Fachhochschule Nordwestschweiz
Hochschule für Technik

Hypermedia driven REST API for linked bibliographic resources

IP-516bb

swissbib

Hypermedia driven REST API for linked bibliographic resources.

Mächler Markus und Stucki Melanie

Projekt 5, Studiengang Informatik und Informatik IComptence
FHNW

Brugg, 09.12.2016

FHNW

Markus.Maechler@students.fhnw.ch
Melanie.Stucki@students.fhnw.ch
www.fhnw.ch

Abstract

Folgende Fragen werden im Verlauf der Arbeit genauer untersucht:
Ziel dieses Dokumentes ist es unsere erstellte Linked Data REST Schnittstelle und ihre Entwicklung zu dokumentieren. Der Fokus wird auf folgende Fragen gelegt:

- *Wie kann im PHP-Umfeld eine unabhängige Komponente erstellt werden, die in anderen Projekten, mit anderen Frameworks wiederverwendet werden kann?*
- *Wie lässt sich die Query DSL des Elasticsearch Servers in PHP-Code abstrahieren?*
- *Wie lassen sich Suchabfragen per Konfiguration (ohne Änderungen am Source Code) möglichst flexibel definieren und anpassen?*
- *Wie soll mit der neu erstellten Schnittstelle interagiert werden?*
- *Wie stellen wir sicher, dass die Schnittstelle selbstbeschreibend ist?*

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Weltweit gibt es eine riesige Menge an Daten. Doch Daten an sich sind wertlos ohne einen entsprechenden Kontext. Wenn wir das Format von einem Datensatz nicht kennen, können wir keine Informationen daraus ziehen. Und selbst wenn das Format bekannt ist, ist es meist schwierig diese Daten in einen grösseren Zusammenhang zu setzen und mit anderen Datenquellen zu verbinden.
Genau damit befasst sich unser Projekt. Wie können wir Daten wie ein Bild oder eine Beziehung für einen Computer verständlich machen. Wieso weiss der Computer, dass ein Bild ein Portrait von Paul Klee zeigt? Von wo holt der Computer die Information, was hinter der Beziehung «geschrieben von» steckt?
Diese Fragen lassen sich alle durch das Konzept von Linked Data lösen. In den folgenden Seiten lesen Sie wie wir das Konzept von Linked Data mit einer REST Schnittstelle verknüpfen um die vorhandenen Daten der Universitätsbibliothek Basel nicht nur für Menschen sondern auch für maschinelle Clients lesbar und verständlich zu machen.
Wie bei «normalen» Daten gibt es auch bei Linked Data verschiedene Datenformate. Wir untersuchen diese unterschiedlichen Formate, wie z.B. JSON-LD, Turtle oder RDF/XML. Im Rahmen der REST Schnittstelle werden wir uns mit der Architektur der Software und der Definition der Schnittstelle gegen Aussen beschäftigen. Insbesondere werden wir uns vertieft damit auseinandersetzen, wie wir Suchabfragen auf einen Elasticsearch Server möglichst dynamisch und einfach konfigurierbar gestalten können.
Dank den eingesetzten Konzepten von Hydra und Linked Data ermöglicht die API eine zuvor nicht dagewesene Möglichkeit der Interaktion. Die Daten können damit komplett automatisiert von maschinellen Clients abgefragt und interpretiert werden.

2 Ausgangslage

Unser Projekt *Linked Data REST Schnittstelle* ist Teil eines, sehr viel grösseren, SUK P-2[1] Projektes namens linked.swissbib.ch[2]. Ziel des gesamten Projektes ist es, durch die Anreicherung der bestehenden swissbib Datensätze mit Linked Data Konzepten, einen Mehrwert für die Benutzerinnen und Benutzer zu schaffen.
Es ist bereits eine aufwändige Pipeline vorhanden, um Daten von verschiedenen Bibliotheksverbunden zusammenzutragen, zu deduplizieren, zu clustern und schlussendlich mit weiteren Linked Data Quellen (z.B. DBPedia) anzureichern. Diese Daten werden schlussendlich im Format JSON-LD in einem Elasticsearch Server indexiert. Die gesamte Prozesskette steht zwar schon seit einiger Zeit, es wird aber noch sehr viel Arbeit hinein gesteckt um den Prozess und die Resultate zu verbessern.
Was dem Projekt noch komplett fehlt ist eine öffentliche Schnittstelle für den Zugriff auf diese Daten. Die Schnittstelle soll dabei auf der Basis der Daten im Elasticsearch Server aufgebaut werden und über REST verschiedene Serialisierungsformate anbieten.

[1] https://www.swissuniversities.ch/de/organisation/projekte-und-programme/suk-p-2-wissensch-information-zugang-verarbeitung-speicherung/
[2] http://www.swissbib.org/wiki/index.php?title=Linked_swissbib

3 Systemübersicht

Wie bereits in der Einleitung erwähnt, ist unser Projekt Linked Data REST Schnittstelle nur ein Teilprojekt eines grösseren Projektes. Die gesamte Pipeline der Datenbeschaffung und Anreicherung bis hin zur Ausgabe ist in der folgenden Grafik illustriert.

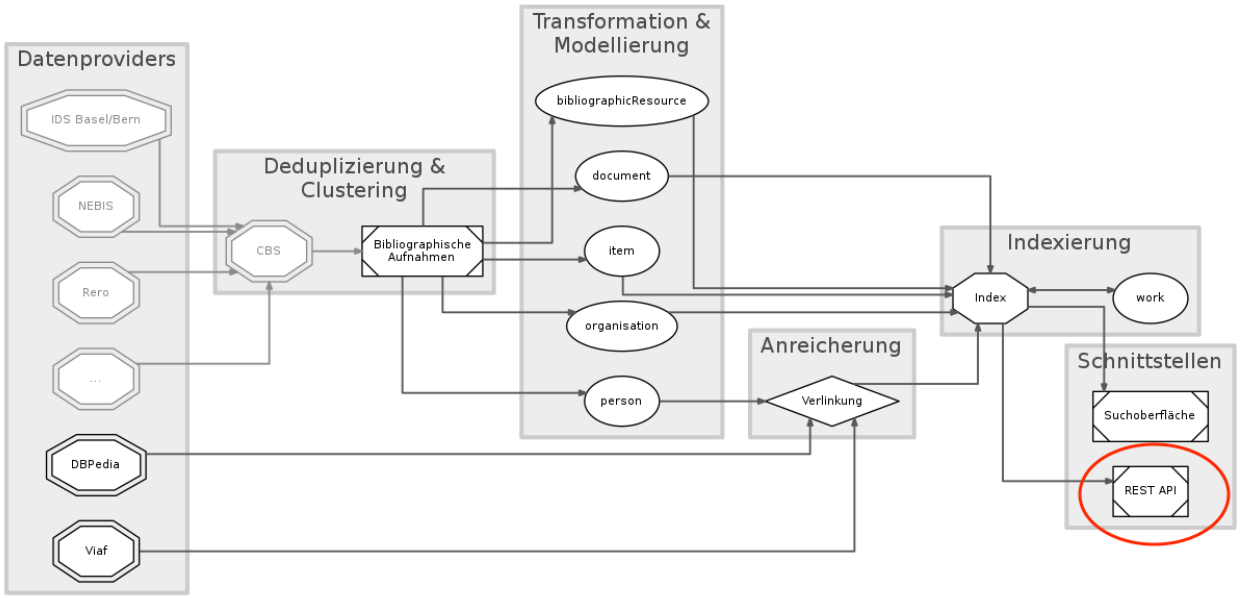


Abbildung 1: Systemübersicht Linked Swissbib [1]

Der einzige Berührungspunkt unseres Teilprojektes zum Gesamtprojekt besteht im Zugriff auf eine geteilte Elasticsearch Server Instanz (In Abbildung 1 „Index“ genannt.). Im Folgenden werden wir uns auf unser Teilprojekt bestehend aus der REST API und dem Index beschränken.

Die folgende Abbildung zeigt die entwickelte REST API im Zusammenspiel mit dem Elasticsearch Server und einem Klienten, der eine Abfrage an die API schickt.

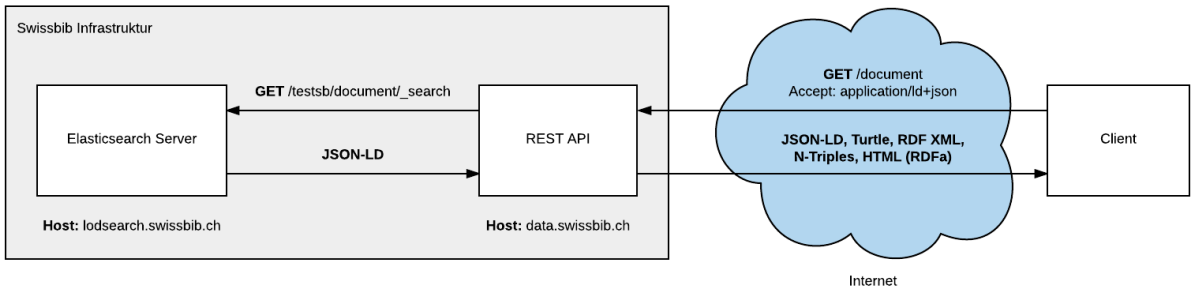


Abbildung 2: Systemübersicht des Teilprojekts REST API

4 Technologieübersicht

4.1 Symfony und API-Platform

4.1.1 Symfony

Um den Aufbau der Applikation zu verstehen, ist es essentiell ein gewisses Grundverständnis für die verwendeten Frameworks zu haben. Symfony ist ein Komponenten basiertes Framework, das für praktisch jede Applikation eine geeignete Kombination der unzähligen Komponenten[5] bereitstellt. Eine der wichtigsten Komponenten für jede Web-Applikation ist wohl der HttpKernel[6]. Der HttpKernel definiert den Ablauf der grundlegendsten Aufgabe jeder Web-Applikation überhaupt: Einen HTTP Request in eine HTTP Response zu verwandeln. Damit ist er das Herzstück unserer Applikation.

Die folgende Grafik zeigt den Ablauf, wie er vom HttpKernel definiert wird:

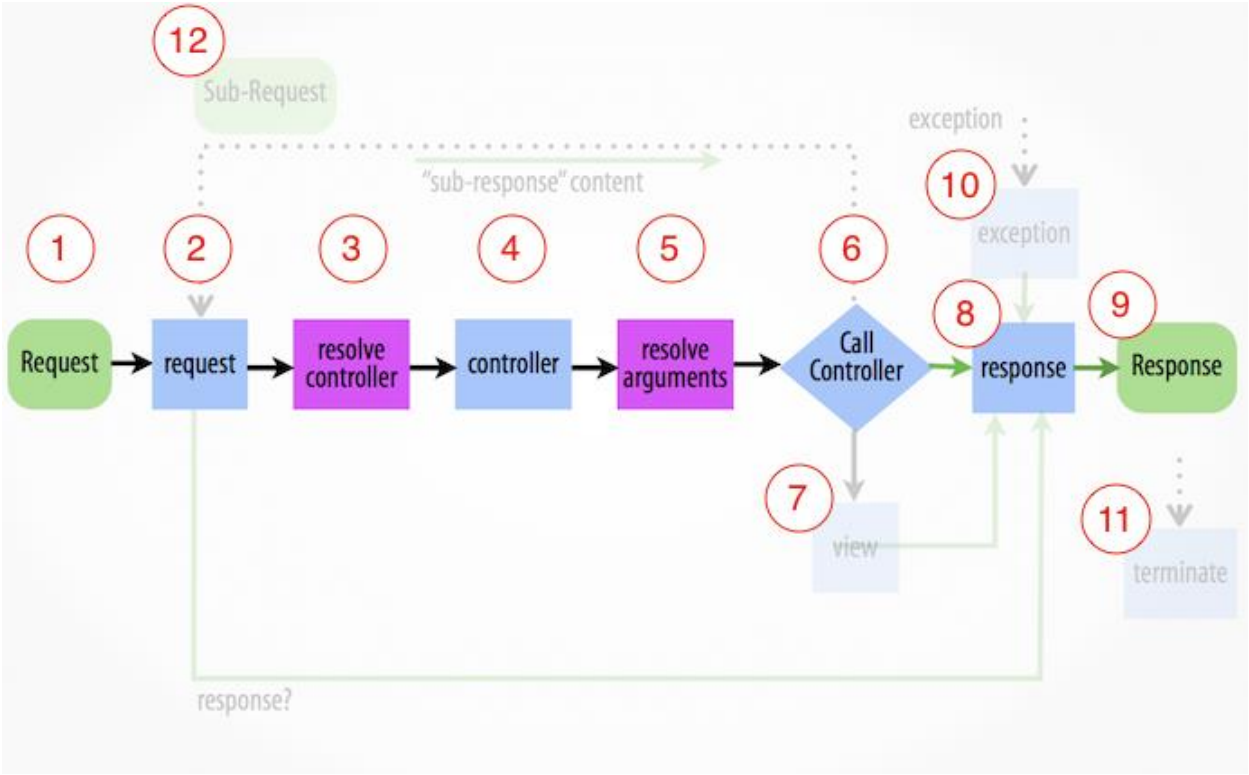


Abbildung 3: Ablauf eines HTTP Request mit der HttpKernel Komponente [2]

[5] https://packagist.org/packages/symfony/

[6] http://symfony.com/doc/current/components/http_kernel.html

1. Der HttpKernel erhält ein Request-Objekt, das aus dem Inhalt der superglobalen[7] PHP Variablen erstellt wird.
2. Es wird ein Event *kernel.request* ausgelöst. EventListener, die auf diesen Event hören, können bereits eine vollständige Antwort (z.B. 404 Not Found) zurückgeben und die Ausführung beenden. Oder die EventListener können das Request-Objekt mit zusätzlichen Attributen versehen. Ein wichtiger Listener in diesem Schritt ist der *RouterListener*, der wenn möglich den Controller auflöst und das Attribut *_controller* setzt.
3. Der HttpKernel ruft auf dem Objekt ControllerResolver die Methode *getController* auf, die ein PHP-Callable zurückgeben muss.
4. Der Event *kernel.controller* wird ausgelöst. Er gibt den Listener die Möglichkeit den Controller noch zu ändern oder dessen Initialisierung zu beeinflussen.
5. Der HttpKernel ruft auf dem Objekt ArgumentResolver die Methode getArguments auf, die die Argumente für den Controller zurückgeben muss.
6. Das Controller Callable wird ausgeführt, dabei gibt es zwei mögliche Ausgänge:
    a. Der Controller gibt ein Response-Objekt zurück.
    b. Der Controller gibt kein Response-Objekt zurück, dann wird der Event *kernel.view* ausgelöst.
7. Die Listener des Events *kernel.view* haben die Möglichkeit aus der Controller Antwort ein Response Objekt zu erstellen.
8. Der Event *kernel.response* wird ausgelöst, dieser gibt den Listener die Möglichkeit das Response-Objekt noch zu verändern, bevor die Antwort an den Client geschickt wird.
9. Die Response wird an den Client geschickt.
10. Wenn eine Exception nicht im eigenen Code behandelt wird, wird sie vom HttpKernel gefangen und der Event *kernel.exception* wird ausgelöst. Listener dieses Events sollten mit der Exception gleich ein Response Objekt erzeugen können.
11. Nach dem die Antwort bereits an den Client geschickt wurde, wird noch der Event *kernel.terminate* ausgelöst. Listener dieses Events können letzte Aufräumarbeiten machen oder zeitaufwändige Routinen ausführen, die die Response nicht mehr beeinflussen.
12. Zusätzlich gibt es die Möglichkeit den ganzen Prozess mehrfach zu durchlaufen mit der Hilfe von Sub-Requests.

[7] http://php.net/manual/en/language.variables.superglobals.php

4.1.2 API Platform

Das Framework API-Platform setzt auf diesem Mechanismus des Symfony HttpKernel an und erweitert diesen mit API spezifischen Komponenten. Eine vollständige Liste der verwendeten Events findet man in der Dokumentation[8].
Die folgende Grafik zeigt exemplarisch einen Ablauf, wie er bei einem Request mit API-Platform durchgeführt wird. Wir beschränken uns dabei nur auf die relevanten Elemente.

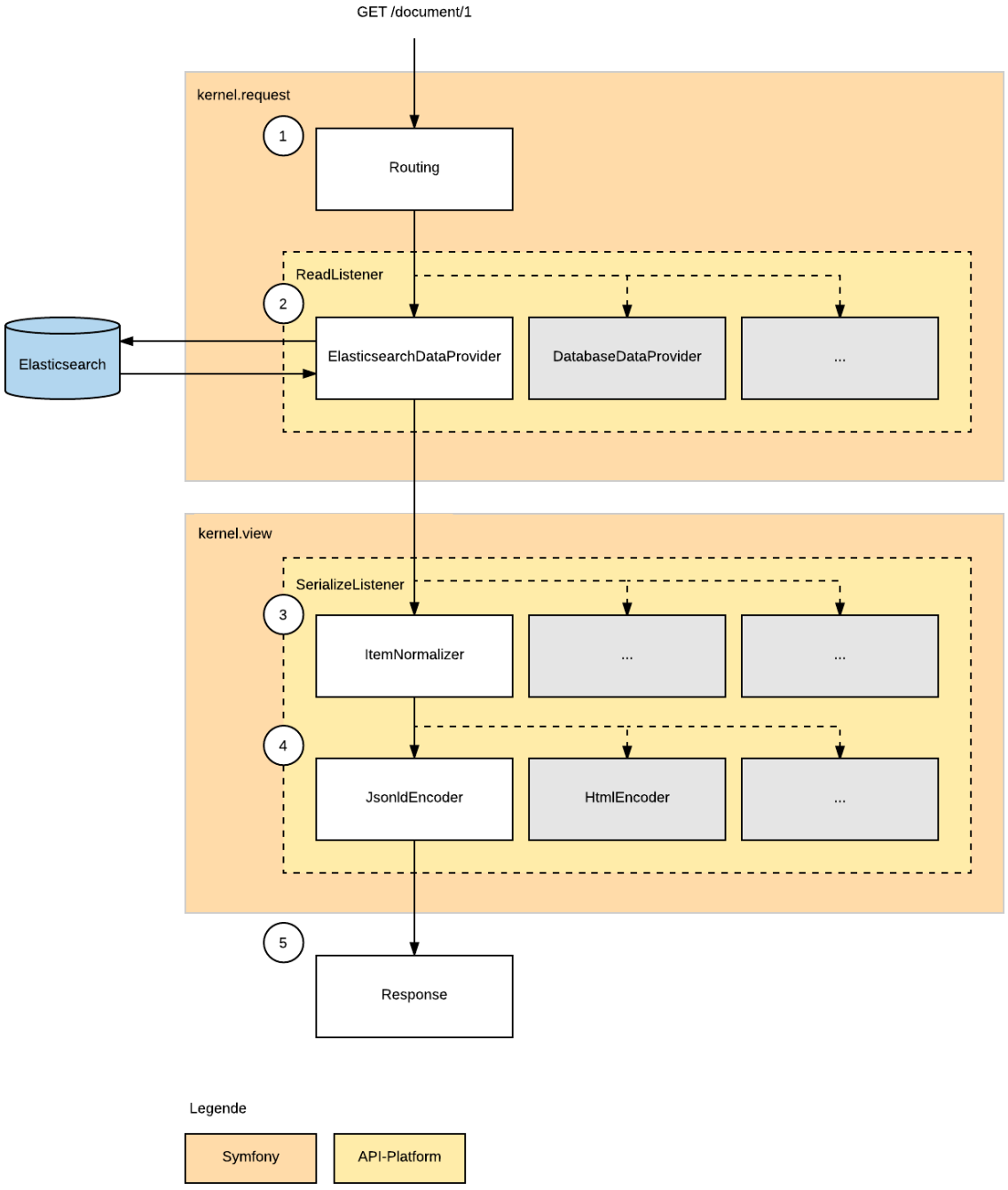


Abbildung 4: Exemplarische Darstellung eines Requests mit dem Framework API Platform

[8] https://api-platform.com/docs/core/events

1. Anhand der Request-Informationen und den registrierten Routen wird geprüft ob die URL valide ist und welche Aktionen darauffolgen sollen. Für alle registrierten Entitäten werden automatisch Routen zur Verfügung gestellt.
2. Für GET-Requests wird innerhalb des ReadListeners aus verschiedenen DataProvidern ausgesucht. In unserem Fall hat der ElasticsearchDataProvider die höchste Priorität und kann alle Anfragen beantworten.
3. Innerhalb des SerializeListener werden die Daten (Entity Objekte) zuerst normalisiert. In unserem Fall bedeutet das, dass sie in ein Array umgewandelt werden und mit Hydra spezifischem Kontext erweitert werden.
4. Nach der Normalisierung werden die Daten in das entsprechende Ausgabeformat konvertiert (z.B. JSON-LD, HTML oder Turtle).
5. Aus der Ausgabe des Encoders wird nun ein Response-Objekt erstellt und an den Client gesendet.

4.1.3 Konzepte

Ein zentrales Stück unserer Applikation ist der Symfony **Service Container**[9]. Er managet die Instanzierung von Objekten und all ihrer Abhängigkeiten und regelt den zentralen Zugriff auf diese Objekte. Die Abhängigkeiten werden mittels **Dependency Injection**[10] aufgelöst. Das macht das System sehr einfach erweiterbar und testbar. Um bestehende Services von Symfony oder API-Platform zu erweitern, wird das **Decorator**[11] Pattern benutzt.
Die Konfiguration aller Services befindet sich in der Datei *app/config/services.yml*.
Ein Auszug aus dieser Datei sieht wie folgt aus:

```
services:
   linked_swissbib.elasticsearch_data_provider:
      class: LinkedSwissbibBundle\DataProvider\ElasticsearchDataProvider
      tags: [ { name: 'api_platform.item_data_provider', priority: 1 },
            { name: 'api_platform.collection_data_provider', priority: 1 } ]
      arguments:
        - '@linked_swissbib.elasticsearchadapter_adapter'
        - '@linked_swissbib.elasticsearchadapter_templatesearchbuilder'
        - '@linked_swissbib.entity_simple_entity_builder'
        - '@linked_swissbib.elasticsearch_context_mapper'
        - '@request_stack'
        - '@linked_swissbib.elasticsearch_resource_name_converter'
        - '@linked_swissbib.elasticsearch_params_builder'
```

Diese Konfiguration registriert die Klasse *LinkedSwissbibBundle\DataProvider\Elasticserach-DataProvider* als Service. Der Service hat Tags, die ihn für das Framework API-Platform als DataProvider erkennbar machen. Als Konstruktor-Argumente erhält der Service weitere Services mittels Dependency Injection.

[9] http://symfony.com/doc/current/service_container.html
[10] https://de.wikipedia.org/wiki/Dependency_Injection
[11] https://de.wikipedia.org/wiki/Decorator

4.2 Module / Source Code

Weitere Implementierungsdetails zu einzelnen Klassen können im Dokument *Developer Guide IP-516bb.docx* nachgelesen werden.

4.3 Performance Optimierung

Um die Performance der PHP Applikation zu optimieren können folgende Schritte unternommen werden:

- APCu[12] Memory Caching aktivieren in der Datei *app/config/config.prod.yml*.
- Den Composer Class-Autoloader aufwärmen mit dem Befehl:
  composer dump-autoload –optimize
  (siehe auch *development.md* im Root-Verzeichnis der Applikation)
- Alle nicht benötigten PHP und Apache Module deaktivieren, insbesondere XDebug.
- Als Prozess Manager PHP-FPM[13] einsetzen.

Des Weiteren könnte der PHP Prozessmanager[14] die Performance signifikant steigern. Dieses Projekt ist allerdings noch stark in der Entwicklung und von einem produktiven Einsatz wird ausdrücklichabgeraten.

4.4 Evolutionsszenarien

4.4.1 Weitere Konzepte

Eines der wohl wahrscheinlichsten Evolutionsszenarien ist, dass zusätzliche Entitäten über die API angeboten werden in Zukunft. Dies kann sehr einfach damit erreicht werden, dass eine neue Klasse mit dem Namen der Entität und entsprechenden Annotationen erstellt wird. Damit wird die Entität bereits über die API angeboten. Um die Suche zu ermöglichen, muss in der Datei *app/config/elasticsearch_adapter.yml* ein Such-Template erstellt werden mit den Feldern, die standardmässig durchsucht werden können. So eine Erweiterung ist damit mit minimalem Aufwand verbunden.

4.4.2 Discovery Search

Bis jetzt ist es nur möglich innerhalb von einzelnen Konzepten (z.B. Document) zu suchen. Für Benutzer, die noch nicht genau wissen wonach sie suchen, sollte es einen Endpunkt der API geben, der es erlaubt über mehrere Konzepte zu suchen. Von der Seite von Elasticsearch und unserem Elasticsearch Adapter lässt sich das einfach konfigurieren, indem mehrere Typen angegeben werden. Durch Einschränkungen des Frameworks API-Platform gibt es einige Herausforderungen bei diesem Szenario:

- Die automatisierte Dokumentation funktioniert nur auf Ebene von Entitäten.
- Das automatisierte Routing funktioniert nur für Entitäten.
- Die Hydra Normalisierung funktioniert nur für Entitäten, die vom gleichen Typ sind.
- Im JSON-LD Kontext gibt es keine Unterstützung für Namespaces, damit kann es zu Namenskonflikten unter mehreren Entitäten kommen.

[12] https://pecl.php.net/package/APCu
[13] https://php-fpm.org/
[14] https://github.com/php-pm/php-pm

4.4.3 Erweiterte Suche

Das Such-Interface ist bewusst sehr einfach gehalten. In Zukunft könnte es aber gewünscht werden, auch sehr komplexe Abfragen auf der Schnittstelle absetzen zu können. Die gesamte Hydra Normalisierung sowie die Serialisierung der Daten wäre aufgrund unserer Architektur gar nicht betroffen von so einer Änderung. Unsere Komponente Elasticsearch Adapter deckt schon ein breites Spektrum der Elasticsearch Query DSL ab. Die verwendete Bibliothek zum Erstellen der Abfragen deckt sogar alles ab. Falls es nötig wäre, könnte der Adapter somit einfach um gewisse Features, wie z.B. Aggregationen, erweitert werden.

5 Elasticsearch Adapter

Ein zentrales Stück der Applikation und dieses Projektes ist es die Abfragen auf den Elasticsearch Server möglichst flexibel und einfach definieren und anpassen zu können. Im Rahmen dieses Projektes wurde eine unabhängige Komponente zur Abfrage auf den Elasticsearch Server entwickelt. Diese Komponente soll in Zukunft auch in anderen Projekten an der Universität Basel und vielleicht sogar darüber hinaus verwendet werden. Für die Umsetzung dieser unabhängigen Komponente stellen sich folgende Fragen:

Wie kann im PHP-Umfeld eine unabhängige Komponente erstellt werden, die in anderen Projekten, mit anderen Frameworks wiederverwendet werden kann?
Wie lässt sich die Query DSL[15] des Elasticsearch Servers in PHP-Code abstrahieren?
Wie lassen sich Suchabfragen per Konfiguration (ohne Änderungen am Source Code) möglichst flexibel definieren und anpassen?

5.1 Ausgangslage

Im Umfeld um Elasticsearch und PHP gibt es im wesentlichen vier Projekte, die auf die eine oder andere Art Abfragen auf Elasticsearch vereinfachen sollen. Im Folgenden werden diese Projekte kurz vorgestellt:

elastic/elasticsearch[16]
Dieses Projekt ist der offiziell von Elasticsearch entwickelte PHP Client. Der Client ist bewusst sehr einfach gehalten und dadurch sehr flexibel. Es ist ein low-level Client, wie ihn Elasticsearch auch für andere Programmiersprachen anbietet, bei dem die gesamte Abfrage in ein assoziatives Array geschrieben wird.

linked-swissbib/adapterElasticsearch[17]
Aufbauend auf dem offiziellen Elasticsearch Client hat Günter Hipler, Projektverantwortlicher für das Projekt swissbib, begonnen eine objektorientierte Abstraktion für die Generierung des assoziativen Arrays begonnen. Es ist damit möglich einige Teile der Query DSL objektorientiert zusammen zu bauen.

ruflin/Elastica[18]
Elastica ist eine Implementierung, die ohne den offiziellen Elasticsearch Client, direkt JSON Abfragen für den Server generiert. Im Gegensatz zum offiziellen Client bietet es ein objektorientiertes Interface. Der Entwickler von Elastica selbst gibt aber zu bedenken, dass es aus Gründen der Performance und Kompatibilität mit neuen Elasticsearch Versionen wohl mehr Sinn machen würde das objektorientierte Interface aufbauend auf dem offiziellen Elasticsearch Client zu implementieren.

ongr-io/ElasticsearchDSL[19]
Die Bibliothek ElasticsearchDSL hat genau das gemacht, nämlich eine objektorientierte Abstraktion der Query DSL, die schlussendlich ein assoziatives Array generiert, welches dem offiziellen Elasticsearch Client übergeben werden kann. Die Bibliothek deckt alle Aspekte der Elasticsearch Query DSL ab.

[15] https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html
[16] https://github.com/elastic/elasticsearch-php
[17] https://github.com/linked-swissbib/adapterElasticsearch/tree/0c893836de579a16ed6c9d1e56d5d147bbc8ea25
[18] https://github.com/ruflin/Elastica
[19] https://github.com/ongr-io/ElasticsearchDSL/

5.1.1 Fazit

Es gibt bereits verschiedene Ansätze, wie Abfragen auf den Elasticsearch Server abstrahiert werden können. Dabei scheint der Ansatz, der den offiziellen Elasticsearch low-level Client verwendet und darauf aufbauend eine objektorientierte Abstraktion für die Query DSL anbietet, sich gegenüber anderen Varianten durchzusetzen. So profitiert man von den Vorzügen eines objektorientierten Interfaces, kann aber darauf vertrauen, dass die Abfragen so effizient wie möglich ausgeführt werden und die Fehlerbehandlung alle Aspekte des Servers abdeckt. Man hat mit dieser Variante auch immer die Möglichkeit sehr spezifische Abfragen direkt als assoziatives Array zu definieren.
Was all diesen Varianten zurzeit jedoch fehlt, ist die Möglichkeit Suchen per Konfiguration zu definieren und anpassen zu können. Im Umfeld der bibliographischen Daten ist es oft verlangt, dass Suchen auf gewisse Felder eingeschränkt werden müssen, oder bestimmte Filter auf Daten gesetzt werden können. Dabei möchte man nicht ständig die Abfragen im PHP Code ändern müssen.
Für unser Projekt haben wir uns deshalb entschieden, aufbauend auf dem offiziellen Elasticsearch Client und der Query DSL Bibliothek ongr-io/ElasticsearchDSL, einen Mechanismus für die Konfiguration von Suchen zu implementieren.

5.2 Komponente

5.2.1 Versionskontrolle

Der erste Schritt um eine unabhängige Komponente zu erstellen ist, ein eigenes Repository mit einer eigenen Versionskontrolle zu führen. Das ermöglicht die unabhängige Weiterentwicklung der Komponente sowie eine eigene Versionierung mit einem eigenen Release Zyklus.

5.2.2 Dependency Management

Für das Management der Abhängigkeiten gibt es im wesentlichen drei Möglichkeiten:

1. Die Komponente hat keine Abhängigkeiten zu anderen Komponenten.
2. Die Abhängigkeiten werden direkt in das Projekt integriert. Jeder, der die Komponente verwendet muss nichts zusätzlich installieren.
3. Die Abhängigkeiten werden über ein bekanntes Dependency Management System verwaltet.

Variante 1 kommt für uns nicht in Frage da wir, wie in Kapitel Abfragen auf Elasticsearch / Query DSL weiter erläutert wird, Abhängigkeiten zu anderen Komponenten haben. Variante 2 hat den Vorteil, dass die Komponente in sich voll funktionsfähig ist. Sie hat aber auch den Nachteil, dass es zu Konflikten kommen kann, wenn die Komponente in einem Umfeld verwendet wird indem Abhängigkeiten zu den gleichen Komponenten vorkommen.
Variante 3 hat den Vorteil, dass Konflikte bei der Integration in bestehende Systeme vom Dependency Management System aufgelöst werden können, hat aber den Nachteil, dass das spezifische Dependency Management System verwendet werden muss.

Wir haben uns für Variante 3 entschieden, da es mit Composer einen Quasi-Standard für das Dependency Management in PHP gibt. Alle grösseren PHP Projekte und Frameworks setzen Composer für das Dependency Management ein. Composer wird auch in allen Projekten bei swissbib bereits verwendet. Zusätzlich hat Composer den Vorteil, dass das Class-Autoloading standardisiert funktioniert.

5.2.3 Unabhängigkeit von Frameworks

Damit die Komponente in verschiedensten Projekten und Kontexten eingesetzt werden kann, muss die Komponente unabhängig von Framework spezifischen Eigenheiten sein. Es sollte möglich sein die Komponente über einfaches Initialisieren vom PHP Objekten benutzen zu können. Wir haben darauf verzichtet Framework spezifisches Dependency- oder Configuration-Management zu verwenden.

5.3 Abfragen auf Elasticsearch / Query DSL

5.3.1 Abfragen

In der Ausgangslage haben wir bereits verschiedene Möglichkeiten für die Abfragen auf Elasticsearch aus PHP heraus vorgestellt. Wir haben uns aus folgenden Gründen für eine Kombination des offiziellen Elasticsearch low-level Client und der Bibliothek ongr-io/ElasticsearchDSL entschieden:

- Objektorientierte Abstraktion der Query DSL.
- Vollständige Unterstützung aller Elasticsearch Features dank der Verwendung des offiziellen Elasticsearch Clients.
- Zukunftssichere Implementierung, da der Elasticsearch Client von Elasticsearch selbst weiterentwickelt wird
- Genügend Flexibilität um unsere Konfiguration der Suche zu implementieren.

5.3.2 Query DSL

Elasticsearch bietet eine vollständige Query DSL im Format JSON. Die Query DSL besteht im Wesentlichen aus zwei Typen von Klauseln:

Leaf Query Clauses
Solche Klauseln können benutzt werden um nach einzelnen Werten in einem einzelnen Feld zu suchen. Die Term-Klausel kann z.B. genutzt werden um zu prüfen ob ein Feldwert exakt mit dem Suchbegriff überein stimmt:

```
"query": {
   "term" : { "firstname" : "Markus" }
}
```

Compound Query Clauses
Eine Compound-Klausel kann verwendet werden um Leaf-Klauseln zu kombinieren mit z.B. einer OR- oder AND-Verknüpfung.

```
"query": {
   "bool": {
      "must": [
         {
            "term" : { "firstname" : "Markus" }
         },
         {
            "term" : { "lastname" : "Mächler" }
         }
      ]
   }
}
```

Diese Struktur der Query DSL macht es notwendig, dass Query-Klauseln beliebig tief verschachtelt werden können um beliebig komplexe Abfragen zu formulieren. Diese Struktur kann in der objektorientierten Welt sehr gut mit dem Composite-Pattern[20] abgedeckt werden. Die Bibliothek ongr-io/ElasticsearchDSL tut genau dies. In unserer Konfiguration der Such-Templates werden wir diesem Umstand auch Rechnung tragen.

5.3.3 Architektur

Der offizielle Elasticsearch Client hat als einzige Schnittstelle assoziative Arrays. Das gilt sowohl für Queries als auch für Antworten des Elasticsearch Servers. Für den Elasticsearch Client ist das deshalb so naheliegend, weil sich Daten sehr leicht zwischen JSON und assoziativen Arrays hin und her konvertieren lassen.
Die Bibliothek ongr-io/ElasticsearchDSL bietet zumindest für die Generierung des Query-Teils eine objektorientierte Abstraktion. Das Ziel unserer Komponente ist es, die gesamte Abfrage sowie Antwort des Elasticsearch Servers in einer objektorientierten Abstraktion anzubieten.
Die folgende Grafik zeigt eine Übersicht dieser drei Ansätze:

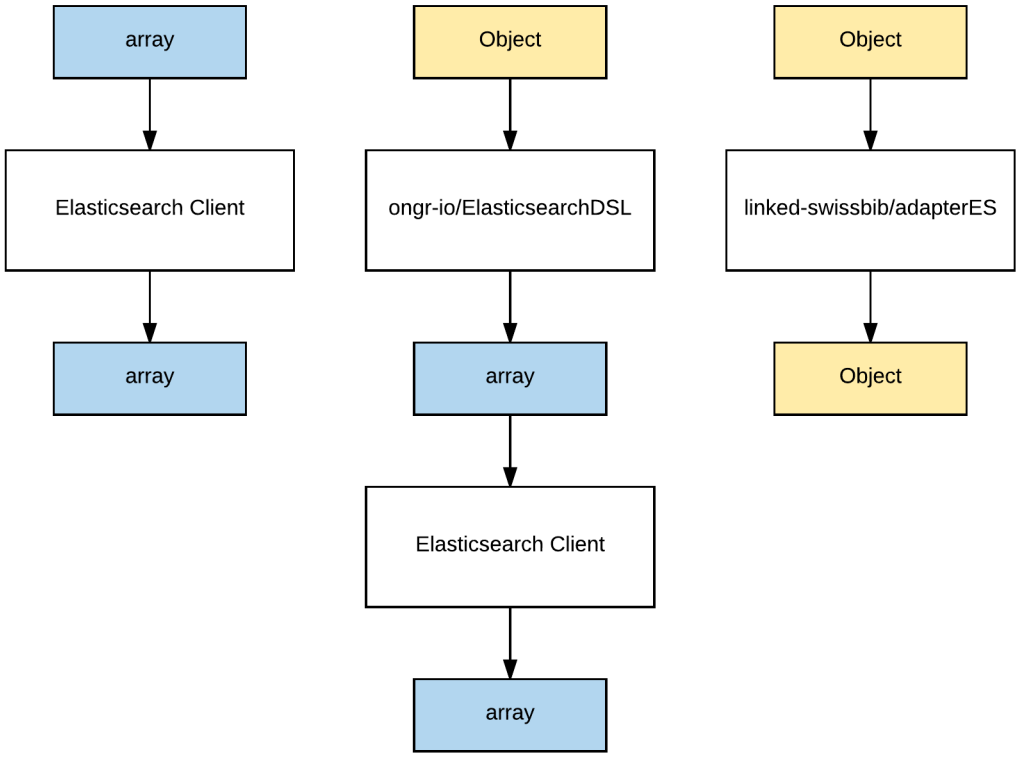


Abbildung 5: Übersicht über In- und Output von verschiedenen Elasticsearch Abstraktionen

[20] https://en.wikipedia.org/wiki/Composite_pattern

Bei der Architektur haben wir darauf geachtet, dass alle Komponenten nur Assoziationen über Interfaces herstellen. Das macht jede Komponente unabhängig von der konkreten Implementierung einer anderen Komponente. Dadurch wird das ganze System sehr flexibel erweiterbar indem konkrete Implementierungen ausgetauscht werden.
Die einzelnen Komponenten werden in den weiterführenden Kapiteln erklärt. Folgende Grafik zeigt eine Übersicht der definierten Interfaces und ihrer Abhängigkeiten:

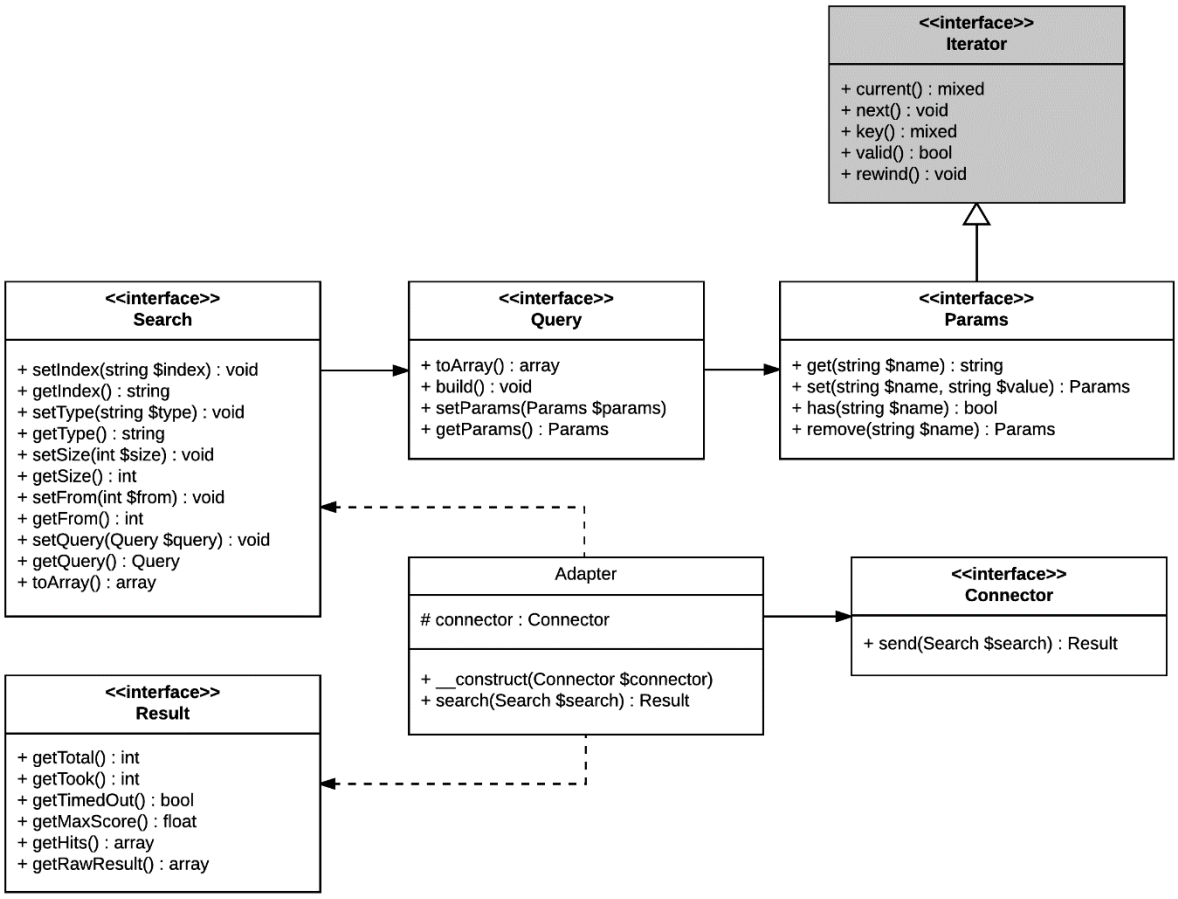


Abbildung 6: UML Klassendiagramm des Elasticsearch Adapters

Der Adapter ist das zentrale Objekt einer jeder Abfrage. Er bietet nur eine Methode an, die ein Search-Objekt verlangt und als Antwort ein Result-Objekt zurückgibt. Diese Komponente wird für jede Abfrage auf Elasticsearch verwendet.

5.3.3.1 Connector

Der Connector ist dafür verantwortlich die konkrete Verbindung zum Elasticsearch Server herzustellen. Eine konkrete Implementierung eines Connectors ist die Klasse **ElasticsearchAdapter\Connector\ElasticsearchClientConnector**. Diese Klasse verwendet für Ihre Abfragen den offiziellen Elasticsearch low-level Client.

5.3.3.2 Search
Die Search-Klasse repräsentiert eine Suche für den Elasticsearch Server. Darin enthalten ist z.B. der Index, der Typ, das Query oder die Size (Limit für Resultate) einer Suche.

5.3.3.3 Query
Das Query-Objekt ist Teil einer Suche. Es enthält die Suchabfrage wie z.B. ein Bool- oder Term-Query für den Elasticsearch Server. Eine konkrete Implementierung des Query-Interfaces ist die Klasse **ElasticsearchAdapter\Query\TemplateQuery**, welche anhand einer Konfiguration erstellt werden kann.

5.3.3.4 Params
Das Params-Interface kann verwendet werden um Parameter für ein Query zu setzen. Es speichert Strings anhand eines weiteren Strings als Index. Eine konkrete Implementierung ist das Objekt **ElasticsearchAdapter\Params\ArrayParams**, welches die Daten als assoziatives Array abspeichert.

5.3.3.5 Result
Das Result-Interface repräsentiert eine Antwort des Elasticsearch Servers. Eine konkrete Implementierung ist die Klasse **ElasticsearchAdapter\Result\ElasticsearchClientResult**, die ein Result-Objekt aus einer Antwort des offiziellen low-level Elasticsearch Client erstellen kann.

5.3.3.6 TemplateSearchBuilder
Um die sich wiederholenden Aufgaben zum Erstellen eines Search-Objekts nur einmal programmieren zu müssen, haben wir einen TemplateSearchBuilder eingeführt. Dieses Objekt erhält ein Array von Template-Definitionen und kann aus den Template Definitionen eine vollständige Suche konstruieren. Was genau mit Template gemeint ist, werden wir in Kapitel Konfiguration der Suche sehen.

5.3.4 Beispiel-Code

Folgender Code zeigt wie die ElasticsearchAdapter Klassen verwendet werden können um eine einfach Abfrage abzusetzen:

```
$templates = loadTemplates(); //load template definitions
$connector = new ElasticsearchClientConnector(['localhost']);
$adapter = new Adapter($connector);
$searchBuilder = new TemplateSearchBuilder($templates);

$search = $searchBuilder->buildSearchFromTemplate('template_name');
$search->setSize(50); //changing search if wanted

$result = $adapter->search($search); //execute search

echo 'Total elements found: ' . $result->getTotal();
```

5.4 Konfiguration der Suche

Die Konfiguration der Suche erfolgt über sogenannte Search-Templates. Das sind Definitionen für eine Suche, die auch Platzhalter enthalten können, die durch Parameter ersetzt werden. Diese Templates sollen als assoziatives PHP-Array zur Verfügung gestellt werden. Damit können sie einfach aus verschiedensten Datei-Formaten (z.B. YAML, JSON oder PHP) eingelesen werden.
Die folgende Grafik zeigt die Struktur, der ein Template folgt. Es handelt sich dabei nicht um ein Klassendiagramm, sondern eine schematische Darstellung der Zusammenhänge.

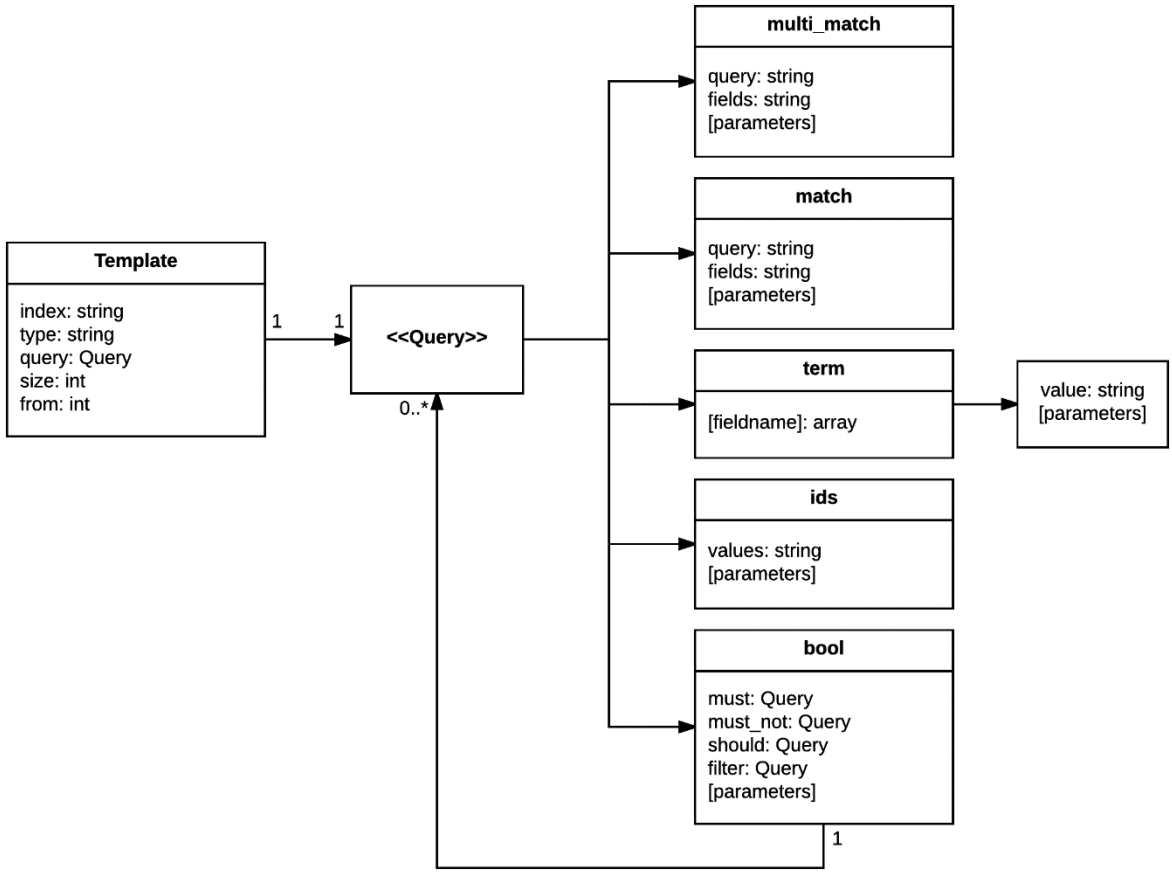


Abbildung 7: Schematische Übersicht der rekursiven Template Struktur

Die implementierten Query-Typen decken die Ealsticsearch DSL nur teilweise ab, wir haben uns für das Projekt auf die für uns relevanten Query-Typen beschränkt. Für die meisten Queries können noch zusätzliche Parameter wie z.B. der Operator (OR oder AND) bei einem multi_match Query definiert werden. Eine vollständige Liste der Parameter kann auf der offiziellen Elasticsearch Webseite[21] [22] nachgeschlagen werden. Die Template-Struktur erlaubt den selben rekursiven Aufbau, wie die Query DSL selbst. In unserem Subset der Query DSL ist es nur das bool Query, das für die Rekursion verwendet werden kann.

[21] https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html
[22] https://www.elastic.co/guide/en/elasticsearch/reference/current/compound-queries.html

5.4.1 Template

Ein Beispiel eines solchen Templates im Format YAML könnte so aussehen:

```yaml
person_template:
  index: 'index1, index2'
  type: 'person'
  size: 20
  query:
    bool:
      must:
        multi_match:
          query: '{q}'
          fields: 'firstname,lastname,address'
          type: 'cross_fields'
          operator: 'and'
      must_not:
        - term:
            status:
              value: 'deleted'
        - term:
            status:
              value: 'inactive'
```

Für die term- und match-Queries gibt es eine Short-Syntax, wenn keine zusätzlichen Parameter verwendet werden:

```yaml
term:
  status: 'deleted'

match:
  status: 'deleted'
```

5.4.2 Parameter

In einem Template können in jedem Text-Feld Parameter definiert werden, die zur Laufzeit ersetzt werden. Ein Parameter muss immer in geschweiften Klammern geschrieben werden (z.B. {parameterName}). Die Parameter können als optionaler, zweiter Parameter bei der Instanzierung eines TemplateSearchBuilder-Objekts mitgegeben werden.
Parameter sind dazu gedacht um dynamischen Inhalt wie z.B. den Suchtext in ein Template einzubetten. Folgender YAML-Code zeigt ein Beispiel eines Parameters „q“:

```yaml
multi_match:
  query: '{q}'
  fields: 'name'
```

5.4.3 Modifiers

Manchmal reicht es nicht einfach einen Parameter zu definieren, sondern man möchte einen Wert noch in irgendeiner Form transformieren, bevor er in das Query eingefügt wird. Um das zu erreichen, haben wir das Konzept der „Modifiers“ eingeführt. Ein Modifier ist im Wesentlichen eine Funktion, die auf einen Parameter angewendet werden kann. Ein neuer Modifier kann ganz einfach dadurch erstellt werden, dass eine Klasse erstellt wird, die das Interface ElasticsearchAdapter\Params\Modifiers\Modifier implementiert und im Namespace ElasticsearchAdapter\Params\Modifiers abgelegt wird. Ein Beispiel für einen solchen Modifier ist der DefaultModifier, der einen Default-Wert entgegennimmt und den Default-Wert zurückgibt, wenn die Variable (im Beispiel „test“) nicht gesetzt ist.

```
index: 'person_index'
type: 'person'
size: '{default(test, 10)}'
query:
  term:
    name: 'Markus'
```

Falls die Variable „test“ im Parameter Objekt gesetzt ist, wird ihr Wert für „size“ eingesetzt. Falls keine Variable „test“ vorhanden ist, wird der Wert 10 für „size“ eingesetzt.

5.5 Fazit

Wir sind überzeugt mit dieser Komponente etwas Neues geschaffen zu haben, das einen signifikanten Mehrwert bietet zu den bestehenden Möglichkeiten innerhalb einer PHP Applikation Abfragen auf Elasticsearch auszuführen. Es ist damit möglich Abfragen alleine über die Konfiguration komplett umzustellen oder gewisse Feineinstellungen über Parameter vorzunehmen, ohne eine Zeile PHP-Code ändern zu müssen. Das macht die Suche nicht nur einfacher änderbar, sondern insbesondere auch für Personen zugänglich, die nicht Programmieren können. Gerade im Umfeld von Bibliotheken-Software ist es immer eine grosse Herausforderung die Lücke zwischen Informatikern und Bibliothekaren zu schliessen. Es gibt nur wenige Leute, die ein ausgeprägtes Verständnis für beide Domänen haben. Mit unserer Komponente denken wir, dass wir die Konfiguration der Suche auch für Nicht-Informatiker etwas zugänglicher gemacht haben. Das wird sich nicht zuletzt in einem besseren Benutzererlebnis für die Benutzer der Suchschnittstelle niederschlagen.
Unsere Komponente deckt zwar noch nicht die komplette Elasticsearch DSL ab, kann aber sehr einfach um zusätzliche Queries oder Optionen erweitert werden. Die konkrete Suche kann über Parameter und Modifiers bereits stark dynamisch gestaltet werden. Durch unsere Komponente ist es nun möglich Abfragen auf den Elasticsearch Server in einer komplett objektorientierten Art und Weise zu gestalten. Das erhöht nicht nur den Komfort in der Entwicklung, sondern auch die Chancen auf eine fehlerfreie Implementierung.

6 Testing und CI

Um die Funktionsfähigkeit unserer Software sicher zu stellen verwenden wir eine Kombination von Unit Tests, System- und Integrationstests sowie einen Continuous Integration Server für die automatisierte Ausführung der Tests.

6.1 Continuous Integration

Als Continuous Integration Server wird, der für Open Source Projekte freie Server, **Travis CI**[23] verwendet. Die genaue Konfiguration des Builds befindet sich im Root-Verzeichnis der Applikation in der Datei *.travis.yml*.
Der Build wird bei jedem Commit in eines unserer GitHub Repositories ausgeführt und beinhaltet folgende Schritte:

1. Installation des Projektes und allen Abhängigkeiten mit Composer
2. Prüfung des Code-Style Standards PSR-2
3. Ausführung der Unit Tests mit Code Coverage
4. Ausführung der Integrations- und Systemtests mit Code Coverage

Der aktuelle Build kann unter folgenden URLs für unsere zwei Projekte eingesehen werden:

- https://travis-ci.org/linked-swissbib/hydra-swissbib.ch
- https://travis-ci.org/linked-swissbib/adapterElasticsearch

6.2 Coverage

Um sicher zu stellen, dass unsere Tests möglichst den ganzen Source Code abdecken, setzen wir das ebenfalls freie Tool **Coveralls**[24] ein um die Code Coverage zu messen.

Aktuelle Statistiken zur Coverage sind unter folgenden URLs einsehbar:

- https://travis-ci.org/linked-swissbib/hydra-swissbib.ch
- https://travis-ci.org/linked-swissbib/adapterElasticsearch

6.3 Unit Tests

Für das Erstellen der Tests wird das bekannte PHP Framework **PHPUnit**[25] verwendet. Die Tests befinden sich im Root-Verzeichnis der Applikation unterhalb des Ordners *tests*. Die Konfiguration für die Ausführung der Tests befindet sich in der Datei *phpunit.xml* ebenfalls im Root-Verzeichnis der jeweiligen Applikation. Mit Hilfe dieser Konfiguration können die Tests über die Konsole, IDE oder auf dem CI Server ausgeführt werden.

Bei einem Unit Test steht immer das Testen einer einzelnen Klasse im Fokus. Bei der Programmierung wird darauf geachtet, dass alle Klassen Ihre Abhängigkeit zu anderen Objekten per Dependency Injection erhalten. Das macht es einfach einzelne Klassen zu testen indem alle Abhängigkeiten zu anderen Objekten über Mock-Objekte[26] erfüllt werden. PHPUnit selbst bringt bereits Mocking-Funktionalität mit sich.
Nachfolgend wird ein einfaches Beispiel für das Testen einer Klasse mit Hilfe eines Mock-Objekts gezeigt:

[23] https://travis-ci.org
[24] https://coveralls.io
[25] https://phpunit.de/
[26] https://en.wikipedia.org/wiki/Mock_object

```php
class SimpleEntityBuilderTest extends TestCase
{
    /**
     * @var EntityBuilder
     */
    private $entityBuilder;

    /**
     * @var MockObject
     */
    private $mockLogger;

    /**
     * @return void
     */
    public function setUp()
    {
        $this->mockLogger = $this->getMockBuilder(Logger::class)
            ->disableOriginalConstructor()
            ->getMock();

        $this->entityBuilder = new SimpleEntityBuilder($this->mockLogger);
    }

    /**
     * @return void
     */
    public function testNotExistingEntity()
    {
        $this->mockLogger->expects($this->once())->method('error');

        $entity = $this->entityBuilder->build(
            'Tests\\LinkedSwissbibBundle\\NonExistingEntityName',
            ['attrA' => 'attrAValue']
        );

        $this->assertNull($entity);
    }
}
```

6.4 Integrations- und Systemtests

Im Gegensatz zu den Unit Tests sollen die Integrations- und Systemtests sicherstellen, dass die einzelnen Klassen korrekt zusammenarbeiten und das System als Ganzes seine Anforderungen erfüllt.
Für das Erstellen unserer Tests verwenden wir das PHP Framework **Behat**[27]. Behat ist ein Framework für die Implementierung von Tests nach dem Prinzip von Behaviour Driven Development[28]. Der Fokus liegt damit auf der Funktionsfähigkeit des Gesamtsystems sowie einer domain-nahen Sprache für die Tests. Behat selbst ist eine PHP spezifische Implementierung der Cucumber[29] Spezifikation.
Cucumber erlaubt es Tests in natürlicher Sprache zu schreiben. Mit Hilfe von Schlüsselwörtern innerhalb des Tests und entsprechenden Annotationen in PHP Klassen wird der natürlich-sprachige Text in auszuführende Anweisungen übersetzt.
Die Dateien für die Tests befinden sich im Root-Verzeichnis der Applikation im Ordner *features*.
Getestet werden alle für die API relevanten Endpunkte. Insbesondere werden auch Negativ-Tests durchgeführt, bei denen das Verhalten der API im Fehlerfall geprüft wird. Es wurden keine separaten Testszenarien definiert, da diese Integrationstests bereits ausreichend Szenarien beschreiben.
Tests werden immer innerhalb eines **Features** gruppiert. Für ein Feature kann es dann eine Reihe von **Scenarios** geben, die verschiedene Aspekte dieses Features testen. Ein einfacher Test für einen Web-Request sieht so aus:

```
Feature: Responses for documents
  In order to be able to retrieve documents the API should
  be able to respond with single documents or collections of documents.

  Scenario: Single document
    When I add "Accept" header equal to "application/ld+json"
    And I send a "GET" request to "/document/1"
    Then the response status code should be 200
    And the header "Content-Type" should be equal to "application/ld+json;"
    And the JSON should be equal to:
    """
    {
       "@context": "http:\/\/example.com\/contexts\/Document",
       "@id": "http:\/\/example.com\/document\/000000051",
       "@type": "http:\/\/purl.org\/ontology\/bibo\/document",
       "id": "000000051",
       "local": [
          "OCoLC\/775794624",
          "ABN\/000300043"
       ],
       "contributor": [
          "http:\/\/d-nb.info\/gnd\/1046905-9",
          "http:\/\/data.swissbib.ch\/agent\/ABN"
       ],
       "issued": "2016-04-26T08:41:49.227Z",
       "modified": "2014-08-14T16:40:57+01:00",
       "primaryTopic": "http:\/\/data.swissbib.ch\/resource\/000000051\/about"
    }
    """
```

Im Rahmen der Integrations- und Systemtests wird jeweils die gesamte Symfony Applikation oder bestimmte Module davon initialisiert und getestet.

[27] http://behat.org/en/latest/guides.html
[28] http://behaviourdriven.org/
[29] https://cucumber.io

6.5 Lasttests

Auf Last- sowie Performance-Tests wurde, in Absprache mit dem Kunden, aus zeitlichen Gründen verzichtet. Während der Entwicklung wurde immer darauf geachtet möglichst performante Lösungen umzusetzen. So werden z.B. Context-Informationen, die vom Elasticsearch Server abgefragt werden auf Seite unserer Applikation in einen lokalen Cache geschrieben (siehe ContextMapper im Dokument *Developer Guide IP-516bb.docx*). Oder bei der Serialisierung in die verschiedenen RDF-Formate haben wir darauf geachtet, dass der JSON-LD Context nicht über einen separaten Request abgefragt werden muss, sondern direkt eingebettet wird. Bezüglich Lasttests gilt es zu sagen, dass unsere Applikation problemlos auf mehreren Servern parallel betrieben werden könnt. Es gibt keine Synchronisationsprobleme, da wir rein lesenden Zugriff auf die Daten erlauben. Die API würde also problemlos skalieren bei sehr vielen Requests sofern das Hosting Setup skaliert (z.B. AWS Cloud).

7 Datenmodell

Die REST API besitzt 6 unterschiedliche Entitäten:
Document, Item, Organisation, Person, Work und BibliographicResource.

Folgende Grafik zeigt alle Felder der einzelnen Entitäten und wie sie sich untereinander referenzieren.

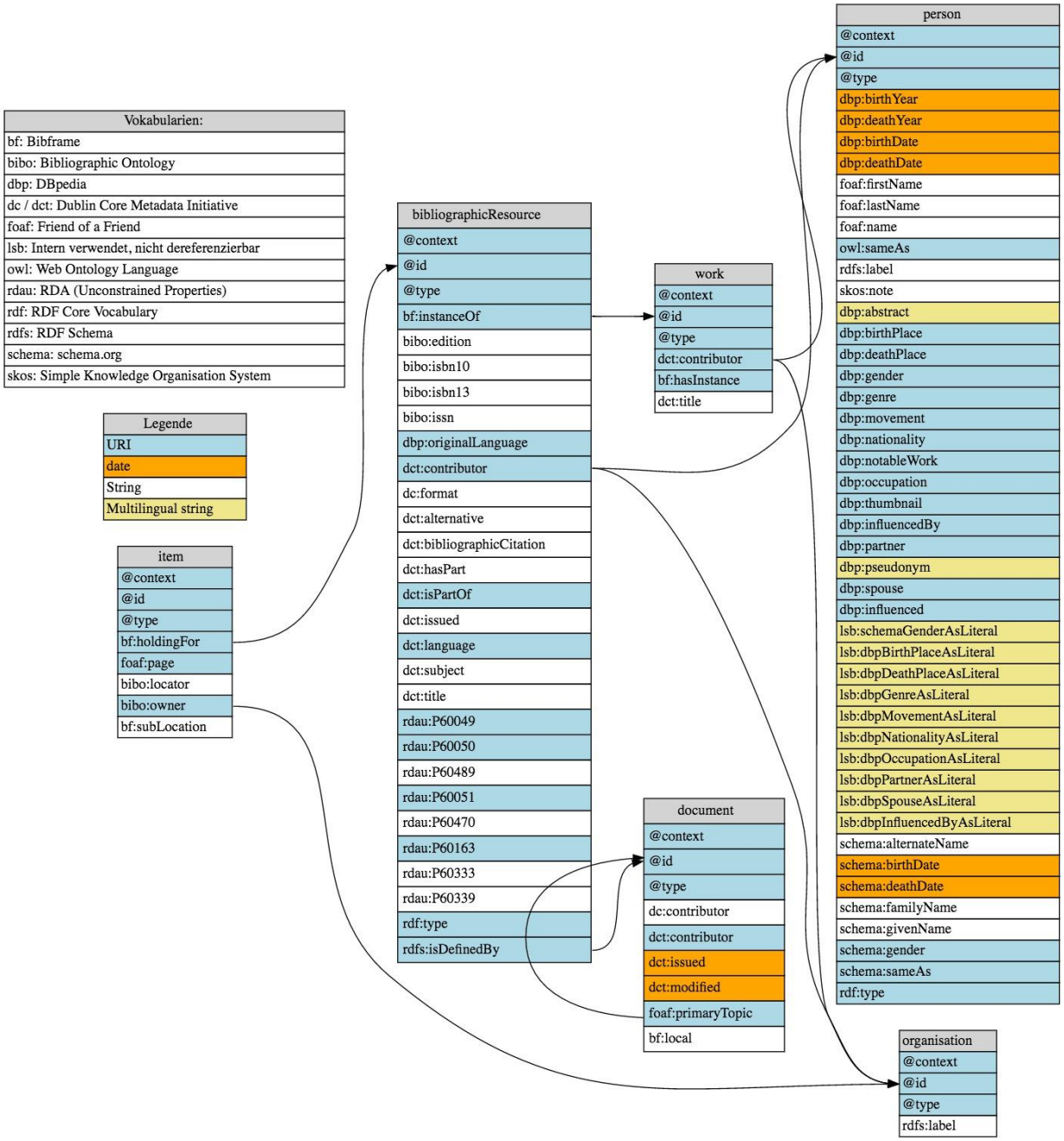


Abbildung 8: Datenmodel

8 REST API

8.1 Wieso ein REST Service?

REST (REpresentational State Transfer), ein Programmierparadigma für verteilte Systeme, hat sich in den letzten Jahren gegenüber anderen Ansätzen (z.B. SOAP) aus verschiedenen Gründen durchgesetzt. REST ist leichtgewichtig, skalierbar, einfach verständlich für Menschen und auch relativ einfach zu implementieren für Clients in jeglichen Programmiersprachen. Die Grundsätze eines REST Service sind vereinfacht:

8.1.1 Client – Server

Es gibt eine klare Trennung zwischen Client und Server. Der Client steuert die Interaktionen und der Server verarbeitet Daten oder liefert diese aus.

8.1.2 Stateless

Der Server benutzt nur Daten aus einem einzelnen HTTP Request um die Anfrage zu beantworten und nutzt keine weiteren Kontextinformationen.

8.1.3 Cacheable

Der Inhalt einer Server-Antwort muss implizit oder explizit als cachebar oder nicht cachebar gekennzeichnet sein.

8.1.4 Layered System

Für Server und Client ist transparent ob das Gesamtsystem aus mehreren Abstraktionsschichten besteht oder nicht

8.1.5 Uniform interface

- Jede Ressource (z.B. eine bibliographische Ressource) ist über eine URL ansprechbar.
- Eine Ressource kann syntaktisch und semantisch verstanden werden anhand der enthaltenen Daten und Metadaten aus einer Antwort. Das beinhaltet unter anderem die Information in welchem Format (z.B. JSON-LD) die Antwort ist.
- Wenn ein Client eine Ressource abgefragt hat, hat er damit genug Informationen um die Ressource zu bearbeiten oder zu löschen (vorausgesetzt er hat das Recht dazu).
- HATEOAS (Hypermedia as the Engine of Application State)[30]

[30] Siehe Kapitel HATEOAS Implementierungen

9 Interaktion mit der Schnittstelle

Eine klare Definition der zur Verfügung stehenden Funktionen und unter welcher URI sie zu finden sind ist für eine Schnittstelle essentiell.
Nachfolgende Kapitel klären die Frage *wie mit der neu erstellten Rest Schnittstelle interagiert werden kann.*

9.1.1 Content Negotiation

Alle in dem Kapitel 9 aufgeführten Ressourcen (Ausnahme */docs*) werden beim Aufruf im Browser standardmässig als HTML, angereichert mit RDFa Ansicht gerendert.

Möchte man ein anderes RDF Format erhalten, hat man dafür zwei Optionen:

1. Das Format kann mit *URL.format* ausgewählt werden, z.B. /document.jsonld
2. Das gewünschte Format kann über den «Accept Header» gewählt werden

Folgende Formate stehen zur Verfügung.

Format	URL Ergänzung	Accept header
JSON-LD	.jsonld	application/ld+json
RDF/XML	.rdfxml	application/rdf+xml
N-Triples	.ntriples	application/n-triples
Turtle	.turtle	text/turtle
HTML	.html	text/html

Tabelle 1: Content Negotiation über accept header oder per URL

Weitere Beschreibungen der angebotenen RDF Formate finden Sie unter Kapitel RDF Formate.

9.1.2 Einstiegspunkt

Der Einstiegspunt für die Hypermedia getriebene REST API befindet sich unter

http://data.swissbib.ch/.

9.1.3 Dokumentation

Unsere Schnittstelle bietet 2 unterschiedliche Dokumentationen an.

Swagger:
Mit einer Swagger-fähigen API garantieren wir eine interaktive Dokumentation, eine SDK Generierung des Clients und eine hohe Auffindbarkeit. [3]

http://data.swissbib.ch/docs

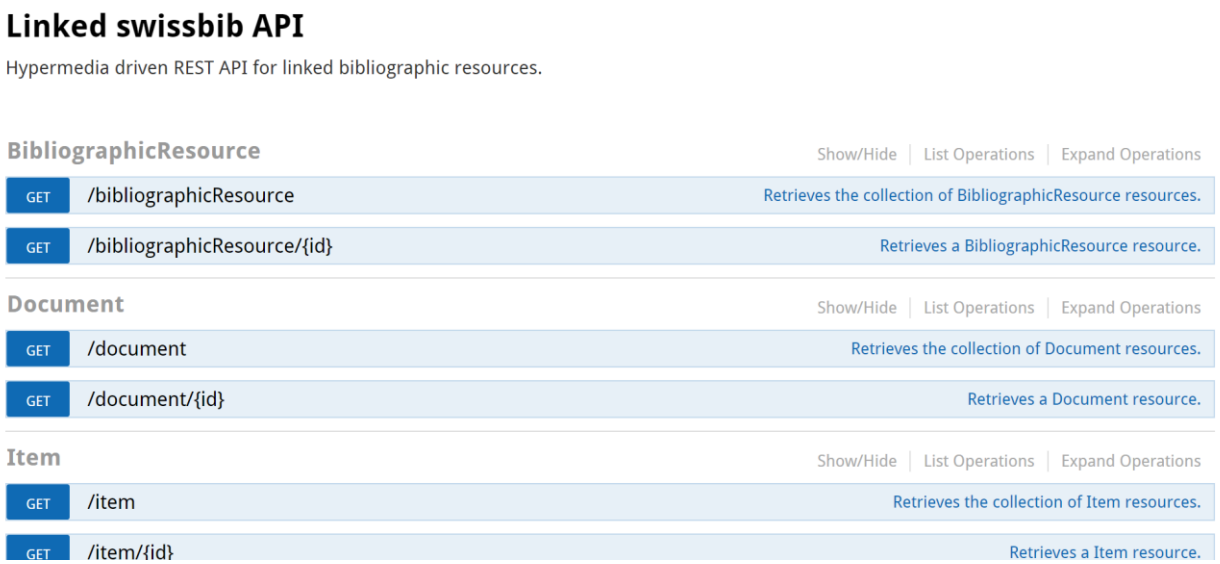


Abbildung 9: Ausschnitt aus Abfrage data.swissbib.ch/docs

HydraDoc:

Hydra[31] macht die Dokumentation vollständig maschinell verarbeitbar. Dadurch, dass alle Definitionen per URL identifizierbar sind, weisen sie eine sehr hohe Wiederverwendbarkeit auf. [4]

http://data.swissbib.ch/docs.jsonld


```
{
  - @context: {
        @vocab: "http://data.swissbib.ch/docs.jsonld#",
        hydra: "http://www.w3.org/ns/hydra/core#",
        rdf: "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
        rdfs: "http://www.w3.org/2000/01/rdf-schema#",
        xmls: "http://www.w3.org/2001/XMLSchema#",
        owl: "http://www.w3.org/2002/07/owl#",
      - domain: {
            @id: "rdfs:domain",
            @type: "@id"
        },
      - range: {
            @id: "rdfs:range",
            @type: "@id"
        },
      - subClassOf: {
            @id: "rdfs:subClassOf",
            @type: "@id"
        },
      - expects: {
            @id: "hydra:expects",
            @type: "@id"
        },
      - returns: {
            @id: "hydra:returns",
            @type: "@id"
        }
    },
@id: "http://data.swissbib.ch/docs.jsonld",
hydra:title: "Linked swissbib API",
hydra:description: "Hypermedia driven REST API for linked bibliographic resources.",
hydra:entrypoint: "http://data.swissbib.ch/",
hydra:supportedClass: [
  - {
        @id: "http://purl.org/ontology/bibo/document",
        @type: "hydra:Class",
        rdfs:label: "Document",
        hydra:title: "Document",
      - hydra:supportedProperty: [
          - {
                @type: "hydra:SupportedProperty",
              - hydra:property: {
                    @id: "#Document/id",
                    @type: "rdf:Property",
                    rdfs:label: "id",
                    domain: "http://purl.org/ontology/bibo/document",
                    range: "xmls:string"
                },
                hydra:title: "id",
                hydra:required: false,
                hydra:readable: true,
                hydra:writable: true,
                hydra:description: "identifier of a document"
            },
          - {
                @type: "hydra:SupportedProperty",
              - hydra:property: {
```


Abbildung 10: Ausschnitt aus der Abfrage data.swissbib.ch/docs

[31] Siehe Kapitel Was ist Hydra (hypermedia-driven web APIs)?

9.1.4 Collections / Suche

Die API bietet die Möglichkeit eine Collection/Sammlung von Daten abzurufen. So ist es z.B. möglich alle vorhandenen bibliographischen Ressourcen abzufragen.

Die Collections sind unter folgender URL zu finden *data.swissbib.ch/nameOfResource.*

http://data.swissbib.ch/bibliographicResource
http://data.swissbib.ch/document
http://data.swissbib.ch/item
http://data.swissbib.ch/organisation
http://data.swissbib.ch/person
http://data.swissbib.ch/work

9.1.4.1 Filtermöglichkeiten

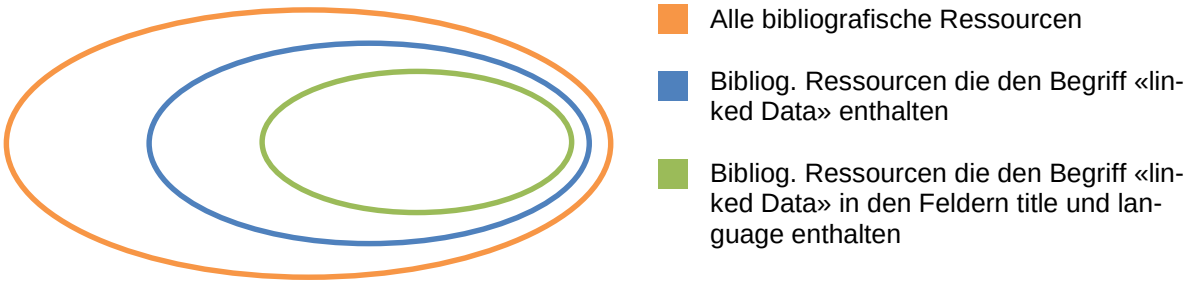
Die Rest API bietet 2 Möglichkeiten die Datenmenge du filtern.

1. Filtern nach Wörtern
   Es ist möglich die gewünschte Collection nach bestimmten Wörtern zu filtern. Die gewünschten Wörter können als GET Parameter *q (Query)* der URL angehängt werden.

   Beispiel:
   http://data.swissbib.ch/bibliographicResource.jsonld?q=linked+data
   → Alle bibliographischen Ressourcen werden nach «linked data» durchsucht und falls gefunden angezeigt.

2. Filtern nach Wörtern in bestimmten Feldern
   Die Resultatmenge kann weiter gefiltert werden, wenn bestimmte Felder angegeben werden, die das Suchworts enthalten sollen. Die entsprechenden Felder können mit dem GET Parameter *fields* definiert werden.

   Beispiel:
   http://data.swissbib.ch/bibliographicResource?q=linked+data&fields=title,language
   → In den Feldern title und language der bibliographischen Ressourcen wird nach «linked data» gesucht.



Siehe auch Kapitel Parameter

9.1.4.2 Pagination

Da bei einer Abfrage einer Collectionsehr viele Daten zurückkommen können, besitzt die API eine Pagination.
Die API gibt schrittweise immer 20 Entitäten zurück (die Anzahl kann in den Konfigurationen geändert werden). Die Pagination wird mit Hydra Konzepten realisiert (siehe Kapitel hydra:collection).

9.1.5 Eintrag mit bekannter ID

Die API bietet die Möglichkeit Entitäten mit einer bestimmten Id direkt abzufragen.
Die URI stellt sich wie folgt zusammen

data.swissbib.ch/*type/id.*

Beispiel: http://data.swissbib.ch/bibliographicResource/314420207
→ Die bibliographische Ressource mit der Id 314420207 wird zurückgegeben

9.1.5.1 Error Handling

Verständliche Fehlermeldungen und Status Codes machen ein gutes Errorhandling aus.
Wir nutzen dafür Hydra:Error.
Hydra[32] bietet von sich aus ein definierter Umgang mit Fehlermeldungen an. Sie beschriebt wie eine Fehlermeldung aussehen soll und welche Attribute sie haben soll. Hydra wandelt also eine nicht abgefangene Fehlermeldung in das gewünschte Standardformat um und gibt sie so dem Client weiter. [33]

```
HTTP/1.1 400 Bad Request
Content-Type: application/ld+json

{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@type": "Error",
  "title": "An error occurred",
  "description": "Typically, a specialization of this class is used in practice.",
  ...
}
```

Abbildung 11: Hydras Error Klasse um Error zu beschreiben

In unserer API findet man den Error context unter

http://data.swissbib.ch/contexts/Error

9.2 Diskussion

9.2.1 Einzahl vs. Mehrzahl

Während der Definition der URIs stellte sich uns die Frage, ob wir die Ressourcen im Plural oder Singular ansprechen.
Es stellte sich heraus, dass dies ebenfalls eine grosse Diskussion unter den Rest Schnittstellen Entwicklern im Internet ist. Die Rest Naming Convention sowie die Hydra Convention definiert keinen allgemein gültigen Standard.

Beispiele:

data.swissbib.ch/bibliographicResource	vs	data.swissbib.ch/bibliographicResources
data.swissbib.ch/bibliographicResource/1	vs	data.swissbib.ch/bibliographicResources/1

[32] Siehe Kapitel Was ist Hydra (hypermedia-driven web APIs)?
[33] Weitere Informationen unter http://www.hydra-cg.com/spec/latest/core/#description-of-http-status-codes-and-errors

Folgende Gründe für die einzelnen Varianten haben wir gefunden.

Gründe für Einzahl:
- Sehr verbreitet beim Arbeiten von Linked Data
- Linked Data Guidelines geben vor, dass die Ressourcen im Singular angesprochen werden sollen, ausser sie existiert nur in der Mehrzahl [5]
- Wenn über eine ID eine Ressource abgefragt wird kommt ein einzelner Eintrag zurück

Gründe für Mehrzahl:
- URLs bleiben konsistent in allen HTTP Methoden
- Am meisten verbreitete Praxis bei REST Schnittstellen
- In einer Abfrage auf eine Collection kommen mehrere Objekte zurück

Es zeigt sich, dass für das Arbeiten mir REST APIs Ressourcen im Plural bevorzugt werden und beim Arbeiten mit Linked Data empfohlen wird mit dem Singular der Ressource zu arbeiten.

Nach Absprache mit dem Kunden haben wir uns fürs Singular entschieden, da der Fokus dieser Schnittstelle auf der Arbeit mit Linked Data liegt.

9.2.2 Gleichnamige Attribute mit mehreren Vokabularen

Die API Plattform sieht vor, dass die jeweiligen Vokabulare der Ressourcenattribute wie in Abbildung 12 über die Annotation ApiProperty definiert werden können.

```
/**
 * @var string|array
 *
 * @ApiProperty(iri="http://purl.org/dc/terms/contributor",writable=false, attributes={
 *     "jsonld_context"={
 *         "@type"="@id"
 *     }
 * })
 */
private $contributor;
```

Vokabular von

Abbildung 12: Bestimmung des Vokabulars über die Annotation @ApiProperty

Mit der gegebenen Definition von API Plattform kann pro Attribute nur genau ein Vokabular angegeben werden.

Im Datenbestand des Kunden gibt es jedoch Entitäten die gleichnamigen Attribute mit unterschiedlichen Vokabularen besitzen.
Ein Beispiel sind die contributor Attribute in der document Entity. Ein Contributor soll vom Vokabular *dc* und der andere von *dct* definiert sein. Zudem unterscheiden sie sich im Typ. Ein Contributor soll eine Verlinkung sein (@id) und der zweite ein Text (@literal).

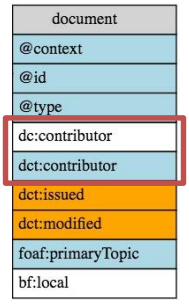


Abbildung 13: Entität document mit Attributen

Da es mit unserer Implementierung nicht möglich ist diese Daten so umzusetzten, haben wir folgende Punkte diskutiert:

Entspricht es dem LinkedData Konzept gleichnamigen Attribute mit unterschiedlichen Vokabularen zuzulassen?
Linked Data baut darauf auf, dass alle Ressourcen eindeutig zugewiesen werden können und auch für einen maschinellen Client ohne Konflikt interpretiert werden kann. Gibt es zwei gleichnamige Informationsquellen muss der Client gezwungenermasse das genaue

Vokabular des gesuchten wissen. Die Idee des Vokabulars ist jedoch die Informationsquelle genauer zu definieren undzu spezifizieren – soll also zusätzliche Informationen bieten.

Sind beide Attribute überhaupt notwendig?
Bei der Untersuchung der vorhandenen Datensätze, fiel uns auf, dass wenige Einträge wirklich beide Attribute gesetzt haben. Viele haben entweder den verlinkten Contributor oder den einfachen Contributor mit Textinhalt definiert.
Nach Absprache mit dem Kunden stellte sich heraus, dass der Contributor von ihrer Seite her eine Verlinkung sein sollte, den nur mit einem Text ist es einem Client nicht möglich weitere Informationen abzufragen.

Somit wurde entschieden den Contributor im *document* sowie die anderen doppelten Attribute zu entfernen. Somit wurde das Attribut, das einen Link enthält beibehalten und das textuelle Attribut entfernt, unabhängig davon zu welchem Vokabular sie gehören.

9.3 Fazit

Die Ressourcen können mithilfe unserer Rest API eindeutig identifiziert und abgerufen werden. Die URIs wurden aufgrund der oben erwähnten Gründe auf Namen im Singular aufgebaut. Dadurch, dass wir einen klaren Einstiegspunkt definiert haben sind die unterschiedlichen Entitäten leicht zu finden.
Um die Abfrage auf die Ressourcen zu individualisieren bieten wir 2 Möglichkeiten an um das Suchresultat zu filtern. Wenn keine zusätzlichen Bedingungen für eine Abfrage gemacht werden, erhält der Client das Resultatset stückweise zurück um nicht überfordert zu werden.
Das Einpflegen der Attribute und ihren Vokabularen stellte uns vor grosse Herausforderungen. Schlussendlich konnten wir aber mit dem Kunden eine für ihn gute Lösung finden.
Einer der wichtigsten Punkte ist die Content Negotiation. Dadurch, dass man 2 Möglichkeiten hat das Format wählen zu können wir dem Client die Wahl überlassen, wie er seine Anfrage gestalten will.

10 Selbstbeschreibende Schnittstelle

Im nachfolgenden Kapitel wird auf die Anforderung eingegangen, dass die Schnittstelle selbstbeschreibend sein soll.
Es wird folgende Frage geklärt:
Wie stellen wir sicher, dass die Schnittstelle selbstbeschreibend ist?

10.1 HATEOAS Implementierungen

HATEOS bedeutet ausgeschrieben: Hypermedia As The Engine Of Application State.

HATEOAS ist eine der wichtigsten Anforderungen einer REST API und doch ist sie am meisten vernachlässigte.
Das IT Magazin heise Developer schrieb Anfangs Dezember sogar, dass REST APIs erst mit HATEAOAS den höchsten Reifegrad erreicht haben und es daher unverzichtbar ist HATEOAS in REST Applikationen anzubieten. [6]

Doch was sagt HATEOAS den überhaupt?
HATEOAS sagt, dass die API selbstbeschreibend sein muss. Das heisst ein maschineller Client sollte über einen einzigen Einstiegspunkt in der Lage sein zu verstehen welche Aktionen auf welchen Ressourcen über die API ausgeführt werden können. Abstrakt betrachtet stellt ein HATEOAS-konformer REST-Service einen endlichen Automaten dar, dessen Zustandsveränderungen durch die Navigation mittels der bereitgestellten URLs erfolgen.

10.2 Was ist Hydra (hypermedia-driven web APIs)?

REST selbst schreibt keinen Standard vor, wie eine selbstbeschreibende API aufgebaut sein soll. Genau an diesem Punkt setzt Hydra, mit dem Versuch das Vokabular für Hypermedia-getriebene Web APIs zu standardisieren, an.
Das Hydra Core Vocabulary ist noch ein inoffizieller Entwurf, der von Google Mitarbeiter Markus Lanthaler vorangetrieben wird und sich besonders im Linked Data Umfeld bereits grosser Beliebtheit erfreut.
Das gesamte Vokabular ist schon relativ umfangreich. Um das Prinzip zu veranschaulichen möchten wir hier deshalb zwei Beispiele genauer betrachten.

10.2.1 hydra:entrypoint

Der Hydra:Entrypoint ist der Startpunkt jeder API.
Alle weiterführenden Links zu Ressourcen sind im Entrypoint zu finden.

Der Einstiegspunkt unserer API befindet sich unter data.swissbib.ch und liefert das abgebildete Resultat in Abbildung 14.

Abbildung 14: hydra:entrypoint

10.2.2 hydra:collection

Eine Hydra:Collection ist die Sicht auf eine Menge von zusammenhängenden Ressourcen. In unserem Fall könnte das eine Liste der Suchresultate für bibliographische Ressourcen sein. Da eine solche Menge von Ressourcen sehr schnell sehr gross werden kann gibt es im Hydra Vocabulary die Hydra:PartialCollectionView. Diese beschreibt eine Sicht auf eine Untermenge aller Elemente einer Collection.

In einer Hydra:PartialCollectionView sind unter anderem folgende Links enthalten:
hydra:member: Hier befinden sich die Ressourcen selbst.
hydra:first: Erste zur Verfügung stehende Seite mit Ressourcen
hydra:last: Letzte Seite mit Ressourcen
hydra:next: Unter diesem Link gelangt man auf die nächste Seite und somit auf die nächsten Ressourcen
hydra:previous: Unter diesem Link gelangt man auf die vorherige Seite

hydra:previous und hydra:next werden nur angezeigt wenn man sich nicht auf der letzten beziehungsweise der ersten Seite befindet.

Mithilfe dieser PartialCollectionView lässt sich also eine sogenannte Pagination aufbauen.
In der Abbildung 15 ist eine PartialCollectionView mit ihren Elementen ersichtlich.

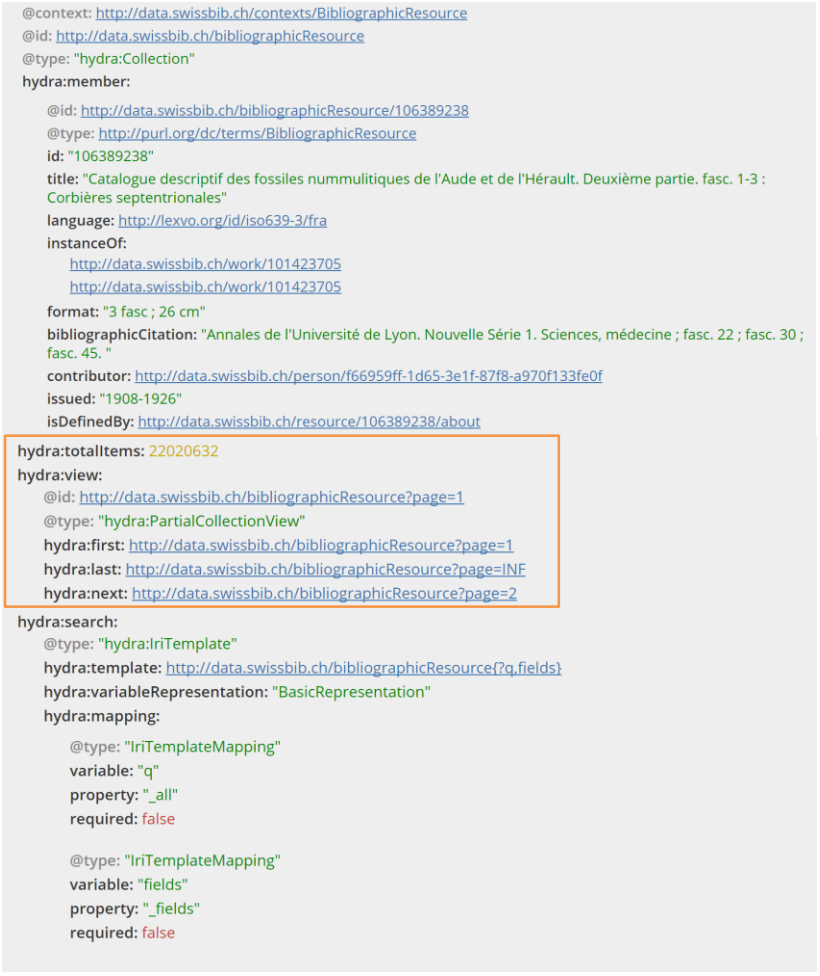


Abbildung 15: Die Ausgabe einer hydra:collection des swissbib REST API Prototyps

Diese Standardisierung mittels des Hydra Core Vocabulary erlaubt es nun einen generischen Client zu schreiben, der die möglichen Interaktionen mit der Web API auflisten kann.

10.3 Alternativen zu Hydra

Hypertext Application Language (HAL) stellt sich als spannende Alternative zu Hydra heraus. Auch sie verlinkt Ressourcen untereinander, bietet eine generierte Dokumentation und lässt sich leicht in eine REST Schnittstelle integrieren.
HAL definiert verschiedene Konventionen um Hyperlinks und JSON oder XML anzubieten. [7]
Der Autor Mike Kelly preist HAL als menschen- sowie computerfreundlich an.

In der Abbildung 16 finden Sie ein Einblick in ein Beispiel Code.
Man erkennt einige Gemeinsamkeiten mit Hydra.
Zum Beispiel wiederspiegelt das Attribut *self* den Link zur momentanen Ressource, diesen finden wir in Hydra unter *id*. Der Namespace ns wird in *curries* beschrieben, in Hydra werden die Namespaces unter *context* angegeben.

```
{
    "_links":{
        "self":{
            "href":"/book/123"
        },
        "curries":[
            {
                "name":"ns",
                "href":"http://booklistapi.com/rels/{rel}",
                "templated":true
            }
        ],
        "ns:authors":[
            {
                "href":"/author/4554",
                "title":"Leonard Richardson"
            },
            {
                "href":"/author/5758",
                "title":"Mike Amundsen"
            },
            {
                "href":"/author/6853",
                "title":"Sam Ruby"
            }
        ]
    },
    "Title":"RESTful Web APIs",
    "Price":"$31.92",
    "Paperback":"408 pages",
    "Language":"English"
}
```

Abbildung 16: HAL Code Beispiel [8]

10.4 Fazit

Wir implementieren in unserer Schnittstelle das HATEOAS Konzept mit Hilfe von Hydra und können dadurch gewährleisten, dass der Client einen klaren Einstiegspunkt kennt und von diesem selbst die unterschiedlichen Ressourcen problemlos findet.
Hydra ist deshalb für unsere Applikation um einiges besser geeignet als HAL, da Hydra selbst im Linked Data Format aufgebaut ist.

11 RDF Formate

RDF = Resource Description Framework

Das Resource Description Framework fasst verschiedene verlinkte Formate zusammen. Sie alle repräsentieren dabei Informationen, die eindeutig durch eine Internetadresse (URI) identifizierbar ist.
Im RDF werden die Informationen zu Aussagen aus drei Elementen dargestellt; einem Subjekt, einem Prädikat und einem Objekt. Das Subjekt und das Prädikat sind immer Ressourcen. Das Objekt hingegen kann entweder eine Ressource oder ein Literal sein. RDF-Ressourcen sind eindeutig durch eine URI identifizierbar.
RDF Dateien sind XML basierte Textdateien.

Nachfolgend werden 5 RDF Formate genauer erläutert. Es sind die 4 Formate die unsere API dem Client anbietet.
Zu jedem Format ist angegeben wie die 3 grundlegenden Attribute Context, Id und Type angegeben werden. Zudem wird jeweils ein Beispiel diskutiert um Subjekt, Prädikat und Objekt unterscheiden zu können.

11.1.1 JSON LD

Die Daten im JSON-LD Format sind in leichtgewichtigen JSON geschrieben. Der grosse Vorteil von JSON-LD gegenüber den anderen Formaten ist, dass viele Services bereits mit JSON Daten kommunizieren. Eine Umstellung von JSON zu JSON-LD ist schnell implementierbar, da die beiden reibungslos zusammenarbeiten.

Context	Id	Type
Link zu den Kontexten: @context: «…/contexts/Work	@id	@type


```
{
    @context: "http://data.swissbib.ch/contexts/Work",
    @id: "http://data.swissbib.ch/work/025316931",
    @type: "http://bibframe.org/vocab/Work",
    id: "025316931",
  - hasInstance: [
        "http://data.swissbib.ch/bibliographicResource/025316931",
        "http://data.swissbib.ch/bibliographicResource/061176214"
    ],
  - contributor: [
        "http://data.swissbib.ch/person/8f39768b-6dd8-3381-bf36-5020d02ba067",
        "http://data.swissbib.ch/person/d51411e7-1a96-31e7-b751-bcb27b8f2e43"
    ],
  - title: [
        "Die Lage von 100 Bergbauernfamilien im Bezirk Einsiedeln und den Gemeinden Alpthal und Unteryberg",
        "Die Lage von 100 Bergbauernfamilien im Bezirk Einsiedeln und den Gemeinden Alphthal und Unteryberg"
    ]
}
```


Abbildung 17: Abfrage auf data.swissbib.ch/work/025316931.jsonld

Subjekt	Prädikat	Objekt
@id: http://.../work/025316931	hasInstance	«http://data.swissbib.ch/resource/025316931, http://data.swissbib.ch/resource/061176214»

11.1.2 RDF/XML

RDF/XML ist wie der Namen vermuten lässt XML basiert. RDF/XML war das erste Serialisierungsformat, dass von W3C entwickelt wurde. Heute wird es wird immer seltener genutzt um Daten zu serialisieren. Dies aus dem Grund, dass viele andere RDF Formate mehr «humen-friendly» sind.

Context	Id	Type
<rdf:RDF … >	Attrbute rdf:about	<bf:Work …> <*type:Subjekt* …>

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:bf="http://bibframe.org/vocab/"
         xmlns:ns0="http://data.swissbib.ch/docs.jsonld##Work/"
         xmlns:dc="http://purl.org/dc/terms/">
  <bf:Work rdf:about="http://data.swissbib.ch/work/025316931">
    <bf:hasInstance rdf:resource="http://data.swissbib.ch/bibliographicResource/025316931"/>
    <bf:hasInstance rdf:resource="http://data.swissbib.ch/bibliographicResource/061176214"/>
    <ns0:id rdf:datatype="http://www.w3.org/2001/XMLSchema#string">025316931</ns0:id>
    <dc:contributor rdf:resource="http://data.swissbib.ch/person/8f39768b-6dd8-3381-bf36-5020d02ba067"/>
    <dc:contributor rdf:resource="http://data.swissbib.ch/person/d51411e7-1a96-31e7-b751-bcb27b8f2e43"/>
    <dc:title rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Die Lage von 100 Bergbauernfamilien im
    Bezirk Einsiedeln und den Gemeinden Alpthal und Unteryberg</dc:title>
    <dc:title rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Die Lage von 100 Bergbauernfamilien im
    Bezirk Einsiedeln und den Gemeinden Alphthal und Unteryberg</dc:title>
  </bf:Work>

</rdf:RDF>
```

Abbildung 18: Abfrage auf data.swissbib.ch/work/025316931.rdfxml

Subjekt	Prädikat	Objekt
<bf:Work rdf:about =…>	<bf:hasInstance …>	Attribute rdf:resource= falls Prädikat ein Link ist oder Text mit dazugehörigem rdf:datartype

11.1.1 Turtle

Turtle, Terse RDF Triple Language, ist textbasiert. Es repräsentiert den RDF Graph textuell. [10]

Context	Id	Type
Mit @prefix gekennzeichnet	<http://.../work/025316931> Subjekt	bf:Work

```
@prefix bf: <http://bibframe.org/vocab/> .
@prefix ns0: <http://data.swissbib.ch/docs.jsonld##Work/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dc: <http://purl.org/dc/terms/> .

<http://data.swissbib.ch/work/025316931>
  a bf:Work ;
  bf:hasInstance <http://data.swissbib.ch/bibliographicResource/025316931>, <http://data.swissbib.ch/bibliographicResource/061176214> ;
  ns0:id "025316931"^^xsd:string ;
  dc:contributor <http://data.swissbib.ch/person/8f39768b-6dd8-3381-bf36-5020d02ba067>, <http://data.swissbib.ch/person/d51411e7-1a96-31e7-b751-bcb27b8f2e43> ;
  dc:title "Die Lage von 100 Bergbauernfamilien im Bezirk Einsiedeln und den Gemeinden Alpthal und Unteryberg"^^xsd:string, "Die Lage von 100 Bergbauernfamilien im Bezirk
Einsiedeln und den Gemeinden Alphthal und Unteryberg"^^xsd:string .
```

Abbildung 19: Abfrage auf data.swissbib.ch/work/025316931.turtle

Subjekt	Prädikat	Objekt
Erstes Element in <>	prefix:name	Drittes Element in <> oder Text im Format *«abc»^^dataTyp*

3er Paar immer abgeschlossen mit einem Semikolon.

11.1.2 N-Triples

N-Triples ist ein textbasierter RDF-Graph und eine Untermenge von Turtle.[9]

Context	Id	Type
Kontext im Objekt ausgeschrieben	Id ist gleichzeitig das Subjekt	Prädikat der ersten Aussage

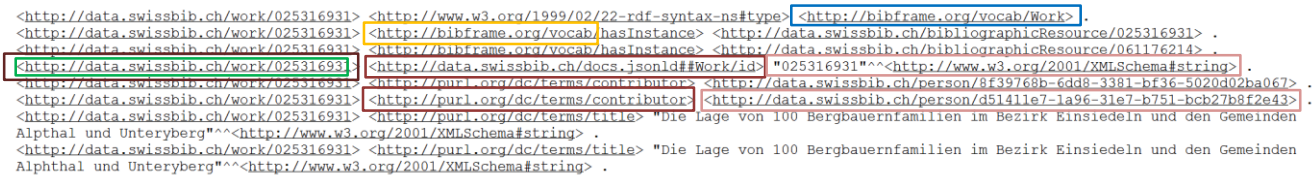


Abbildung 20: Abfrage auf data.swissbib.ch/work/025316931.ntriples

Die Ressource beschreibt sich im ersten 3er selbst.
<http://.../**work/025316931**> hat den http://../22-rdf-syntax-ns#**type** <http://…/vokab/Work>.

Die einzelnen 3er Paare sind mir einem Punkt voneinander getrennt.

Subjekt	Prädikat	Objekt
Erstes Element in <>	Zweites Element in <>	Drittes Element in <> oder Text im Format *«abc»^^dataTyp*

11.1.3 RDFa

RDFa ist eine Erweiterung von HTML5 um Ressourcen zu verlinken. Da RDFa auf HTML basiert kann es auch problemlos in HTML ähnlichen Dokumenten wie XHTML oder XML eingesetzt werden.
RDFa ermöglicht es uns die Daten für einen Menschen schön darzustellen. [11]

Context	Id	Type
Attribut vocab in div-Tag	Attribut resource in div-Tag	Attribut typeof div-Tag

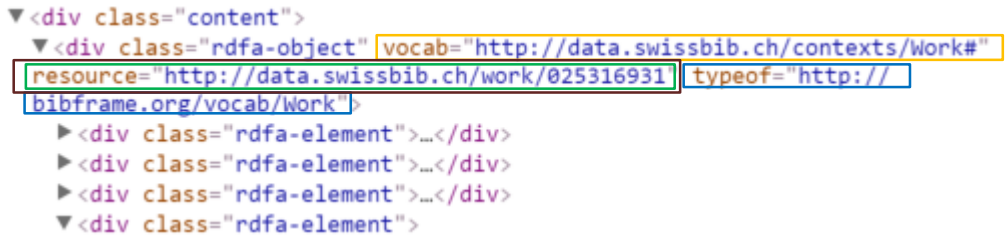


Abbildung 21: Ausschnitt aus Abfrage auf http://data.swissbib.ch/work/025316931 : Context, Id, Type

Abbildung 22: Ausschnitt aus Abfrage auf http://data.swissbib.ch/work/025316931 | Prädikat und Objekt

Abbildung 23: Abfrage auf http://data.swissbib.ch/work/025316931

Subjekt	Prädikat	Objekt
Attribut resource in div-Tag	Attribut property in a-Tag	Wert zwischen den entsprechenden a-Tags

11.2 Fazit

Alle Formate sind in der Lage die Daten darzustellen. Entscheidend für die Wahl des Formates ist, auf welchem «Basis-Format» aufgebaut werden soll. In eine HTML Umgebung lässt sich z.B. RDFa leicht einbinden, in einer JSON basierte Umgebung macht JSON-LD wenig Aufwand.
Der zweite grosse Unterschied ist der Umgang mit der Typisierung. Während einige nur den Objektnamen angeben und die Zuteilung des Vokabulars einem Kontext überlassen, schreiben andere den vollen Objektnamen samt Vokabular aus.
Und dadurch, dass Hydra auch in Linked Data umgesetzt ist, lassen sich auch ihre Konzepte in die unterschiedlichen RDF Formate umwandeln.Um die Content Negotiation in der Applikation umzusetzen wurde für jedes Format ein Encoder programmiert. Dieser wandelt die Daten mithilfe des easyRDFGraphen[34] um.

[34] http://www.easyrdf.org/docs/api/EasyRdf_Graph.html

12 Ähnliche Projekte

Natürlich gehört es bei einer neuen Entwicklung auch dazu, sich umzusehen wer schon ähnliche Applikationen auf dem Markt hat.
Eine solche Applikationen wird im folgenden Kapitel kurz beschrieben.

12.1 Lobid.org

Auch lobid.org bietet Zugang zu verlinkten Daten z.B. für bibliographic resources, libraries and organizations.
In der Diskussion der Anforderungen an unsere Rest API war lobid immer wieder ein Thema, da es viele Aspekte enthält die für swissbib auch wichtig waren.

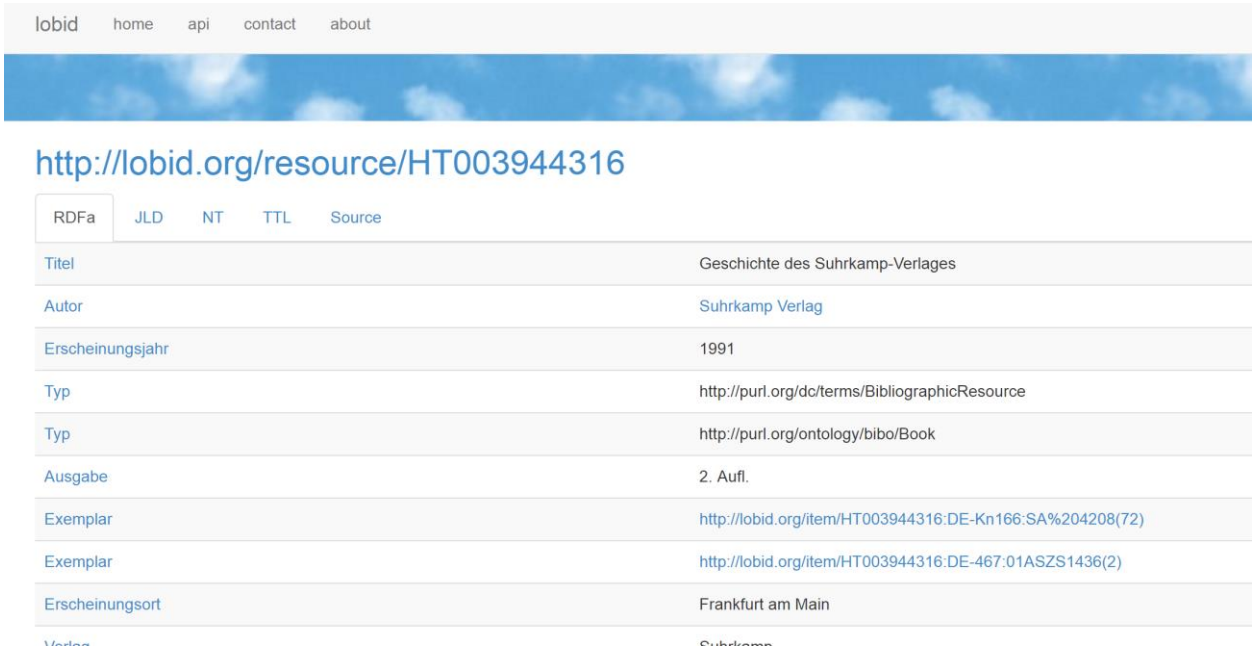


Abbildung 24: Abfrage auf http://lobid.org/resource/HT003944316/about

Jedoch haben wir uns für einige unterschiedliche Implementierungen entschieden.
Wie in der Abbildung 2421 zu sehen ist, bietet lobid mehrere RDF Formate auf der HTML Ausgabe an. Wir haben uns gegen eine solche Vermischung entschieden, da wir unter dem HTML Format auch nur HTML Repräsentierungen haben möchten.
Weiter unterscheidet sich unsere Ansicht indem, dass wir die Hydra Konzepte im HTML anzeigen; lobid verzichtet auf diese Erweiterungen.
Eine Gemeinsamkeit findet man in der URL. Wir haben uns wie auch lobid für einzahlige URLs entschieden.

13 Schlussfolgerung

Ziel des Projektes war es, die verlinkten Daten von swissbib über eine HATEOAS-konforme Schnittstelle anzubieten. Zu diesem Zweck wurden verschiedene Standards für HATEOAS analysiert und verglichen. Dabei ergab sich, dass das Vokabular von Hydra mit markanten Vorteilen hervorstach und insbesondere dadurch geeignet ist, da es selbst im Linked Data Format definiert ist. Dadurch wurde es uns möglich die Prinzipien von HATEOAS in allen RDF Serialisierungsformaten anzubieten.
Was die Anbindung des Elasticsearch Servers betrifft, so konnte eine neuartige Komponente geschaffen werden, die eine einfache Anpassung der Suchabfrage per Konfiguration erlaubt. Dadurch, dass dafür keine Programmierkenntnisse mehr notwendig sind, ist die Konfiguration der Suche für ein viel breiteres Publikum zugänglich.
Abschliessend kann festgestellt werden, dass die Prinzipien von Hydra und Linked Data sich ausgezeichnet ergänzen.
Wir sind überzeugt, dass wir mit unsere Linked Data REST Schnittstelle und dem Elasticsearch Adapter zwei Applikationen entwickeln konnten, die es so noch nicht gibt und die einen signifikanten Mehrwert bieten.

14 Literaturverzeichnis

[1] Universität Basel, „Swissbib,“ 27 09 2016. [Online]. Available: https://www.swissbib.ch/.

[2] N. Prongué, „swissbib.blogspot.ch,“ [Online]. Available: https://2.bp.blogspot.com/-ll97yQcSvLg/Vwy5wbpZeKI/AAAAAAAAAAY/lFlotTXjdjQk2HfDwFYbnIJm0tR8YJ8BwCLcB/s1600/overview2_oF.png. [Zugriff am 6 11 2016].

[3] «swagger,» 2016. [Online]. Available: http://swagger.io/. [Zugriff am 1 11 2016].

[4] „Hydra Core Vocabulary,“ 2016. [Online]. Available: http://www.hydra-cg.com/spec/latest/core/#documenting-a-web-api. [Zugriff am 1 11 2016].

[5] A. G. L. D. W. Group, „URI Guidelines for publishing Linked Datasets,“ 22 07 2016. [Online]. Available: https://github.com/AGLDWG/TR/wiki/URI-Guidelines-for-publishing-linked-datasets-on-data.gov.au-v0.1#h.blxw9q9b2rlf. [Zugriff am 03 09 2016].

[6] J. A. Andreas Würl, „heise Developer,“ 06 12 2016. [Online]. Available: https://www.heise.de/developer/artikel/Hoechster-Reifegrad-fuer-REST-mit-HATEOAS-3550392.html. [Zugriff am 07 12 2016].

[7] M. Kelly, „Hypertext Application Language,“ 18 08 2013. [Online]. Available: http://stateless.co/hal_specification.html. [Zugriff am 19 11 2016].

[8] D. Noranovich, „Introduction to Hypertext Application Language (HAL),“ 07 03 2015. [Online]. Available: https://dzone.com/articles/introduction-hypertext-0. [Zugriff am 19 11 2016].

[9] W3C, „RDF 1.1 Turtle,“ 24 02 2014. [Online]. Available: https://www.w3.org/TR/turtle/. [Zugriff am 19 11 2016].

[10] D. Beckett, „RDF 1.1 N-Triples,“ W3C, 25 02 2014. [Online]. Available: https://www.w3.org/TR/n-triples/. [Zugriff am 17 11 2016].

[11] „Linked Data in HTML,“ unbekannt. [Online]. Available: https://rdfa.info/. [Zugriff am 19 11 2016].

15 Glossar

Linked Data	Linked Data bezeichnet strukturierte Daten, die per URI eindeutig identifizierbar sind.
RDF	Resource Description Framework, formuliert Daten in einer logischen Aussage. RDF Daten sind in Subjekt, Prädikat und Objekt gegliedert.
REST	Representational State Transfer, beschreibt Anforderungen an eine Schnittstelle, insbesondere Webservices.
API	Application Programming Interface, ist die Abkürzung für einen Programmteil, der für die Anbindung in weitere System zur Verfügung gestellt wird.
PHP	Hypertext Preprocessor, ist eine Programmiersprache, die speziell für die Programmierung von Web-Inhalten geschaffen ist.
HTML	Hypertext Markup Language, ist eine textbasierte Auszeichnungssprache, die dazu verwendet wird Inhalte aus dem Internet mit Hilfe eines Browsers darzustellen.
CSS	Cascading Stylesheets, ist eine Sprache zur Definition des Aussehens von HTML-Dokumenten.
HTTP	Hypertext Transfer Protocol, ist ein Protokoll für den Austausch von Daten in einem Netz. Es findet insbesondere Anwendung im World Wide Web.
YAML	YAML ist eine einfache Auszeichnungssprache für die Serialisierung von Daten in einem Textformat.

16 Anhang

16.1 Quellcode

Der Quellcode des gesamten Projektes ist Open Source und auf GitHub[35] veröffentlicht. GitHub wurde bereits zu Beginn für die Kollaboration zwischen den Entwicklern benutzt.
Der Code ist auf zwei Repositories verteilt:

linked-swissbib/hydra-swissbib.ch
Repository der REST API mit all ihren Abhängigkeiten.
URL: https://github.com/linked-swissbib/hydra-swissbib.ch

linked-swissbib/adapterElasticsearch
Repository einer unabhängigen Komponente zur Definition von Elasticsearch Suchtemplates. Diese Komponente wird innerhalb des API Projekts, sowie weiteren swissbib Projekten verwendet. Weitere Informationen zu dieser Komponente können im Kapitel 5 Elasticsearch Adapter nachgelesen werden.
URL: https://github.com/linked-swissbib/adapterElasticsearch

16.2 Git Branching Strategie

Als Strategie für das Management dir Git Branches wird **git-flow** verwendet. Details über git-flow können hier nachgelesen werden: http://danielkummer.github.io/git-flow-cheatsheet/

16.3 Code Style

Als Code Style Standard wird **PSR-2**[36] verwendet.

16.4 Code Dokumentation

Die Dokumentation innerhalb des Source Code folgt der Spezifikation von PHPDoc[37].

16.5 IDE Setup und Debugger

Zur Entwicklung wurde die speziell auf PHP Bedürfnisse zugeschnittene IDE PhpStorm[38] zusammen mit dem Debugger XDebug[39] verwendet. Informationen zur Installation finden sich auf der Seite der Hersteller.

16.6 Installation und Deployment

Der Prozess für die vollständige Installation der Applikation ist mit Composer automatisiert. Es reicht den Befehl *composer install* im Root-Verzeichnis der Applikation auszuführen. Die Dokumentation dazu befindet sich im Projekt selbst in der Datei *development.md*.

[35] https://github.com/
[36] http://www.php-fig.org/psr/psr-2/
[37] http://www.phpdoc.de/
[38] https://www.jetbrains.com/phpstorm/
[39] https://xdebug.org/