Abschlussbericht Software Projekt

Raphael Matile, 12-711-222

14. August 2015

Zusammenfassung

Seit 2008 bietet RaiseNow AG Dienstleistungen im Bereich Digital Fundraising an. RaiseNow ist inzwischen ein führender Anbieter von Online-Fundraising-Lösungen. Der Fokus liegt auf leistungsstarken, benutzerfreundlichen Online-Fundraising-Tools, die einfach zu bedienen und anzuwenden sind. Die Philosophie: Hervorragendes intuitives Nutzererlebnis auch bei starken Features. RaiseNow gestaltet damit das Online-Fundraising effizient und gleichzeitig bequem und bietet eine umfassende Lösung, die auf die Bedürfnisse von (Nonprofit-) Organisationen angepasst ist.

Die RaiseNow Produktpalette besteht aus verschiedenen Applikationen, welche dedizierte Aufgaben übernehmen und somit eine Software-as-a-Service-Plattform darstellen. Damit die Authentifizierung zwischen diesen Applikationen sichergestellt werden kann, wird das standardisierte Authorisierungsframework OAuth2 eingesetzt. Die eigentliche Kommunikation zwischen den Applikationen findet entweder über spezielle Events (WebHooks) statt oder kann über das Nachrichten-orientierte Protokoll AMQP erfolgen.

Weiterhin bietet RaiseNow auch eine Lösung in Crowdfunding für Nonprofit-Organisationen an, die *RaiseNow P2P Plattform*. Sie erlaubt es jedem, eine eigene Spendenkampagne für einen bestimmten Zweck zu erstellen und ein dafür eigenes SMS-Kennwort zu generieren, welches eine Spende an die gewünschte Kampagne sicherstellt. Der Kunde als Kampagnenersteller hat dann die Möglichkeit, die hereinkommenden Zahlungen zu überwachen und weitere Informationen über die Kampagne zu verbreiten.

Durch die grosse Anzahl von Kunden und mehreren tausend Transaktionen ergibt sich eine grosse Menge an Daten, welche in gewissen Applikationen durchsucht werden muss. Zurzeit werden die Datensätze mittels SQL auf relationalen Datenbanken durchsucht. Durch die Komplexität sowie der Grösse der Datenmenge dauern einzelne Anfragen für ein hervorragendes Nutzererlebnis zu lange.

Die zu implementierende Aufgabe besteht darin, einen einheitlichen, wiederverwendbaren Service zur Verfügung zu stellen, um Daten sowie Objekte zu indexieren und schliesslich sogenannte Facettenfilter darauf anzuwenden. Des Weiteren soll dieser Service später nicht nur den hauseigenen, sondern auch Applikationen von Drittentwicklern zur Verfügung stehen.

Inhaltsverzeichnis

1 Projektübersicht **3**

1.1 Notwendigkeit einer Such-API 3

1.2 Bestehende Implementierungen 3

1.3 Aufgabenstellung 4

2 Anforderungen **5**

2.1 Architektur 5

2.2 Schemaübertragung 5

2.3 Such-API 6

3 Lösungsansätze **7**

3.1 Architektur 7

3.2 Schemaübertragung 8

3.3 Such-API 10

4 Implementation **11**

4.1 EsIntegrationBundle 11

4.2 EsSearchBundle 12

4.3 Referenz Implementation RaiseNow P2P 13

4.4 Dokumentation 14

5 Ausblick und Fazit **15**

1. Projektübersicht

Dieses Kapitel beinhaltet die Begründung und die gegebenen Voraussetzungen für eine Erweiterung der bestehenden Such-Infrastruktur und die daraus resultierenden Aufgabenstellungen sowie Anforderungen.

1.1 Notwendigkeit einer Such-API

In den bereits bestehenden Applikationen[1] fielen über die vergangenen Jahre eine grössere Menge an Daten an, gespeichert in einem relationalen Datenbankmodell. Mit dem wachsenden Kundenstamm von RaiseNow vergrösserte sich die Anzahl der Bedürfnisse seitens der Auftraggeber. Eine dadurch entstandene Anforderung, die Mehrsprachigkeit unserer Applikationen, stellte uns schliesslich vor eine Herausforderung:

Eine erste Implementation unter Verwendung einer bestehenden Library schien zuerst vielversprechend. Jedoch stellte sich heraus, dass bei komplexeren Suchanfragen auf unsere Datenbank die Anzahl an `JOIN`s zwischen Tabellen stark anstieg und die Dauer der Suchanfrage unseren Anforderungen bei Weitem nicht genügte.

Die strategische Entscheidung, um diesen Sachverhalt zu verbessern, fiel auf eine von Grund auf neu zu bauende Such-API. Da relationale Datenbanken im Allgemeinen eher weniger geeignet sind, grosse Mengen an Daten unabhängig von Relationen zu durchsuchen, soll eine andere Technologie eingesetzt werden: Elasticsearch, ein verteilter Suchindex [1]. Elasticsearch beschreibt sich selbst als einen verteilten, Echtzeit unterstützenden Dokumentenspeicher, in dem jedes Feld grundätzlich indexiert wird und durchsuchbar ist.

1.2 Bestehende Implementierungen

In einigen der bestehenden internen Applikationen werden Daten bereits über Elasticsearch zur Verfügung gestellt. Diese Schnittstelle wird über HTTP bereit gestellt. Gesucht wird über ein proprietäres Format mit URI-Query Parametern. Im Hinblick auf die Unterschiedlichkeit der zu durchsuchenden Datensätze über Applikationsgrenzen hinweg, fiel ein Einsatz dieser Implementation aber weg.

[1] Im Nachfolgenden wird der Begriff *Applikation* sowie *Anwendung* auf Entwicklungen von *RaiseNow* bezogen. Applikationen von Drittentwicklern werden als solche beschrieben.

1.3 Aufgabenstellung

Durch die beschriebenen Voraussetzungen zeigte sich für mein durchzuführendes Projekt eine grobe Aufgabenstellung:

Die neu aufzubauende Implementation eines Suchservices soll

- auf Elasticsearch beruhen
- eine REST-API für Suchanfragen bereitstellen
- bestehende Standards für URIs einhalten
- im Hinblick auf eine Öffnung für Drittentwickler und Drittapplikationen gestaltet werden
- eine Synchronisation zwischen den Daten im relationalen Datenbankmodell und in Elasticsearch ermöglichen
- eine generische Grundfunktionalität bereitstellen, welche in verschiedenen internen Anwendungen verwendet werden kann
- die Performance der bisherigen Such-APIs übertreffen

Zusätzlich soll nach der generischen Entwicklung dieses Frameworks eine Referenzimplementation für eine unserer Applikationen durchgeführt werden: Der *RaiseNow P2P Plattform*. Jeder kann auf dieser Plattform eigene Spendenkampagnen zu einem bestimmten Zweck erstellen. Zahlungen werden über ein generiertes SMS-Kennwort der Kampagne zugeordnet. Zur Überwachung der Kampagne bietet sich dem Ersteller die Möglichkeit, eingegangene Zahlungen zu analysieren und weitere Informationen über die Kampagne zu verbreiten.

2. Anforderungen

Das folgende Kapitel führt die vorangegangenen, kurz umrissenen Anforderungen weiter im Detail aus. Es ist in die drei Bereiche *Architektur, Schemaübertragung* und *Such-API* gegliedert.

2.1 Architektur

Die Architektur des zu erstellenden Frameworks soll mit den im Folgenden beschriebenen Anforderungen zurechtkommen.

Mehrmandantenfähigkeit Um unbeabsichtigte Veränderungen am Datenbestand – verursacht von anderen Applikation – ausschliessen zu können, soll eine Mehrmandantenfähigkeit sichergestellt werden. Verschiedene Applikationen dürfen keine Möglichkeit besitzen, um auf für sie fremde Daten zuzugreifen. Eine Mehrmandantenfähigkeit im Bezug auf die einzelnen Kunden der jeweiligen Applikationen soll in Erwägung gezogen werden, stellt aber kein Muss-Kriterium dar.

Synchronisation der Daten Um die Datensätze im Speicher des Such-Services persistent mit den Korrespondierenden der relationalen Datenbank zu halten, soll ein Weg gefunden werden, um die beiden Datenbestände zu synchronisieren.

Elasticsearch Als Backend-Technologie soll Elasticsearch in der Version 1.4.3 eingesetzt werdeb.

Integrationsaufwand und Wartbarkeit Als weitere Anforderung gilt es, den Integrationsaufwand der zu erstellenden Such-Schnittstelle in bestehende Projekte zu beachten. Das Bereinigen von Fehlern sollte im Idealfall über die Applikationsgrenzen hinweg erfolgen, eine abstrakte, gemeinsame Codebasis soll verwendet werden.

2.2 Schemaübertragung

CRUD-Operationen auf Mappings Da sich die Schemen der relationalen Datenbank über die verschiedenen Applikationen komplett unterscheiden können, muss es eine

Möglichkeit geben, mit der das Pendant des Schemas der relationalen Datenbank – das Mapping – im Suchindex erzeugt, gelesen, geändert und schliesslich auch wieder gelöscht[1] werden kann.

Integrationsaufwand Im Hinblick auf einen zukünftigen Einsatz des zu erstellenden Services in weiteren Applikationen soll der Aufwand, um solche Mappings anhand der bestehenden Entitäten für die relationale Datenbank zu generieren, tief bleiben. Ein weiteres Anliegen besteht darin, auf Konfigurationsdateien für das Festhalten von Mappings zu verzichten, da unsere Infrastruktur von einem Provisioning-Service betrieben wird und dadurch zusätzlich eine weitere Konfiguration unterhalten werden müsste.

2.3 Such-API

Art und Weise des Zugriffs Die Such-API soll über eine RESTful-Schnittstelle über HTTP ansprechbar sein.

Standards für URIs und RESTful Such-Schnittstellen Bestehende Best-Practices im Umgang mit URIs sollen eingehalten werden. Dies betrifft vor allem die unterstützte Länge eines Queries, reservierte sowie nicht reservierte Zeichen (vgl. RFC3986 [2]). Auch sollen die REST-Verben möglichst korrekt eingesetzt werden.

Restriktionen bei Suchanfragen Bei bestimmten Suchanfragen sollen Veränderungen an der Anfrage vorgenommen werden können.

Lernkurve für Drittentwickler Um für die spätere Öffnung der Such-Schnittstelle keine zusätzlichen Eintrittshürden aufzubauen, soll kein neues, proprietäres Suchformat eingeführt werden. Des Weiteren sollen bereits Libraries für Client-seitige Programmiersprachen bestehen, welche Anfragen von einem abstrakteren Modell in das Suchformat transformieren.

[1] CRUD = **C**reate, **R**ead, **U**pdate, **D**elete

3. Lösungsansätze

Die vorgängig definierten Anforderungen im Blick, erarbeitete ich die im Folgenden beschriebenen Lösungsansätze.

3.1 Architektur

Der ausgearbeitete Lösungsvorschlag für die architektonischen Anforderungen beinhaltet vier verschiedene Schichten, wie in Abbildung 3.1 dargestellt. Zuunterst befinden sich die Drittapplikationen sowie der Client, welcher Anfragen an die zu erstellende Such-API stellt.

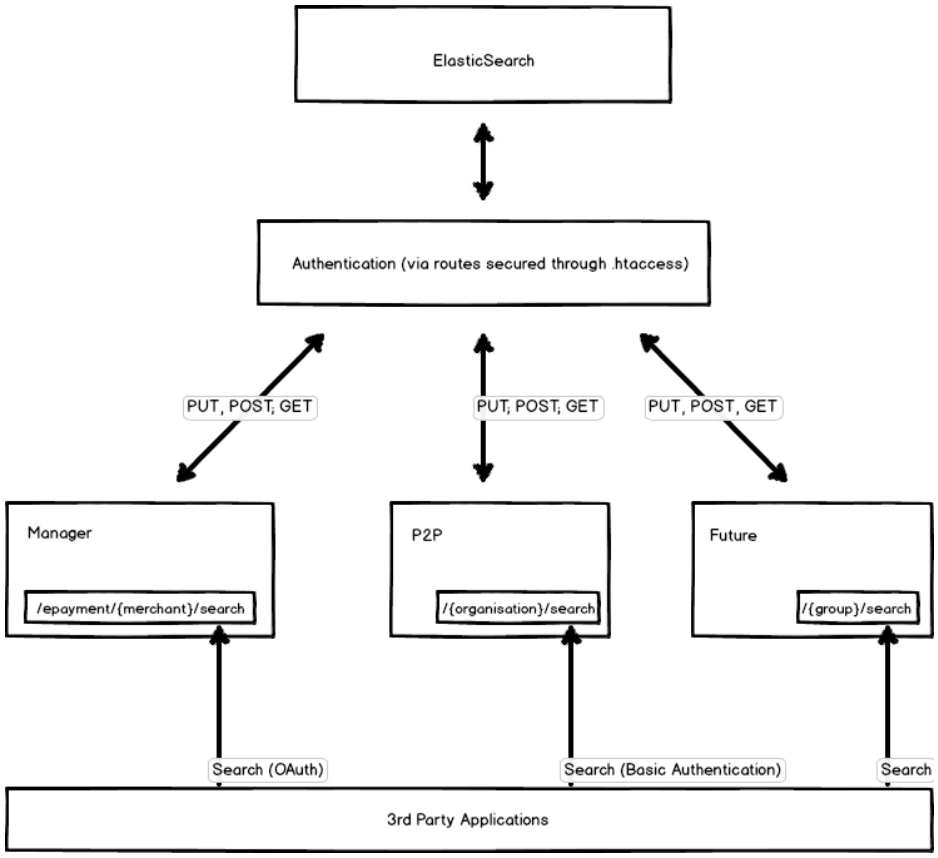

Abbildung 3.1: Überblick vorgeschlagene Architektur

Auf der darüber liegenden Schicht befinden sich die einzelnen Applikationen. Im Vordergrund steht hier die Unabhängigkeit der jeweiligen Anwendung von einer zentralen Implementation für die Authentifizierung sowie die Authorisierung der Clients. Dies ist insofern erwünscht, als dass die konkreten Ausbildungen des Authorisierungs-Systems sehr unterschiedlich ausfallen können. Ein weiterer zentraler Grund für die Integration des Zugriffsberechtigungs-Systems auf dieser Ebene besteht in der Tatsache, dass der Suchservice, unabhängig vom Client und seinen Rollen, Suchanfragen beantworten können und sich nicht mit seinen Restriktionen beschäftigen soll. Jede Applikation erhält somit – unabhängig von den anderen – die Möglichkeit, Daten komplett neu zu indexieren, zu verändern, hinzuzufügen und die zugrunde liegenden Mappings zu modifizieren oder zu löschen und diese Aktionen an ihre Benutzer offenzulegen.

Zwischen dem Elasticsearch-Cluster – der obersten Schicht – und den internen Applikationen befindet sich eine Authentifizierungs- sowie Authorisierungsschicht, die sicherstellt, dass der Zugriff nur auf die zugehörigen Datensätze erlaubt wird. Dies verhindert ein (un-)beabsichtiges Verändern der Indizes und den Mappings. Sie fungiert nach dem Authentifizierungs- und Authorisierungsprozess als Proxy und leitet die Suchanfragen an den Cluster weiter.

3.2 Schemaübertragung

Um das Schema einer relationalen Datenbank in Elasticsearch zu übertragen, zog ich die folgenden drei Möglichkeiten in Betracht.

3.2.1 Entität zu JSON zu Mapping

```
class Foo {

  protected $name;

  /**
   * @var Baz A nested Baz object
   */
  protected $foo;
}

class Baz {

  protected $name;
}
```

Listing 3.1: Class definitions

```

// Foo
{
  "name": "",
  "foo": { // is a Baz object
    "name": ""
  }
}

// Baz
{
  "name": ""
}
```

Listing 3.2: Serialisierte Form in JSON

Einen generischen Ansatz bestand in der Generierung von JSON-Repräsentationen anhand der zu speichernden Entität und die daraus folgende Konvertierung in ein Mapping. Bei genauerem Hinsehen erwies sich dieser Ansatz jedoch als nicht zielführend. Dies aus den folgenden Gründen:

- Relationen zwischen Entitäten können verloren gehen, wenn mehrere die gleiche Struktur aufweisen. Die Zuordnung zur richtigen Entität wäre nicht mehr eindeutig. Vergleiche auch Listings 3.1 und 3.2.

- Es würden Mappingkonfigurationen in Dateien benötigt werden, um solche Ambiguitäten aufzuheben oder um Datentypen der einzelnen Felder festzuhalten.

- Vater-Kind-Beziehungen[1] zwischen Entitäten können hiermit nicht dargestellt werden. Als einzige Möglichkeit blieben Nested-Objects[2].

3.2.2 FOSElasticaBundle

Eine weitere, schon sehr spezifische Implementation einer Integration mit Elasticsearch besteht im Framework "FOSElasticaBundle" [3]. Es stellt bereits eine Konfiguration für Mappings bereit und implementiert darüber hinaus die Anbindung zum Elasticsearch-Cluster. Um den Entitäten auch dynamisch Felder hinzufügen zu können, können selbst zu implementierende "Model-Transformer" eingesetzt werden.

Unerwünschte Eigenschaften einer Integration dieses Framework kristallisierten sich vor allem in den wieder benötigten Konfigurationsdateien heraus. Des Weiteren sind die Funktionalitäten ungenügend dokumentiert, so dass ein zielstrebiges Arbeiten behindert werden würde.

3.2.3 Doctrine/search

Der letzte der untersuchten Ansätze besteht in einer Implementation von den Entwicklern unserer eingesetzten "Object-relational Mapping"-Technologie. Sie wird `doctrine/search` [4] genannt und bietet eine Integration der Mappingkonfiguration anhand von Annotationen direkt in den Entitäten, so dass Schemadefinitionen sowie das Mapping an ein und dem selben Ort zu liegen kommen: Ein klarer Vorteil für die Wartbarkeit. Viel Funktionalität zur Interaktion mit Elasticsearch ist bereits durch die Implementation gegeben. Der grösste Pluspunkt für einen Einsatz dieses Frameworks besteht in der Art wie mit Objekten umgegangen werden kann: Eine Transformation zu bzw. von der serialisierten Version der einzelnen Objekte fällt für den Entwickler weg, sie ist im Framework schon implementiert.

Jedoch birgt auch der Einsatz dieser Technologie einige Risiken. Als grösstes Hindernis wurde der unklare Stand in der Weiterentwicklung angesehen. Gemäss dem Code-Repository haben die Entwickler den Fokus auf andere Projekte gerichtet. Auch ist die Dokumentation nur spärlich vorhanden.

---

[1] **Vater-Kind-Beziehungen** in Elasticsearch bilden eine Möglichkeit Fremdschlüssel von der relationalen Datenbank abzubilden. Jedoch sind sie pro Mapping auf eine solche Relation beschränkt.

[2] **Nested-Objects** bilden die andere Möglichkeit, Beziehungen zwischen Objekten darzustellen. Sie verschachteln Objekte in einander.

3.2.4 Abwägung

Durch die unklare Situation im Bezug auf die einzusetzenden Technologien erstellte ich in Absprache mit dem Entwicklungsteam einen Spike für unseren Favoriten `doctrine/-search`. Durch seine vielfältigen Konfigurationsmöglichkeiten sowie der Einfachheit, um Mapping-Konfigurationen zu erstellen und zu warten, stellte sich dieser Ansatz schlussendlich als zielführend heraus.

3.3 Such-API

Um die früher genannten Anforderungen betreffend der Such-API zu berücksichtigen, blieb nur ein Ansatz übrig: Den Einsatz des gleichen Suchformats welches bei Elasticsearch verwendet wird. Es ist ausgiebig dokumentiert und verhindert somit eine übermässig grosse Einstiegshürde für Drittentwickler. Ausserdem ermöglicht uns seine Verwendung von der Entwicklungsarbeit Dritter zu profitieren (vor allem im Bezug auf die Wartbarkeit, Dokumentation und Support).

Ein weiterer Vorteil besteht im Einsatz von JSON als Query-Sprache. Gemäss der gängigen Literatur über REST-Schnittstellen [5], soll das Eingabe-Format dem der Ausgabe entsprechen. Da bei Elasticsearch Query-Resultate in JSON zurückgegeben werden, kann auch diese Anforderung eingehalten werden.

Auch gibt es für Client-seitige Programmiersprachen – wie JavaScript – bereits Bibliotheken, welche anhand eines Query-Builders Suchanfragen zusammenstellen können.

4. Implementation

Aus meinem Spike, der die Eignung von `doctrine/search` darstellte, entwickelte sich das EsIntegrationBundle. Es integriert die bestehenden Entitäten mit ihren relationalen Schemadefinitionen in Elasticsearch. Für die Suchfunktionalität und ihre Abstraktion auf den Elasticsearch-Cluster implementierte ich das EsSearchBundle. Es bündelt die Funktionalität zur Validation und Modifizierung von Queries, bevor sie an die Elasticsearch Instanz gesendet werden.

4.1 EsIntegrationBundle

Dieses Bundle besteht aus mehreren Komponenten, welche unterschiedliche Aufgaben zur Interaktion mit Elasticsearch übernehmen.

4.1.1 Elasticsearch-Handler

Diese Komponente stellt den zentralen Einstiegspunkt für das Verwalten der Mapping-Definitionen dar. Verschiedene Methoden für CRUD-Operationen auf Mappings sowie für eine direkte Kommunikation mit dem Elasticsearch-Cluster werden zur Verfügung gestellt. Des Weiteren können die Mapping-Definitionen aus den Klassen-Annotationen gelesen und zurückgegeben werden.

4.1.2 Search-Manager

`Doctrine/search` stellt für das Persistieren der annotierten Entitäten bereits einen sogenannten Search-Manager zur Verfügung, der im Stil eines ORM-Entity-Managers eine pseudo-transaktionale Verarbeitung der Datensätze erlaubt.

Durch die Architektur von Elasticsearch sind Nested Objects zu bevorzugen, da sie eine raschere Verarbeitung der Suchanfrage versprechen als Vater-Kind-Relationen (im Verhältnis 1:10). Beim Einsatz der Ersteren werden jedoch keine Nested Objects aktualisiert, wenn die Entität nicht selbst auch neu reindexiert wird. Aus diesem Grund implementierte

ich sogenannte Dependency Resolver, welche konfigurierbar pro Entität, auch Nested Objects für eine erneute Reindexierung einplanen. Dadurch können einerseits solche Abhängigkeiten über Accessor-Methoden definiert werden, andererseits über die Query-Sprache unseres ORMs. Ferner können auch Abhängigkeiten von Entitäten für eine Reindexierung berücksichtigt werden, welche nicht über eine Accessor-Methode ansprechbar sind. Um zyklische Pfade bei der Auflösung solcher Abhängigkeiten nicht erneut zu traversieren, wird von jeder bereits verarbeiteten Instanz ein Hash gebildet und gespeichert. Eine Entität wird nur dann zur Reindexierung zugelassen, falls ihr Hash noch nicht bekannt ist.

4.1.3 Database Provider

Um einen Abgleich der Daten zwischen der relationalen Datenbank und Elasticsearch zu ermöglichen, kommen sogenannte Database Provider zum Einsatz. Sie erlauben es, eine bestimmte Menge an Daten einer Entität (resp. einer Tabelle), beginnend an einem definierten Offset, aus der relationalen Datenbank abzurufen. Dass keine Zeile ein erneutes Mal verarbeitet wird, stellt ein vorheriges Sortieren anhand der Primary Keys sicher.

4.1.4 Commands

Ausserdem implementiert dieses Bundle noch Konsolen-Kommandos, mit welchen man die grundlegende Funktionalität in Elasticsearch ansprechen kann. Dazu gehören Kommandos zur Ausführung von CRUD-Operationen auf Indizes sowie Mappings und zum Bestücken des Clusters mit den Daten der relationalen Datenbank.

4.2 EsSearchBundle

Das zweite Bundle beinhaltet Funktionalität für die Integration der Such-API in unsere Applikationen. Zusätzlich zu den nachfolgenden Komponenten, stellt es ein Interface für die Weiterleitung der Suchanfrage an Elasticsearch bereit. Dieses muss die Applikation aufgrund besserer Trennung der Komponenten selbst implementieren.

4.2.1 Query-Handler

Auch hier stellt der Handler den zentralen Einstiegspunkt dar, um Suchanfragen zu beantworten und an den Elasticsearch-Cluster weiter zu reichen. Bevor aber eine Weiterleitung stattfindet, werden sie validiert und gegebenenfalls modifiziert.

4.2.2 Query-Validator

Um ungültige Suchanfragen schon im vornherein abzuweisen, können sogenannte Query-Validator implementiert werden, welche die Gültigkeit des Queries beurteilen. Wird ein Fehler erkannt, kann der Query-Handler die Anfrage direkt mit einer Exception beantworten.

4.2.3 Query-Modifier

Zusätzlich zu den Query-Validatoren, gibt es mit den Query-Modifier eine mächtige Möglichkeit, Suchanfragen so zu verändern, dass sie den Filterkriterien für den jeweiligen Client entsprechen. Die Applikation, welche dieses Bundle verwendet, kann somit abhängig von den Benutzerberechtigungen mehr oder weniger Einschränkungen in eine Suchanfrage einbauen.

4.3 Referenz Implementation RaiseNow P2P

Zum Schluss meiner Projektarbeit integrierte ich die oben genannten Bundles in die *RaiseNow P2P Plattform*. Der Aufwand für die Integration hielt sich stark in Grenzen, zum Annotieren der einzelnen Entitäten kamen schlussendlich noch Hilfeklassen für dynamische Änderungen bei der Serialisierung hinzu.

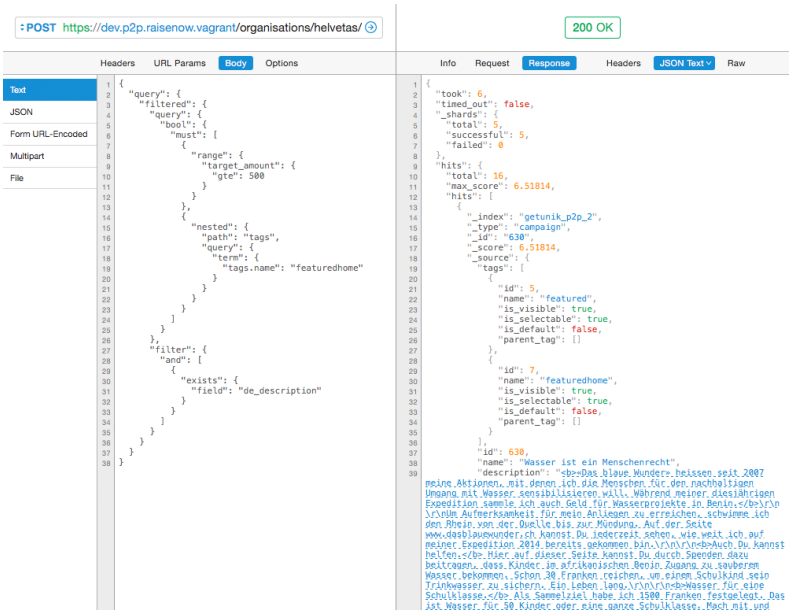

Abbildung 4.1: Such-Anfrage an die *RaiseNow P2P Plattform*:
HTTP/1.1 POST `/organisations/helvetas/api/v1/campaign/search?size=100`

Um den Elasticsearch-Cluster mit den Daten in der Datenbank synchron zu halten, wurden Lifecycle-Events unseres ORMs verwendet, die eine erneute Reindexierung für das

geänderte Dokument auslösten. Diese Implementation wird nicht von einem der Bundles bereitgestellt, damit sie möglichst unabhängig von der Verwendung eines ORMs bleiben. Mittels den vorgängig beschriebenen Dependency Resolver ist der Index im Cluster somit immer auf dem gleichen Stand wie die Datenbank.

Ein Query an die Referenz-Implementation ist in Abbildung 4.1 abgebildet. Es gibt alle Dokumente zurück, für welche der `target_amount` grösser oder gleich 500 ist, ein Tag mit dem Namen `featuredhome` existiert und ein Feld `de_description` vorhanden ist. Zusätzlich zu den abgebildeten Filtern, wird für den User ein Filter auf den Namen der Organisation, für welche er autorisiert ist, angelegt. Das Ergebnis beinhaltet 16 Treffer, wobei die Ergebnisse nach dem Feld `_score` geordnet sind. `_score` ist ein Indikator für die Relevanz, die das Dokument für die Suchanfrage aufweist. Sie wird anhand einer Kombination von Term-Frequency[1], Inverse-Document-Frequency[2] und Field-Length-Norm[3] berechnet.

4.4 Dokumentation

Zusätzlich zu der Implementation dieses Suchservices dokumentierte ich die Bundles ausgiebig in unserer firmeninternen Wissensdatenbank.

[1] $tf(t) = \sqrt{f}$, wobei t für den Term im Dokument steht, und f die Anzahl Vorkommen von t im Dokument repräsentiert

[2] $idf(t) = 1 + \log \frac{n}{f+1}$, mit n = Anzahl Dokumente im Index, f = Anzahl Dokumente welche t enthalten

[3] $norm(d) = \frac{1}{\sqrt{n}}$, mit d = das Dokument, n = Anzahl Terms im Feld

5. Ausblick und Fazit

Zum Zeitpunkt des Verfassens dieses Schlussberichts befand sich die Referenzimplementation für die *RaiseNow P2P Plattform* sowie der beiden Bundles im Review-Prozess. Falls Verbesserungen oder Änderungen an der Architektur vonnöten sind, werden sie innherhalb meiner regulären Arbeitszeit von mir selbst oder von meinen Teamkollegen umgesetzt.

Während der Durchführung meines Projektes konnte ich wertvolle Erfahrungen mit der Technologie von Suchindizes sammeln. Auch durfte ich mich mit den Feinheiten von REST beschäftigen, um die Such-API konform den geltenden Standards zu gestalten. Die darauffolgende Implementation der gewählten und spezifizierten Architektur sowie die Integration in die *RaiseNow P2P Plattform* hat mir grosse Freude bereitet.

Meinen herzlichen Dank gilt Tsuy Ito, Technical Director von RaiseNow sowie dem ganzen Team, welches mich während dieser Zeit begleitet und unterstützt hat.

Literaturverzeichnis

[1] Elasticsearch, `https://www.elastic.co/products/elasticsearch`, abgerufen am 13.08.2015

[2] RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax, `https://www.ietf.org/rfc/rfc3986.txt`, abgerufen am 13.08.2015

[3] FOSElasticaBundle, `https://github.com/FriendsOfSymfony/FOSElasticaBundle`, abgerufen am 13.08.2015

[4] Doctrine/search, `https://github.com/doctrine/search`, abgerufen am 13.08.2015

[5] Subbu Allamaraju, *RESTful Web Services Cookbook - Solutions for Improving Scalability and Simplicity*, O'Reilly Media / Yahoo Press, Februar 2010