufgrund der teils noch nicht ausgereiften Frameworks - in einer realen Situation vermutlich anders ausfallen. Dennoch soll die Entscheidung an dieser Stelle kurz begründet werden.

Da Rust problemlos in Maschinencode als auch Webassembly (Entwicklung 2018) kompiliert werden kann, verzichten die meisten Webframeworks, die in Rust geschrieben sind, komplett auf Javascript. Systemnahe Sprachen, typischerweise Assembler, C++ oder Rust, aber auch interpretierte Sprachen wie C# können mit der Laufzeitumgebung Webassembly in bytecode kompiliert werden, welcher plattformunabhängig und extrem schnell im Browser, zunehmend aber auch auf verteilten Systemen ausgeführt wird. Das verwendete Framework Perseus zeichnet sich durch seinen Reactive Ansatz (ähnlich Svelte oder Solid.js), sowie eine sehr hohe Performanz aus (s. auch Abb.21). Zudem ist auch das Deployment für Mobiles und Desktopplattformen möglich. Es unterstützt SSR (Serverside Rendering) und benötigt kein Virtual DOM (was i.d.R. durch den zusätzlichen Overhead die Performance etwas drückt)¹².

¹²<https://github.com/flosse/rust-web-framework-comparison>

Duration in milliseconds \pm 95% confidence interval (Slowdown = Duration / Fastest)

Name Duration for...	solid-v1.4.4	sycamore-v0.7.1	elm-v0.19.1-3	vue-v3.2.37	svelte-v3.48.0	yew-v0.19.3	angular-v13.0.0	dioxus-v0.2.4	react-v17.0.2
Implementation notes									
create rows creating 1,000 rows (5 warmup runs).	97.1 \pm 1.4 (1.00)	120.9 \pm 1.7 (1.24)	112.6 \pm 2.5 (1.16)	119.3 \pm 1.2 (1.23)	113.5 \pm 1.5 (1.17)	153.3 \pm 1.2 (1.58)	118.5 \pm 1.4 (1.22)	140.4 \pm 3.8 (1.45)	126.2 \pm 1.4 (1.30)
replace all rows updating all 1,000 rows (5 warmup runs).	101.2 \pm 1.1 (1.00)	125.7 \pm 1.5 (1.24)	116.9 \pm 3.4 (1.16)	113.2 \pm 1.5 (1.12)	116.5 \pm 2.0 (1.15)	166.6 \pm 1.5 (1.65)	130.7 \pm 1.0 (1.29)	145.2 \pm 2.8 (1.44)	126.7 \pm 0.9 (1.25)
partial update updating every 10th row for 1,000 rows (3 warmup runs). 16x CPU slowdown.	312.7 \pm 7.1 (1.00)	336.2 \pm 8.2 (1.08)	335.9 \pm 6.6 (1.07)	359.0 \pm 11.2 (1.15)	334.0 \pm 8.1 (1.07)	359.9 \pm 15.4 (1.15)	343.0 \pm 6.7 (1.10)	621.9 \pm 13.9 (1.99)	399.5 \pm 5.4 (1.28)
select row highlighting a selected row. (5 warmup runs). 16x CPU slowdown.	37.1 \pm 1.5 (1.00)	41.8 \pm 1.3 (1.13)	50.0 \pm 2.3 (1.35)	53.7 \pm 1.2 (1.45)	49.7 \pm 2.3 (1.34)	48.4 \pm 3.0 (1.30)	43.6 \pm 0.8 (1.18)	99.1 \pm 1.9 (2.67)	105.2 \pm 3.7 (2.84)
swap rows swap 2 rows for table with 1,000 rows. (5 warmup runs). 4x CPU slowdown.	67.8 \pm 2.9 (1.00)	70.5 \pm 2.4 (1.04)	68.5 \pm 5.0 (1.01)	73.6 \pm 4.5 (1.08)	69.4 \pm 4.8 (1.02)	69.5 \pm 2.3 (1.03)	512.8 \pm 3.4 (7.56)	88.2 \pm 17.4 (1.30)	494.1 \pm 7.7 (7.28)
remove row removing one row. (5 warmup runs).	25.1 \pm 0.7 (1.00)	25.5 \pm 0.5 (1.01)	31.4 \pm 3.8 (1.25)	28.0 \pm 0.6 (1.11)	25.8 \pm 0.4 (1.03)	25.8 \pm 0.5 (1.03)	25.7 \pm 0.4 (1.02)	50.3 \pm 3.8 (2.00)	27.8 \pm 0.5 (1.11)
create many rows creating 10,000 rows. (5 warmup runs with 1k rows).	1,026.0 \pm 10.9 (1.00)	1,281.4 \pm 17.8 (1.25)	1,156.5 \pm 4.3 (1.13)	1,149.2 \pm 5.1 (1.12)	1,171.2 \pm 9.7 (1.14)	2,309.3 \pm 12.1 (2.25)	1,215.1 \pm 11.5 (1.18)	1,314.5 \pm 4.9 (1.28)	1,488.1 \pm 10.2 (1.45)
append rows to large table appending 1,000 to a table of 10,000 rows. 2x CPU slowdown.	241.0 \pm 4.4 (1.00)	288.9 \pm 9.7 (1.20)	264.8 \pm 5.9 (1.10)	268.8 \pm 7.3 (1.12)	278.7 \pm 6.7 (1.16)	377.9 \pm 4.7 (1.57)	303.2 \pm 2.8 (1.26)	324.3 \pm 12.3 (1.35)	322.5 \pm 4.0 (1.34)
clear rows clearing a table with 1,000 rows. 8x CPU slowdown. (5 warmup runs).	79.8 \pm 1.9 (1.00)	88.1 \pm 2.6 (1.10)	88.4 \pm 4.1 (1.11)	90.0 \pm 2.9 (1.13)	118.9 \pm 3.5 (1.49)	190.1 \pm 6.0 (2.38)	231.0 \pm 6.8 (2.90)	183.1 \pm 26.2 (2.30)	101.0 \pm 2.9 (1.27)
geometric mean of all factors in the table	1.00	1.14	1.14	1.16	1.17	1.48	1.60	1.69	1.70

Abbildung 21: Benchmark Frontend Webframeworks. Perseus wurde nicht getestet, hat aber in Einzelbenchmarks etwas bessere Werte als das zugrundeliegende Sycamore (hier zu sehen)

Zu bedenkende Nachteile, insbesondere im produktiven Einsatz, könnte beim Frontendframework die im Gegensatz zu den Backendframeworks die fehlende Maturität sein. Zudem kommt hier durch den Einsatz von WebAssembly eine weitere spezifische Anforderung an die Fähigkeiten der Entwickler hinzu, was die Verfügbarkeit weiter einschränken dürfte.

3.2.1.3.3 Datenbank

Als Datenbanksystem wurde Surreal DB ausgesucht, weil hier die Bandbreite der Möglichkeiten am höchsten ist. Insbesondere die Nutzung sowohl als relationale und nicht relationale Datenbank, die integrierte, sichere REST Schnittstelle und die hohe Performanz, haben in Bezug auf die genannten Anforderungen zu der Entscheidung geführt. Weitere Vorteile, die zur Auswahl geführt haben:

- Single-node oder verteilt
- strukturierte und unstrukturierte Daten
- GraphQL, REST, WebSockets als Schnittstellen (wir nutzen REST)
- Real time sync

- flexibles modelling (relational/ nicht relational, keine joins)
- Web-native access (JWT, OAuth, Basic)
- open source
- flexible Speichertechnologien (in memory, kv, document)

Zudem bietet das System hinsichtlich der Zugänglichkeit durch die Abfragesprache SurrealQL, welche stark an SQL angelehnt ist, es jedoch z.B. wie in Listing 5 zu sehen ist, um Funktionen wie die Integration von Javascript erweitert.

Listing 5: SQL ähnliche Syntax in Surreal DB

```

1 INSERT INTO company {
2   name: 'SurrealDB',
3   founded: "2021-09-10",
4   founders: [person:tobie, person:jaime],
5   tags: ['big data', 'database']
6 };
7
8 CREATE user:test SET
9   session_timeout = function() {
10     return new Duration('1w');
11   },
12   best_friend = function() {
13     return new Record('user', 'joanna');
14   },
15   identifier = function() {
16     return new Uuid('03412258-988f-47cd-82db-549902cdafffe');
17 };

```

Nachteil ist hier analog zu den anderen Komponenten die geringe Verbreitung. Andererseits wird der Ansatz der Nutzerfreundlichkeit (auch aufgrund der guten Dokumentation) für Entwickler u.E. so gut umgesetzt, dass eine sehr schnelle Einarbeitung im Gegensatz zu anderen DBMS möglich ist.

3.2.1.4 Eingesetzte Frameworks und Libraries

Actix wurde aufgrund der guten Performanz (s. auch Abb.18), Dokumentation, Aktualität und Kompatibilität mit Concurrency ausgewählt. Sycamore (bzw. darauf aufbauend Perseus) aufgrund der Performanz und relativ weiten Verbreitung und somit guten Unterstützung aus der Entwicklercommunity. Utoipa für die Dokumentation mit OpenAPI/Swagger ist quasi alternativlos, ebenso Serde für den Umgang mit Json Objekten. Weitere, transitive Abhängigkeiten (die in Rust qua Architektur des Abhängigkeitenmanagements mit Traits und Crates sehr kleinteilig und quantitativ hoch sind) sind hier nicht dargestellt, können aber beispielhaft für die REST API der toml- Konfigurationsdatei entnommen werden.

Servicekomponente	Name	Version	Autor	Funktion
Client	sycamore	0.8.1	sycamore-rs	Webassembly Webframework
Client	perseus	0.4.0-rc.1	framesurge	Sycamore Erweiterung, ermöglicht Hot State reloading, i18n Support
Client, REST API	tokio	1.0	tokio	Concurrency (async main)
Client, REST API	reqwest / reqwasm	0.11.17 / 0.5.0	Rust Foundation	HTTP Client (Standard und für Webassembly)
REST API	serde	1.0.152	Serde	JSON (De)serialisierung
REST API	actix-Web	4	actix	Webserver, Webframework
REST API	utoipa	3.3	juhaka	Open API Doc Generation
Datenbank	Surreal DB	1.0.0-beta.9	Tobie und Jaime Morgan Hitchcock	vollständiges DBMS und integrierter Server

Tabelle 8: Verwendete, externe Abhängigkeiten

Listing 6: cargo.toml Datei zur Organisation der Abhängigkeiten in Rust hier für die REST API

```

1 [package]
2 name = "rust-actix-surreal-rest-api"
3 version = "0.1.0"
4 edition = "2023"
5 authors = ["Hannes Roever"]
6
7 [dependencies]
8 actix-web = "4"
9 actix-cors = "*"
10 serde = {version = "1.0.152", features = ["derive"]}
11 serde_json = {version = "1.0.93"}
12 tokio = {version = "1", features = ["full"]}
13 mini-redis = "0.4"
14 env_logger = "0.10.0"
15 log = "0.4"
16 futures = "0.3"
17 utoipa = {features = ["actix_extras"]}
18 utoipa-swagger-ui = {features = ["actix-web"]}
19 chrono = "*"
20 reqwest = {features = ["json"]}

```

Listing 7: cargo.toml Client

```

1 [package]
2 name = "perseus-website"
3 version = "0.1.0"
4 edition = "2023"
5 authors = ["Hannes Roever"]
6
7 [dependencies]
8 perseus = {version = "0.4.0-rc.1", features = ["hydrate"]}
9 sycamore = {version = "^0.8.1", features = ["suspense"]}

```

```

10 serde = "1.0.152"
11 serde_json = "1.0.93"
12 reqwest = {features = ["json"]}
13 # futures = "0.3"
14 chrono = "*"
15 dotenv = "0.15.0"
16
17 [target.'cfg(engine)'.dependencies]
18 tokio = {version = "1", features = ["macros", "rt", "rt-multi-thread"] }
19 perseus-axum = {version = "0.4.0-rc.1", features = ["dflt-server"] }
20
21 [target.'cfg(client)'.dependencies]
22 web-sys = {version = "0.3", features = ["Storage"] }
23 reqwasm = "0.5.0"

```

3.2.1.5 Konfiguration Entwicklungsumgebung

Voraussetzung für die dargestellten Schritte ist, dass Docker bereits installiert ist (Docker Client auf Windows, Docker Engine auf Linux). Da dies, analog zum Vorhandensein einer geeigneten IDE oder eines Editors, zu den Basiswerkzeugen in der Entwicklung gehört, wird der allgemeine Installations- und Konfigurationsprozess nicht weiter ausgeführt (zumal er sich je nach OS auch unterscheidet und bestens dokumentiert ist).

3.2.1.5.1 Datenbank

Die Datenbank kann sehr unkompliziert als Docker-Container gestartet werden. Das entsprechende CLI Kommando bzw. der Inhalt und das Kommando zum Ausführen der docker-compose.yml sind in den Listings 8-10 dargestellt. s sollte nur eine der Optionen genutzt werden. Anschließend läuft die Datenbank mit in-memory Option (weitere sind möglich) unter Port 8000 des localhost.

Listing 8: CLI Command zum Starten des Datenbankcontainers

```
1 docker run --rm --pull always -p 8000:8000 surrealdb/surrealdb:latest start
```

Listing 9: Alternative mit docker-compose zum Starten des Datenbankcontainers

```

1 version: '3.8'
2 services:
3   db:
4     image: surrealdb/surrealdb:latest
5     restart: always
6     command: start --user root --pass root memory
7     ports:
8       - '8000:8000'
9     volumes:
10      - db:/var/lib/surrealdb/data
11 volumes:
12   db:
13     driver: local

```

Listing 10: CLI Command zum Ausführen der docker-compose Datei. Das Kommando muss im Verzeichnis ausgeführt werden in dem die Datei liegt oder der Pfad der Datei über die flag -f spezifiziert werden

```
1 docker-compose up -d
```

3.2.1.5.2 REST API

Für die Entwicklung in Rust wird die Rust Toolchain benötigt (bestehend aus rustup, rustc und cargo). Die Installation erfolgt über die Kommandozeile oder für Windows mit einem Installer, welcher unter <https://www.rust-lang.org/tools/install> heruntergeladen werden kann. Ggf. muss noch die

ensprechende Umgebungsvariable gesetzt werden. Die Toolchain umfasst alle notwendigen Commandlinien tools für die Kompilierung, Codeformatierung, Abruf von Dokumentation (ähnlich zu MAN Pages), Tests und Deployment.

Listing 11: CLI Command zur Installation von Rust in Linux und macOS

```
1 curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh
```

Für die Erstellung eines neuen Projekts muss das Kommando cargo new projektname ausgeführt werden. Im entsprechenden Verzeichnis wird ein Ordner mit den Konfigfiles, main und Gitrepository angelegt. Die Bearbeitung des Codes kann mit einem einfachen Editor (z.B. Vim, Neovim, Emacs, Sublime, Nano), einem erweiterten Editor (VS Code) oder einer vollumfänglichen IDE (IntelliJ IDEA, CLion) vorgenommen werden. Wir nutzen IntelliJ und für die schnelle Bearbeitung, z.B. auf einem über SSH verbundenen Server, Nano.

Weitere Schritte sind nicht notwendig, die Abhängigkeiten können in der cargo.toml (s.a. Listing 7) Datei hinzugefügt werden und werden beim nächsten Build, so noch nicht lokal vorhanden, automatisch gezogen und kompiliert. Mit cargo run (bauen, ausführen) bzw cargo build (bauen), fürs publishing mit –release flag, wird das Programm ausgeführt.

3.2.1.5.3 Client WebApp

Um die Kompilierung in WASM zu ermöglichen sind zwei weitere, global bereitzustellende Abhängigkeiten notwendig, die Installation ist in Listing 12 zu sehen.

Listing 12: CLI Command zur Installation der Laufzeitumgebung webassembly und des WASM-Buildtools Trunk für Rust

```
1 rustup target add wasm32-unknown-unknown
2 cargo install --locked trunk
```

Der Start eines bereits erstellten Projektes kann mit trunk –serve durchgeführt werden, durch das Buildtool wird automatisch ein lokaler Webserver bereitgestellt. Perseus baut auf Sycamore auf und kann mit den Commands aus Listing 13 installiert und ausgeführt werden.

Listing 13: CLI Command zur Installation der Perseus CLI und Ausführung eines Projektes

```
1 cargo install perseus-cli
2 perseus serve -w
```

3.2.1.6 Deployment

Das Deployment wird, dem Industriestandard folgend, als Containerlösung realisiert. Um den Rahmen nicht zu sprengen, haben wir uns für Docker (mit docker-compose) entschieden und auf einen Orchestrierungslayer, z.B. mit K8, verzichtet. Die physische Bereitstellung erfolgt bei einem IAAS Anbieter, aufgrund der Nutzung von Docker ist die Linux-Distribution zweitrangig - Debian oder Ubuntu als etablierte Serverdistros oder Alpine als Low-Footprint Distro sind naheliegende Optionen. Windows Server oder spezielle oder proprietäre Lösungen sind nicht notwendig und u.E. auch nicht sinnvoll, weil sie eine Abhängigkeit von einer bestimmten Firma bzw. Technologie schaffen. Zudem haben Umgebungen wie Java Application Server einen extrem hohen Overhead, den wir vermeiden wollen um die gesteckten Ziele nicht zu gefährden.

3.2.2 Umsetzung

3.2.2.1 Design

Der Ansatz folgt einem einfachen Client-Server Paradigma (Client, Serverbackend, Datenbank). In Abb. 22 ist das auf Serviceebene und unter Einsatz von Containern aufgebaute Diagramm gezeigt. In den verschiedenen Phasen einer auf schnelle Deployments ausgerichteten, agilen Entwicklung sind die Bezüge zu den Sourcecoderepositories und einzelnen Phasen der Pipelines, sowie den Repositories (Registries) für die Artefakte dargestellt. Das Diagramm (eigene Quelle) wurde im Rahmen eines

anderen Projekts für die Hochschule erstellt und wenn auch in diesem Kontext die Testphasen keine Rolle spielten, so stellt es doch aus High-Level Sicht gut die Zusammenhänge der zu entwickelnden Komponenten dar.

Weitere Überlegungen zum Design mussten up-front nicht erfolgen, Best Practices wie der Einsatz von Design Patterns und Clean Code Standards können während der Entwicklung eingesetzt werden und aufgrund der Einfachheit der Anforderungen würde eine “Designphase” wahrscheinlich länger dauern als die Entwicklung selbst und die Hälfte davon wird am Ende erfahrungsgemäß ohnehin verworfen.

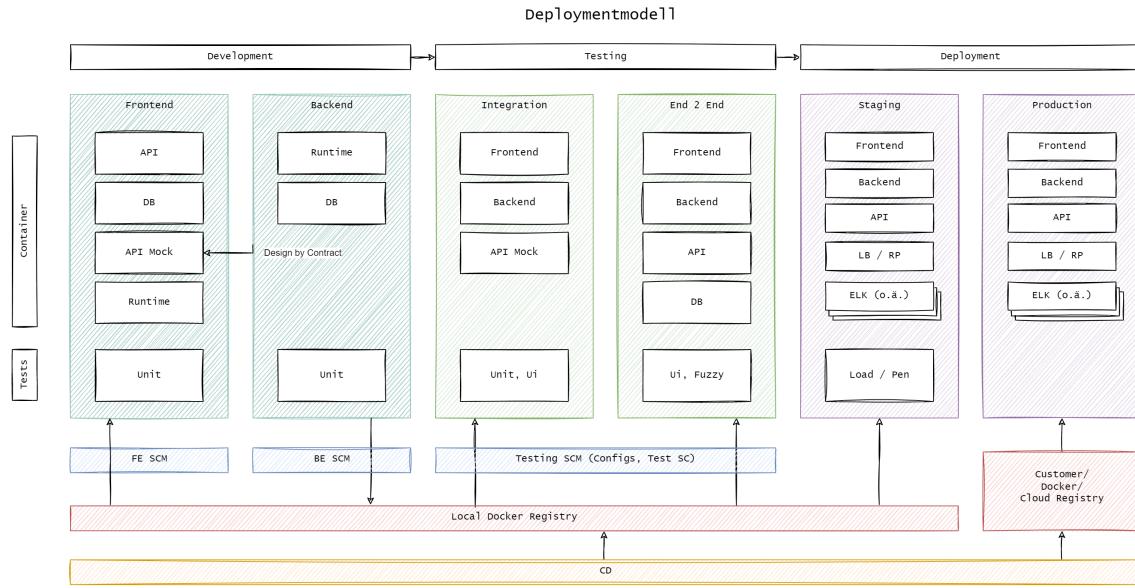


Abbildung 22: Deploymentdiagramm

3.2.2.2 Implementierung

Die Implementierung soll für alle Bestandteile jeweils anhand kurzer Codeabschnitte erläutert werden. Der Fokus ist dabei themenbedingt auf die Komponenten beschränkt, die auf der Schnittstellenebene miteinander kommunizieren, welche jeweils die Boundary der jeweiligen Komponente bilden (also jeweils die REST-Schnittstelle zwischen Client und Backendserver, sowie Backendserver und Datenbank).

3.2.2.2.1 Client

Der Client ist eine einfache Webansicht (eine Seite) auf der alle Repositories angezeigt werden, die in der Datenbank eingetragen sind. Die Möglichkeit ein Repository anzulegen wurde nicht implementiert, ist aber im Backend funktional und kann über Swagger auch genutzt werden. Dinge wie z.B. Pagination, wie sie in einer “richtigen” Anwendung enthalten sein sollten, fehlen ebenfalls, da es ja primär um die REST API spezifischen Fragestellungen ging. Deshalb werden an dieser Stelle nur die beiden wesentlichen codeseitigen Funktionalitäten des Clients beschrieben (Rendering Frontend, Request der Ressourcen vom Backend).

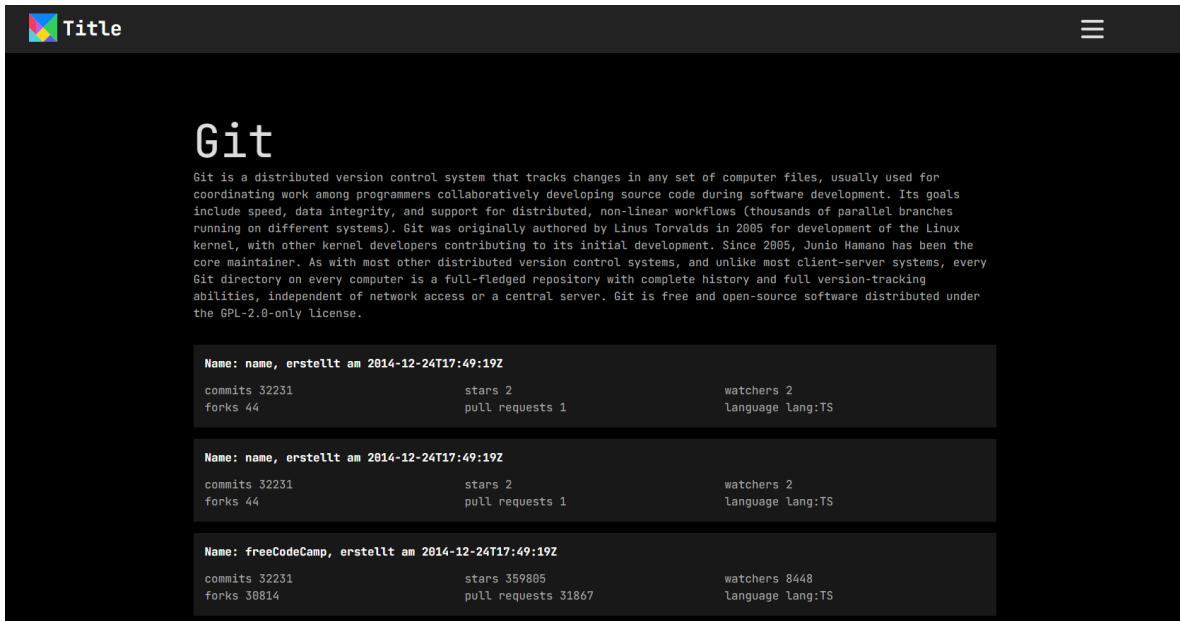


Abbildung 23: Screenshot des Clients

Die Funktion `create_feature_card` (Listing 14) erstellt eine HTML-Ansicht für eine Liste von Repository-Objekten. Sie akzeptiert eine Scope-Referenz `cx` sowie einen Vektor von Repository-Objekten namens `repos` und gibt eine HTML-Ansicht zurück, die jede Repository-Karte enthält. `Vector` ist die Rust Klasse entsprechend Listen in anderen Sprachen, `cx` ist der Reactive Architektur geschuldet. Jede Karte enthält Informationen zu dem Repository, einschließlich Name, Erstellungsdatum, Anzahl der Commits, Anzahl der Forks, Anzahl der Sterne, Anzahl der Pull Requests, Anzahl der Beobachter und der primären Programmiersprache. Die zurückgegebene View wird dann als Teil des Markups in der Indexpage (Hauptseite) gerendert (hier nicht gezeigt).

Listing 14: Markup im Framework Perseus/Sycamore in Rust

```

1 fn create_feature_card<G: Html>(cx: Scope<'_>, repos: Vec<Repository>) -> View<G>
2 {
3     let mut all = Vec::with_capacity(repos.capacity());
4     for repo in repos.clone() {
5         all.push(view! { cx,
6             div(class = "card") {
7                 div(class = "feature-title") { "Name: " (repo.name.clone()) " ",
8                     erstellt am " (repo.created_at.clone()) " }
9                 div(style= "display:flex;") {
10                     div(class = "feature-text") {
11                         p {"commits " (repo.commit_count.clone()) }
12                         p {"forks " (repo.forks_count.clone()) }
13                     }
14                     div(class = "feature-text") {
15                         p {"stars " (repo.stars_count.clone()) }
16                         p {"pull requests " (repo.pull_requests.clone()) }
17                     }
18                     div(class = "feature-text") {
19                         p {"watchers " (repo.watchers.clone()) }
20                         p {"language " (repo.primary_language.clone()) }
21                     }
22                 }
23             });
24     let markup = View:: new_fragment(all);
25     markup
26 }
```

Die Funktion `get_build_state` in Listing 15 ist eine asynchrone Funktion, die Informationen von der REST-API abruft und sie in die `IndexPageState`-Struktur „einfügt“. Die Funktion verwendet die `StateGeneratorInfo`-Struktur, um den aktuellen Status zu überwachen und zu aktualisieren (ebenfalls aufgrund der Reactive Architektur). Die Funktion baut eine URL aus einer Basis-URL und einem Port auf und sendet eine Anfrage an die REST-API. Die Antwort wird in eine JSON-Struktur gepackt und dann in die `IndexPageState`-Struktur eingefügt. Wenn ein Fehler auftritt, wird ein Standardwert zurückgegeben. Die Übergabe des Timestamp war an dieser Stelle ein Workaround um durch ein Caching zu verhindern, das der Status nicht aktualisiert wird.

Listing 15: Aufruf des Get Endpunktes für die Repositories im Client

```

1 #[engine_only_fn]
2 async fn get_build_state(
3     _info: StateGeneratorInfo<()>,
4 ) -> IndexPageState {
5
6     let base_url = env::var("REST_URL").expect("No url env. variable found");
7     let port = env::var("REST_PORT").expect("No port env. variable found");
8     let url = format!("http://{}:{}.repositories", base_url, port);
9     println!("{}", url);
10    let client = reqwest::Client::new();
11    let res = match client
12        .get(url)
13        .header(CONTENT_TYPE, "application/json")
14        .header(ACCEPT, "application/json")
15        .query(&[("timestamp", chrono::Utc::now().timestamp())])
16        .send()
17        .await {
18            Ok(resp) => resp,
19            Err(e) => {
20                return IndexPageState {
21                    response: ResponseInfo {
22                        result: vec![],
23                        status: reqwest::StatusCode::INTERNAL_SERVER_ERROR.to_string()
24                        ,
25                        time: "".to_string(),
26                    }
27                };
28            }
29        }
30    let val = match res.json().await {
31        Ok(json) => {
32            let body: Value = json;
33            body
34        },
35        Err(e) => { Value::String(String::from("")) }
36    };
37    let mut response: Vec<ResponseInfo> = serde_json::from_value(val).unwrap_or(
38        vec![]);
39    let body = Ok::<ResponseInfo, reqwest::Error>(response[0].clone()).unwrap_or(
40        ResponseInfo {
41            result: vec![],
42            status: "".to_string(),
43            time: "".to_string(),
44        });
45    IndexPageState {
46        response: body,
47    }
48}

```

3.2.2.2 REST API

Interaktion der Funktionen: `add_repository` fügt zunächst die Eigenschaften, welche dem Aufrufer nicht bekannt sein sollen, aber in der Datenbank gespeichert werden müssen, hinzu. Die aufgerufene

Funktion `create_client_post_request` stellt dann eine HTTP-POST-Anfrage an die Datenbank, um das Repository-Objekt zu speichern. Wenn die Datenbank antwortet, wird die `handle_response`-Funktion aufgerufen, um die Antwort zu verarbeiten und als JSON zurückzugeben. `create_client_post_request` gibt die Antwort an `add_repository` und damit den Aufrufer des Endpunktes zurück.

Listing 16: Programmatischer Endpunkt mit Dokumentation für Swagger (mit CodeGen)

```

1 #[utoipa::path(
2   request_body = PostRepository,
3   responses(
4     (status = 201, description = "File created successfully.", body = Vec<ResponseInfo
5       >),
6     (status = 500, description = "Internal Server Error", body = String),
7     (status = 404, description = "Files not found", body = String)))
8 #[post("/repositories")]
9 pub async fn add_repository(req: web::Json<PostRepository>) -> impl Responder {
10   let repo = Repository {
11     id: String::from(""),
12     name: String::from(&req.name),
13     created_at: format!("{}", Local::now()),
14     license: String::from(&req.license),
15     primary_language: String::from(""),
16     commit_count: 0,
17     forks_count: 0,
18     pull_requests: 0,
19     stars_count: 0,
20     watchers: 0,
21     languages_used: Vec::new(),
22   };
23
24   let base_url = env::var("BASE_URL").expect("Base URL not found as environment
25   variable");
26   let url = format!("{}/key/repository", base_url);
27   info!("URL POST: {}", url);
28   let response = create_client_post_request(repo, url).await;
29   response
30 }
```

Listing 17: Senden eines HTTP Requests (hier an die Datenbank) mit Headern

```

1 pub async fn create_client_post_request(req: Repository, url: String) -> impl
2   Responder{
3     let user = env::var("DB_USER").expect("No username env. variable found");
4     let pw = env::var("DB_PASSWORD").expect("No password env. variable found");
5     let client = reqwest::Client::new();
6     let res = client.post(url)
7       .header("Accept", "application/json")
8       .header("NS", "base")
9       .header("DB", "base")
10      .basic_auth(user, Some(pw))
11      .json(&req)
12      .send()
13      .await
14      .expect("Database not reachable.");
15     let res = handle_response(res, true).await;
16     res
17 }
```

Listing 18: Auflösung des Responses (als Option) mit Pattern Matching

```

1 async fn handle_response(res: reqwest::Response) -> impl Responder{
2   match res.status() {
3     reqwest::StatusCode::OK => {
4       match res.json().await {
5         Ok(json) => {
6           let body: Value = json;
7           HttpResponse::Created().json(body)
8         }
9       }
10     }
11   }
12 }
```

```

8         },
9         Err(e) => HttpResponse::InternalServerError()
10            .json(format!("Json parsing error: {}", e)),
11    }
12  },
13  requestCode::StatusCode::FORBIDDEN => {
14    HttpResponse::Forbidden()
15      .json("You are not allowed to access this resource")
16  },
17  other => {
18    HttpResponse::InternalServerError()
19      .json(format!("Something went wrong: {}", other))
20  },
21 }
22 }
```

The screenshot shows the Swagger UI interface. At the top, there is a list of API endpoints categorized by method and path:

- GET /repositories**
- POST /repositories**
- GET /repositories/{id}**
- PUT /repositories/{id}**
- DELETE /repositories/{id}**

Below this, under the heading "Schemas", are three defined models:

- PostRepository** (grey background)
- Repository** (light grey background)
- ResponseInfo** (light grey background)

Abbildung 24: Implementierte Ressourcen mit Routen und Datenmodellen (Swagger UI Integration)

3.2.2.3 Tests

Wir haben (außer den während der Entwicklung nebenbei durchgeführten “Tests”) keine Funktionstests durchgeführt, in der Realität wären natürlich auf allen Ebenen (Unittests, Integrationstests, End2End Tests, FlakyTests) Tests angebracht. Der interessante und laut den Anforderungen ein wesentlicher Aspekt war die Performanz, deshalb wurde diese mit einigen Tests überprüft. Dafür haben wir das HTTP benchmarking Loadtesttool wrk mit Kali Linux verwendet.

Ein weiterer wichtiger Punkt war ein geringer Footprint und gute Skalierbarkeit. Ein dafür wesentlicher Indikator ist die Größe der Images bzw. ausgeführten Container, sowie die Ressourcenallokation auf dem Server an die Services. Deshalb wurde die Größe des Containers optimiert und mit einem groben Monitoring des Servers der Ressourcenverbrauch geprüft. Die Ergebnisse sind im folgenden zu sehen und kurz erläutert.

```
(hannes@LT-6FGD373) [~]
$ wrk -t100 -c100 -d5s http://139.144.71.117:8088/repositories
Running 5s test @ http://139.144.71.117:8088/repositories
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    301.64ms  111.74ms  1.11s   84.66%
    Req/Sec    3.38      1.55     10.00   56.26%
  1582 requests in 5.10s, 10.63MB read
Requests/sec:  310.17
Transfer/sec:   2.08MB
```

Abbildung 25: Loadtest mit 100 Anfragen auf 100 Threads für 5 Sekunden an den Repositories Endpunkt, welcher 100 relativ große Json Objekte zurückgibt. Der Service war hier analog zu einem produktiven Einsatz auf einem öffentlich erreichbaren Cloudserver deployed. Durchschnittlich wurden 300 Anfragen pro Sekunde realisiert, bei einer Latenz von 300ms. Letztere ist nur im mittleren Bereich, unter Betrachtung der Anzahl der Anfragen und hinsichtlich der Serverkapazitäten (1GB RAM, 1 CPU Core) jedoch sehr gut.

```
(hannes@LT-6FGD373) [~]
$ wrk -t100 -c100 -d5s http://localhost:8088/repositories
Running 5s test @ http://localhost:8088/repositories
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    27.99ms  25.25ms 202.10ms   89.88%
    Req/Sec    40.63      31.37   131.00   75.50%
  20204 requests in 5.10s, 4.44MB read
Requests/sec:  3961.09
Transfer/sec:   0.87MB
```

Abbildung 26: Vergleich auf einem lokalen System um netzwerkbedingte Verzögerungen auszuschließen (Docker stehen auf dem System 6GB RAM und 4 physische Prozessorkerne zur Verfügung). Alle Metriken liegen ca. um den Faktor 10 höher, der Service hat nicht annähernd die ihm zugewiesenen Ressourcen beansprucht. Dies deutet auf eine sehr hohe Performanz des Services hin, etwaige Einschränkungen im ersten Test könnten auch durch Restriktionen des Cloudhosters (Requestthrottling) hervorgerufen werden.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES	SIZE
c5729c2b2ecc	hansendockedin/rust-be:latest	"/rust-actix-surreal"	8 hours ago	Up 8 hours	0.0.0.0:8088->8088/tcp, ::1:8088->8088/tcp	root_rest-api_1	0B (virtual 30MB)
893699d003ac	surrealdb/surrealdb:latest	"/surreal start -us"	8 hours ago	Up 8 hours	0.0.0.0:8000->8000/tcp, ::1:8000->8000/tcp	root_db_1	0B (virtual 50.7MB)

Abbildung 27: Footprint der Container - ein vollwertiges DBMS (Daten selbst werden auf einem im Host gemounteten Volume gespeichert bzw. In-memory) und die REST API belegen weniger als 100MB Speicherplatz. Das Image für die REST API hat inklusive der Linux Runtime eine Größe von 28MB.

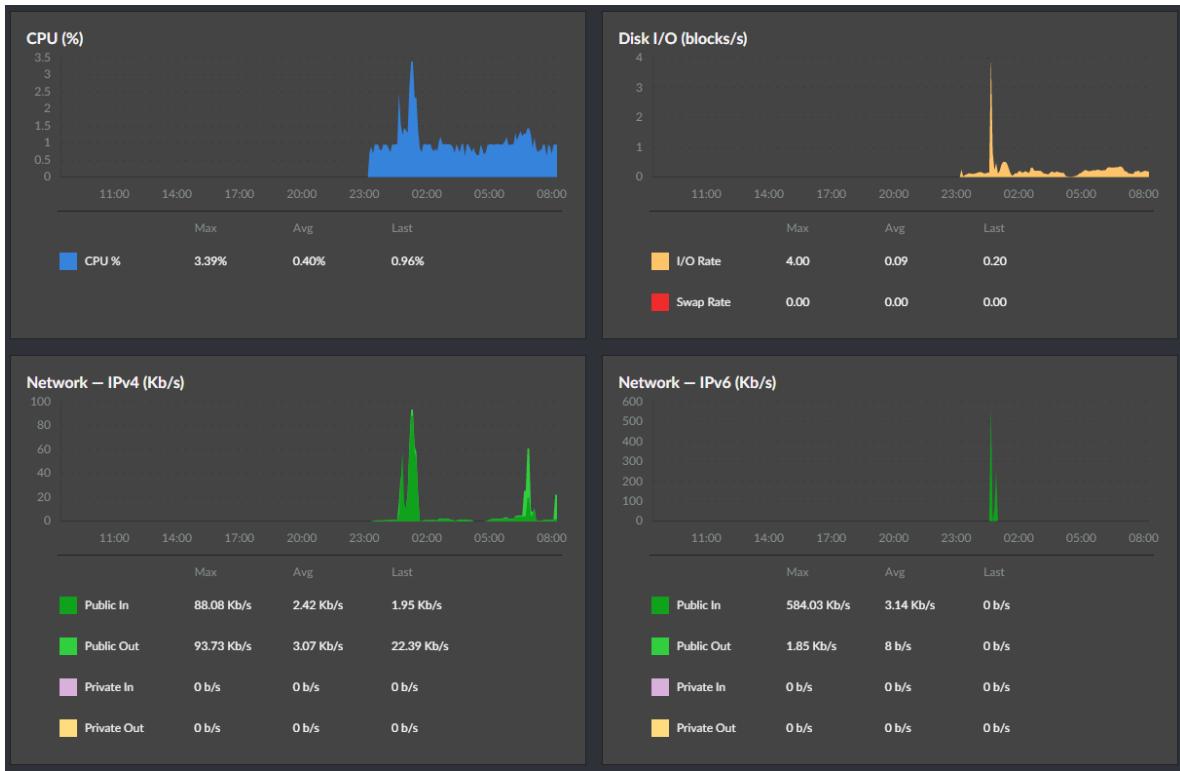


Abbildung 28: CPU Auslastung des Servers - der Spike zeigt den Zeitpunkt in welchem über Lasttests ca. 300 Anfragen pro Sekunde gestellt wurden. Sowohl im Idle (1%, als auch unter Last (3%) läuft der Service sehr effizient. Probleme durch Spalten während Idlephasen, wie sie z.B. bei Einsatz von Garbagecollectoren regelmäßig auftreten, oder der extrem hohe Overhead den z.B. ein Java Application Server verursacht, sind hier behoben.

Insbesondere die letzten beiden Tests zeigen, dass ein Deployment mit replizierbaren Containern selbst auf diesem einen, extrem kleinen Server mit bis zu 10 - 20 parallel laufenden Containern ohne Probleme möglich wäre. Kritisch ist zu betrachten, dass die Tests sehr schnell und nicht wiederholt durchgeführt wurden und deshalb nur eine begrenzte, repräsentative Aussagekraft haben.

3.2.2.3 Deployment

Für beide selbst entwickelten Services wird mit Hilfe von Dockerfiles (Listing 19) ein Image erstellt und aufs Dockerhub gepusht. In einem produktiven Szenario, insbesondere wenn der Code Closed Source ist, sollte stattdessen eine eigene Docker Registry verwendet werden. Die Schritte zur Öffentlichung sind jedoch bis auf den Host im Command docker push ... dieselben.

Listing 19: Dockerfile für die Erstellung des REST-API Images

```

1 FROM rust:1.60.0-bullseye AS build
2 WORKDIR /app
3 COPY . .
4 RUN cargo build --release
5 RUN mkdir -p /app/lib
6 RUN cp -LR $(ldd ./target/release/rust-actix-surreal-rest-api | grep ">" | cut -d
   , -f 3) /app/lib
7
8 FROM scratch AS app
9 WORKDIR /app
10 COPY --from=build /app/lib /app/lib
11 COPY --from=build /lib64/ld-linux-x86-64.so.2 /lib64/ld-linux-x86-64.so.2
12 COPY --from=build /app/target/release/rust-actix-surreal-rest-api rust-actix-
   surreal-rest-api
13 ENV LD_LIBRARY_PATH=/app/lib
14 ENTRYPOINT ["./rust-actix-surreal-rest-api"]

```

Auf dem Server, auf welchem die Services laufen sollen, muss ein Pull der Images erfolgen oder der Pfad zur Registry im docker-compose File angegeben sein, dann wird das Image automatisch bezogen. Die Ausführung von docker-compose up -d erstellt dann aus den Images die Container mit der dargestellten Konfiguration. Die virtuell erstellten Netzwerke in Docker (s. Listing 20 in Zeile 11, 18 und 31) ermöglichen eine zusätzliche Kapselung, der einzige nach außen geöffnete Port ist im Beispiel 8080. In einer produktiven Umgebung wäre hier entweder noch ein weiterer Service in Form eines Reverse Proxys (z.B. nginx oder traefik) vorhanden, welcher über Port 443 erreichbar ist und über LetsEncrypt ein Zertifikat bezieht. Alternativ könnte auf dem Server direkt ein nginx Webserver bereitgestellt werden, Hauptsache die über HTTP erreichbaren Services sind in einem gekapselten Netzwerk nur über die Weiterleitung der Anfragen des Reverse Proxys erreichbar.

Listing 20: docker-compose.yml zur Bereitstellung des kompletten Stacks

```

1 version : '3.8'
2 services:
3   db:
4     image: surrealdb/surrealdb:latest
5     restart: always
6     command: start --user root --pass root memory
7     expose:
8       - 8000
9     volumes:
10    - db:/var/lib/surrealdb/data
11   networks:
12     - backend
13
14   rest-api:
15     image: rust-actix-surreal-rest-api
16     expose:
17       - 8088
18     networks:
19       - backend
20       - frontend
21     depends_on:
22       - db
23     environment:
24       - BASE_URL=http://db
25       - CORS_ALLOW=http://localhost:8080
26
27   client:
28     image: rust-client
29     ports:
30       - '8080:8080'
31     networks:
32       - frontend
33     depends_on:
34       - rest-api
35     environment:
36       - SERVICE_URL=http://rest-api
37
38 networks:
39   backend:

```

Die dargestellte Form des Deployments ermöglicht eine sehr schnelle Aktualisierung der Services. Der Veröffentlichung des neuen Images würde i.d.R. natürlich ein umfangreiches, automatisiertes Testing vorausgehen, das Image selbst ist dann das Artefakt. Die erneute Ausführung von docker-compose up -d würde dann ausschließlich die Container neu starten, für welche Änderungen der Images festgestellt wurden. Dies dauert maximal einige Sekunden. Um auch dies zu vermeiden wäre es mit wenigen Zeilen zusätzlicher Konfiguration möglich, die Container zu replizieren und nach Terminierung der Verbindung im Reverse Proxy dynamisch die Last zu verteilen. Der Reverse Proxy hat dann dementsprechend gleichzeitig die Funktion eines Loadbalancers.

Um den Rahmen nicht zu sprengen, haben wir kein Monitoring und Remote Logging realisiert, auch dies wäre jedoch durch die Nutzung von Docker einfach umzusetzen. Images, z.B. für den oft verwendeten ELK Stack oder alternativ die Kombination von Grafana und Prometheus, sind vorhanden und mit wenigen Anpassungen als weitere Services innerhalb der docker-compose File einsetzbar. Des weiteren

würden die Services in einem produktiven Umfeld als Cluster auf physisch getrennten Systemen laufen. Das Deployment kann auf jedem beliebigen Linuxserver erfolgen, auf dem Docker installiert ist. In unserem Fall haben wir Linode (Akamai) als IAAS Anbieter ausgewählt und die Anwendung auf einem Alpine Server mit 1GB RAM bereitgestellt.

3.2.3 Anbindung Datenbank (Hannes)

Listing 21: CLI Kommandos zur lokalen Installation der Datenbank für Windows Linux und macOS

```
1 iwr https://windows.surrealdb.com -useb | iex
2 curl -sSf https://install.surrealdb.com | sh
3 brew install surrealdb/tap/surreal
```

Listing 22: CLI Kommando zur Übertragung der Daten aus der Datei in Listing 23

```
1 cat schemashort.sql | surreal sql --conn http://localhost:8000 --user root --pass
root --ns base --db base
```

Listing 23: Ausschnitt der sql Setupdatei

```
1 INSERT INTO repository (name, stars_count, forks_count, watchers, pull_requests,
    primary_language, languages_used, commit_count, created_at, licence) VALUES ('  

        react', 159266, 30464, 8497, 2911, lang:JavaScript, [lang:JavaScript, lang:  

        HTML, lang:CSS], 5562, '2013-05-24T16:15:54Z', 'MIT License');
2 INSERT INTO repository (name, stars_count, forks_count, watchers, pull_requests,
    primary_language, languages_used, commit_count, created_at, licence) VALUES ('  

        scikit-learn', 38327, 18225, 4968, 1701, lang:Python, [lang:Python, lang:  

        Cython, lang:HTML, lang:CSS], 4085, '2010-01-10T09:58:52Z', 'BSD-3-Clause  

        License');
3 INSERT INTO repository (name, stars_count, forks_count, watchers, pull_requests,
    primary_language, languages_used, commit_count, created_at, licence) VALUES ('  

        angular', 68521, 24536, 6779, 2197, lang:TypeScript, [lang:TypeScript, lang:  

        JavaScript, lang:HTML, lang:CSS], 4248, '2014-09-18T16:12:01Z', 'MIT License')
```

4 Übung 4d: Sicherheit von Web APIs

4.1 Sicherheitsrisiken in Verbindung mit dem HTTP Protokoll

4.1.1 HTTPS und TLS

HTTP Eigenschaften:

- ist das Standardprotokoll für die Internetkommunikation
- arbeitet nach dem Client Server Modell
- nutzt TCP
- hat zwei Typen: non-persistent (weniger Overhead, einmalige Verbindung, wird nicht aufrechterhalten) und persistent (Verbindung wird nach Aufbau aufrechterhalten, s. ??)
- funktioniert über Request und Response Message
- Benutzer- und Serverstatus werden über Cookies aufrechterhalten
- Webcache kann Geschwindigkeit erhöhen (lokal im Browser oder serverseitig auf Proxy), Response ist 304 (conditional GET)
- HTTPS (S steht für secure) ist eine Erweiterung von HTTP

TLS Eigenschaften: (Wiki)

- Nachfolger von SSL, steht für Transport Layer Security
- Verbindung zu HTTPS: kommt im TCP/IP Stack zwischen Transport und Anwendungsebene zum Einsatz und wird i.d.R. zusätzlich zum TCP Protokoll eingesetzt.
- ermöglicht eine Ende zu Ende Verschlüsselung von Data in Transit (und ist deshalb z.B. bei Mails durch zusätzliche Teilnehmer zwischen den “Enden” nur eingeschränkt sicher)
- Im TLS Handshake findet ein sicherer Schlüsselaustausch und eine Authentifizierung statt.
- Für den Schlüsselaustausch sind in den älteren TLS-Versionen verschiedene Algorithmen mit unterschiedlichen Sicherheitsgarantien im Einsatz. Die neueste Version TLS 1.3 verwendet allerdings nur noch das Diffie-Hellman-Schlüsselaustausch Protokoll (DHE oder ECDHE) auf Basis elliptischer Kurven.
- Dabei wird für jede Verbindung ein neuer Sitzungsschlüssel (Session Key) ausgehandelt. Da dies ohne Verwendung eines Langzeitschlüssels geschieht, erreicht TLS 1.3 Perfect Forward Secrecy.
- Alle TLS-Handshakes verwenden eine asymmetrische Kryptographie (öffentlicher und privater Schlüssel), aber nicht alle nutzen den privaten Schlüssel beim Generieren von Sitzungsschlüsseln.

Vorteile TLS (1.3): TLS 1.3 hat die Unterstützung für ältere, weniger sichere kryptografische Features eingestellt und unter anderem TLS-Handshakes schneller gemacht. Die Hauptvorteile von TLS 1.3 gegenüber TLS 1.2 sind schnellere Geschwindigkeiten und verbesserte Sicherheit. TLS und verschlüsselte Verbindungen erzeugen naturgemäß einen Overhead bei der Übertragung. HTTP / 2 hat bei diesem Problem durch die Verringerung der Schritte beim Aufbau der TCP Verbindung geholfen, aber TLS 1.3 beschleunigt verschlüsselte Verbindungen durch Funktionen wie TLS false start und Zero Round Trip Time (0-RTT) noch weiter. Die Einführung elliptischer Kurven verbessert zudem bei gleicher Schlüssellänge die Sicherheit und vermeidet beispielsweise Angriffe wie LogJam, die auf dem Number field sieve Algorithmus basieren (welcher die Tatsache ausnutzt, dass immer dieselbe Primzahl verwendet wird). Auch export-grade Funktionalitäten (welche demselben Angriff zugrunde lagen) sind in TLS 1.3 nicht mehr eingebaut.

4.1.2 Authentifizierungsmöglichkeiten HTTP(S)

Es gibt mehrere Möglichkeiten, Benutzer (Clients) im Rahmen des HTTP-Protokolls zu authentifizieren. Verbreitet sind (Wiki, ssl.com):

- Basic Authentication: Die Basic Authentication (Basisauthentifizierung) wird seit 2015 durch RFC 7617 spezifiziert und ist eine häufig verwendete Art der HTTP-Authentifizierung. Der Webserver fordert mit Eingabe von Benutzername und Passwort eine Authentifizierung an. Ein Vorteil der Basic Authentication ist ihre Einfachheit in der Implementierung. Ein Nachteil ist, dass die Anmeldeinformationen im Klartext übertragen werden und daher leicht abgefangen werden können. Deshalb sollte diese Methode nur für den Hobbybereich eingesetzt werden.
- Digest Access Authentication: Die Hashwertauthentifizierung ist ein Verfahren, das die Basic-Authentifizierung ersetzen soll(te). Der Server sendet eine Zeichenfolge zufälliger Daten, auch Nonce genannt, als Challenge an den Client. Der Client reagiert mit einem Hash, der neben anderen Informationen den Benutzernamen, das Kennwort und die Nonce enthält. Die Digest Access Authentication bietet mehr Sicherheit als die Basic Authentication, da sie einen Hash verwendet und somit die Anmeldeinformationen nicht im Klartext übertragen werden.

Weitere Verfahren, die im Zusammenhang mit HTTPS eingesetzt werden, sind nicht den direkten Verfahren zuzurechnen, weil sie alle eine dritte Instanz hinzuziehen müssen (Validierung Zertifikate über PKI, Authorisierungs- bzw. Authentifizierungsprüfung über Drittanbieter bei OAuth2 und OIDC).

4.1.3 Cookies

4.1.3.1 Begriff

Cookies sind kleine Textdateien, die von einer Webseite im Internetbrowser eines Nutzers gespeichert werden können. Sie dienen dazu, Informationen über den Nutzer und seine Interaktionen mit der Webseite zu speichern.

Cookies können nützlich sein, indem sie beispielsweise Einstellungen im Webbrowser abspeichern oder dafür sorgen, dass ein Warenkorb beim Online-Shopping zu einem späteren Zeitpunkt wieder aufgerufen werden kann. Sie können auch dazu verwendet werden, das Surfverhalten von Nutzern im Internet über einen längeren Zeitraum zu verfolgen und detaillierte Nutzerprofile anzulegen (was eher aus Sicht des Betreibers ein Vorteil ist...).

Grundsätzlich werden zwei Arten von Cookies unterschieden: technisch notwendige und technisch nicht notwendige Cookies. Technisch notwendige Cookies sind für das Funktionieren der Webseite notwendig, während technisch nicht notwendige Cookies für Zwecke wie das Verfolgen des Surfverhaltens verwendet werden können.

Es ist zwar möglich, eine Webseite ohne Cookies zu betreiben, allerdings kann dies zu Einschränkungen in der Funktionalität führen. Beispielsweise müssten Nutzer bei jedem Besuch der Webseite erneut ihre Einstellungen vornehmen oder sich erneut anmelden. Sofern jedoch eine Persistierung von Nutzerdaten nicht notwendig ist, z.B. bei statischen, rein informativen Seiten, kann auf Cookies verzichtet werden (was oft nicht passiert, weil durch den Einsatz von Tracking, eingebetteten Webfonts usw. auch auf solchen Seiten, Cookies gesetzt werden). Dabei sollte jedoch zwischen tendenziell unkritischen Sessioncookies, die lokal und verschlüsselt gespeichert werden und Trackingcookies, welche übertragen werden, unterschieden werden.

Ein großer Nachteil von Cookies ist mit ihnen einhergehende Sicherheitsrisiko. Da sie Informationen über den Nutzer und seine Interaktionen mit der Webseite speichern, können sie von Dritten abgefangen und missbraucht werden.

4.1.3.2 Sicherheitsrisiken

Zwei häufige, in Verbindung mit Cookies durchgeführte Angriffe sind Cookie Poising und XSS (Cross Site Scripting). Diese sollen hier kurz erläutert werden.

Cookie Poisoning ist eine Art von Angriff, bei dem ein Angreifer versucht, die Daten in einem Cookie zu manipulieren. Er kann versuchen, die Daten in einem Cookie zu ändern oder einen gefälschten Cookie zu erstellen, um eine bösartige Aktion auszulösen. Beispielsweise könnte ein Angreifer ein Cookie

mit einer falschen Sitzungs-ID erstellen und sich so Zugang zu einer geschützten Ressource (Passwort) verschaffen. Ein weiteres Beispiel wäre, ein Cookie mit einem bösartigen Skript zu injizieren, das beim Öffnen der Seite ausgeführt wird und den Browser des Benutzers kompromittiert (Weiterleitung, Keylogging).

Bei XSS versucht der Angreifer, schädlichen Code in eine Website einzuschleusen (ebenfalls um sensible Informationen von Benutzern zu stehlen oder die Website zu kompromittieren) - hierbei wird oft auch versucht Cookies abzugreifen, weil sie oftmals sensible Daten enthalten. Der Angreifer nutzt Schwachstellen in der Website aus, um eine bösartige Skriptzeile in der Website einzufügen, die dann von anderen Benutzern ausgeführt wird, wenn sie die Website besuchen (z.B. in einem Formular).

4.2 Möglichkeiten zur Risikominderung

4.2.1 OWASP

Dem Ranking liegt das OWASP API Security Threat Model zugrunde.¹³ Es ist eine Methodologie zur Identifizierung und Bewertung von Bedrohungen für APIs. Es folgt einem strukturierten Ansatz, der aus vier Schritten besteht:

1. API-Beschreibung: Hier werden die API-Funktionen und Datenobjekte beschrieben.
2. Bedrohungsmodellierung: In diesem Schritt werden potenzielle Bedrohungen identifiziert und priorisiert. Die Bedrohungsmodelle werden auf der Basis der API-Beschreibung entwickelt.
3. Risikobewertung: Die Bedrohungsmodelle werden bewertet, um diejenigen mit dem höchsten Risiko zu identifizieren.
4. Empfehlungen: Basierend auf den identifizierten Bedrohungen und Risiken werden Empfehlungen für Sicherheitsmaßnahmen abgeleitet, um das Risiko zu minimieren.

Die OWASP klassifiziert die Bedrohungen folgendermaßen (mehr Punkte = höhere Gefahr):

Punkte	Exploitability (Ausnutzbarkeit)	Weakness Prevalence (Häufigkeit)	Weakness Detectability (Aufspürbarkeit)	Technical Impact (Auswirkungen)
3	Easy	Widespread	Easy	Severe
2	Average	Common	Average	Moderate
1	Difficult	Difficult	Difficult	Minor

Tabelle 9: OWASP Thread Model, allgemeine Form

Die Methodologie soll helfen, API-Designs zu verbessern und Entwicklern zu helfen, sicherere APIs zu erstellen. Für die ausgewählten Lücken geben wir zunächst die (aufgrund der technischen Fachtermini im englischen belassene) Beschreibung der OWASP Foundation an. Anschließend zeigen wir je anhand eines negativen und positiven Beispiels, wie das Ausnutzen der Lücke eingeschränkt werden kann. Dabei wird im jeweiligen Beispiel nur die entsprechende Schwachstelle beachtet, d.h. andere Teile im Beispiel können hinsichtlich weiterer Schwachstellen durchaus anfällig sein. Zudem vereinfachen die Beispiele die oftmals komplexen Problematiken natürlich.

¹³Vgl. OWASP (2019)

4.2.1.1 Ausgewählte Schwachstellen (5)

Vulnerability	Exploitability (Ausnutzbarkeit)	Weakness Prevalence (Häufigkeit)	Weakness Detectability (Aufspürbarkeit)	Technical Impact (Auswirkungen)
Broken Object Level Authorization	3	3	2	3
Excessive Data Exposure	3	2	2	2
Broken Function Level Authorization	3	2	1	2
Injection	3	2	3	3
Insufficient Logging & Monitoring	2	3	1	2

Tabelle 10: OWASP Thread Model für APIs für die 5 ausgewählten Sicherheitslücken

4.2.1.1.1 Broken Object Level Authorization

APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.

Das erste Beispiel (Listing 24) stellt ein Sicherheitsrisiko dar, weil keine Überprüfung der Zugriffsberechtigungen für den Abruf einer bestimmten Ressource aus der Datenbank vorgenommen wird. Jeder Benutzer, der auf diese Route zugreift, kann alle Elemente abrufen, unabhängig von den Zugriffsberechtigungen.

Listing 24: Negativbeispiel Broken Object Level Authorization

```

1 [HttpGet("items/{id}")]
2 public Task<IActionResult<Item>> GetItem(int id)
3 {
4     var item = await _context.Items
5         .Where(i => i.Id == id)
6         .SingleOrDefaultAsync();
7     return Ok(item ?? NotFound());
8 }
```

Das zweite Beispiel (Listing 25) verbessert die Sicherheit, da überprüft wird, ob der angemeldete Benutzer (current user) das Eigentümer-Attribut des Elements besitzt, bevor das Element zurückgegeben wird. Auf diese Weise wird sichergestellt, dass nur der Eigentümer des Elements darauf zugreifen kann und andere Benutzer keinen Zugriff darauf haben.

Listing 25: Positivbeispiel Broken Object Level Authorization

```

1 [HttpGet("items/{id}")]
2 public async Task<IActionResult<Item>> GetItem(int id)
3 {
4     var currentUser = HttpContext.User;
5     var item = await _context.Items
```

```

7     .Where(i => i.Id == id && i.OwnerId == currentUser.FindFirstValue(
8         ClaimTypes.NameIdentifier))
9     .SingleOrDefaultAsync();
10
11 }  


```

4.2.1.1.2 Excessive Data Exposure

Looking forward to generic implementations, developers tend to expose all object properties without considering their individual sensitivity, relying on clients to perform the data filtering before displaying it to the user.

In diesem Beispiel (Listing 26) enthält der Response zum Abrufen eines Order-Objekts das Kundenobjekt als Property. Es mag Gründe für ein solches Datenmodell geben, in dem Fall werden aber ohne Einschränkung durch die direkte Rückgabe alle Informationen preisgegeben. Eine Erweiterung des Objektes um weitere Eigenschaften durch einen Entwickler würde über diese Route die neue Eigenschaft ungeprüft mit zurückgeben. Wenn ein Angreifer auf diesen Endpunkt zugreifen würde, könnte er diese sensiblen Informationen abrufen, obwohl sie für die Erfüllung der Anfrage nicht erforderlich sind.

Listing 26: Negativbeispiel Excessive Data Exposure

```

1 [HttpGet("order/{id}")]
2 public IActionResult<Order> GetOrder(int id)
3 {
4     var order = _context.Orders.Include(o => o.Customer).FirstOrDefault(o => o.Id
5         == id);
6     return Ok(order);
7 }
8 public record Order(int Id, int Total, DateTime Date, int CustomerId)

```

In diesem Beispiel (Listing 27) gibt der API-Endpunkt nur die für die Erfüllung der Anforderung erforderlichen Informationen zurück (d. h. die Auftragskennung, den Gesamtbetrag und das Datum). Das Response Model entspricht nicht dem Rückgabeobjekt von `_context.Orders` und die notwendigen Properties werden auf den response gemappt. Das o.g. Problem, z.B. bei einer Erweiterung, tritt nicht auf und neue Property müsste explizit auch dem Response Model hinzugefügt werden und zusätzlich gemappt werden.

Listing 27: Positivbeispiel Excessive Data Exposure

```

1
2 [HttpGet("orders/{id}")]
3 public IActionResult<OrderResponse> GetOrder(int id)
4 {
5     var order = _context.Orders.FirstOrDefault(o => o.Id == id);
6
7     if (order == null)
8     {
9         return NotFound();
10    }
11
12     OrderResponse response = new {
13         Id = order.Id,
14         Total = order.Total,
15         Date = order.Date
16     };
17
18     return Ok(response);
19 }
20 public record Order(int Id, int Total, DateTime Date, int CustomerId)
21 public record OrderResponse(int Id, int Total, DateTime Date)

```

4.2.1.1.3 Broken Function Level Authorization

Complex access control policies with different hierarchies, groups, and roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws. By exploiting these issues, attackers gain access to other users' resources and/or administrative functions.

Die folgenden beiden Beispiele (Listings 28 und 29) unterscheiden sich lediglich in der Annotation in der zweiten Methode, welche vorgibt, welche Rollen berechtigt sind, den Endpunkt überhaupt aufzurufen. Die Überprüfung innerhalb der Methode wäre bei Implementierung der Standardlibrary von .NET gar nicht nötig und ist hier nur zur Verdeutlichung eingefügt. Die erste Methode könnte hingegen über die entsprechende Route von jedem Nutzer aufgerufen werden und die entsprechenden Informationen über den zurückgegebenen User unabhängig der Autorisierung des Aufrufers eingesehen und verwendet werden.

Listing 28: Negativbeispiel Broken Function Level Authorization

```
1 [HttpGet("orders/{id}")]
2 public IActionResult<Order> GetOrder(int id)
3 {
4     Order order = _context.Orders.Find(id);
5     return Ok(order ?? NotFound());
6 }
```

Listing 29: Positivbeispiel Broken Function Level Authorization

```
1 [HttpGet("orders/{id}")]
2 [Authorize(Roles = "admin, manager")]
3 public IActionResult<Order> GetOrder(int id)
4 {
5     var currentUser = HttpContext.User;
6     if (!currentUser.IsInRole("admin") || !currentUser.IsInRole("manager"))
7     {
8         return Forbid();
9     }
10    Order order = _context.Orders.Find(id);
11    return Ok(order ?? NotFound());
12 }
```

4.2.1.1.4 Injection (hier SQL)

Injection flaws, such as SQL, NoSQL, Command Injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

Das negative Beispiel (Listing 30) weist eine SQL-Injection-Vulnerability auf, da der Wert der Variable "id" direkt in die SQL-Abfrage eingefügt (injected) wird. Ein Angreifer kann die Eingabe manipulieren, um die Abfrage zu verändern und unautorisierten Zugriff auf die Datenbank zu erlangen oder schädlichen Code einzufügen.

Listing 30: Negativbeispiel Injection

```
1 [HttpGet("users/{id}")]
2 public IActionResult<User> GetUser(string id)
3 {
4     string query = $"SELECT * FROM Users WHERE Id = '{id}'";
5     // ...
6 }
```

Im Gegensatz dazu verwendet das positive Beispiel in Listing ?? parameterisierte Abfragen und schützt damit gegen SQL-Injection-Attacken, da der Wert von id als Parameter an die Abfrage übergeben wird, anstatt direkt in die Abfrage eingefügt zu werden. Als zusätzliche Sicherheitsmaßnahme wird statt einer id eine GUID verwendet, welche im richtigen Format vorliegen muss und es bei der Abfrage erschwert, id's zu „erraten“ (was Security by Obscurity wäre - also für sich keinerlei Sicherheit bietet, in Kombination jedoch sinnvoll ist).

Listing 31: Positivbeispiel Injection

```

1 [HttpGet("users/{id}")]
2 public IActionResult GetUser(Guid id)
3 {
4     string query = "SELECT * FROM Users WHERE Id = @id";
5     var command = new SqlCommand(query, connection);
6     command.Parameters.AddWithValue("@id", id);
7     // ...
8 }
```

4.2.1.1.5 Insufficient Logging & Monitoring

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems to tamper with, extract, or destroy data. Most breach studies demonstrate the time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

Wir greifen die Problematik der SQL-Injection auf - in diesem Beispiel (Listing 32) wird durch Parametrisierung versucht, einen SQL-Injection-Angriff zu verhindern. Allerdings wird im Falle eines Fehlers beim Ausführen des SQL-Statements nur eine allgemeine Fehlermeldung an den Client zurückgegeben. Es gibt keine weiteren Maßnahmen zur Überwachung oder Protokollierung des Vorfalls.

Listing 32: Negativbeispiel Improper Assets Management

```

1 [HttpGet("users/{id}")]
2 public IActionResult GetUser(string id)
3 {
4     try
5     {
6         string query = "SELECT * FROM Users WHERE Id = @id";
7         var command = new SqlCommand(query, connection);
8         command.Parameters.AddWithValue("@id", id);
9         // ...
10    }
11    catch (Exception ex)
12    {
13        _logger.LogError(ex, "Error occurred while getting user with ID {id}", id);
14    }
15 }
16 }
```

In positiven Beispiel (Listing 34) wird ebenfalls versucht, einen SQL-Injection-Angriff zu verhindern, indem ein parametrisiertes SQL-Statement verwendet wird. Wenn jedoch ein Fehler beim Ausführen des Statements auftritt, wird der Fehler sowohl protokolliert als auch an einen Remote-Logger gesendet, der in das mit einem Monitoring Tool verbundenen Sink schreibt. Je nach Incidence oder bei wiederholten Vorfällen kann dann entsprechend (automatisch) reagiert werden, um möglicherweise eine weitere Attacke zu verhindern (z.B. durch ein Blacklisting der IP).

Listing 33: Positivbeispiel Improper Assets Management

```

1 [HttpGet("users/{id}")]
2 public IActionResult GetUser(string id)
3 {
4     try
5     {
6         string query = "SELECT * FROM Users WHERE Id = @id";
7         var command = new SqlCommand(query, connection);
8         command.Parameters.AddWithValue("@id", id);
9         // ...
10    }
11    catch (SqlException ex)
12    {
13        _logger.LogError(ex, "Error occurred while getting user with ID {id}", id);
14        _monitoringService.LogEvent(new EventLog {
15            EventType = "Security",
16            EventLevel = "Warning",
17            EventMessage = "Failed attempt to get user with ID " + id
18        });
19        return StatusCode(500);
20    }
21    catch (Exception ex)
22    {
23        _logger.LogError(ex, "Unknown error occurred while getting user with ID {id}", id);
24        _monitoringService.LogEvent(new EventLog {
25            EventType = "Common",
26            EventLevel = "Error",
27            EventMessage = "Unknown error occurred while getting user with ID " +
28            id
29        });
30        return StatusCode(500);
31    }
}

```

4.2.1.2 Zwei ausgewählte Testmöglichkeiten (Hannes)

OWASP ZAP, Burp Suite, Postman Interceptor, Wireshark injection: sqlmap, XSSStrike, SQLNinja
ELK Stack

4.2.2 OAuth 2 und OIDC (Danny)

OAuth2 ist ein Protokoll, das zur Autorisierung von Zugriffen auf geschützte Ressourcen verwendet wird. Es ermöglicht autorisierten Zugriff auf diese Ressourcen, ohne dass Benutzer ihre Anmeldeinformationen direkt an Dritte weitergeben müssen. Dazu wurden die vier Rollen Resource Owner, Authorisation Server, Client und Resource Server definiert, welche im Folgenden erläutert werden.

- Der Resource Owner ist der Besitzer der zu schützenden Resource. Dies kann ein Endbenutzer oder eine Organisation sein. Er hat die Kontrolle darüber, welche Ressourcen geschützt werden und wer darauf zugreifen darf. Der Resource Owner muss in der Lage sein, autorisierten Zugriff auf die Ressourcen zu gewähren, entweder durch direkte Interaktion mit dem Authorisation Server oder durch Delegierung an einen Client, der im Namen des Resource Owners handelt.
- Der Authorisation Server ist der Server, der die Authentifizierung und Autorisierung von Clients verwaltet. Er ist dafür verantwortlich, die Identität des Benutzers zu überprüfen und sicherzustellen, dass der Client nur auf diejenigen Ressourcen zugreift, auf die er zugreifen darf. Dazu wird ein Access Token generiert, welches dann vom Client verwendet wird, um auf die geschützten Ressourcen zuzugreifen.
- Der Client ist die Anwendung, die auf die geschützten Ressourcen zugreifen möchte. Der Client muss sich beim Authorization Server anmelden und das Access Token anfordern, das zur Authentifizierung bei der API verwendet wird.

- Der Resource Server ist der Server, auf dem die geschützten Ressourcen gespeichert sind. Er ist für den Schutz und die Bereitstellung dieser Ressourcen zuständig und prüft das Access Token, das vom Client gesendet wird, um sicherzustellen, dass der Client berechtigt ist, auf die Ressourcen zuzugreifen. Wenn das Access Token gültig ist, gewährt der Resource Server dem Client den Zugriff auf die angeforderten Ressourcen.

OpenID Connect (OICD) und Open Authorisation (OAuth) sind zwei verwandte Protokolle, die häufig gemeinsam verwendet werden, um die Authentifizierung und Autorisierung von Benutzern in Webanwendungen zu verwalten. Die Autorisierung des Clients zur zum Zugriff auf eine geschützte Resource ist dabei im Zuständigkeitsbereich von OAuth2. OpenID ist für die Autentifizierung des Clients zuständig. Die Anwendungsfälle dieser beiden Protokolle sind folglich völlig verschieden[1]. Ein typischer Anwendungsfall für OAuth ist, dass ein Nutzer Daten von Webservice A beim Webservice B nutzen möchte. Der Nutzer muss Webservice B also Zugriff auf Webservice A geben, möchte allerdings nicht seine Anmeldeinformationen an Webservice B weitergeben. Mit OAuth2 kann der Nutzer, welcher den Resource Owner darstellt, dem Webservice B (in diesem Fall der Client) mithilfe eines Authorisation Servers Zugriff auf Daten des Webservices A gewähren. Webservice A stellt hierbei den Resource Server dar. Wird dieses Beispiel nun um die Verwendung von OICD erweitert, würde sich der Benutzer sich bei Webservice B über OICD anmelden und dann auf die Ressourcen von Webservice A zugreifen können, ohne dass er seine Anmeldeinformationen direkt an Webservice B weitergeben muss. Wie OICD funktioniert kann der Abbildung 29 entnommen werden.

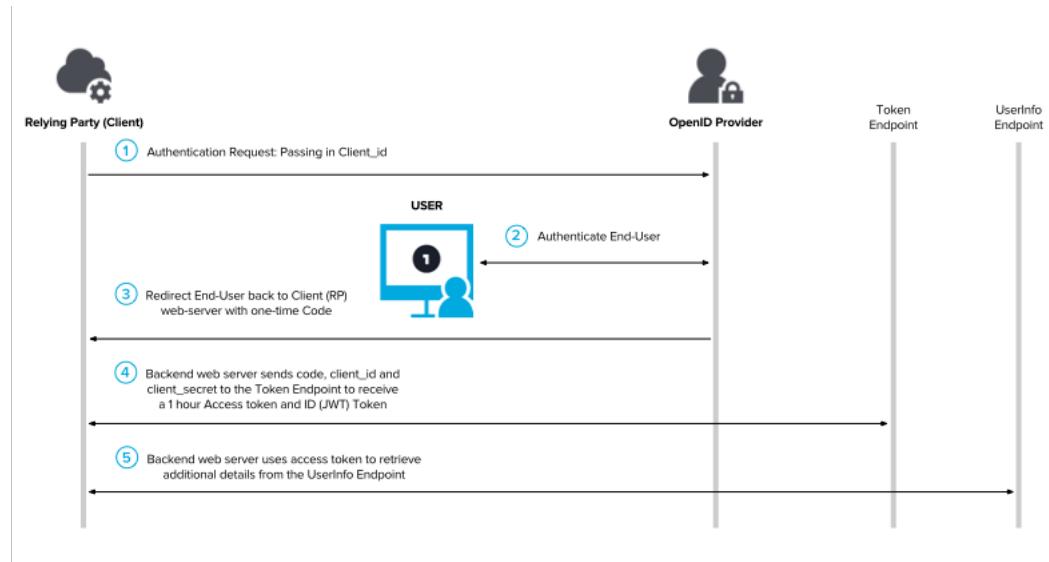


Abbildung 29

4.3 Praktische Anwendung von OAuth2

4.3.1 Testwerkzeuge

Da nicht ganz klar ist, was explizite "Testwerkzeuge" für OAuth2 sein sollen, was überhaupt getestet werden soll und es vermutlich eher um die "Funktionsaufrufe" geht (gemeint sind wohl die Operationen für REST Requests?) stellen wir hier anhand von zwei OAuth Flows den Ablauf zum Abruf eines AccessTokens dar. Die Beispiele sind vom Playground von <https://www.oauth.com> entnommen. Das "Tool" zum Aufrufen (sei es nun curl, Postman, die Aufruffunktion für HTTP wie sie inzwischen in fast jeder IDE integriert ist, oder was auch immer) ist dabei nebensächlich, wir beschränken uns also auf den Ablauf, die Requests und Responses.

4.3.1.1 Authorization Code Flow

Bevor die Autorisierung beginnt, wird zunächst eine zufällige Zeichenfolge für den Parameter `state` erzeugt. Der Client (läuft unter `oauth.com`) muss diese Zeichenkette speichern, um sie im nächsten Schritt zu verwenden. Anschließend stellt er folgende Anfrage an den Authorization Server (GET):

```
https://authorization-server.com/authorize?  
response_type=code  
&client_id=Oh_FuJbgdnMBNCrBTonIIffFr  
&redirect_uri=https://www.oauth.com/playground/authorization-code.html  
&scope=photo+offline_access  
&state=PmmmiQakBxCiZpVS
```

Nun erfolgt die Weiterleitung und der Nutzer wird aufgefordert sich einzuloggen (üblicherweise mit seinem Google, Facebook, Apple, Github oder Microsoft Account, je nachdem welche OAuth2 Anbieter eingebunden wurden).



Abbildung 30

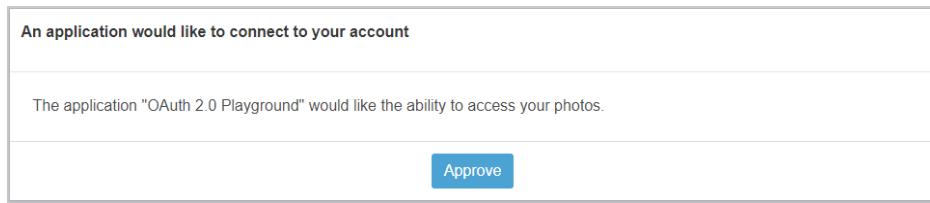


Abbildung 31

Der Nutzer wird an den Client zurückgeleitet (welcher in der Konfig beim OAuth Anbieter angegeben werden muss), im Beispiel über folgende URL: `https://www.oauth.com/playground/authorization-code.html?state=PmmmiQakBxCiZpVS&code=NyA5xLq-xr8a3l8U57nTY31ILXrpEeSn3uhFipgF1bE6tN5D`. Wichtig ist hier ein Vergleich des States (Parameter `state=PmmmiQakBxCiZpVS`) - stimmen die states nicht überein ist eine mögliche Cross-site request forgery (CSRF) Attacke der Grund. Dies wird jedoch in aller Regel automatisch von Bibliotheken übernommen wenn OAuth2 implementiert wird. Der zweite wichtige Parameter ist der Authorisierungscode. Dieser wird nun in einem POST geschickt:

```
https://authorization-server.com/token?  
grant_type=authorization_code  
&client_id=Oh_FuJbgdnMBNCrBTonIIffFr  
&client_secret=VjRYnxDo14LGxfN9ZTAHNTv78CVX-wXSkrjneu3KdtvdbJZ5  
&redirect_uri=https://www.oauth.com/playground/authorization-code.html  
&code=E3qtUPB9NyiZIYnbGiyC2udH-7FWjQp_9SoWG1jLiMPT47Kz
```

Statt wie hier im Body werden die Credentials auch oft im Header verschickt. Dies hängt von der Konfiguration des Authorizationservers ab. Der Server schickt bei Erfolg nun das Accesstoken zurück, welches vom Client für alle weiteren Validierungen verwendet werden kann:

```
{
  "token_type": "Bearer",
  "expires_in": 86400,
  "access_token": "ÖD3iBGptBchoMh6ZIVgxnrfgQr0sptq0_5zJVy8kt7UB2CV3tBoQu3xp1vSmhO32cezsY1xK",
  "scope": "photo offline_access",
  "refresh_token": "qzwneRoYmySW7AAb258kucz9"
}
```

4.3.1.2 OIDC Flow

Der erste Aufruf unterscheidet sich nur durch einen zusätzlichen Parameter, die Nonce (Number used once). Zudem sind die Scopes andere, weil die Authorisierung nicht direkt erfolgt, sondern über die Authentifizierung mit OIDC die verfügbaren Scopes für OAuth erst abgefragt werden müssen.

```
https://authorization-server.com/authorize?
response_type=code
&client_id=Oh_FuJbgdnMBNCrBTonIIffFr
&redirect_uri=https://www.oauth.com/playground/oidc.html
&scope=openid+profile+email+photos
&state=TCklNPN2HH78XVc2
&nonce=cJSfg33Gp7zS0tT3
```

Das anschließende Fenster (Login “On behalf of”) hat wohl jeder schonmal gesehen. Hier werden auch nochmal die Berechtigungen benannt, die den Scopes des Requests entsprechen.

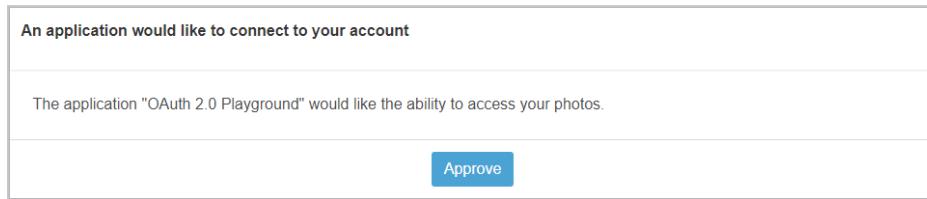


Abbildung 32

Der nun folgende POST Request entspricht exakt dem im vorigen Beispiel und wird deshalb nicht wiederholt. Der Response enthält nun allerdings zusätzlich ein id_token mit Informationen zum Nutzer (Nutzername, Email etc., je nach Scopes). In diesem Fall handelt es sich um ein jwt Token, der Base64 codierte String kann z.B. mit Tools wie <https://jwt.io> aufgelöst werden (Abb.??).

```

1 {
2   "token_type": "Bearer",
3   "expires_in": 86400,
4   "access_token": "
      _kn999A3ZjF6MlxKr3PfN9VeZTUPt5E4jNXvdmGdMF2Lu0iA2vk3P5Qhm9hqnN71sf9HtAO",
5   "scope": "openid profile email photo",
6   "id_token": "
      eyJraWQiOiJzMTZ0cVNtODhwREo4VGZCXzdrSEtQUkFQRjg1d1VEVGxteW85SUxUZTdzIiwiYXnIjoiUlMyNTYifQ
      .
      eyJzdWIiOiJiYWQtc2t5bGFyaOBleGFtcGx1LmNvbSIsIm5hbWUiOiJCYWQgU2t5bGFyayIsImVtYWlsIjoiYmFkLXNreW
      .ZoPvZPaomd0nnz2GFRGbgw7PPWIMFDqSBp0gbN4An4a9F-Bc-4
      _T9EBGV8aGetyjZYAON0gjNVOpONGFiwettePWKuxBzusuGCEd9iXWWU09-
      WTF5e2AGr3_jkg34dbxfiFXY3KgH7m0czm809cMaiZ_ofLYgJHVD81qMQoWifhoNhpjPqa19Svc3nCHzSYHugTXQWvA56N
      -b0d7FL39jeam2-
      k1TFSDoxyJE513aC00ssRADr_TWvtL8xoapkXM_7bXYs9_7erXmzF91aOhvm0uasieetpLh0vFeoioJWCU9xhxj4Q
      "

```

The screenshot shows a token decoding interface with two main sections: 'Encoded' and 'Decoded'.

Encoded: PASTE A TOKEN HERE

```
eyJraWQiOjJzMTZ0cVNtODhwREo4VGZCXzdrSET
QUkFQRjg1d1VEVGxteW85SUxUZTdzIiwiYWxnIj
oiUlMyNTYifQ.eyJzdWIiOiJiYWQtc2t5bGFya0
BleGFtcGxlLmNvbSIIm5hbWUiOjCYWQgU2t5b
GFyayIsImVtYWlsIjoiYmFkLXNreWxhcmTAZxhh
bXBsZS5jb20iLCJpc3Mi0iJodHRwczovL3BrLWR
1bW8ub2t0YS5jb20vb2F1dGgyL2R1ZmF1bHQiLC
JhdWQiOjPaF9GdUpiZ2RuTUJ0Q3JCVG9uSU1mR
nIiLCJpYXQiOjE20DM1NTU2ODIsImV4cCI6MTY4
NjE0NzY4MiwiYW1yIjpbInB3ZCJdfQ.ZoPvZPao
mdOnnz2GFRGbgaW7PPWIMFdqSBp0gbN4An4a9F-
Bc-
4_T9EBGV8aGetyjZYAON0gjNV0p0NGFiwettePW
KuxBzusuGCEd9iXWWU09-
WTF5e2AGr3_jkg34dbxfiFXy3KgH7m0czm809cM
aiz_ofLYgJHVD81qMQoWifhoNhpjPqa19SvC3C
HzSYHUGTXQWvA56NmQvyVPh_0M7GMpc6zHopmih
Jqt3eREof8N-b0d7FL39jeam-
k1TFSDoxyJE513aC00ssRADr_TWvtL8xoapKXM_
7bXYs9_7erXmzF91a0hvmoUasieetpLh0vFeoi0
JWCU9xhxj4Q
```

Decoded: EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE
<pre>"kid": "s16tqSm88pDj8TfB_7KHPRAPFB5wUDTlmyo9ILTe7s", "alg": "RS256" }</pre>
PAYOUT: DATA
<pre>{ "sub": "bad-sylark@example.com", "name": "Bad Skylark", "email": "bad-sylark@example.com", "iss": "https://pk-demo.okta.com/oauth2/default", "aud": "Oh_FuJbgdnMBNCrBtonIffr", "iat": 1683555682, "exp": 1686147682, "amr": ["pwd"] }</pre>
VERIFY SIGNATURE
<pre>RSASHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), Public Key in SPKI, PKCS #1, X.509 Certificate, or JWK stri ng format.</pre>

Abbildung 33

4.3.2 Implementierung

Abschließend noch ein Codesnippet (bewusst in einem großen Block um die gesamte Funktionalität nachzuvollziehen - Clean Code Standards wurden hier offensichtlich nicht angewendet), welches kurz erklärt werden soll (los gehts ab Z. 56):

Zu Beginn werden die Umgebungsvariablen für den Google-Client (Client-ID und Geheimnis) gelesen, um den Zugriff auf Google-Authentifizierungsdienste zu ermöglichen. Dann wird eine Instanz des CoreClient erstellt, das für die Interaktion mit dem Identity Provider verwendet wird. Die Provider-Metadaten werden zur Laufzeit abgerufen.

Das CoreClient-Objekt verwendet den Authorization Code Flow (s.o.) und eine zufällige Zeichenfolge als CSRF-Token und Nonce, um den Autorisierungscode abzurufen. Die authorizeleitet dann den Benutzer weiter, der diese URL in einem Webbroweser öffnen und den Authentifizierungsprozess durchführen muss. Nach Abschluss des Prozesses wird der Benutzer zu der Redirect-URL (<http://localhost:8080>) zurückgeleitet, wo der Code und das CSRF-Token vom Server empfangen und extrahiert werden.

Der Autorisierungscode wird dann verwendet, um ein OAuth2-Token anzufordern, welches vom Identity Provider signiert und verschlüsselt wurde. Es gibt viele andere Schritte, die in dem Programm ausgeführt werden, um die Sicherheit und die Vertraulichkeit der Token zu gewährleisten, darunter die Überprüfung der Signatur und die Verwendung von Verschlüsselungsalgorithmen, um die Integrität und Vertraulichkeit der Daten zu schützen, diese sollen hier aber nicht weiter ausgeführt werden. Interessant ist noch der revocation Endpunkt, welcher beim Beenden de Programms jedes Mal ausgerufen wird und das Token somit revoziert (zurückgezogen wird). Der Nutzer muss sich also jedes Mal neu einloggen. Dies verhindert natürlich genau einen der Vorteile, wurde hier aber als POC eingesetzt um die Revozierung zu demonstrieren.

Listing 34: OAuth2 und OICD in Rust

```

1 use std::env;
2 use std::io::{BufRead, BufReader, Write};
3 use std::net::TcpListener;
4 use std::process::exit;
5 use dotenv::dotenv;
6 use serde::{Deserialize, Serialize};
7 use url::Url;
8
9 use openidconnect::core::{
10     CoreAuthDisplay, CoreClaimName, CoreClaimType, CoreClient,
11     CoreClientAuthMethod, CoreGrantType,
12     CoreIdTokenClaims, CoreIdTokenVerifier, CoreJsonWebKey, CoreJsonWebKeyType,
13     CoreJsonWebKeyUse,
14     CoreJweContentEncryptionAlgorithm, CoreJweKeyManagementAlgorithm,
15     CoreJwsSigningAlgorithm,
16     CoreResponseMode, CoreResponseType, CoreRevocableToken,
17     CoreSubjectIdentifierType,
18 };
19
20 use openidconnect::reqwest::http_client;
21 use openidconnect::{
22     AdditionalProviderMetadata, AuthenticationFlow, AuthorizationCode, ClientId,
23     ClientSecret,
24     CsrfToken, IssuerUrl,Nonce, OAuth2TokenResponse, ProviderMetadata,
25     RedirectUrl, RevocationUrl,
26     Scope,
27 };
28
29 fn handle_error<T: std::error::Error>(fail: &T, msg: &'static str) {
30     let mut err_msg = format!("ERROR: {}", msg);
31     let mut cur_fail: Option<&dyn std::error::Error> = Some(fail);
32     while let Some(cause) = cur_fail {
33         err_msg += &format!("\n    caused by: {}", cause);
34         cur_fail = cause.source();
35     }
36     println!("{}", err_msg);
37     exit(1);
38 }
39
40 #[derive(Clone, Debug, Deserialize, Serialize)]
41 struct RevocationEndpointProviderMetadata {
42     revocation_endpoint: String,
43 }
44
45 impl AdditionalProviderMetadata for RevocationEndpointProviderMetadata {}
46 type GoogleProviderMetadata = ProviderMetadata<
47     RevocationEndpointProviderMetadata,
48     CoreAuthDisplay,
49     CoreClientAuthMethod,
50     CoreClaimName,
51     CoreClaimType,
52     CoreGrantType,
53     CoreJweContentEncryptionAlgorithm,
54     CoreJweKeyManagementAlgorithm,
55     CoreJwsSigningAlgorithm,
56     CoreJsonWebKeyType,
57     CoreJsonWebKeyUse,
58     CoreJsonWebKey,
59     CoreResponseMode,
60     CoreResponseType,
61     CoreSubjectIdentifierType,
62 >;
63
64 fn main() {
65     env_logger::init();
66     dotenv().ok();
67
68     let google_client_id = ClientId::new(
69         env::var("GOOGLE_CLIENT_ID").expect("Missing the GOOGLE_CLIENT_ID
70             environment variable."),
71     );
72     let google_client_secret = ClientSecret::new(
73

```

```

64     env::var("GOOGLE_CLIENT_SECRET")
65         .expect("Missing the GOOGLE_CLIENT_SECRET environment variable."),
66     );
67     let issuer_url =
68         IssuerUrl::new("https://accounts.google.com".to_string()).expect("Invalid
69             issuer URL");
70     let provider_metadata = GoogleProviderMetadata::discover(&issuer_url,
71         http_client)
72         .unwrap_or_else(|err| {
73             handle_error(&err, "Failed to discover OpenID Provider");
74             unreachable!();
75         });
76     let revocation_endpoint = provider_metadata
77         .additional_metadata()
78         .revocation_endpoint
79         .clone();
80     println!(
81         "Discovered Google revocation endpoint: {}",
82         revocation_endpoint
83     );
84
85     let client = CoreClient::from_provider_metadata(
86         provider_metadata,
87         google_client_id,
88         Some(google_client_secret),
89     )
90         .set_redirect_uri(
91             RedirectUrl::new("http://localhost:8080".to_string()).expect("Invalid
92                 redirect URL"),
93         )
94         .set_revocation_uri(
95             RevocationUrl::new(revocation_endpoint).expect("Invalid revocation
96                 endpoint URL"),
97         );
98
99     let (authorize_url, csrf_state, nonce) = client
100        .authorize_url(
101            AuthenticationFlow::<CoreResponseType>::AuthorizationCode,
102            CsrfToken::new_random,
103            Nonce::new_random,
104        )
105        .add_scope(Scope::new("email".to_string()))
106        .add_scope(Scope::new("profile".to_string()))
107        .url();
108
109     println!("Open this URL in your browser:\n{}\n", authorize_url);
110
111     let listener = TcpListener::bind("127.0.0.1:8080").unwrap();
112     let (mut stream, _) = listener.accept().unwrap();
113
114     let code;
115     let state;
116     {
117         let mut reader = BufReader::new(&stream);
118
119         let mut request_line = String::new();
120         reader.read_line(&mut request_line).unwrap();
121
122         let redirect_url = request_line.split_whitespace().nth(1).unwrap();
123         let url = Url::parse(&("http://localhost".to_string() + redirect_url)).unwrap();
124
125         let code_pair = url
126             .query_pairs()
127             .find(|pair| {
128                 let &(ref key, _) = pair;
129                 key == "code"
130             })

```

```

129         .unwrap();
130
131     let (_, value) = code_pair;
132     code = AuthorizationCode::new(value.into_owned());
133
134     let state_pair = url
135         .query_pairs()
136         .find(|pair| {
137             let &(ref key, _) = pair;
138             key == "state"
139         })
140         .unwrap();
141
142     let (_, value) = state_pair;
143     state = CsrfToken::new(value.into_owned());
144 }
145
146 let message = "Go back to your terminal :)";
147 let response = format!(
148     "HTTP/1.1 200 OK\r\ncontent-length: {}\r\n\r\n{}",
149     message.len(),
150     message
151 );
152 stream.write_all(response.as_bytes()).unwrap();
153
154 println!("Google returned the following code:\n{}\n", code.secret());
155 println!(
156     "Google returned the following state:\n{}\n(expected '{}')\n",
157     state.secret(),
158     csrf_state.secret()
159 );
160
161 let token_response = client
162     .exchange_code(code)
163     .request(http_client)
164     .unwrap_or_else(|err| {
165         handle_error(&err, "Failed to contact token endpoint");
166         unreachable!();
167     });
168
169 println!(
170     "Google returned access token:\n{}\n",
171     token_response.access_token().secret()
172 );
173 println!("Google returned scopes: {:?}", token_response.scopes());
174
175 let id_token_verifier: CoreIdTokenVerifier = client.id_token_verifier();
176 let id_token_claims: &CoreIdTokenClaims = token_response
177     .extra_fields()
178     .id_token()
179     .expect("Server did not return an ID token")
180     .claims(&id_token_verifier, &nonce)
181     .unwrap_or_else(|err| {
182         handle_error(&err, "Failed to verify ID token");
183         unreachable!();
184     });
185 println!("Google returned ID token: {:?}", id_token_claims);
186
187
188 let token_to_revoke: CoreRevocableToken = match token_response.refresh_token()
189 {
190     Some(token) => token.into(),
191     None => token_response.access_token().into(),
192 };
193
194 client
195     .revoke_token(token_to_revoke)
196     .expect("no revocation_uri configured")
197     .request(http_client)
198     .unwrap_or_else(|err| {

```

```
198         handle_error(&err, "Failed to contact token revocation endpoint");
199         unreachable!();
200     );
201 }
```

Literatur

- OWASP (2019), “Owasp api security top 10 2019.” Technical report, OWASP.
- Postman (2022), “2022 state of the api report.” Technical report, Postman.
- Schnepf, Michael (2021), “Edi vs api vs schnittstellen: Systemintegration in der praxis.” <https://systempilot.net/edi-rest-api-schnittstellen-systemintegration>, zuletzt abgerufen: Mai 2023.
- Wikipedia (2023a), “Programmierschnittstelle.” <https://de.wikipedia.org/wiki/Programmierschnittstelle>, zuletzt abgerufen: Mai 2023.
- Wikipedia (2023b), “Spiegelneurone.” <https://de.wikipedia.org/wiki/Spiegelneuron>, zuletzt abgerufen: Mai 2023.