# Modul 165: LB2 Projektarbeit

## SkyGraph: Waypoint Router für die kommerzielle Luftfahrt in der Schweiz

| | |
|---|---|
| Modul | IET-165 – NoSQL-Datenbanken einsetzen |
| Eingereicht von | Simon Leutert und Jonas Vetsch<br>Klasse INFW2025a |
| Eingereicht bei | Nicolas Dumermuth |
| Datum | 23. März 2026 |

# Inhaltsverzeichnis

# 1 NoSQL-Datenbank

## 1.1 Projektidee und Begründung der Wahl der NoSQL-Datenbank

Am Anfang unseres Projekts stand zunächst die Entscheidung für die verwendete Technologie. Die Idee, die Projektarbeit mit Neo4j umzusetzen, kam von Jonas. Für uns beide war schnell klar, dass wir diese Gelegenheit nutzen wollten, um einmal praktisch mit einer Graphdatenbank zu arbeiten. In vielen typischen Softwareprojekten werden vor allem relationale Datenbanken eingesetzt, während Graphdatenbanken eher in spezialisierten Anwendungsfällen genutzt werden. Deshalb ist die Wahrscheinlichkeit klein, dass wir im zukünftigen beruflichen Umfeld regelmässig mit Neo4j arbeiten werden. Genau aus diesem Grund wollten wir die Chance nutzen, im Rahmen dieser Projektarbeit bewusst ein Projekt umzusetzen, das die Möglichkeiten einer Graphdatenbank gezielt nutzt und deren Stärken möglichst gut ausschöpft.

Neo4j hat uns insbesondere deshalb interessiert, weil es besonders gut geeignet ist, Netzwerke und Beziehungen zwischen Objekten abzubilden. Während in relationalen Datenbanken Beziehungen meist über Tabellenverknüpfungen modelliert werden (welche mit ressourcenhungrigen JOINs wieder verbunden werden müssen), stehen in einer Graphdatenbank die Verbindungen zwischen einzelnen Knoten direkt als eigene Datensätze zur Verfügung. Dadurch lassen sich Netzwerke mit vielen miteinander verbundenen Elementen effizient darstellen und analysieren. Zudem ermöglicht Neo4j eine anschauliche grafische Darstellung solcher Strukturen, was komplexe Zusammenhänge leichter verständlich macht.

Nachdem feststand, dass wir mit Neo4j arbeiten möchten, suchten wir nach einer passenden Anwendung. Da Graphdatenbanken besonders gut für Probleme geeignet sind, bei denen Wege innerhalb eines Netzwerks analysiert werden, lag der Gedanke nahe, ein System zur Routenberechnung zu entwickeln. Ein bekanntes Beispiel für solche Probleme sind Navigationssysteme, die innerhalb eines grossen Netzwerks den optimalen Weg zwischen zwei Punkten bestimmen.

Ein vollständiges Strassennavigationssystem wäre jedoch für den Umfang unseres Projekts deutlich zu komplex gewesen. Deshalb suchten wir nach einem vergleichbaren, aber überschaubareren Szenario. Dabei entstand die Idee, ein System zur Berechnung von Flugrouten innerhalb der Schweiz zu entwickeln.

Diese Idee hat auch einen persönlichen Bezug: Jonas ist Segelflugpilot und beschäftigt sich somit bereits intensiver mit Luftfahrt und Navigation. Auch Simon interessiert sich für Luftfahrt und plant langfristig, die Privatpilotenlizenz (PPL) zu erwerben. Dadurch lag es nahe, ein Thema aus diesem Bereich aufzugreifen.

In der Luftfahrt verlaufen Flugrouten in der Regel nicht einfach als direkte Linie zwischen zwei Flughäfen. Stattdessen orientieren sich Flugpläne an sogenannten Navigationspunkten oder Fixpunkten im Luftraum. Ein Flugplan besteht aus einer Reihe solcher Punkte, die nacheinander überflogen werden. Diese Punkte dienen als Referenz für die Navigation und werden auch an die Flugsicherung übermittelt.

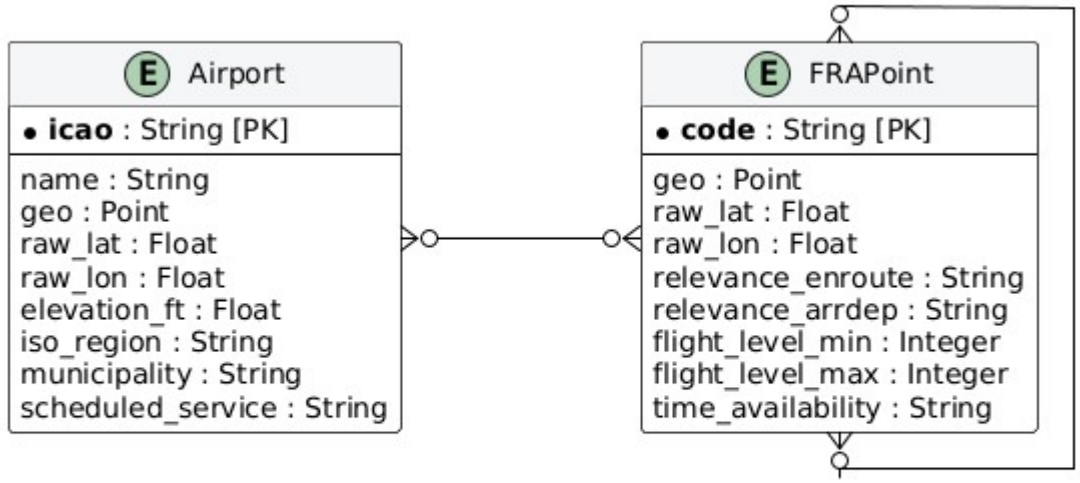
Dadurch entsteht ein Netzwerk aus Flughäfen und Navigationspunkten, zwischen denen verschiedene Verbindungen bestehen. Genau diese Struktur lässt sich sehr gut mit einer Graphdatenbank modellieren: Flughäfen und Navigationspunkte werden als Knoten, mögliche Flugverbindungen als Beziehungen dargestellt. Innerhalb dieses Netzwerks kann anschliessend berechnet werden, welche Route über diese Punkte die kürzeste Verbindung zwischen zwei Flughäfen darstellt.

Für unser Projekt haben wir uns bewusst auf die Schweiz beschränkt. Dadurch bleibt die Datenmenge überschaubar und das Projekt realistisch umsetzbar. Gleichzeitig orientiert sich das Modell weiterhin an realen Konzepten aus der Luftfahrt. Die Anwendung soll es ermöglichen, zwei Flughäfen auszuwählen und anschliessend die kürzeste Flugroute über vorhandene Navigationspunkte zu bestimmen.

Die Projektidee verbindet damit mehrere Aspekte: Einerseits wollten wir bewusst mit einer Graphdatenbank arbeiten und deren Funktionsweise besser verstehen. Andererseits bietet das Netzwerk aus Flughäfen und Navigationspunkten ein anschauliches Beispiel für ein Problem, das sich gut als Graph modellieren lässt und bei dem die Stärken von Neo4j sinnvoll genutzt werden können.

## 1.2 Konzeptionelles Datenmodell

**Konzeptionelles Datenmodell (ERD): Flugnetz**




| Symbol | Bedeutung |
|---|---|
| [PK] | **Primary Key** (Unique Constraint laut Skript) |
| }o--o{ | **N-zu-M** (Many-to-Many Beziehung) |
| geo | **Neo4j Point** (Koordinaten aus dms_to_decimal) |

Die Abbildung zeigt das konzeptionelle Datenmodell des Flugnetzes in Form eines reduzierten Entity-Relationship-Diagramms. Es definiert die zwei zentralen Entitäten Airport und FRAPoint (Navigationspunkte der kommerziellen Luftfahrt, **F**ree **R**oute **A**irspace **Point**) mit ihren jeweiligen Primärschlüsseln und Pflichtattributen.

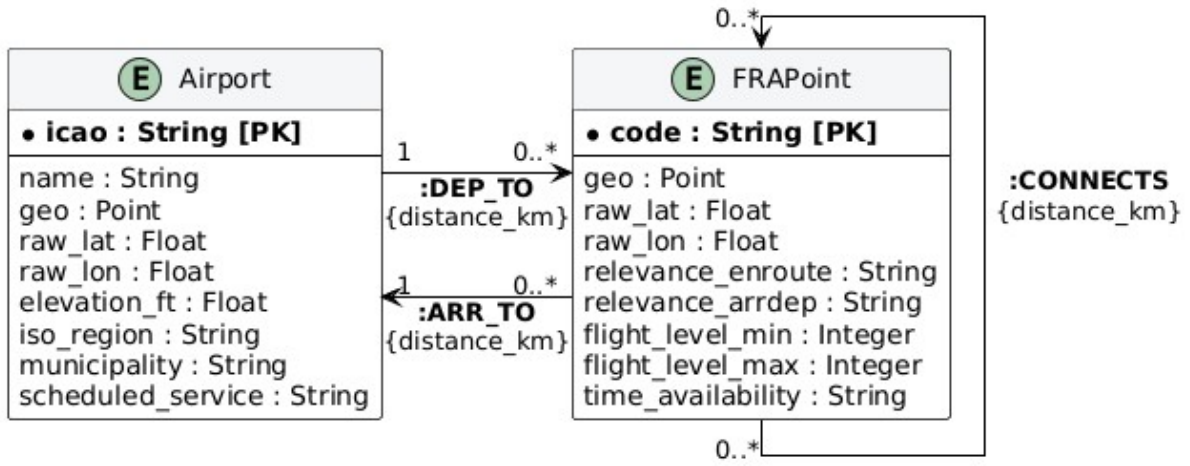
Strukturell verdeutlicht dieses Modell die wichtigste Grundregel des Netzwerks, welches wir nachbilden wollen: Es existiert bewusst keine direkte Beziehung zwischen zwei Flughäfen. Stattdessen sind Flughäfen über eine N:M-Beziehung an die Wegpunkte (FRAPoints) angebunden. Die Wegpunkte wiederum sind durch eine rekursive N:M-Beziehung (Selbstverlinkung) untereinander verknüpft. Diese Modellierung bildet die Grundlage für das vollständig vermaschte Flugnetz, in dem jede Route zwingend über das FRAPoint-Netzwerk verlaufen muss.

## 1.3 Logisches Datenmodell

Da für die Umsetzung eine Neo4j-Graphendatenbank gewählt wurde, bietet sich ein klassisches relationales Tabellenschema hier nicht an. Um den Weg vom abstrakten ERD zur eigentlichen Datenbank trotzdem greifbar zu machen, ist die Dokumentation in zwei Schritte unterteilt.

### 1.3.1 Das logische Datenmodell (Graph-Schema)

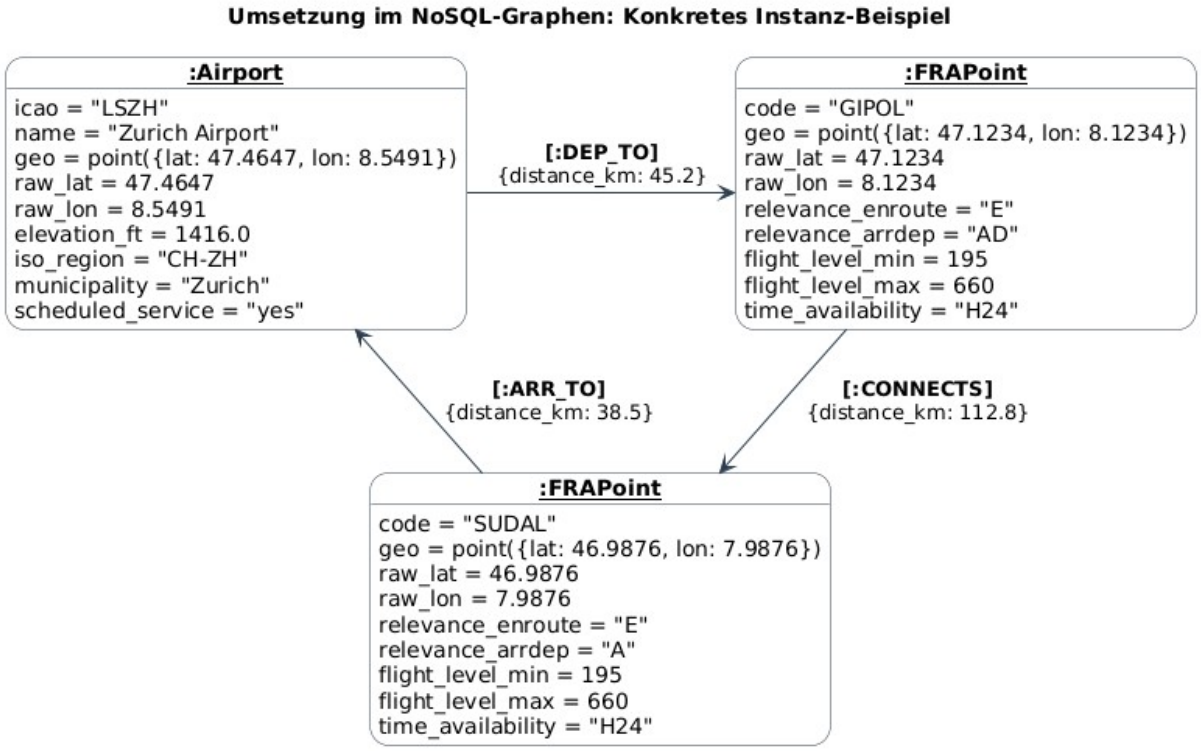
**Logisches Datenmodell / Graph-Schema: Neo4j Flugnetz (Vollständig)**




| Element | Neo4j Mapping laut Skript-Spezifikation |
|---|---|
| **[PK]** | Eindeutiger Schlüssel (Unique Constraint). |
| **raw_lat/lon** | Originale Dezimalwerte nach der DMS-Umwandlung. |
| **geo** | Point-Objekt für räumliche Abfragen (Index-fähig). |
| **flight_level** | Aufgeteilte Ganzzahlen aus 'Level Availability' (FLXXX). |
| **{...}** | Kanten-Attribut: Distanz in Kilometern (Float). |

Die erste Abbildung zeigt den strukturellen Bauplan der Graphendatenbank. Aus den ursprünglichen Entitäten des ERDs sind hier Neo4j-typische Knoten-Labels (:Airport und :FRAPoint) geworden. Die abstrakten n:m-Beziehungen haben wir in konkrete, gerichtete Kanten (:DEP_TO, :ARR_TO, :CONNECTS) umgewandelt. Typisch für einen Property Graph ist hier auch gut erkennbar, dass die Distanz (distance_km) direkt als Eigenschaft auf den jeweiligen Kanten gespeichert wird.

### 1.3.2 Die konkrete NoSQL-Umsetzung (Instanz-Beispiel)



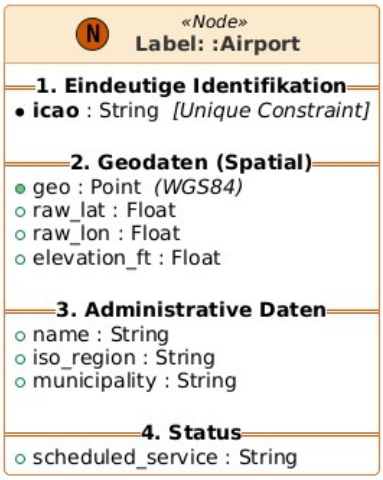

Die Aufgabenstellung verlangt ein konkretes Datenbeispiel, vergleichbar mit einem JSON-Dokument in MongoDB. Um dies graphengerecht zu lösen, zeigt die zweite Abbildung einen realen Datenausschnitt (einen Instanz-Graphen). Am Beispiel des Flughafens Zürich (LSZH) und der Wegpunkte GIPOL und SUDAL sieht man, wie die Eigenschaften als Key-Value-Paare direkt in den Knoten liegen. Gleichzeitig macht das Diagramm deutlich, wie diese einzelnen Datensätze über die Kanten mit den exakten Distanzwerten logisch miteinander vernetzt sind.

### 1.3.3 Fazit zur Umsetzung

Anstatt nur isolierte Datensätze abzubilden (wie es bei einem reinen JSON-Beispiel der Fall wäre), bringt diese Darstellungsform den grössten Vorteil der gewählten NoSQL-Lösung auf den Punkt: Die eigentlichen Daten (Nodes) und ihre Verbindungen (Relationships) sind gleichwertig und lassen sich direkt zusammen abfragen.

## 1.4 Hauptentitätsmenge mit Attributen

**Logisches NoSQL-Schema: Hauptentität (Node)**




Die Abbildung illustriert das logische NoSQL-Schema der Hauptentität des Flugnetzes, repräsentiert durch den Knoten (Node) mit dem Label **`:Airport`**. Um die interne Struktur des Datenbank-Dokuments präzise darzustellen, sind die Eigenschaften in folgende vier fachliche Kategorien gegliedert:

### 1.4.1 Eindeutige Identifikation

Das Attribut **`icao`** dient als Primärschlüssel. In der Neo4j-Datenbank wird dieses Feld zwingend mit einem *Unique Constraint* versehen, um die Eindeutigkeit der Flughäfen sicherzustellen und Duplikate beim Import auf Basis des ICAO-Codes zu verhindern.

### 1.4.2 Geodaten (Spatial)

Die räumliche Position wird in den nativen Neo4j-Datentyp **`Point`** (WGS84) konvertiert. Zusätzlich werden die ursprünglichen Dezimalwerte in **`raw_lat`** und **`raw_lon`** sowie die Höhenangabe in **`elevation_ft`** gespeichert. Dies ermöglicht sowohl performante räumliche Abfragen als auch die präzise Nachvollziehbarkeit der konvertierten Quelldaten.

### 1.4.3 Administrative Daten

Die Attribute **`name`**, **`iso_region`** und **`municipality`** definieren die geografische und administrative Zuordnung des Flughafens (Name, Region und Gemeinde) und dienen der detaillierten Filterung im globalen Flugnetz.

### 1.4.4 Status

Das Feld **`scheduled_service`** gibt Aufschluss darüber, ob der Flughafen für den regulären Linienverkehr vorgesehen ist, was für die Relevanzbewertung bei der Routenplanung entscheidend ist.

### 1.4.5 Fazit zur Umsetzung

Mit dieser Struktur werden alle für den Import-Prozess relevanten Kerneigenschaften des Flughafen-Knotentyps vollständig, strukturiert und datenbankgerecht für die Graph-Umgebung abgebildet.

## 1.5 Einfügen von Daten

### 1.5.1 Datenquelle

Die in diesem Projekt verwendeten Daten stammen aus zwei Quellen: einer offenen Flughafen-Datenbank sowie einem Datensatz zu Free-Route-Airspace-Punkten von Eurocontrol.

Die Flughafen-Daten wurden aus der öffentlich zugänglichen Datenbank OurAirports bezogen. Konkret wurde die Datei *airports.csv* verwendet, welche eine weltweit gepflegte Liste von Flughäfen und Flugplätzen enthält. Der Datensatz umfasst unter anderem Informationen wie ICAO-

Code, Name des Flughafens, Typ, geografische Koordinaten sowie Länderzugehörigkeit. Die Daten werden als CSV-Dateien bereitgestellt und können frei verwendet werden.

Für dieses Projekt wurden daraus ausschliesslich Flughäfen in der Schweiz berücksichtigt. Zusätzlich wurden nur diejenigen Flughäfen übernommen, die im Eurocontrol-Datensatz auch tatsächlich als Departure- oder Arrival-Point vorkommen. Dadurch wird sichergestellt, dass nur Flughäfen im Modell enthalten sind, die im betrachteten Datensatz auch operativ relevant sind.

Die FRA-Points (Free Route Airspace Points) stammen aus einem Datensatz von Eurocontrol (AIRAC Cycle 2603). Diese Punkte sind Wegpunkte im europäischen Free-Route-Luftraum, die für die Flugroutenplanung verwendet werden. Aus dieser Liste wurden nur diejenigen Punkte verwendet, die unter der Verantwortung der Schweizer Flugsicherung (Skyguide) stehen. Im Datensatz sind diese mit dem Präfix **„**LSFRA**“** gekennzeichnet.

Durch diese Filterung beschränkt sich der verwendete Datensatz bewusst auf relevante Infrastruktur im Schweizer Luftraum, was eine konsistente und realitätsnahe Modellierung der Flugrouten ermöglicht.

### 1.5.2 Automatisierter Datenimport via Python-Skript

Für den effizienten und fehlerfreien Import der Daten, die uns als CSV -Dateien vorlagen, in die Neo4j-Datenbank, haben wir ein Python-Skript entwickelt. Im Skript nutzen wir den offiziellen Neo4j-Datenbanktreiber, bereinigen die Daten und generieren die Kanten.

Der Import-Prozess gliedert sich dabei in folgende Kernschritte:

- **Vorbereitung & Datenintegrität:** Zunächst stellt das Skript die Verbindung zur Datenbank her und legt *Unique Constraints* auf die Primärschlüssel Airport.ident und FRAPoint.code an. Dies verhindert die Erstellung von Duplikaten während des Imports.
- **Knoten-Generierung & Koordinaten-Parsing:** Das Skript liest die Entitäten iterativ aus den CSV-Dateien aus. Ein zentraler Bestandteil ist die Funktion dms_to_decimal, welche die Wegpunkt-Koordinaten aus dem rohen Format (z. B. "N460935") in Dezimalgrade umrechnet. Anschliessend werden die FRAPoint- und Airport-Knoten per Cypher-MERGE-Befehl in der Datenbank angelegt. Die Koordinaten werden dabei direkt in den nativen räumlichen Neo4j-Datentyp (Point) konvertiert.
- **Vermeidung verwaister Knoten:** Um die Vorgaben des logischen Modells strikt einzuhalten, filtert das Skript die Flughäfen vor dem Import. Es werden ausschliesslich jene Airport-Knoten in die Datenbank geschrieben, die auch tatsächlich als Start- oder Zielpunkt in den Wegpunkt-Daten (nur Wegpunkte im Verantwortungsgebiet von Skyguide, der Schweizer Luftsicherung) referenziert sind.
- **Beziehungsaufbau & Distanzberechnung:** Im letzten Schritt iteriert das Skript erneut über die Daten, um die Beziehungen (:DEP_TO, :ARR_TO) gemäss der Fluglogik zu knüpfen, sowie das Netz der Wegpunkte untereinander vollständig zu vermaschen (:CONNECTS). Ein massgeblicher Vorteil des Skripts ist dabei, dass es die exakte Distanz zwischen den Knotenpunkten mithilfe der räumlichen Neo4j-Funktion point.distance() direkt beim Import berechnet und als Eigenschaft (distance_km) auf den jeweiligen Beziehungen speichert.

Mit Ausführung dieses Skripts wird das zuvor definierte logische Datenmodell vollständig und vollautomatisiert als physischer Graph in der NoSQL-Datenbank instanziiert.

### 1.5.3 Manueller Import

#### *1.5.3.1 Einzelne Entitäten (Knoten) erstellen*

Um einen Flughafen und einen FRA-Punkt anzulegen, nutzt man den CREATE oder MERGE Befehl. MERGE ist vorzuziehen, da es Dubletten verhindert (es erstellt den Knoten nur, wenn er noch nicht existiert).

**Einen Flughafen anlegen**

```
MERGE (a:Airport {icao: 'LSZH'})
SET a.name = 'Zurich Airport',
    a.geo = point({latitude: 47.458, longitude: 8.548}),
    a.elevation_ft = 1416;
```

**Einen FRA-Punkt (Wegpunkt) anlegen**

```
MERGE (p:FRAPoint {code: 'DEGES'})
SET p.geo = point({latitude: 47.635, longitude: 9.321}),
    p.relevance_enroute = 'YES';
```

#### *1.5.3.2 Entitäten verknüpfen (Beziehung erstellen)*

Um eine Verbindung zwischen zwei bereits existierenden Knoten herzustellen, müssen diese zuerst mit MATCH "gesucht" und dann mit MERGE verbunden werden.

**Beispiel: Abflug von Zürich zum Punkt DEGES**

```
MATCH (a:Airport {icao: 'LSZH'})
MATCH (p:FRAPoint {code: 'DEGES'})
MERGE (a)-[r:DEP_TO]->(p)
SET r.distance_km = point.distance(a.geo, p.geo) / 1000;
```

**Erläuterung der Syntax**

| | |
|---|---|
| `(a:Airport)` | Das a ist eine Variable (ein Platzhalter), :Airport ist das Label (die Kategorie). |
| `{ident: 'LSZH'}` | Das ist die Eigenschaft (Property), nach der gesucht wird. |
| `-[r:DEP_TO]->` | Das definiert die Richtung der Beziehung (r ist die Variable, :DEP_TO der Typ). |
| `point.distance(...)` | Eine eingebaute Funktion, die den Abstand zwischen zwei Geokoordinaten in Metern berechnet (daher / 1000 für Kilometer). |

## 1.6 Ändern von Daten

Nach erfolgreichem Import der Daten führen wir nachfolgend pro Entitätstyp einen Befehl auf, um eine Entität zu ändern und zu löschen.

### 1.6.1 :FRAPoint – Attribut ändern

```
MATCH (p:FRAPoint {code: 'GIPOL'})
SET p.time_availability = 'H24'
RETURN p
```

### 1.6.2 :Airport – Attribut ändern

```
MATCH (a:Airport {icao: 'LSZH'})
SET a.scheduled_service = 'no'
RETURN a
```

## 1.7 Löschen von Daten

Da beide Knotentypen immer mit Beziehungen verbunden sind (keine verwaisten Knoten), muss zwingend **`DETACH DELETE`** verwendet werden – andernfalls wirft Neo4j einen Fehler.

**`DETACH DELETE`** entfernt den Knoten zusammen mit allen zugehörigen Beziehungen **`(:DEP_TO, :ARR_TO, :CONNECTS)`** in einem einzigen Befehl. Ein einfaches **`DELETE`** ohne **`DETACH`** würde fehlschlagen, solange noch Kanten am Knoten hängen. Dies ist bei unserem Modell per Design immer der Fall, da keine verwaisten Knoten existieren.

### 1.7.1 :FRAPoint – löschen (inkl. aller Beziehungen)

```
MATCH (p:FRAPoint {code: 'GIPOL'})
DETACH DELETE p
```

### 1.7.2 :Airport – löschen (inkl. aller Beziehungen)

```
MATCH (a:Airport {icao: 'LSZH'})
DETACH DELETE a
```

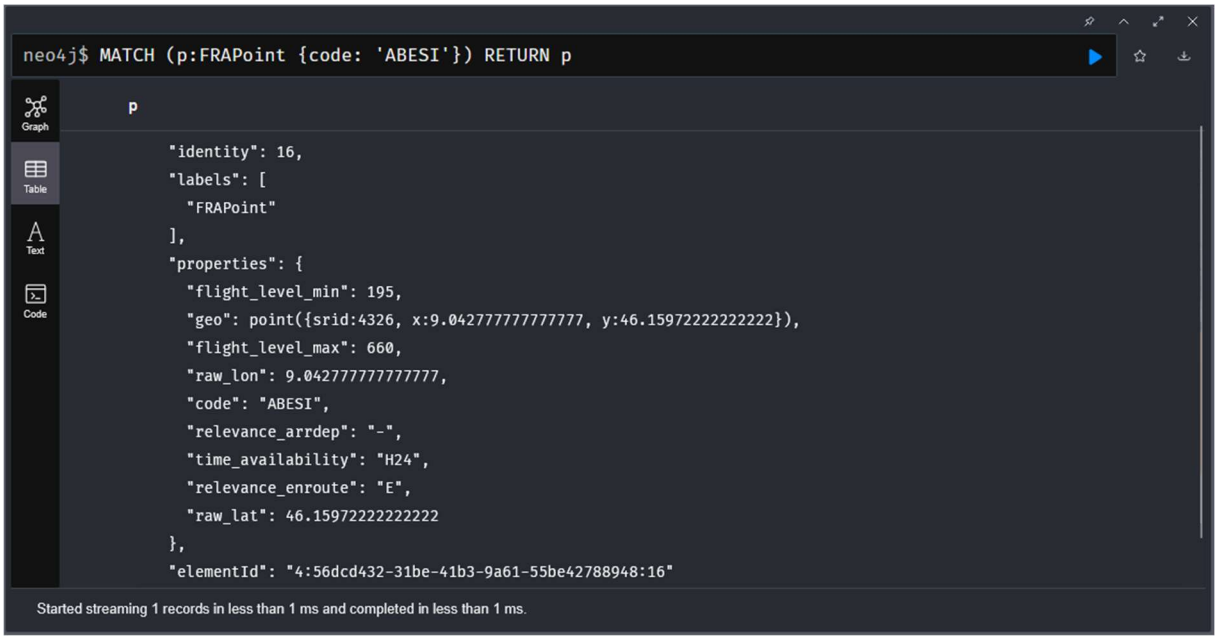
## 1.8 Dynamische Daten

Unsere Entitäten haben unterschiedliche Attribute: Ein FRAPoint hat andere Attribute als ein Airport. Wenn für ein Attribut kein Wert vorhanden ist, wird das Attribut auch nicht gelistet.

Nachfolgend ein Beispiel für zwei Datensätzen mit unterschiedlichen Attributen.

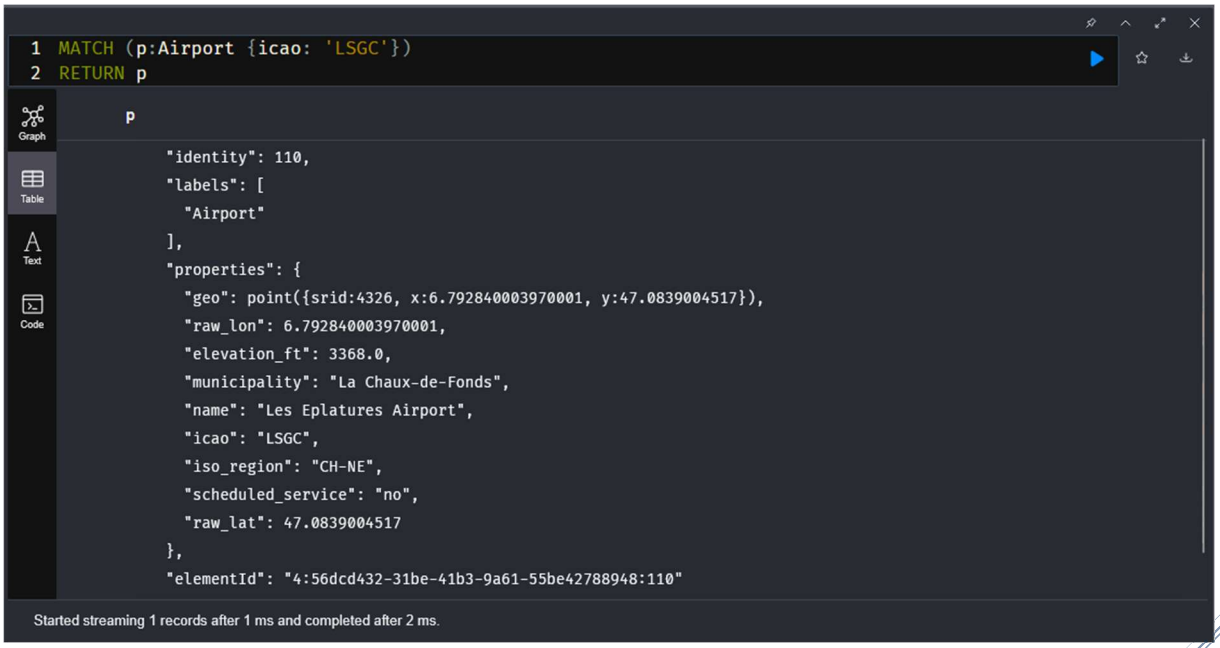
### 1.8.1 FRAPoint ABESI

```
MATCH (p:FRAPoint {code: 'ABESI'})
RETURN p
```




### 1.8.2 Regionalflugplatz Les Eplatures

```
MATCH (p:Airport {icao: 'LSGC'})
RETURN p
```



## 1.9 Anzeigen von Daten

Nachfolgend führen wir einige Befehle auf, um Daten anzuzeigen.

### 1.9.1 Eine Entität anzeigen

Um nur eine Entität anzuzeigen setzen wir den Filter für die Entitäten (nach «:») auf FRAPoint bzw. Airport. Danach suchen wir nach dem Punkt bwz. Flughafenkürzel, welches wir finden wollen.

```
MATCH (p:FRAPoint {code: 'ABESI'})

RETURN p
```

*Ausgabe siehe oben.*

```
MATCH (p:Airport {ident: 'LSGC'})

RETURN p
```

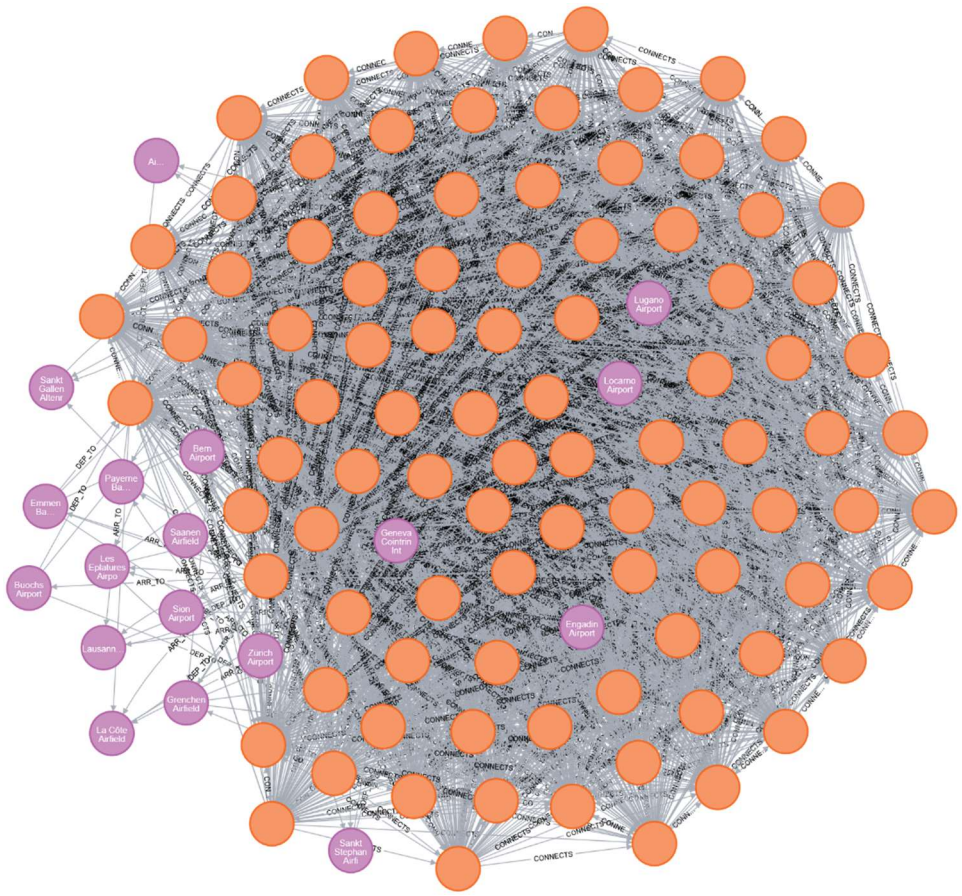
*Ausgabe siehe oben.*

### 1.9.2 Alle Entitäten anzeigen

Um alle Entitäten anzuzeigen, führen wir folgenden Befehl aus.

```
match (n) return n;
```

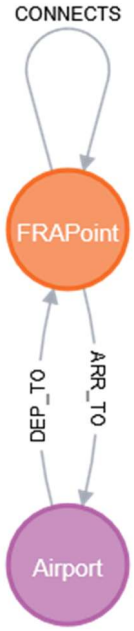
Diese Abfrage dauert einige Sekunden, da wir damit mehr als 4500 Datensätze abfragen. Die Ausgabe sieht in graphischer Form wie folgend aus.



### 1.9.3 Weitere praxisnahe Befehle, um Daten anzuzeigen

#### *1.9.3.1 Das Datenbankschema grafisch anzeigen*

```
CALL db.schema.visualization();
```




#### *1.9.3.2 Die Anzahl Entitäten in der Datenbank anzeigen*

```
MATCH (n)
RETURN count(n)
```

```
1 MATCH (n)
2 RETURN count(n)
```

| | count(n) |
|---|---|
| 1 | 112 |

#### *1.9.3.3 Die Anzahl der Entitäten pro Entität anzeigen*

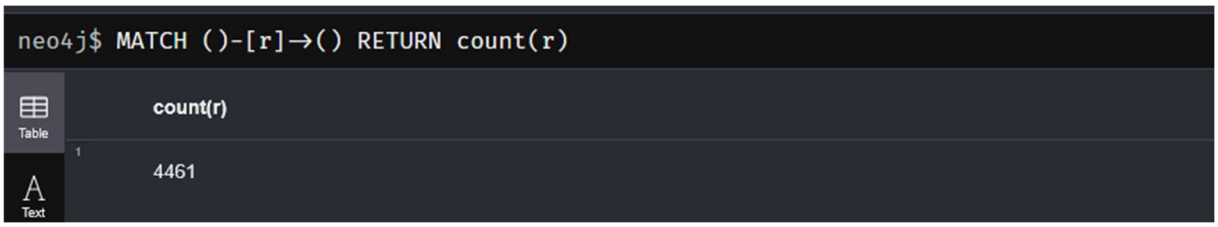
```
MATCH (n)
RETURN labels(n) AS label, count(n) AS anzahl
```

```
1 MATCH (n)
2 RETURN labels(n) AS label, count(n) AS anzahl
```

| | label | anzahl |
|---|---|---|
| 1 | ["FRAPoint"] | 94 |
| 2 | ["Airport"] | 18 |

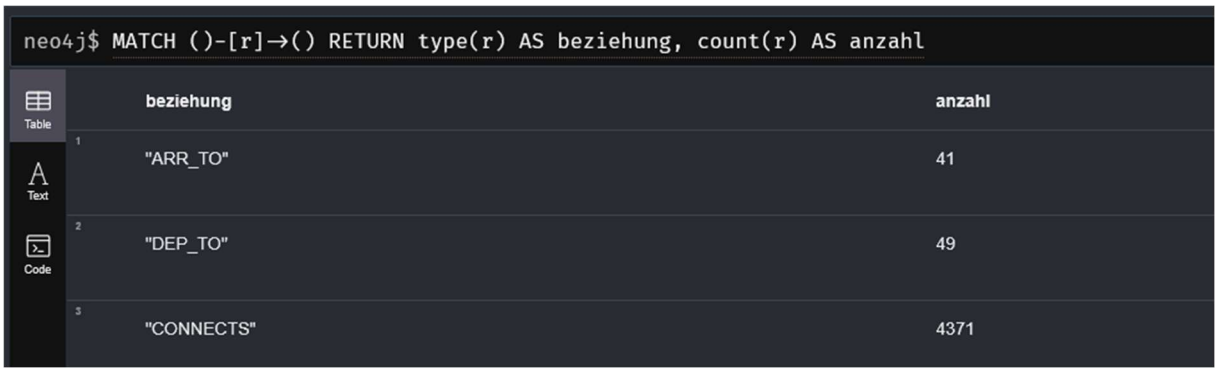
#### 1.9.3.4 Die Anzahl der Beziehungen (Kanten) anzeigen

```
MATCH ()-[r]->()
RETURN count(r)
```




#### 1.9.3.5 Die Anzahl der Beziehungen (Kanten) pro Beziehungsart anzeigen

```
MATCH ()-[r]->()
RETURN type(r) AS beziehung, count(r) AS anzahl
```




#### 1.9.3.6 Kürzeste Flugroute zwischen zwei Flughäfen berechnen

Die spannendste Abfrage für den Use-Case unserer Datenbank ist die Berechnung der kürzesten Flugroute zwischen zwei Flughäfen. Grundsätzlich bildet die direkte Luftlinie zwischen zwei Flughäfen die kürzeste Route. In der kommerziellen Luftfahrt werden Flugrouten allerdings entlang offizieller Navigationspunkte (FRAPoints) geplant, weshalb auch wir im Rahmen dieses Projekts die kürzeste Flugroute via dieser FRAPoints berechnen wollen.

Dazu suchen wir zuerst nach einem FRAPoint, der zum Ausgangsflughafen mit einer Departure-Beziehung verbunden ist. Das bedeutet, dass es für den Ausgangsflughafen eine standardisierte Abflugroute gibt, die zu diesem FRAPoint führt. In gewissen Fällen hat dieser Punkt, aufgrund der überschaulichen geographischen Grösse der Schweiz, bereits eine Arrival-Beziehung zum gewünschten Zielflughafen. Das bedeutet, dass es für den Zielflughafen eine standardisierte Anflugroute gibt, die diesen FRAPoint und den Flughafen verbindet.

Hat der erste FRAPoint keine Arrival-Beziehung zum gewünschten Zielflughafen, verbinden wir zu einem FRAPoint, der eine solche Beziehung hat. Daraus ergibt sich ein Kandidat für die kürzeste Flugroute.

Im Anschluss generieren wir alle weiteren Möglichkeiten (Kandidaten) für Flugrouten zwischen den zwei gewünschten Flughäfen und wählen daraus die kürzeste berechnete Strecke aus. Damit erhalten wir die kürzeste Flugroute zwischen zwei Flughäfen, die so als offizielle Flugroute für die kommerzielle Luftfahrt gelten könnte.

Der Umfang dieses Projekts ist begrenzt und der Luftraum der Schweiz sehr klein. Entsprechend haben wir uns darauf beschränkt, dass eine Flugroute aus jeweils maximal zwei

FRAPoints besteht. Damit bliebt die Komplexität und der Rechenaufwand einer Abfrage im Rahmen.

In nachfolgendem Beispiel fragen wir die kürzeste Flugroute zwischen den Flughäfen Genf und Lugano ab. Die Route führt über die FRAPoints KORED und SOSON und ist 286 km lange.

```
MATCH (start:Airport {icao: 'LSGG'}), (target:Airport {icao: 'LSZA'})


// Erlaubt Wege mit 2 bis 3 Kanten (also 1 bis 2 FRAPoints dazwischen)
MATCH p = (start)-[:DEP_TO|CONNECTS|ARR_TO*2..3]->(target)


// Summiert die Kilometer für alle diese Routen
WITH p, reduce(total = 0, r IN relationships(p) | total + r.distance_km) AS
 total_distance_km


// Sortiert nach der geringsten Distanz und gibt den Gewinner aus
ORDER BY total_distance_km ASC
LIMIT 1


RETURN
  [n IN nodes(p) | coalesce(n.icao, n.ident, n.code)] AS path,
  total_distance_km
```

```
1  MATCH (start:Airport {icao: 'LSGG'}), (target:Airport {icao: 'LSZA'})
2  // Erlaubt Wege mit 2 bis 3 Kanten (also 1 bis 2 FRAPoints dazwischen)
3  MATCH p = (start)-[:DEP_TO|CONNECTS|ARR_TO*2..3]→(target)
4  // Summiert die Kilometer für alle diese Routen
5  WITH p, reduce(total = 0, r IN relationships(p) | total + r.distance_km) AS total_distance_km
6  // Sortiert nach der geringsten Distanz und gibt den Gewinner aus
7  ORDER BY total_distance_km ASC
8  LIMIT 1
9
10 RETURN
11   [n IN nodes(p) | coalesce(n.icao, n.ident, n.code)] AS path,
12   total_distance_km
```

Table / Text

| | path | total_distance_km |
|---|---|---|
| 1 | ["LSGG", "KORED", "SOSON", "LSZA"] | 286.34861292774434 |

# 2 Datenbankoperationen und -architektur

## 2.1 Zugriffsberechtigungen: Konzept

Das Sicherheitskonzept von SkyGraph basiert auf dem Prinzip des geringsten Privilegs (Principle of Least Privilege). Jeder Akteur und jeder Systemdienst erhält nur die Rechte, die für seine spezifische Aufgabe zwingend notwendig sind. Wir definieren drei funktionsbezogene Hauptrollen mit entsprechenden Service- und Administrationskonten:

### 2.1.1 Rolle 1: Role_Admin (Datenbank-Administration)

- **Benutzer:** simon, jonas
- **Zweck:** Vollständige Verwaltung der Neo4j-Datenbank, des Servers (VPS) und der Benutzerarchitektur.
- **Berechtigungen:** Uneingeschränkte Lese- und Schreibrechte auf alle Graphen (ALL GRAPH PRIVILEGES) sowie administrative Rechte zur Benutzer- und Rollenverwaltung (ALL DATABASE MANAGEMENT PRIVILEGES). Darf Indizes und Constraints erstellen oder löschen.

### 2.1.2 Rolle 2: Role_Import (Datenimport)

- **Benutzer:** svc_import
- **Zweck:** Diese Rolle wird exklusiv vom Python-Skript für den automatisierten Datenimport und die Aktualisierung der Wegpunkte verwendet.
- **Berechtigungen:** Darf Knoten (Airport, FRAPoint) und Beziehungen (DEP_TO, ARR_TO, CONNECTS) lesen, erstellen, aktualisieren und löschen (CRUD-Operationen). Hat jedoch keine Berechtigung, neue Benutzer anzulegen oder das Schema (Constraints/Indizes) zu verändern.

### 2.1.3 Rolle 3: Role_app (Backend)

- **Benutzer:** svc_app
- **Zweck:** Wird von der Backend-Applikation genutzt, um die Flugrouten für den Endanwender zu berechnen und anzuzeigen, Datensätze zu verändern und neue zu erstellen.
- **Berechtigungen:** Darf Knoten (Airport, FRAPoint) und Beziehungen (DEP_TO, ARR_TO, CONNECTS) lesen, erstellen, aktualisieren und löschen (CRUD-Operationen). Hat jedoch keine Berechtigung, neue Benutzer anzulegen oder das Schema (Constraints/Indizes) zu verändern.

### 2.1.4 Berechtigungsmatrix

Zur besseren Übersichtlichkeit fassen wir die Berechtigungen in folgender Matrix zusammen:

| Rolle | Knoten & Kanten (Lesen) | Knoten & Kanten (Schreiben/Löschen) | Schema (Indizes/Constraints) | Benutzerverwaltung |
|---|---|---|---|---|
| **Role_Admin** | Ja | Ja | Ja | Ja |
| **Role_Import** | Ja | Ja | Nein | Nein |
| **Role_App** | Ja | Ja | Nein | Nein |

**Hinweis zur technischen Umsetzung:** Das vorliegende Konzept wurde gemäss Best Practices für Enterprise-Umgebungen entworfen. Da für dieses Projekt die Neo4j Community Edition zum Einsatz kommt, ist die feingranulare Rollenverteilung (RBAC) systemseitig eingeschränkt. Das logische Konzept bleibt jedoch als architektonische Richtlinie für die Applikation bestehen.

## 2.2 Zugriffsberechtigung: Umsetzung

Die Umsetzung unseres Berechtigungskonzepts gliedert sich aufgrund technischer Rahmenbedingungen in zwei Teile: Die theoretische Best-Practice-Umsetzung für Enterprise-Systeme und unsere tatsächliche Umsetzung, die die Limitierungen der genutzten Version umgeht.

### 2.2.1 Ideale Umsetzung (Best Practice mit Neo4j Enterprise)

In einer professionellen Produktionsumgebung würde das in Kapitel 2.1 definierte Rollenkonzept (Role-Based Access Control) nativ auf Datenbankebene umgesetzt. Die Datenbank erzwingt dabei die Rechte, unabhängig davon, welche Applikation darauf zugreift. Die Cypher-Befehle dafür lauten:

```
// 1. Rollen erstellen
CREATE ROLE Role_Admin;
CREATE ROLE Role_Import;
CREATE ROLE Role_App;

// 2. Granulare Berechtigungen zuweisen
GRANT ALL PRIVILEGES ON DATABASE neo4j TO Role_Admin;
GRANT TRAVERSE, READ, WRITE ON GRAPH neo4j TO Role_Import;
GRANT TRAVERSE, READ, WRITE ON GRAPH neo4j TO Role_App;
```

```
// 3. Benutzer erstellen
CREATE USER simon SET PASSWORD 'SecurePass_Simon_1!' CHANGE REQUIRED;
CREATE USER jonas SET PASSWORD 'SecurePass_Jonas_2!' CHANGE REQUIRED;
CREATE USER svc_import SET PASSWORD 'Import_Service_3!' CHANGE REQUIRED;
CREATE USER svc_app SET PASSWORD 'App_Service_4!' CHANGE REQUIRED;

// 4. Rollen den Benutzern zuweisen
GRANT ROLE Role_Admin TO simon, jonas;
GRANT ROLE Role_Import TO svc_import;
GRANT ROLE Role_App TO svc_app;
```

### 2.2.2 Das Problem der Neo4j Community Edition

Für unser SkyGraph-Projekt läuft auf dem VPS die kostenlose Neo4j Community Edition. Diese Version unterstützt das Erstellen von Benutzern (CREATE USER), blockiert jedoch sicherheitsrelevante Features wie die Rollenverwaltung (CREATE ROLE) und feingranulare Rechtevergabe (GRANT) mit einem "Feature Restricted"-Fehler.

**Der logische Konflikt:** In der Community Edition erhält jeder angelegte Benutzer standardmässig volle Schreib- und Leserechte. Ein nativer, rein lesender Benutzer (Role_App) lässt sich auf Datenbankebene technisch nicht realisieren.

### 2.2.3 Tatsächliche Umsetzung (Sicherheit auf Applikationsebene)

Da wir die Privilegien in der Datenbank nicht einschränken können, führen wir auf unserem Server lediglich die Befehle zur reinen Benutzererstellung aus:

```
// Tatsächlich ausgeführte Befehle in unserer SkyGraph-Datenbank
CREATE USER simon SET PASSWORD 'Admin_Simon_2026!' CHANGE REQUIRED;
CREATE USER jonas SET PASSWORD 'Admin_Jonas_2026!' CHANGE REQUIRED;
CREATE USER svc_import SET PASSWORD 'Service_Import_26!' CHANGE REQUIRED;
CREATE USER svc_app SET PASSWORD 'Service_App_26!' CHANGE REQUIRED;
```

Obwohl nun alle vier Benutzer datenbankseitig Schreibrechte besitzen, setzen wir unser Sicherheitskonzept aus Kapitel 2.1 durch eine strenge Architektur logisch durch:

**Administration (simon, jonas):** Wir nutzen diese Konten für manuelle administrative Eingriffe (z. B. Indizes setzen, Backups) über das Neo4j-Webinterface.

**Datenimport (sv c_import):** Das Python-Importskript authentifiziert sich mit diesem Benutzer. Da der Python-Code ausschliesslich Befehle zum Erstellen und Verknüpfen von Knoten (MERGE, MATCH) enthält, kann dieser Benutzer faktisch keine administrativen Schäden (wie das Löschen anderer User) anrichten, da die Logik dafür im Skript schlichtweg fehlt.

**Applikation/Frontend (svc_app):** Um zu verhindern, dass Endanwender mutwillig Daten verändern, greift das Frontend niemals direkt auf die Datenbank zu. Stattdessen schalten wir eine

Backend-API dazwischen. Das Backend loggt sich als svc_app in Neo4j ein. Der entscheidende Schutzmechanismus: Im Backend sind ausschliesslich lesende Cypher-Abfragen (MATCH ... RETURN) für die Wegberechnung fest programmiert. Da das Backend keine Schnittstellen für modifizierende Befehle anbietet, ist ein Read-Only-Zugriff für den Endanwender absolut sichergestellt – ganz ohne Enterprise-Lizenz.

## 2.3 Backup der DB

### 2.3.1 Strategie und Automatisierung

- **Erstellung:** Die Backups werden automatisiert über einen **Linux-Cronjob** erstellt. Dieser führt nachts um 02:00 Uhr ein Bash-Skript aus.
- **Betriebsmodus:** Es handelt sich um ein **Offline-Backup**. Der Neo4j-Dienst wird kurzzeitig gestoppt, um einen konsistenten Datenbank-Dump zu garantieren.
- **Backup-Typ:** Wir führen ein **Full Backup** (Datenbank-Dump) durch. Da Neo4j Community keine inkrementellen Backups unterstützt, wird jede Nacht der gesamte Graph (Airports, FRAPoints und Beziehungen) gesichert.
- **Aufbewahrung (Retention):** Um Speicherplatz effizient zu nutzen, bewahren wir die Backups der letzten 7 Tage sowie das jeweils letzte Backup der vergangenen 4 Wochen auf.

### 2.3.2 Authentifizierung und Berechtigungen

Das Backup wird durch den Systembenutzer mit Sudo-Rechten ausgeführt. Da der Befehl neo4j-admin direkt auf die Dateiebene des Betriebssystems zugreift, ist keine separate Datenbank-Authentifizierung (Bolt-Protokoll) für den Dump erforderlich. Der Zugriff auf den VPS selbst ist jedoch, wie im Sicherheitskonzept erwähnt, via SSH-Key geschützt.

### 2.3.3 Durchführung des Backups

Die folgenden Befehle sind im automatisierten Skript hinterlegt:

```
# 1. Den Neo4j-Dienst stoppen
sudo systemctl stop neo4j


# 2. Den Datenbank-Dump erstellen (sichert die Standard-DB 'neo4j' in den Ordner /tmp)
sudo neo4j-admin database dump neo4j --to-path=/tmp/


# 3. Den Neo4j-Dienst wieder starten
sudo systemctl start neo4j
```

*Nach erfolgreicher Ausführung befindet sich im Ordner /tmp/ eine Datei namens neo4j.dump, welche die vollständige Sicherung der Datenbank enthält und lokal oder auf einem externen Speicher archiviert werden kann.*

## 2.4 Restore eines DB-Backups

Um ein zuvor erstelltes Backup (die .dump-Datei) wiederherzustellen, muss der Datenbankdienst ebenfalls gestoppt sein. Der Parameter --overwrite-destination=true stellt sicher, dass die bestehenden Daten der aktuellen Datenbank mit dem Stand des Backups überschrieben werden.

```
# 1. Den Neo4j-Dienst stoppen
sudo systemctl stop neo4j


# 2. Das Backup wiederherstellen (lädt den Dump aus dem Ordner /tmp)
sudo neo4j-admin database load neo4j --from-path=/tmp/ --overwrite-
destination=true


# 3. Den Neo4j-Dienst wieder starten
sudo systemctl start neo4j
```

*Nach dem Neustart des Dienstes befindet sich die Datenbank wieder exakt in dem Zustand, in dem das Backup erstellt wurde.*

## 2.5 Konzept für horizontale Skalierung

Unser Projekt ist aktuell auf die Schweiz beschränkt und läuft zentral auf einem einzelnen VPS-Server in Helsinki. Wenn wir das System in Zukunft auf den weltweiten Luftverkehr ausdehnen sollten, würde die Datenmenge (Wegpunkte, Flughäfen und vor allem die Verbindungen dazwischen) massiv ansteigen. Um Ausfallsicherheit und Performance zu gewährleisten, planen wir eine horizontale Skalierung unserer Neo4j-Datenbank wie folgend.

Da unsere Applikation primär leselastig ist (die Endanwender berechnen Flugrouten, was die Pathfinding-Algorithmen beansprucht), während Schreiboperationen (Datenimport) eher selten Batches stattfinden, setzen wir auf eine Cluster-Architektur mit einer klaren Trennung von Lese- und Schreibknoten.

### 2.5.1 Cluster-Architektur und Knotenverteilung (Replikation)

Wir setzen für SkyGraph ein Causal Clustering (die Standard-Cluster-Architektur von Neo4j) ein. Dieses Konzept löst das klassische Master-Slave-Modell ab und kombiniert Multimaster-Eigenschaften für Schreibvorgänge mit hochskalierbaren Lese-Knoten.

Wir unterteilen unser Cluster in zwei Arten von Knoten (Nodes):

**Core-Nodes (Replikation & Schreib-Konsens)**

Diese Server bilden das Herzstück. Sie nehmen Schreibvorgänge (z.B. durch unseren automatisierten Python-Import) entgegen. Sie nutzen ein Multimaster-ähnliches Raft-Protokoll, bei dem die Mehrheit der Core-Nodes eine Änderung bestätigen muss, bevor sie gültig ist.

**Read-Replica-Nodes**

Diese Server enthalten eine asynchron synchronisierte Kopie (Replikation) der Datenbank. Sie beantworten ausschliesslich Leseanfragen der Applikation (Role_App).

**Anzahl und örtliche Verteilung der Knoten**

Um global niedrige Latenzen und hohe Ausfallsicherheit zu bieten, verteilen wir insgesamt 6 Server (Knoten) wie folgt:

| Server-Typ | Anzahl | Örtliche Verteilung (Rechenzentren) | Funktion im System |
|---|---|---|---|
| **Core-Nodes** | 3 | Frankfurt (DE), Zürich (CH), Helsinki (FI) | Gewährleisten Ausfallsicherheit in Europa (unserem Kernmarkt). 3 Knoten sind das Minimum, um bei Ausfall eines Servers weiterhin eine Mehrheit (Quorum) für Schreibvorgänge zu haben. |
| **Read-Replicas** | 3 | New York (USA), Singapur (SG), Zürich (CH) | Liefern die Antworten für die Benutzer-Applikation schnell aus, da sie geografisch nahe am Endnutzer platziert sind. |

### 2.5.2 Partitionierung (Sharding) bei globalen Datenmengen

Eine klassische Partitionierung ist bei Graphdatenbanken komplex, da das Zerschneiden von Beziehungen (Kanten) über mehrere Server hinweg («Dangling Edges») die Performance bei Abfragen wie der kürzesten Route drastisch verschlechtert.

Sollte unser Datenbestand jemals so gross werden, dass er nicht mehr in den RAM eines einzelnen Servers passt, nutzen wir die Neo4j Fabric-Technologie. Damit können wir den Graphen logisch aufteilen (Sharding).

**Konkrete Umsetzung der Partitionierung**

**Partition Key:** Wir nutzen das bereits im logischen Datenmodell vorhandene Attribut iso_region.

**Verteilungsschema:** Wir fassen die iso_region-Codes zu übergeordneten Kontinental-Graphen zusammen. So entstehen separate Partitionen für Europa, Nordamerika und Asien.

**Lokale Routen:** Ein Flug von Zürich (LSZH) nach London wird komplett auf der europäischen Partition berechnet, ohne Netzwerk-Overhead.

**Globale Routen:** Ein Flug von Zürich nach New York wird über einen zentralen «Fabric-Node» orchestriert, der die Leseanfrage an die europäische und die amerikanische Partition splittet und die Ergebnisse für den Anwender wieder zusammenführt.

### 2.5.3 Bemerkung zur Umsetzung der horizontalen Skalierung

Da wir die Neo4j Community Edition nutzen, konnten wir unser Konzept für die horizontale Skalierung nicht umsetzen. Wir sind mit unserer Lehrperson so verblieben, dass dies in Ordnung so ist und wir stattdessen die dadurch freigewordene Zeit und Energie in die Applikation (siehe nächstes Kapitel) investieren.

# 3 Applikation

## 3.1 Technologie / Aufbau der Applikation

Für die Applikation von SkyGraph setzen wir auf eine klassische Client-Server-Architektur, bestehend aus Frontend, Backend und Datenbank. Ziel der Anwendung ist es, dem Benutzer eine übersichtliche Weboberfläche zur Verfügung zu stellen, über die Flugrouten zwischen zwei Flughäfen innerhalb der Schweiz berechnet und visuell dargestellt werden können.

Die Anwendung trennt dabei klar zwischen Benutzeroberfläche, Geschäftslogik und Datenhaltung. Das Frontend übernimmt die Darstellung und Interaktion mit dem Benutzer, das Backend stellt eine API zur Verfügung und verarbeitet die Anfragen, während die Neo4j-Datenbank die eigentlichen Daten des Flugnetzwerks speichert.

## 3.2 Eingesetzte Technologien und Begründung

Für das Backend verwenden wir Java mit Spring Boot sowie Gradle als Build-Tool. Spring Boot ermöglicht es, REST-Schnittstellen strukturiert umzusetzen und eine klar gegliederte Applikationsarchitektur aufzubauen. Zudem bietet das Framework eine gute Integration mit Datenbanksystemen und erleichtert die Entwicklung von Web-APIs erheblich. Java eignet sich für diesen Zweck besonders gut, da es stabil, weit verbreitet und gut dokumentiert ist.

Gradle wird verwendet, um das Projekt zu strukturieren und Abhängigkeiten zu verwalten. Dadurch lassen sich Build-Prozesse automatisieren und externe Bibliotheken sauber integrieren.

Für das Frontend setzen wir auf HTML, CSS und JavaScript. Diese Technologien ermöglichen eine schlanke Weboberfläche, die direkt im Browser ausgeführt wird und keine zusätzliche Client-Software benötigt. Der Fokus liegt dabei auf einer klaren und intuitiven Darstellung der Daten sowie auf einer einfachen Benutzerinteraktion.

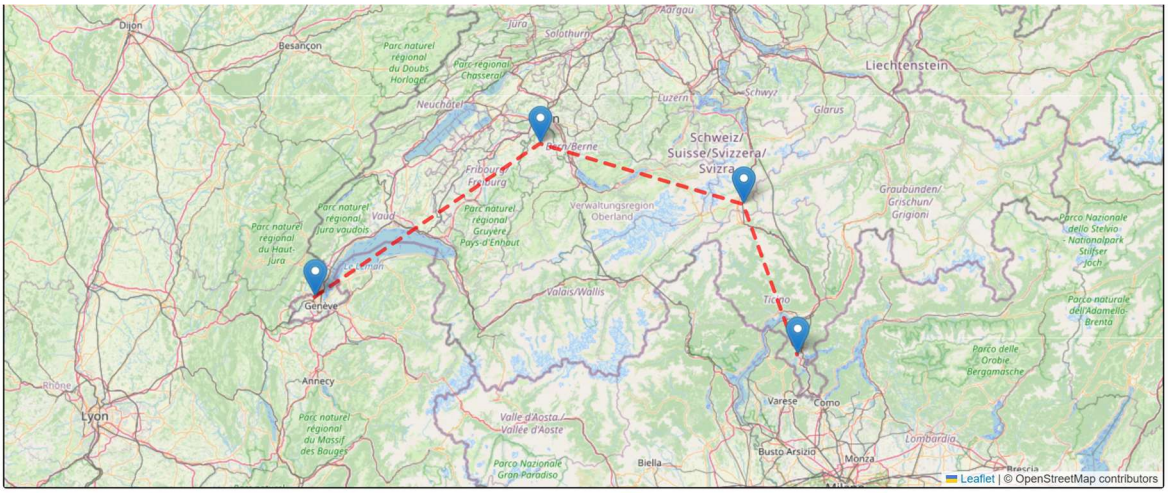
Als Datenbank verwenden wir Neo4j, eine Graphdatenbank, die sich besonders gut für Netzwerkstrukturen eignet. In unserem Anwendungsfall werden Flughäfen und Navigationspunkte als Knoten modelliert, während Flugverbindungen als Beziehungen zwischen diesen Knoten gespeichert werden. Dadurch können Routen innerhalb dieses Netzwerks effizient analysiert werden, insbesondere bei der Berechnung der kürzesten Verbindung zwischen zwei Flughäfen.

## 3.3 Funktionaler Aufbau der Anwendung

Die Anwendung stellt dem Benutzer eine Weboberfläche zur Verfügung, über die Flugrouten innerhalb des modellierten Netzwerks berechnet und dargestellt werden können.

Im Zentrum der Anwendung steht die Berechnung der kürzesten Flugroute zwischen zwei Flughäfen. Der Benutzer wählt dazu im Frontend über zwei Dropdown-Menüs einen Start- und einen Zielflughafen aus. Diese Anfrage wird an das Backend gesendet, welches anschliessend eine passende Abfrage an die Neo4j-Datenbank ausführt. Die Datenbank berechnet anhand der vorhandenen Beziehungen zwischen Airports und FRA-Points die kürzeste Route.

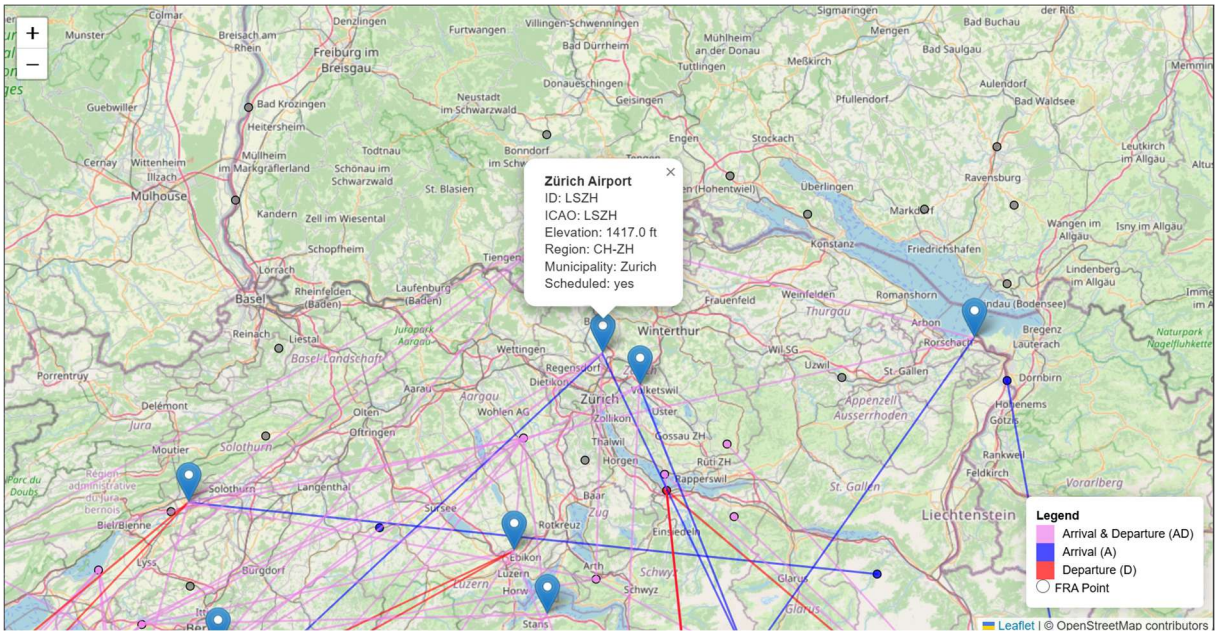
Das Resultat wird anschliessend an das Frontend zurückgegeben und dort visuell auf einer Karte dargestellt. Flughäfen und FRA-Points werden dabei als Punkte auf der Karte angezeigt. Sobald eine Route berechnet wurde, werden die entsprechenden Punkte entlang der Route miteinander verbunden und hervorgehoben, sodass der Verlauf der Flugstrecke für den Benutzer klar sichtbar ist.




**Route: ["LSGG", "KORED", "SOSON", "LSZA"]**

Shortest path visualized from Geneva to Lugano via waypoints.

Zusätzlich bietet die Anwendung eine interaktive Kartenansicht. Alle importierten Flughäfen werden dort als Punkte angezeigt und können angeklickt werden. Beim Anklicken eines Flughafens öffnet sich eine Detailansicht mit Informationen zu diesem Airport. Dasselbe gilt für die FRA-Points. Auch diese sind auf der Karte sichtbar und können ausgewählt werden, woraufhin ihre Detailinformationen angezeigt werden.




Die Interaktionsmöglichkeiten bleiben auch nach der Berechnung einer Route erhalten. Der Benutzer kann weiterhin auf einzelne Punkte entlang der Route klicken und zusätzliche Informationen abrufen.

## 3.4 Struktur der Applikation

Die Applikation ist in drei zentrale Komponenten unterteilt:

- Frontend
- Backend
- Datenbank

Zwischen Frontend und Datenbank besteht keine direkte Verbindung. Alle Anfragen werden über das Backend verarbeitet. Dadurch bleibt die Datenbank vor direkten Zugriffen durch den Benutzer geschützt und die Geschäftslogik kann zentral im Backend implementiert werden.

Das Backend ist nach einem Schichtenmodell (Layered Architecture) aufgebaut. Die einzelnen Schichten übernehmen dabei klar definierte Aufgaben.

**Controller-Schicht**

Diese Schicht nimmt HTTP-Anfragen aus dem Frontend entgegen und stellt REST-Endpunkte zur Verfügung. Die Controller verarbeiten die eingehenden Requests und geben die Ergebnisse als JSON-Antwort an das Frontend zurück.

**Service-Schicht**

Die Service-Schicht enthält die fachliche Logik der Anwendung. Hier werden beispielsweise Routenberechnungen koordiniert oder Daten für Detailansichten aufbereitet.

**Repository-Schicht**

Diese Schicht kapselt den Zugriff auf die Neo4j-Datenbank. Sie enthält die konkreten Datenbankabfragen und stellt Methoden zur Verfügung, über die das Backend auf die gespeicherten Daten zugreifen kann.

**Model- und Entity-Schicht**

Diese Schicht bildet die zentralen Datenstrukturen der Anwendung ab. Sie repräsentiert die Objekte der Datenbank, beispielsweise Airports oder FRA-Points.

**DTO-Schicht (Data Transfer Objects)**

DTOs dienen dazu, Daten zwischen Backend und Frontend zu übertragen, ohne die internen Datenstrukturen direkt nach aussen zu geben. Dadurch können die übertragenen Daten gezielt kontrolliert und vereinfacht werden.

Diese Architektur verbessert die Wartbarkeit und Erweiterbarkeit der Applikation, da jede Schicht klar definierte Aufgaben übernimmt und Änderungen lokal umgesetzt werden können.

## 3.5 Administrationsbereich und CRUD-Operationen

Neben der normalen Benutzeroberfläche enthält die Anwendung auch eine Administrationsseite, über die CRUD-Operationen auf den zentralen Entitäten der Datenbank ausgeführt werden können.

CRUD steht für:

- Create
- Read
- Update
- Delete

Über diese Administrationsseite können somit neue Airports oder FRA-Points erstellt, bestehende Einträge bearbeitet oder vorhandene Datensätze gelöscht werden. Die Administrationsfunktionen greifen über das Backend auf entsprechende REST-Endpunkte zu, welche die notwendigen Datenbankoperationen in Neo4j ausführen.

Die Administrationsseite ist direkt in das Frontend der Anwendung integriert. Aus Gründen der Einfachheit und des begrenzten Projektumfangs wird jedoch keine separate Authentifizierung implementiert. Das bedeutet, dass grundsätzlich jeder Benutzer der Anwendung auch auf diese Seite zugreifen und dort CRUD-Operationen ausführen kann.

Diese Entscheidung wurde bewusst getroffen, um den Implementationsaufwand innerhalb des gegebenen Zeitrahmens überschaubar zu halten. Der Fokus des Projekts liegt primär auf der Modellierung des Graphen, der Datenverarbeitung innerhalb der Neo4j-Datenbank sowie der Visualisierung der Flugrouten.

Die Administrationsseite dient deshalb in erster Linie dazu, die geforderten CRUD-Funktionen innerhalb der Anwendung demonstrieren zu können. In einer produktiven Umgebung würde

dieser Bereich selbstverständlich durch ein Authentifizierungs- und Autorisierungssystem geschützt werden, sodass nur berechtigte Benutzer Änderungen an der Datenbank vornehmen können.

# 4 Weitere optionale Kompetenzen

## 4.1 Transaktionen

Um die Datenintegrität innerhalb des SkyGraph-Systems zu gewährleisten, wurde im Backend eine konsequente Transaktionssteuerung implementiert. Dies ist besonders bei einer Graphdatenbank wie Neo4j entscheidend, da Operationen oft komplexe Abhängigkeiten zwischen Knoten und deren Beziehungen betreffen.

### 4.1.1 Technische Umsetzung

In der Spring-Boot-Applikation nutzen wir die Annotation **`@Transactional`** aus dem Spring-Framework. Diese sorgt dafür, dass alle Datenbankoperationen innerhalb einer Methode als eine einzige atomare Einheit (ACID-Prinzip) behandelt werden.

### 4.1.2 Beispiel FRAPointService

Ein kritisches Beispiel findet sich in der Methode **`saveFRAPoint`** im **`FRAPointService`**. Wenn ein neuer Wegpunkt (**`FRAPoint`**) angelegt wird, geschieht dies in mehreren Schritten:

1. Der Knoten selbst wird per MERGE-Befehl in der Datenbank erstellt oder aktualisiert.
2. Anschliessend wird die Methode **`createFullMeshConnections`** aufgerufen, die den neuen Punkt mit **allen** anderen vorhandenen Wegpunkten über eine **`CONNECTS`**-Beziehung verknüpft, um das vollständige Navigationsnetz zu weben.

Ohne Transaktionssteuerung könnte folgendes Problem auftreten: Wenn nach der Erstellung des Knotens, aber während des Schreibens der hunderte von Beziehungen ein Fehler auftritt (z. B. Verbindungsabbruch), bliebe ein «verwaister» Wegpunkt ohne jede Verbindung in der Datenbank zurück. Durch die **`@Transactional`**-Annotation wird in einem solchen Fehlerfall die gesamte Operation zurückgerollt (Rollback). Die Datenbank kehrt exakt in den Zustand vor dem Aufruf zurück, wodurch inkonsistente Datenbestände verhindert werden.

### 4.1.3 Beispiel AirportService

Auch im **`AirportService`** ist diese Logik essenziell. Bei der Methode **`saveAirport`** werden zuerst alle bestehenden Abflug- (**`DEP_TO`**) und Ankunftsbeziehungen (**`ARR_TO`**) gelöscht, bevor sie basierend auf den neuen Benutzereingaben wieder neu erstellt werden. Die Transaktionsklammer stellt sicher, dass ein Flughafen niemals ohne seine Beziehungen in der Datenbank verbleibt, sollte der Prozess der Neuerstellung fehlschlagen.

Zusammenfassend schützt der Einsatz von Transaktionen das SkyGraph-Projekt vor logischen Fehlern in der Graphstruktur und stellt sicher, dass komplexe Änderungen nach dem «Alles-oder-Nichts-Prinzip» ausgeführt werden.

### 4.2 Grössere Datenmengen importieren

Diese Zusatzleistung haben wir bereits zu Beginn der Projektarbeit erbracht und beschreiben unser Vorgehen im Kapitel «1.5.2. Automatisierter Datenimport via Python-Skript». Das Python-Skript ist ebenfalls in der zip-Datei, mit welcher wir unseren Code von dieser Projektarbeit abgeben, mit dabei.

# 5 Arbeitsjournal, Reflexionen

## 5.1 Journal Simon

| **Datum 5.3.2026** |
|---|
| ***Was habe ich erledigt? Wie bin ich vorgegangen?*** |
| Ich habe die Kapitel 2.1 – 2.5 erarbeitet und mich dabei hauptsächlich mit der Datenbeschaffung der Airport Daten und FRA-Punkten beschäftigt. Danach habe ich die Daten aufbereitet und mich an das ERD und an das logische Datenmodellgesetzt und das gemeinsam mit Jonas ausgearbeitet, bevor ich das Ganze in einem Skript umgesetzt habe, das den korrekten Dateiimport gemäss logischem Datenmodell auf die Datenbank graniert. |
| ***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?*** |
| Das Besorgen der Daten war der Zeit intensivste, da ich zuerst gar nicht wusste was ich genau suche und mich deswegen noch etwas in die Theorie der Luftfahrt einlesen musste. Das Importskript war auch eine kleine Challenge. Da ich noch nie mit Ne4j gearbeitet habe und deswegen noch nicht mit der Syntax vertraut war. Mittlerweile gibt es aber KI Tools die einem das ja schnell erklären können. |
| ***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?*** |
| Gut gelaufen ist die Integration der Daten in das Skript für den automatisierten Datenimport und Verknüpfung auf der Datenbank, was uns viel Handarbeit abnimmt und Fehlerquellen reduziert. Zu viel Zeit habe ich mit der ERD / Logisches Datenmodell Diagrammen Erstellung im PlantUML verbraucht, hier wäre ich in draw.io effizienter gewesen. |
| ***Wie geht es weiter? Was will ich besser machen?*** |
| Der Dateiimport auf der Datenbank ausführen und mit ersten Abfragen / CRUD Operationen testen. Besser machen möchte ich das Zeitmanagement und meine Zeit lieber in den Code stecken als in schöne Diagramme. |

| **Datum 9.3.26** |
|---|
| ***Was habe ich erledigt? Wie bin ich vorgegangen?*** |
| Ich habe das Importscript nochmals überarbeitet, da uns bei den FRA-Points noch einige Attribute gefehlt haben. Konkret haben wir entschieden, die Koordinaten zusätzlich in Rohform zu speichern. Bisher waren sie nur als GeoPoint referenziert, neu speichern wir auch eine rohe West-Ost-Koordinate sowie eine rohe Nord-Süd-Koordinate. Zusätzlich habe ich die Flight-Level-Angaben bei den FRA-Points aufgeteilt, sodass es nun ein Minimum Flight Level und ein Maximum Flight Level gibt. Dadurch ist das Attribut sauberer strukturiert |

und später einfacher auswertbar. Neben diesen Änderungen habe ich noch kleinere Anpassungen am Importscript vorgenommen.

Da sich durch diese Änderungen das Datenmodell angepasst hat, habe ich anschliessend auch die Diagramme in der Dokumentation aktualisiert. Insbesondere habe ich die PlantUML-Diagramme überarbeitet und wieder an den aktuellen Stand des Modells angepasst.

***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?***

Die grösste Schwierigkeit lag beim Zurücksetzen der Datenbank. Wir verwenden die Neo4j Community Edition, die nur eine einzige Datenbank pro Server erlaubt. Dadurch kann die Datenbank nicht einfach gedroppt und neu erstellt werden. Stattdessen musste ich alle Daten manuell entfernen, um wieder eine saubere Ausgangslage zu erhalten. Das war etwas mühsam, weil man dabei darauf achten muss, dass wirklich alle Daten gelöscht sind, bevor man das Importscript erneut ausführt. Letztlich habe ich das Problem gelöst, indem ich die Daten vollständig manuell entfernt und danach die Datenbank erneut sauber aufgesetzt habe.

***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?***

Eine wichtige Erkenntnis ist, dass Neo4j in der Community Version tatsächlich nur eine Datenbank pro Server zulässt. Diese Einschränkung hat mich überrascht, weil sie einige typische Arbeitsschritte wie das einfache Droppen einer Datenbank erschwert. Für unser Projekt ist das zwar kein kritisches Problem, macht das Arbeiten aber teilweise unnötig umständlich.
Gut gelaufen ist die Überarbeitung des Importscripts sowie die Anpassung des Datenmodells. Weniger gut war, dass ich die PlantUML-Texte ursprünglich nicht abgespeichert hatte. Dadurch musste ich die Diagramme teilweise neu rekonstruieren, was unnötig Zeit gekostet hat. Dieses Problem habe ich inzwischen behoben, indem ich die aktuellen PlantUML-Definitionen sauber gespeichert habe.

***Wie geht es weiter? Was will ich besser machen?***

Als nächster Schritt arbeiten wir am Berechtigungskonzept weiter. Parallel dazu beginnen wir bereits mit der Planung des Frontends, zumindest auf funktionaler Ebene. Der Grund dafür ist, dass sich viele Backend-Abfragen direkt aus den Anforderungen des Frontends ergeben. Wenn klar ist, welche Informationen der Benutzer sehen und bearbeiten können soll, lässt sich besser definieren, welche Daten das Backend bereitstellen muss. Für die nächsten Schritte möchte ich zudem darauf achten, Diagramm-Definitionen und ähnliche Artefakte immer direkt mitzuspeichern, damit spätere Anpassungen einfacher und schneller möglich sind.

**16.3**

***Was habe ich erledigt? Wie bin ich vorgegangen?***

Ich habe heute das Berechtigungskonzept für unsere Datenbank ausgearbeitet. Dazu habe ich mir zuerst überlegt, welche Akteure überhaupt auf die Datenbank zugreifen und welche Aktionen sie durchführen müssen. Ausgehend von diesen Use-Cases habe ich dann die Rollen definiert.

Wir haben drei grundlegende Rollen vorgesehen. Eine Administratorrolle, die vollständigen Zugriff auf die Datenbank hat und sämtliche Operationen durchführen darf. Dann eine Rolle für die Applikation selbst, also die Verbindung, über die unsere Anwendung auf die Datenbank zugreift. Diese Rolle benötigt Lese- und Schreibrechte, soll aber keine Berechtigungen oder strukturellen Änderungen an der Datenbank vornehmen können. Zusätzlich habe ich noch eine Import-Rolle vorgesehen, die ebenfalls lesen und schreiben darf, aber primär dafür gedacht ist, dass unser Importskript automatisiert Daten in die Datenbank einspielen und aktualisieren kann.

Neben dem Berechtigungskonzept habe ich mich auch mit der Backup-Strategie beschäftigt und dafür ein Konzept ausgearbeitet. Wir haben uns entschieden, regelmässige Dumps der Datenbank zu erstellen. Ein solcher Dump enthält den vollständigen Datenbestand der Datenbank und kann im Bedarfsfall wieder eingespielt werden, um den ursprünglichen Zustand der Datenbank wiederherzustellen. Dadurch haben wir eine relativ einfache, aber zuverlässige Sicherungsstrategie.

Zusätzlich habe ich die Mockups unserer Benutzeroberfläche nochmals angeschaut. Diese hatte ich bereits vorher erstellt, heute habe ich die entsprechenden Screenshots Nikola gezeigt und sie von ihm abnehmen lassen. Danach habe ich sie in die Dokumentation integriert. Parallel dazu habe ich mir noch Gedanken gemacht, wie wir die Anwendung konzeptionell aufbauen wollen. Die grundlegende User Experience ist durch die Mockups bereits relativ klar definiert. Technisch sind wir aber noch nicht ganz entschieden, wie genau wir die Umsetzung machen wollen.

***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?***

Die grösste Schwierigkeit lag beim Berechtigungskonzept in der technischen Umsetzung. Wir arbeiten mit der Neo4j Community Edition, und diese Version erlaubt keine feingranulare Rechtevergabe. Benutzer können zwar erstellt werden, aber sie erhalten grundsätzlich die gleichen umfassenden Rechte auf der Datenbank. Dadurch lässt sich ein klassisches Rollenmodell mit unterschiedlichen Berechtigungen praktisch nicht sauber umsetzen.

Wir konnten dieses Problem letztlich nicht wirklich lösen. Stattdessen haben wir die Befehle für das Rollen- und Berechtigungskonzept so formuliert, wie sie in der Enterprise-Version von Neo4j funktionieren würden. Auf unserem System konnten wir diese Befehle allerdings nicht ausführen, weil die entsprechenden Funktionen in der Community-Version fehlen.

***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?***

Eine wichtige Erkenntnis war, dass Neo4j in der Community-Version relativ stark auf das kommerzielle Produkt ausgerichtet ist. Viele Funktionen, die man für ein sauberes Rollen- und Berechtigungssystem eigentlich erwarten würde, sind erst in der Enterprise-Version verfügbar. Das hat mich etwas überrascht, weil ich nicht erwartet hätte, dass die Einschränkungen für private oder kleinere Projekte so deutlich sind.

Unabhängig davon hat sich aber nochmals bestätigt, dass es sinnvoll ist, beim Berechtigungsdesign nach dem Prinzip des *Least Privilege* vorzugehen. Man sollte sich zuerst überlegen, welche Akteure welche Aufgaben erfüllen müssen, und daraus dann Rollen ableiten, die nur genau die notwendigen Rechte besitzen.

Beim Backup war ich eher positiv überrascht. Das Erstellen eines Datenbank-Dumps ist relativ einfach und auch die Wiederherstellung einer Datenbank aus einem Dump ist konzeptionell unkompliziert. Für unser Projekt ist diese Backupstrategie daher gut geeignet.

Gut gelaufen ist ausserdem die Abstimmung der Mockups sowie deren Integration in die Dokumentation. Weniger klar ist aktuell noch die technische Architektur der Anwendung.

***Wie geht es weiter? Was will ich besser machen?***

Als nächstes werden wir uns stärker auf die konkrete Umsetzung der Anwendung konzentrieren. Dabei müssen wir vor allem entscheiden, wie wir die Architektur gestalten. Ursprünglich war geplant, eine klassische Aufteilung in Backend und Frontend umzusetzen. Das wäre zwar architektonisch sauber und sehr nahe an einem professionellen Setup, ist für ein kleines Schulprojekt aber möglicherweise unnötig komplex.

Deshalb überlegen wir aktuell, ob wir eine einfachere Lösung wählen, die schneller umzusetzen ist und besser zum Projektumfang passt. In den nächsten Tagen wird ein grosser Teil der Arbeit in die Implementierung der Anwendung fliessen.

Für mich persönlich möchte ich vor allem das Zeitmanagement verbessern. In der letzten Woche haben wir relativ spät mit einzelnen Aufgaben begonnen und sind dadurch zeitlich recht nahe an die Deadline gekommen. Zwar haben wir die Arbeiten noch rechtzeitig abgeschlossen, langfristig ist es aber sinnvoller, früher mit der Umsetzung zu beginnen. So bleibt mehr Spielraum, falls unerwartete Probleme auftreten.

**23.03**

***Was habe ich erledigt? Wie bin ich vorgegangen?***

Ich habe mich vor allem um die Umsetzung des Frontends gekümmert und auch um die Verknüpfung zwischen Frontend und Backend. Jonas und ich haben bereits im Voraus gemeinsam besprochen, wie wir die Applikation grundsätzlich aufbauen wollen, und uns dafür entschieden, sie in ein Backend und ein Web-Frontend zu gliedern. Dadurch war die Architektur relativ klar. Jonas hat dann das Backend mit Java und Spring Boot umgesetzt, sodass die API-Endpunkte bereits funktioniert haben. Meine Aufgabe war es anschliessend, das Frontend so zu bauen, dass es mit diesen Endpunkten kommunizieren kann. Im Frontend hatten wir die Anforderung, dass die berechnete kürzeste Route grafisch dargestellt wird. Das habe ich als erstes umgesetzt, indem ich mich an klassischen Navigations-Apps wie Google Maps orientiert habe. Ich habe ein Interface gebaut, bei dem man einen Start- und Zielflughafen eingeben kann sowie beliebige VIA-Flughäfen hinzufügen kann, und dann die Route berechnen lassen kann. Diese Kernfunktion liess sich relativ zügig umsetzen. Danach habe ich noch ein Einstellungs-Widget implementiert, über das man CRUD-Operationen auf den Entitäten durchführen kann, also insbesondere Flughäfen und FRA-Points. Dabei habe ich stark darauf geachtet, dass die Bedienung möglichst intuitiv ist, zum Beispiel indem man Punkte direkt auf der Karte auswählen kann statt Koordinaten einzugeben. Auch die Verknüpfung von Punkten mit Flughäfen erfolgt grafisch. Die Umsetzung dieser Funktionen hat insgesamt länger gedauert als die eigentliche Routenberechnung, war aber aus meiner Sicht trotzdem sinnvoll.

***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?***

Grössere Schwierigkeiten hatte ich keine, aber es gab einige Punkte, die anspruchsvoll waren. Zum einen war das Arbeiten mit JavaScript für mich noch relativ neu, insbesondere wenn es darum ging, UI-Elemente zur richtigen Zeit ein- und auszublenden. Das hat am Anfang etwas Zeit gebraucht, bis ich das sauber im Griff hatte. Auch die Strukturierung des Codes war eine Herausforderung, also wie ich die JavaScript-Dateien sinnvoll aufteile, damit nicht alles in einer einzigen Datei landet. Das habe ich dann durch Recherche und auch mit Unterstützung von KI schrittweise verbessert. Eine grössere technische Herausforderung war bei den CRUD-Operationen, vor allem im Zusammenhang mit den FRA-Points. Diese sind in der Datenbank stark miteinander vernetzt, sodass beim Hinzufügen oder Aktualisieren eines Punktes auch alle Distanzbeziehungen neu berechnet werden müssen. Dafür musste ich zusätzliche Logik im Backend implementieren und auch Anpassungen an der Datenbank vornehmen. Das war aufwendiger als erwartet, konnte aber schlussendlich sauber gelöst werden.

***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?***

Eine wichtige Erkenntnis für mich ist, dass der eigentliche Mehrwert und der Wow-Effekt stark im Frontend entsteht. Das Backend liefert zwar die Daten, aber erst durch die Darstellung im Frontend wird das Ganze wirklich greifbar. Mir war vorher nicht so bewusst, wie viel das Frontend aus den Daten machen kann, zum Beispiel indem Koordinaten nicht einfach als Text dargestellt werden, sondern direkt als Punkte auf einer Karte visualisiert werden. Das hat einen deutlich grösseren Effekt. Gut gelaufen ist für mich, dass ich insgesamt zügig vorangekommen bin und die Kernfunktion relativ schnell umsetzen konnte. Auch die Zusammenarbeit mit Jonas und die klare Aufteilung der Aufgaben hat gut funktioniert. Weniger gut war, dass ich den Aufwand für die CRUD-Funktionalitäten unterschätzt habe. Diese haben deutlich mehr Zeit gebraucht als gedacht, obwohl sie für die eigentliche Nutzung der Applikation gar nicht so zentral sind, da solche Daten in der Praxis eher selten geändert werden und normalerweise von einem Administrator verwaltet würden.

***Wie geht es weiter? Was will ich besser machen?***

Das Projekt ist grundsätzlich abgeschlossen und ich bin zufrieden mit dem Resultat. Für die Zukunft würde ich aber besser darauf achten, den Nutzen von Features im Verhältnis zum Aufwand realistischer einzuschätzen. Gerade bei komplexen, stark vernetzten Datenstrukturen sollte man früh überlegen, ob sich der Implementationsaufwand wirklich lohnt. Ausserdem möchte ich mein Verständnis für Frontend-Entwicklung weiter vertiefen, da ich gemerkt habe, wie viel Potenzial dort liegt. Optional könnte ich mir noch vorstellen, das Projekt weiterzuführen, indem ich das Backend und eventuell auch das Frontend auf einem VPS hoste, da die Datenbank bereits dort läuft. Das wäre technisch spannend, aber hängt davon ab, ob ich dafür die nötige Zeit habe und ob ich das priorisieren will.

## 5.2 Journal Jonas

| **3.3., 5.3. und 6.3.2026** |
|---|
| ***Was habe ich erledigt? Wie bin ich vorgegangen?*** |
| Zunächst haben Simon und ich die Idee für die Datenbank konkret besprochen und detailliert ausgearbeitet. Danach haben wir die zu erledigenden Arbeitspakete aufgeschrieben und fair aufgeteilt.<br><br>Während Simon sich um die Beschaffung der Daten kümmerte, war es meine Aufgabe, einen VPS aufsetzen. Wir wollten nicht, dass die Datenbank bei nur jemandem auf dem Gerät läuft, sondern dass wir beide Zeit- und Ortsunabhängig darauf arbeiten können. Wir haben uns für Hetzner entschieden. Ich habe einen Ubuntu-Server erstellt mit 2 Kernen, 4 GB RAM und 40 GB HDD (Amd-Architektur) mit Standort Helsinki.<br><br>Ich habe den ssh-Schlüssel von Simon und mir hinterlegt, damit wir uns beide über SSH als root einloggen können. Alle anderen Ports sind für eingehende Verbindungen geschlosen.<br><br>Danach habe ich mich als Root eingeloggt, die Updates gemacht und einen Benutzer für mich selbst angelegt, damit wir nicht als root arbeiten müssen. Dieser Benutzer hat wiederum mein SSH-Key hinterlegt, damit ich mich direkt auf diesen einloggen kann über SSH.<br><br>Ab nun arbeite ich auf diesem Benutzer. Da habe ich zuerst Neo4j installiert und dann ein SSH-Tunnel mit den Ports 7474 (Webinterface) und 7687 (Datenbank) geöffnet. Dann habe ich mich über meinen lokalen Webbrowser auf das Webinterface von Neo4j verbunden und zuerst das Hauptpasswort geändert.<br><br>Nachdem Simon den Datenimport abgeschlossen hatte, habe ich meinen Teil des ersten Teils der Dokumentation geschrieben. Dazu habe ich verschieden Cypher-Befehle ausgeführt und dokumentiert. |
| ***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?*** |
| Ich hatte schon einige erste Erfahrungen mit dem Aufsetzen von VPS machen dürfen, doch einige Dinge waren mir neu. Wie installiere ich Neo4j im Ubuntu Terminal? Hier war das Internet mein bester Freund…<br><br>Die grösste Schwierigkeit waren die unterschiedlichen Anforderungen auf Smartlearn, Dokumentationsvorlage, Folien im Teams. Gewisse Details unterschieden sich in den unterschiedlichen Versionen der Anforderungen. Was gilt nun? Wenn man sein Datenmodell entwirft, sollte man genau wissen, wie viele Attribute beispielsweise pro Entität gefragt sind. Denn nachdem die Datenbank aufgesetzt ist, sind solche grundlegenden Änderungen nicht so einfach umzusetzen: Wir haben nach dem ersten Teil des Projekts schliesslich bereits über 4500 Datensätze in der Datenbank. |
| ***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?*** |
| Es hat erstaunlich gut funktioniert! Ich dachte, dass wir mehr Probleme haben würden. Doch wir hatten nie eine Situation, in der wir längere Zeit von einem Fehler aufgehalten worden wären. |
| ***Wie geht es weiter? Was will ich besser machen?*** |

Ich will genau abklären, welche Anforderungen für uns gelten, damit wir das Projekt auch so umsetzen können, wie es von uns gefordert wird.

**9.3.2026**

***Was habe ich erledigt? Wie bin ich vorgegangen?***

Heute haben Simon und ich noch einige Details angepasst, nachdem wir im Gespräch mit unserer Lehrperson Nicolas einige Digne klären konnten.

So haben wir die Koordinaten nicht nur als Geo-Point (Neo4j Datentyp), sondern jeweils auch noch getrennt («redundant») als Northing und Easting eingetragen. Dies ist offenbar best practice, auch wenn es auf den ersten Blick redundant scheint.

Auch haben wir bemerkt, dass wir bei den Flughäfen zwei Attribute hatten, die denselben Wert speicherten, «ident» und «icao». Da dies eine Redundanz ist, haben wir uns entschieden, das Attribut «ident» zu entfernen und nutzen seither «icao».

Zudem haben wir die Flugflächen, auf welchen der FRAPoint verfügbar ist, aufgeteilt in eine Unter- und Obergrenze, die wir nun in einem separaten Attribut speichern. Das macht das Arbeiten mit diesen Informationen deutlich einfacher in einem Back- bzw. Frontend in Zukunft.

***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?***

Die spannendste und schwierigste Herausforderung heute, war die Praxis zu verstehen. Viele der Fragen, die uns heute beschäftigten (z.B. «redundante» Speicherung der Koordinaten oder ob «ident» und «icao» in der Praxis synonym verwendet werden), sind sehr praxisorientiert. Hier zeigte sich, dass in der Informatik immer wieder ein vertieftes Verständnis des Fachs, für welches man entwickelt, erfordert ist.

***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?***

Simon und ich haben heute mit unserer Lehrperson Nicolas gesprochen, um herauszufinden, welche der unterschiedlichen Anforderungen nun für uns gelten. Ich bin sehr erfreut, dass Nicolas uns heute mitgeteilt hat, dass durchaus ein Spielraum anzunehmen ist bei den Anforderungen. In unserem Fall haben wir beispielsweise nur 18 Datensätze der Hauptentitätsmenge, aber viele Beziehungen (mehr als 4000). Dadurch ist es in Ordnung, wenn wir das Kriterium «mind. 20 Datensätze der Hauptentitätsmenge» unterschreiten. Ich finde es sehr sinnvoll, dass Nicolas unser Projekt neben der Bewertung der genauen Kriterien auch als ein Ganzes wahrnimmt und entsprechend unserer Bemühungen wertschätzt.

***Wie geht es weiter? Was will ich besser machen?***

Als nächsten Schritt werden wir uns um Berechtigungen kümmern und User erstellen auf der Datenbank. Ich freue mich darauf.

**16.03.2026**

***Was habe ich erledigt? Wie bin ich vorgegangen?***

| |
|---|
| Ich habe heute das Konzept für die horizontale Skalierung ausgearbeitet. Dabei habe ich zuerst recherchiert, welche Möglichkeiten Neo4j überhaupt bietet für Sharding und Paritionierung. Daraus habe ich ein Konzept entwickelt und das Kapitel für unsere Dokumentation dazu geschrieben. Zudem habe ich die Befehle fürs Backup und Wiederherstellung auf unserer Datenbank durchgeführt. Dazu habe ich auch einem Screencast aufgenommen und diesen unserer Lehrperson freigegeben. |
| ***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?*** |
| Für die praktische Umsetzung sind wir hier leider eingeschränkt, da Neo4j Sharding und Partitionierung leider in der Community Version nicht zur Verfügung stellt. Nach Rücksprache mit unserer Lehrperson durften wir den praktischen Teil für dieses Kapitel auslassen und haben vereinbart, dass wir die freigewordene Zeit und Energie für die Applikation (nächster Teil der Arbeit) investieren werden. |
| ***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?*** |
| Ich denke die Haupterkenntnis von heute war sicher, dass Neo4j ist ziemlich eingeschränkt ist, was Partitionierung und Sharding in der Communityversion betrifft. |
| ***Wie geht es weiter? Was will ich besser machen?*** |
| Ich freue mich sehr, auf den Teil der Applikation. Wir haben heute bereits ein Konzept erarbeitet und ein erster Prototyp gebaut. Als nächsten Schritt werden wir das Konzept in Realität umsetzen. |

| **23.03.2026** |
|---|
| ***Was habe ich erledigt? Wie bin ich vorgegangen?*** |
| Ich habe die Grundstruktur des Backends als Java Spring Boot Applikation gebaut. Ich habe auch bereits die wichtigsten Endpoints (in den Controllern) angelegt, sodass Simon, der sich um das Frontend gekümmert hatte, nur noch einige Anpassungen im Backend machen musste. Zudem habe ich heute noch Zeit investiert, um die Dokumentation nochmal zu pflegen. Ich habe alles nochmal in Ruhe durchgelesen, ein paar Details angepasst und noch das Kapitel 4 geschrieben. |
| ***Was waren die Schwierigkeiten? Wie habe ich sie gelöst?*** |
| Wir hatten bei der Umsetzung der Applikation die Schwierigkeit, dass die Datenbankabfragen zum Teil etwas lange dauerten. Wir konnten nicht mit Sicherheit eingrenzen, an was es genau lag. Eine Idee war, dass einfach die Verbindung zum VPS-Server (auf dem unsere Datenbank läuft) ggf. etwas langsam war, da diese jeweils immer noch über ein SSH-Tunnel gehen muss. Andererseits waren die Abfragen über das Neo4j-Webinterface aber normal performant, so wie wir es schon kannten. Das spricht als eher gegen diese Theorie.<br><br>Eine andere Theorie ist, dass es an der Art und Weise liegt, wie wir die beste Route berechnen. Hier haben wir aber im Vorfeld eigentlich absichtlich eine optimierte Abfrage entwickelt, siehe Kapitel «Kürzeste Flugroute zwischen zwei Flughäfen berechnen». |
| ***Was sind die Erkenntnisse? Was ist gut gelaufen, was weniger?*** |

| |
|---|
| Meine wichtigste Erkenntnis heute war, wie wertvoll der Backend-üK war, den wir hatten. Was hat das aber mit diesem Modul zu tun? Ich konnte beim Bauen des Backends auf viele Strukturen und gelernte Patterns aus dem Backend-üK zurückgreifen. Ohne das, wäre die Umsetzung dieses Backends sehr viel schwieriger gewesen. Ich bin sehr dankbar, dass wir während dem Backend-üK so viele gute «Best Practices» erlernen durften. |
| ***Wie geht es weiter? Was will ich besser machen?*** |
| Für ein weiterführendes Projekt wäre es sicher spannend, den Algorithmus für die Routenberechnung nochmal genauer unter die Lupe zu nehmen. Ich denke, dass man diesen noch weiter optimieren könnte. |