

Projektdokumentation im Modul Semantic Web

Semantische Suchmaschine für das Web of Data: Trip-Planung für Sehenswürdigkeiten

Peter Nancke

<https://github.com/pnancke/trip-planner>

31.07.2016

Aufgabenstellung: Einen Routenplaner im Sinne des Semantic Webs zu entwickeln, um eine optimale Route zwischen zwei Punkten mit möglichst vielen Sehenswürdigkeiten zu finden.

1 Inhaltliche Interpretation der Aufgabenstellung

Mobilität ist in einer modernen Gesellschaft nicht mehr wegzudenken. Aber nicht nur für den Weg auf Arbeit legen Menschen bereitwillig viele Kilometer zurück. Für die Fahrt zum gewünschten Urlaubsort werden z.B. oftmals weite Strecken zurückgelegt. Hierbei lohnt es sich, immer wieder Pausen einzulegen, um auch die Umgebung mit all dem, was sie zu bieten hat, kennenzulernen. Um interessante Sehenswürdigkeiten zu finden, muss man sich jedoch entweder in lokalen Touristen Informationsstellen beraten lassen oder selber recherchieren.

Hierfür soll eine Anwendung geschaffen werden, die dem Nutzer eine optimale Route zwischen mehreren Punkten vorschlägt, auf denen er eine breite Auswahl an Sehenswürdigkeiten (POI) hat. Zusätzlich zu der Route sollen dem Nutzer nützliche Informationen, wie beispielsweise ein Wikipedia Eintrag zu den Sehenswürdigkeiten, angezeigt werden.

Für diese Anwendung sollen verschiedene Datenquellen und Dienste in benutzt werden, um dem Anspruch einer semantischen Suche für das *Web of Data* gerecht zu werden.

2 Relevante Datenquellen

In den folgenden Abschnitten werden die relevanten Datenquellen kurz vorgestellt.

2.1 OpenStreetMap Daten

OpenStreetMap stellt eine breite Auswahl an Geodaten bereit. Diese beinhalten drei verschiedene Objekttypen. *Nodes* sind hierbei Punkte, wie z.B. ein Gebäude, eine Ampel oder eine Sehenswürdigkeit. *Ways* bestehen aus jeweils mehreren *Nodes* und *Relations*. *Relations* beschreiben die Beziehung zwischen *Nodes* und *Ways* ¹. Alle diese Typen besitzen zusätzlich Attribute, wie etwa dem Namen oder der Position.

Der Zugriff auf die Daten ist entweder über den direkten Datendumb bzw. ein Changeset der letzten Änderungen oder eine REST-API möglich. Aufgrund des Crowdsourcing Ansatzes von OpenStreetMap können die Daten eventuelle kleinere (Rechtschreib-)Fehler enthalten. Die Gesamtheit der OpenStreetMap Daten umfasst unkomprimiert etwa 666 GB und komprimiert etwa 50 GB.

Link	http://download.geofabrik.de/
Datenformat	bz2 , PBF
Schnittstelle	Dumb
Lizenz	Open Data Commons Open Database Lizenz (ODbL)
Open Data	★★★★★

Über die REST-API können alle Nodes innerhalb eines Rechtecks abgefragt werden. Diese API eignet sich für kleine Anfragen. Bei größeren Requests dauert die Abarbeitung signifikant länger.

Link	http://www.overpass-api.de/api/xapi
Datenformat	OSM XML
Schnittstelle	Rest-API
Lizenz	Affero GPL v3
Open Data	★★★

¹<http://wiki.openstreetmap.org/wiki/Elements>

2.2 Autocompletion API

Die Autocomplete API² soll den Nutzer bereits bei der Eingabe eines Ortes mit nützlichen Vorschlägen unterstützen. Sie basiert auf Elasticsearch und nutzt ebenfalls die OpenStreetMap Daten. Es ist möglich, eine Angabe über die gewünschte Sprache der REST-Anfrage beizufügen, um die Ortsnamen in der Sprache des Nutzers zu beziehen.

Link	https://photon.komoot.de/api/
Datenformat	GeoJSON
Schnittstelle	Rest-API
Lizenz	Apache License, Version 2.0
Open Data	★★★★★

2.3 Nomination API

Die Nomination API ermöglicht es, eine Adresse in Textform in Geokoordinaten umzuwandeln. Dies ist notwendig, da andere Dienste, wie z.B. die Routing API, die Koordinaten des Start- und Endpunktes benötigen.

Link	http://nominatim.openstreetmap.org/search
Datenformat	XML, HTML
Schnittstelle	Rest-API, Website
Lizenz	
Open Data	★★★★★

2.4 Routing API

Um eine optimale Route zwischen zwei oder mehr Punkten zu ermitteln, wird die Routing API benötigt, die auf den OpenStreetMap Daten aufbaut. Es gibt eine Vielzahl an Diensten hierfür³. Für unsere Anwendung sind jedoch folgende Kriterien erforderlich:

- Globale Abdeckung
- Bereits existierende online API
- Schnelle Antwortzeiten
- Uneingeschränkte Nutzung

²<http://photon.komoot.de/>

³http://wiki.openstreetmap.org/wiki/Routing/online_routers

- Unlimitierte Unterstützung von Wegpunkten

Keine der APIs erfüllte alle diese Punkte. Aus diesem Grund entschieden wir uns für die Routing API von *yournavigation.org*, da diese die einzige der genannten APIs ist, die bis auf die Unterstützung von Wegpunkten, alle anderen Punkte erfüllt. Wie dieses Feature dennoch umgesetzt worden ist, wird in Kapitel 4.3 erläutert. Im Folgenden ist die benutzte API kurz charakterisiert:

Link	http://www.yournavigation.org/api/1.0
Datenformat	XML
Schnittstelle	Rest-API
Lizenz	Fair Use
Open Data	★★★★★

3 Extraktion relevanter Daten

Es wurde zunächst die OpenStreetMap (OSM) REST-API benutzt, siehe Kapitel 2.1, um die Sehenswürdigkeiten in einem Bereich zu beziehen. Hierbei wurden verschiedene Probleme deutlich. Zum einen ist beträgt die Antwortzeit bereits bei kleineren Strecken von z.B. Leipzig nach Berlin etwa 10 Sekunden. Zusätzlich war lediglich eine Anfrage pro IP Adresse gleichzeitig möglich.

Außerdem ist es lediglich möglich, Sehenswürdigkeiten in einer sog. Bounding Box, also einem Rechteck, abzufragen. Andere Flächenformen, wie z.B. Polygone, werden nicht unterstützt. Für den Routenplaner ist dies zu unflexibel.

Aus den genannten Gründen haben wir uns dazu entschieden, die Sehenswürdigkeiten aus dem OSM Datendumb zu extrahieren und in einer lokalen Datenbank zu speichern. Der grobe Ablauf dieses hierfür implementierten Kommandozeilenscripts `import.sh` ist in folgenden Pseudocode verdeutlicht:

```

Data: OSM Data Dumb (PBF binary)
Result: Points of interest in SQL database
convert binary PBF file to XML;
filter all nodes with tourism=attraction and name=*;
convert remaining nodes to CSV format;
setup local database;
convert CSV to sql import statements;
csv_entry_count ← countLines();
imported_rows_count ← execute import script;
if csv_entry_count ≠ imported_rows_count then
  | print error message;
end
Algorithm 1: Import points of interest from OSM data dumb to database

```

Hierfür kamen u.a. folgende Tools zum Einsatz:

Osmconvert⁴ zum Konvertieren der binären *.osm.pbf Datei in das CSV Format.

Osmfilter⁵ zum Filtern von Nodes, die sowohl einen Namen besitzen, als auch eine Sehenswürdigkeit (erkannbar an `tourism=attraction`) sind.

4 Umsetzung

In diesem Kapitel werden alle umgesetzten Kern-Features der Anwendung erläutert.

4.1 Webserver

Der Anwendungsserver wird durch das Grails⁶ Framework realisiert. Im Mittelpunkt der Entscheidung für Grails steht das „Konvention statt Konfiguration“ Paradigma, welches einen schnellen Start ohne aufwendiges Konfigurieren des Webserverns ermöglicht. Die Möglichkeit, den Code bei laufender Anwendung neu zu kompilieren, ohne den Server neustarten zu müssen, beschleunigt die Entwicklungszeit ebenfalls. Beide Aspekte sind für uns besonders wichtig, da nur ein begrenzter zeitlicher Rahmen für das Projekt zur Verfügung steht.

Mit Grails wurde folgende Schnittstelle realisiert:

HTTP-Operation	GET
Pfad	/getRoute
Parameter	start: Startpunkt (Text) destination: Zielpunkt (Text) waypoints: Mit Komma getrennte Liste aller Wegpunkte (Text) lang: Sprache des Nutzers (Text) searchArea: Breite des Suchraums nach POIs um Route in km (Zahl); $0 \leq \text{searchArea} \leq 100$

Neben der Serverschnittstelle stellt das Grails Framework auch die grafische Oberfläche bereit. Dies geschieht mit sogenannten *Groovy Server Pages*⁷. Diese fungieren als Template der anzuzeigenden HTML Seite. So können neben den HTML Tags auch Grails spezifische Tags genutzt werden.

⁴<http://wiki.openstreetmap.org/wiki/Osmconvert>

⁵<http://wiki.openstreetmap.org/wiki/DE:Osmfilter>

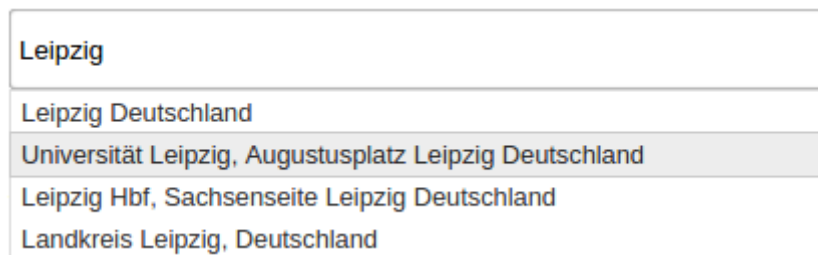
⁶<https://grails.org/>

⁷<http://docs.grails.org/latest/guide/theWebLayer.html#gsp>

Es kann z.B. mit: `{g.createLink(controller: 'home', action: 'getRoute')}` automatisch ein Link zur `getRoute` Ressource des `HomeController` im HTML Code erstellt werden.

4.2 Autocomplete

Für jedes Eingabefeld des Nutzers wird die Autocomplete API, siehe Kapitel 2.2, angefragt. Dies betrifft die Felder für Start- und Endpunkt, sowie die generierten Felder für die Wegpunkte. Abbildung 1 zeigt mögliche Vorschläge dieser API bei der Eingabe des Wortes *Leipzig*.



Leipzig
Leipzig Deutschland
Universität Leipzig, Augustusplatz Leipzig Deutschland
Leipzig Hbf, Sachsenseite Leipzig Deutschland
Landkreis Leipzig, Deutschland

Abbildung 1: Autocomplete API

4.3 Routing

Das Erstellen einer optimalen Route zwischen zwei oder mehreren Punkten ist ein Hauptbestandteil der Anwendung. Hierfür wird die in Kapitel 2.4 vorgestellte Routing API benutzt.

Wie bereits in Kapitel 2.4 erwähnt, unterstützt die API von *yournavigation.org* jedoch keine Wegpunkte, es sind lediglich Start- und Endpunkt einer Route bestimmbar. Um nicht auf dieses Feature verzichten zu müssen, wird zunächst die Route, bestehend aus Start- und Endpunkt, sowie der Wegpunkte, zunächst in Teilstrecken mit je einem Start- und Endpunkt unterteilt. Nun kann für jeder dieser Streckenabschnitte eine Route ermittelt werden. Aus den Teilrouten zusammengesetzt ergibt sich dann die kürzeste Route zwischen zwei Punkte und verschiedenen Wegpunkten. Diese hierfür notwendigen REST-Anfragen werden parallel ausgeführt, damit dieser Teil der Anwendung auch für große Anfragen skaliert.

Nachdem die Route bestimmt wurde, werden alle Sehenswürdigkeiten entlang dieser Route gespeichert und gefiltert, wie in Kapitel 4.4 beschrieben.

Dem Nutzer soll eine direkte Route zu allen Sehenswürdigkeiten angeboten werden. Da die Anzahl der Sehenswürdigkeiten jedoch bereits bei kleineren Routen enorm ist, würden sehr viele Teilrouten entstehen. Aus einer Route würden so schnell mehrere Hundert REST-Anfragen an die Routing API generiert werden, was diese überlasten könnte. Zudem leitet sich kein direkter Nutzen aus einer für den Nutzer so komplizierten Route ab.

Viel sinnvoller ist es, alle Cluster, bestehend aus den POIs, aufzuteilen, damit diese einen maximalen Durchmesser von 10 km haben und jeweils deren Clusterzentren als Anlaufpunkt für das Routing zu benutzen. So kann ein Cluster von Sehenswürdigkeiten jeweils vom Nutzer zu Fuß oder mit Hilfe des öffentlichen Nahverkehrs abgefahren werden.

Hierfür wird der kMeans Cluster Algorithmus verwendet. Es wurde eine Warteschlange implementiert, die über jeden Cluster iteriert. Sobald der Durchmesser eines Clusters über 10 km liegt, werden all Sehenswürdigkeiten dieses Clusters erneut auf mehrere Cluster aufgeteilt und diese der Queue hinzugefügt. Wenn der Durchmesser eines Clusters klein genug ist, wird er aus der Queue entfernt und in einer separaten Liste gespeichert. Der Groovy Code der Methode zum Aufteilen der Cluster ist in Listing 1 vorgestellt.

Listing 1: Aufteilen der Cluster

```
private static List<PointCluster> splitClusters(List<PointCluster> filteredClusters) {
    List<PointCluster> filteredClustersByRange = new ArrayList<PointCluster>()
    Queue<PointCluster> queue = new LinkedList<PointCluster>(filteredClusters)
    while (!queue.isEmpty()) {
        PointCluster cluster = queue.remove()
        if (cluster.clusterRange > MAX_DISTANCE_TO_CLUSTER_CENTER_IN_KILOMETRES) {
            def newClusters = convert(extractClusters(cluster.points,
                K_MEANS_CLUSTER_SIZE, MAX_K_MEANS_ITERATIONS))
            for (it in newClusters) {
                queue.add(it)
            }
        } else {
            filteredClustersByRange.add(cluster)
        }
    }
    filteredClustersByRange
}
```

4.4 Filtern von Sehenswürdigkeiten mithilfe eines Clustering-Algorithmus

Die in Kapitel 2.1 erwähnten *Nodes* können neben diversen Eigenschaften, wie dem Längen- und Breitengrad und einer ID, ebenfalls *Tags* besitzen⁸. Diese Tags sind als Key-

⁸<http://wiki.openstreetmap.org/wiki/Node>

Value Paar aufgebaut und beschreiben ein Merkmal des *Nodes*⁹. Alle Sehenswürdigkeiten besitzen so das Tag `tourism=attraction`. Es sind nur *Nodes* relevant, die ein Tag mit dem Key `name` besitzen, da eine das Anzeigen von Sehenswürdigkeiten ohne Namen dem Anwender wenig nützen.

Nachdem aus dem in 2.1 erwähnten Datendumb alle *Nodes* ohne die beiden Tags `tourism=attraction` und `name=*` entfernt wurden, siehe Kapitel 3, wurde deutlich, dass die verbleibenden Sehenswürdigkeiten zum Teil weit verstreut sind.

Würden auch weit gestreute Sehenswürdigkeiten für das Ermitteln der Route in Betracht gezogen werden, so würde die Fahrtzeit stark zunehmen. Weiterhin hat sich ergeben, dass abgeschiedene Sehenswürdigkeiten in den meisten Fällen weniger Relevant sind als Sehenswürdigkeiten, die geballt auftreten.

Aus den genannten Gründen ist es notwendig, nur geballt auftretende Sehenswürdigkeiten für das Erstellen der Route zu nutzen. Zum Erkennen von verstreuten POIs wird der kMeans Cluster Algorithmus vom Data Mining Framework *ELKI* von der Ludwig-Maximilians-Universität München eingesetzt. Hierfür wurde eigens die Klasse *ElkiWrapper.groovy* implementiert, die alle notwendigen Konfigurationen vornimmt und den Cluster Algorithmus startet.

Nach dem alle POIs in Cluster aufgeteilt werden, können alle *dünnbesetzten* Cluster verworfen werden. Die hierfür implementierte Methode ist in Listing 2 vorgestellt. Für die Konstante `minMeanPercentageClusterSize` wurde durch systematisches Probieren der Wert 0.8 gewählt.

Definition 1 *Dünnbesetzter Cluster:* Ein Cluster wird als dünnbesetzt bezeichnet, wenn die Anzahl der Objekte des Clusters um einen selbstgewählten Faktor kleiner ist, als der durchschnittliche Anzahl der Objekte aller Cluster.

⁹<http://wiki.openstreetmap.org/wiki/Tags>

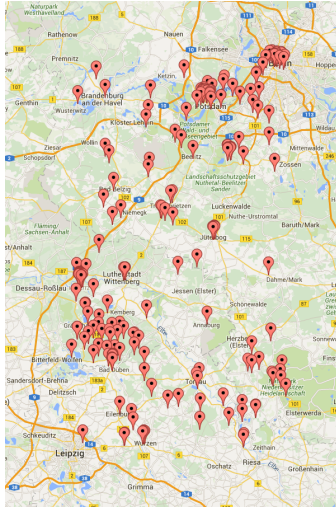


Abbildung 2: Ungefilterte POIs

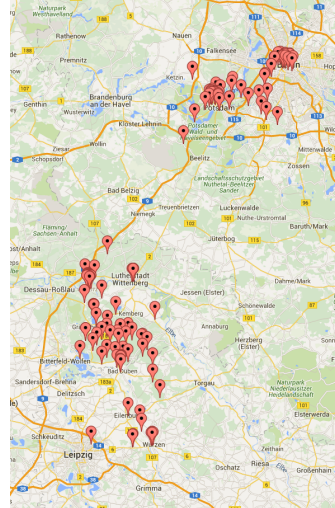


Abbildung 3: POIs ohne *schwache* Cluster

Listing 2: Filtern von dünnbesetzten Clustern

```
private static List<PointCluster> filterOutliers(List<PointCluster> clusters,
                                                int partitionSize,
                                                double minMeanPercentageClusterSize) {

    int allPointsCount = 0
    clusters.each { allPointsCount += it.size() }
    int averageSize = allPointsCount / partitionSize
    clusters.stream().filter {
        it -> it.size() > minMeanPercentageClusterSize * averageSize
    }.collect(Collectors.toList())
}
```

Abbildung 2 zeigt alle Sehenswürdigkeiten in einem gewähltem Bereich. Wie zu erkennen ist, sind diese weit verteilt. Nach dem Clustern und Aussortieren sind nur noch gehäufte Sehenswürdigkeiten vorhanden, wie in Abbildung 3 zu sehen ist.

4.5 Wikipedia Integration

Gerade bei weniger bekannten Sehenswürdigkeiten kann es für einen Anwender sinnvoll sein, weitere Informationen über diese aus Wikipedia zu beziehen.

In den OSM Daten besitzen bereits manche POIs das Attribut `wikipedia`, das aus der Sprache und des Titels des Wikipedia Eintrags besteht. Dies sieht beim z.B. wie folgt aus: `de:Eiffelturm`.

Hieraus lässt sich dann der HTTP Link zum Wikipedia Eintrag rekonstruieren. Um Sehenswürdigkeiten mit Wikipedia Verlinkung sichtbar von solchen ohne abzugrenzen, werden diese mit einem roten Marker gekennzeichnet. Sehenswürdigkeiten ohne Wikipedia Verlinkung besitzen hingegen einen blauen Marker. Abbildung 4 zeigt einen solchen POI mit Wikipedia Link als Popup.



Abbildung 4: POI mit Wikipedia Verlinkung

4.6 Weitere Wegpunkte hinzufügen

Ein Feature des Trip-Planners ist es, dass der Nutzer den Verlauf der Route beeinflussen kann. Dies ist möglich, indem er neben Start- und Endpunkt weitere Wegpunkte hinzufügt.

Die Anzahl dieser Wegpunkte ist unbegrenzt. Für jeden weiteren Wegpunkt wird dem Nutzer durch den Klick auf einen Button (1) ein eigenes Textfeld (2) erstellt.

Diese Textfelder sind ebenso mit der Autocomplete API verknüpft, um den Nutzer bei der Eingabe einer Adresse mit Vorschlägen zu unterstützen. Die Oberfläche für das Hinzufügen von Wegpunkten ist in Abbildung 5 dargestellt.

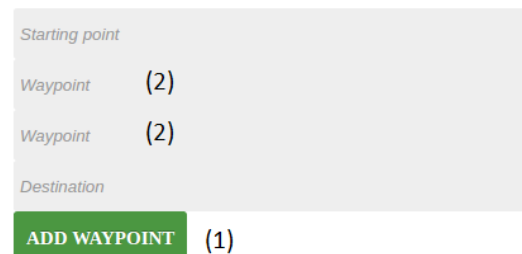


Abbildung 5: Benutzeroberfläche zum Hinzufügen von Wegpunkten

4.7 Nichtfunktionale Anforderungen

Sicherheit: Die Daten der Anwendung sind nach außen abgeschirmt. Der Nutzer kann lediglich durch GET-Requests auf die Daten des Servers zugreifen. Die Anwendung ist client- und serverseitig so konzipiert, dass der Nutzer den Suchradius um die Route nur innerhalb festgelegter Grenzen ändern kann, um den Server nicht mit einem zu großen Radius zu überlasten.

Benutzbarkeit: Die Anwendung ist minimalistisch gehalten. Alle Elemente der grafischen Oberfläche dienen entweder der Eingabe oder der Anzeige der Route inklusive der Sehenswürdigkeiten. Alle Interaktionsmöglichkeiten des Nutzers sind eindeutig beschriftet. Eine zusätzliche Erklärung der einzelnen Komponenten der Anwendung ist so nicht erforderlich. Da das Erstellen der Route bei voneinander weit entfernten Punkten einige Zeit in Anspruch nehmen kann, wird dem Nutzer eine Animation angezeigt, solange die Anfrage bearbeitet wird.

Performance und Skalierbarkeit: Die Reaktionsgeschwindigkeit eines Systems ist von großer Bedeutung. Aus diesem Grund sind alle Datenbankabfragen so weit wie möglich optimiert. So nutzt z.B. die Suche nach Sehenswürdigkeiten in einem Bereich räumliche Indexstrukturen, um die Datenbankabfrage zu beschleunigen. Somit ist sichergestellt, dass auch bei einer sehr großen Anzahl an Sehenswürdigkeiten in der Datenbank nicht alle mit dem Suchbereich verglichen werden müssen.

5 Zusammenfassung

Die Anwendung *Trip-Planner* setzt die Aufgabenstellung um, eine optimale Route zwischen zwei Punkten mit möglichst vielen Sehenswürdigkeiten zu finden. Zusätzlich kann der Nutzer selbstgewählte Wegpunkte angeben, die bei dem Routing berücksichtigt werden sollen. Nachdem eine Route erzeugt wurde, wird diese Visualisiert und zudem noch zu manchen Sehenswürdigkeiten ein Wikipedia Eintrag verlinkt.

Um die Anwendung realisieren, kommen verschiedene Datenquellen und Dienste zum Einsatz. Am Wichtigsten sind die OpenStreetMap Daten, aus denen alle Sehenswürdigkeiten extrahiert wurden. Die Autocomplete API unterstützt den Nutzer durch die Anzeige einer Dropdown-Liste von möglichen Adressen und Orten noch während der Eingabe. Um den eingegebenen Orten Koordinaten zuzuweisen, wird die Nomination API abgefragt. Diese Koordinaten werden von der Routing API benötigt, um wiederum die Route zwischen den vom Nutzer gewählten Punkten zu erstellen.

Es gab einige Geschwindigkeitsoptimierungen, um dem Anwender ein besseres Nutzererlebnis zu bieten. Einer der relevantesten ist die Extraktion der OpenStreetMap Daten. Bei Anfragen, die an die in Kapitel 2.1 erwähnte REST-Schnittstelle etwa 10 Sekunden gedauert haben, war es möglich, diese lokal in unter 2 Sekunden zu realisieren.

Es konnten alle Kern- und einige zusätzliche Features umgesetzt werden. Hierzu zählt z.B. der Anwendungsserver. Das Grails Framework stellt sowohl den Server, als auch die grafische Oberfläche im Client bereit. Nachdem ein Nutzer Start- und Endpunkt, sowie möglicherweise Wegpunkte in der Oberfläche eingegeben hat, wird eine kürzeste Route erstellt. Anhand dieser Route werden in der Datenbank alle nahen Sehenswürdigkeiten abgefragt und eine zweite Route mit diesen Erstellt. Diese wird dem Nutzer abschließend wieder im Client angezeigt. Zusätzlich werden dem Anwender noch Links zu der Wikipedia Ressource einiger POIs angezeigt.

Bei der Umsetzung dieser Features wurde auch auf nichtfunktionale Anforderungen, wie Sicherheit, Benutzbarkeit, sowie Performance und Skalierbarkeit geachtet.

Der Trip-Planer umfasst verschiedenste Datenquellen und Dienste, um die genannten Features umzusetzen. Er wurde somit erfolgreich im Kontext des Semantic Web konzipiert und umgesetzt.

Abbildungsverzeichnis

1	Autocomplete API	6
2	Ungefilterte POIs	9
3	POIs ohne <i>schwache</i> Cluster	9
4	POI mit Wikipedia Verlinkung	10
5	Benutzeroberfläche zum Hinzufügen von Wegpunkten	10

Codefragmente

1	Aufteilen der Cluster	7
2	Filtern von dünnbesetzten Clustern	9