KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
09/2017
Jahrgang 10

Eine Einführung in Gerrit

Warum eigentlich Codereviews?

Von Nikolas May

Mit der Schlange durch das Netz

Aufbereitung relationaler Daten für Neo4J mit Python-Mitteln

von Ilker Yümsek

KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

Sie können die elektronische Form des KaffeeKlatsch
monatlich, kostenlos und unverbindlich
durch eine E-Mail an

abo@bookware.de

abonnieren.

Ihre E-Mail-Adresse wird ausschließlich für den

Versand des KaffeeKlatsch verwendet.

Editorial
Mobiles bezahlen

Oder Mobile Payment wie es so schön heißt, erlebt zurzeit – zumindest medial – einen Auftrieb. Sei es mit Banking-Apps, mit denen man seine Schulden bei Freunden tilgen kann oder die NFC-getriebenen Systeme, die bereits länger auf dem Markt existieren, sich aber in Deutschland noch nicht wirklich durchgesetzt haben, wenn man sich die verschiedenen Veröffentlichungen zu diesem Thema ansieht. Und nun plant auch Whatsapp ein Bezahl-Feature in seinem Messenger zur Verfügung zu stellen. Dies wird sicher auch die Datenschutz- und Sicherheits-Debatte wieder bzw. weiter entfachen. Aber letztlich zeigt dieser Weg, dass wir über kurz oder lang dem Bargeld wahrscheinlich doch irgendwann einmal Ade sagen werden müssen. Bis dies jedoch soweit sein wird, wird es wohl noch eine Weile dauern.

Das Gute ist, dass Sie für den KaffeeKlatsch ja nichts bezahlen müssen, wobei das auch nicht ganz richtig ist. Sie zahlen mit Ihrer Aufmerksamkeit und die zählt für uns umso mehr. Widmen Sie doch also Ihre Aufmerksamkeit den folgenden Beiträgen!

In „*Mit der Schlange durch das Netz*" (Seite 11) befasst sich Ilker Yümsek mit der Aufbereitung bestehender Daten einer relationalen Datenbank für eine Graph-Datenbank. Anhand eines Beispiels aus dem Bereich OpenData zeigt er, welche Aspekte neben der Gestaltung der Datenstruktur noch berücksichtigt werden müssen.

Warum sind Codereviews eigentlich wichtig? Dieser Frage geht Nikolas May in „*Eine Einführung in Gerrit*" (Seite 6) nach und erläutert, welche Vorteile Codereviews haben und stellt das Tool Gerrit vor, das dabei unterstützt.

Etwas Unterhaltsames gibt es auch wieder in „Das Allerletzte".

Ich wünsche Ihnen viel Spaß beim Lesen.

Ihr Oliver Klosa
Chefredakteur

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde. Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.
Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Herrn Oliver Klosa via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-0 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Oliver Klosa via E-Mail unter redaktion@bookware.de oder telefonisch unter 09131/8903-0.

Copyright

Impressum

KaffeeKlatsch Jahrgang 10, Nummer 9, September 2016
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Software GmbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-99
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking / Dr. Oliver Klosa
Anzeigen: Dr. Oliver Klosa
Grafik: Nicole Delong-Buchanan

Inhalt

Eine Einführung in Gerrit

Warum eigentlich Codereviews?

Von Nikolas May

Codereviews werden immer häufiger zum Bestandteil der täglichen Arbeiten eines Software-Entwicklers. Durch das regelmäßige Inspizieren des Codes durch einen anderen Entwickler, wird ein gemeinsames Verständnis für die Software entwickelt. Des Weiteren findet ein kontinuierliches Feedback statt, welches hilft, frühzeitig Fehler in der Logik oder in unvollständig umgesetzten Features zu erkennen und abzustellen. Durch die aus einem Review resultierenden Änderungen wird die Wartbarkeit der Software erhöht. Dies ist natürlich nur ein Auszug der Vorzüge von Codereviews. Sie können zum Beispiel auch zum Wissenstransfer, Prüfen von Coderichtlinien oder für Designvorgaben und mehr genutzt werden.

Hat man sich das Ziel gesetzt jede Änderung zu reviewen, wäre es von Vorteil, wenn sichergestellt werden könnte, dass jeder Code, der in das Repository fließt, reviewt wurde. Hierzu gibt es viele Tools, die einen dabei unterstützen. Eines dieser Tools ist Gerrit.

Was ist Gerrit?

Gerrit ist ein Open-Source-Codereviewtool für das Versionskontrollsystem GIT. Es bietet einem feingranulare, konfigurierbare Zugriffsrechte. Dabei können über die Weboberfläche Codereviews durchgeführt werden und Kommentare für die kompletten Änderungen oder einzelne Zeilen einer Änderung hinzugefügt werden.

Bei einem „normalen" Git-Setup gibt es normalerweise eine zentrale Instanz, von der die Nutzer *fetchen*[1] und Änderungen direkt in den Trunk *pushen*[2] können (siehe Abbildung 1).

Gerrit ersetzt diese und bietet die Möglichkeit Änderungen in einen Bereich für „schwebende" Änderungen zu pushen. Somit landen Änderungen erst im Trunk, wenn diese von einem Reviewer bestätigt wurden (siehe Abbildung 2).

1 Fetchen – Änderungen vom Repository holen

2 Pushen – Änderungen zum Repository schicken

Abbildung 1: Git

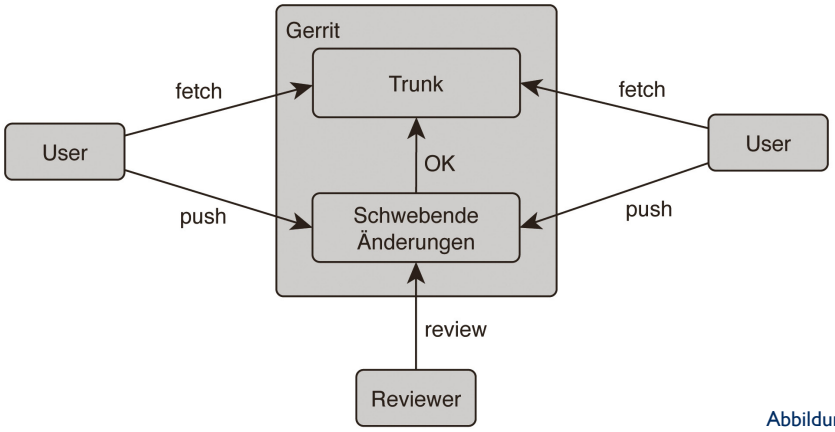


Abbildung 2: Gerrit

Setup

Gerrit ist downloadbar unter *https://www.gerritcodereview.com/*. Das WAR-File kann entweder in einem Applikationsserver *deployt* oder direkt gestartet werden.

Das WAR-File ist in das Verzeichnis zu verschieben, in dem Gerrit installiert werden soll. Mittels *java - jar gerrit.war init* startet man das Setup.

Beim Setup bieten sich verschiedene Optionen. Die wichtigsten sind mit ihrer Bedeutung hier gelistet:

- Location of Git repositories: Ort, an dem das Git-Repository liegt
- Database server type: Gibt an, welche Datenbank Gerrit verwendet (zum Test verwenden wir H2)
- Authentication method: Methode, mit dem User sich am System anmelden (wir verwenden DEVELOPMENT_BECOME_ANY_ACCOUNT, wodurch Benutzer ohne Einschränkungen über das Webinterface angelegt werden können)
- Install verified label: Möglichkeit, um einen Commit als "verified" zu kennzeichnen, was bedeutet „kompiliert und Unit-Tests laufen erfolgreich durch" (wir setzen diese Option auf „Nein")
- SSH Daemon Listen on address/Listen on port: Port auf dem SSH läuft, auf dem später mittels Git gepusht werden kann
- HTTP Daemon Listen on address/Listen on port: Port, auf dem das Webinterface läuft

Nachdem das Setup erfolgreich durchgelaufen ist, kann Gerrits Setup mittels *{Projektverzeichnis}\bin\gerrit.sh* gestartet werden.

Einrichten eines neuen Projekts

Um ein neues Projekt anzulegen, loggen wir uns als Admin ein und legen über *http://127.0.0.1:8080/#/admin/create-project/* ein neues Projekt an (siehe Abbildung 3).

Da wir unsere Arbeit nicht als Administrator erledigen wollen, legen wir uns einen neuen User über *http://localhost:8080/#/register* an (siehe Abbildung 4).

Hierbei tragen wir am besten gleich unseren Public Key ein, mit dem wir uns später gegenüber Gerrit authentifizieren.

Nun können wir das Projekt mittels GIT auschecken:

```
git clone ssh://nico@localhost:29418/Test_Projekt
```

Gerrit benötigt eine Change-ID, um Änderungen zu identifizieren, die zusammengehören. Diese befindet sich im Footer der Commit-Nachricht. Da es umständlich ist, diese immer per Hand einzutragen, empfiehlt es sich, einen Hook zu installieren. Dieser fügt einer Commit-Nachricht eine Change-ID hinzu, falls diese noch nicht vorhanden ist. Um den Hook zu installieren, muss man diesen unter *http://localhost:8080/tools/hooks/commit-msg* downloaden und nach *{Projektverzeichnis}\.git\hooks* verschieben.

Alternativ lässt sich im Projektverzeichnis folgendes Kommando ausführen:

```
scp -p -P 29418 nico@localhost:hooks/commit-msg .git/hooks/
```

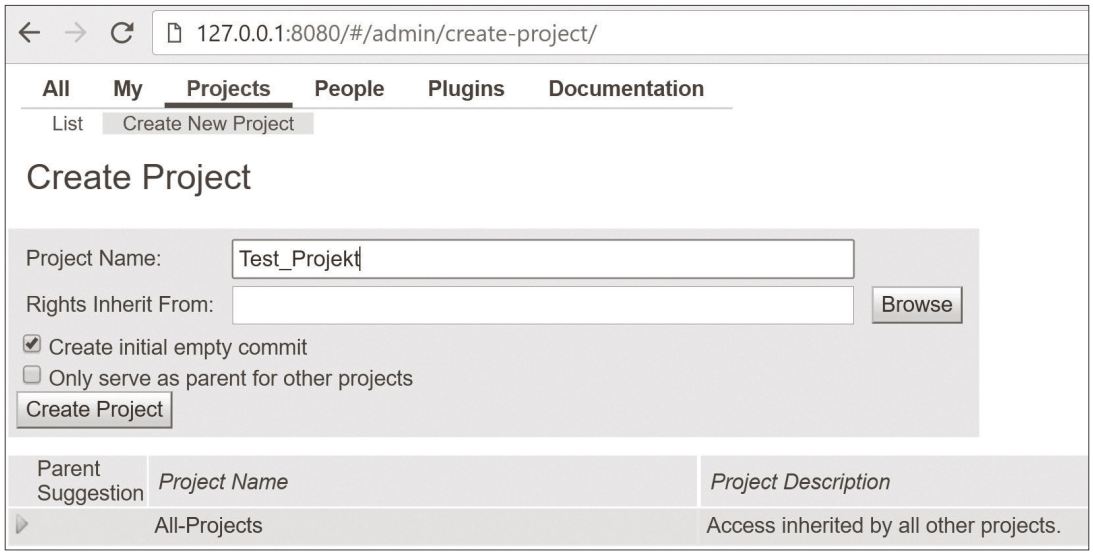


Abbildung 3: Neues Projekt anlegen

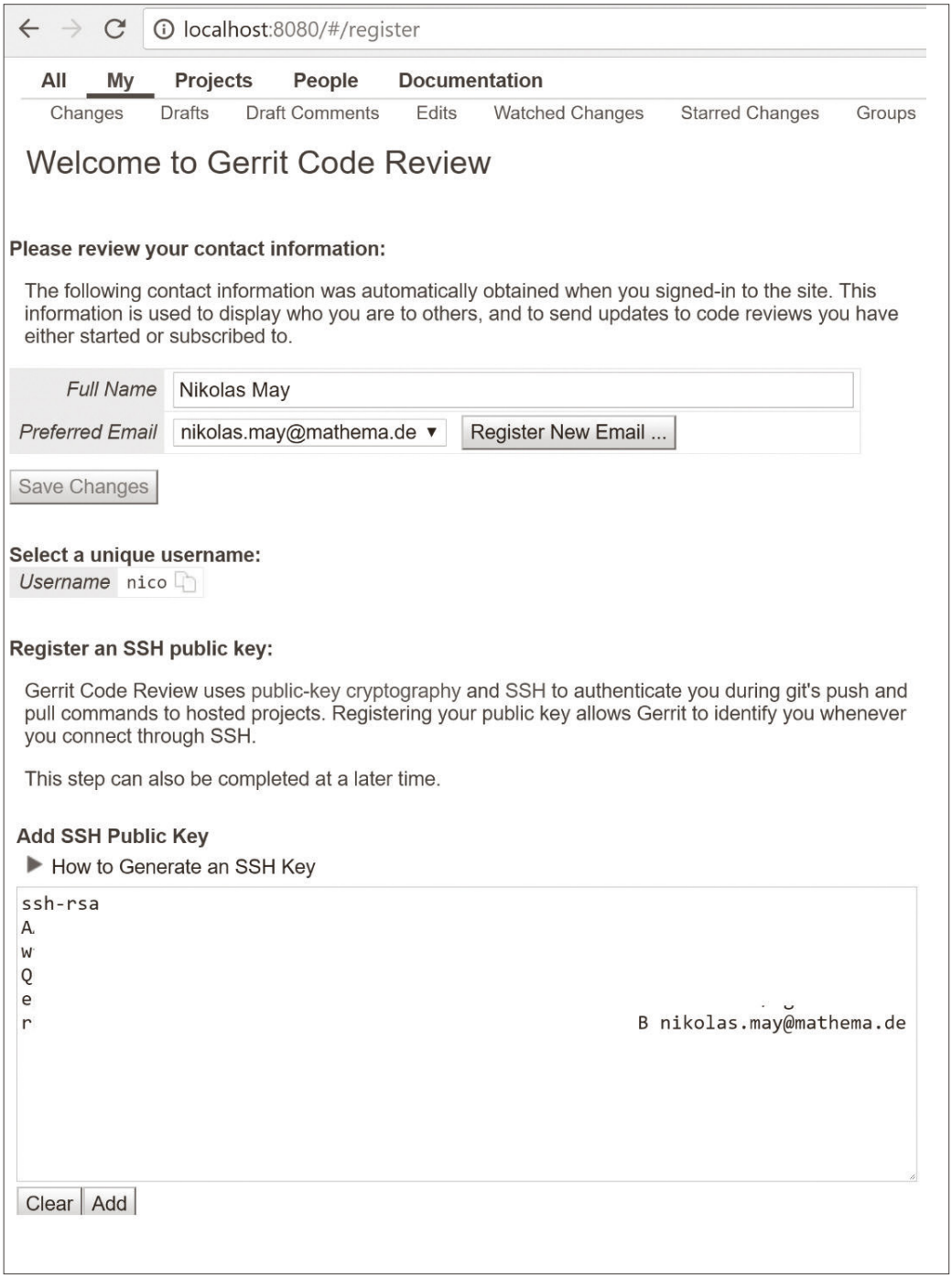


Abbildung 4: Neuen User anlegen

Erster Check-in

Für unseren ersten Check-in geben wir einen beliebigen Text in ein Textfile ein, welches wir commiten:

```
echo FooBar > file.txt
git add file.txt
git commit -m „foobar“
```

Anstelle von git push muss *git push origin HEAD:refs/for/master* verwendet werden.

Möchte man mit dem gewohnten Git-Kommando anstatt *git push origin HEAD:refs/for/master* arbeiten, kann man dies durch eine Anpassung der Git-Config erreichen:

{Projektverzeichnis}\.git\config öffnen und wie folgt ergänzen:

```
[remote "origin"]
    url = ssh://nico@localhost:29418/Test_Projekt
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = HEAD:refs/for/master
```

Nach dem Speichern funktioniert nun „git push".

Codereview

Zum Reviewen loggen wir uns wieder als Administrator ein und öffnen unser Projekt (siehe Abbildung 5).

Hier wählen wir unsere Änderung und öffnen die Detailansicht (siehe Abbildung 6).

In der Detailansicht bietet sich unter anderem die Möglichkeit, die Änderung zu bewerten und zum Trunk hinzuzufügen. Damit ein Urteil über die Änderungen gebildet werden kann, ist es möglich die geänderten Dateien auszuwählen und im Detail zu betrachten (siehe Abbildung 7).

Neben der Verfolgung der Änderungen lassen sich in dieser Ansicht die komplette Datei oder einzelne Zeilen kommentieren.

Ist das Review des Commits beendet, kann man auf der Übersichtsseite den Button *Reply* drücken. Dieser bietet die Möglichkeit, eine abschließende Bewertung der Änderung durchzuführen (siehe Abbildung 8).

Bei der Bewertung kann zwischen –2 und +2 gewählt werden. Dabei gilt, dass ein +2 Vote es ermöglicht den Commit in den Trunk zu mergen. Der –2 Vote ist ein Veto und verhindert somit das Mergen. Die –1 bis +1 Votes haben keinen Einfluss auf das Mergen. Sie bilden lediglich die Meinung des Reviewers ab. Es obliegt dabei denjenigen mit dem Recht +2 bzw. –2 zu wählen, ob der Commit in den Trunk fließt oder nicht.

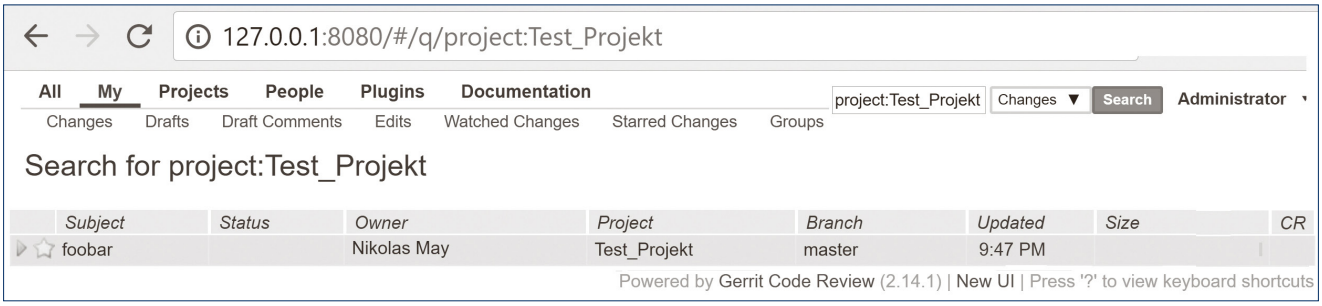


Abb. 5: Projektübersicht

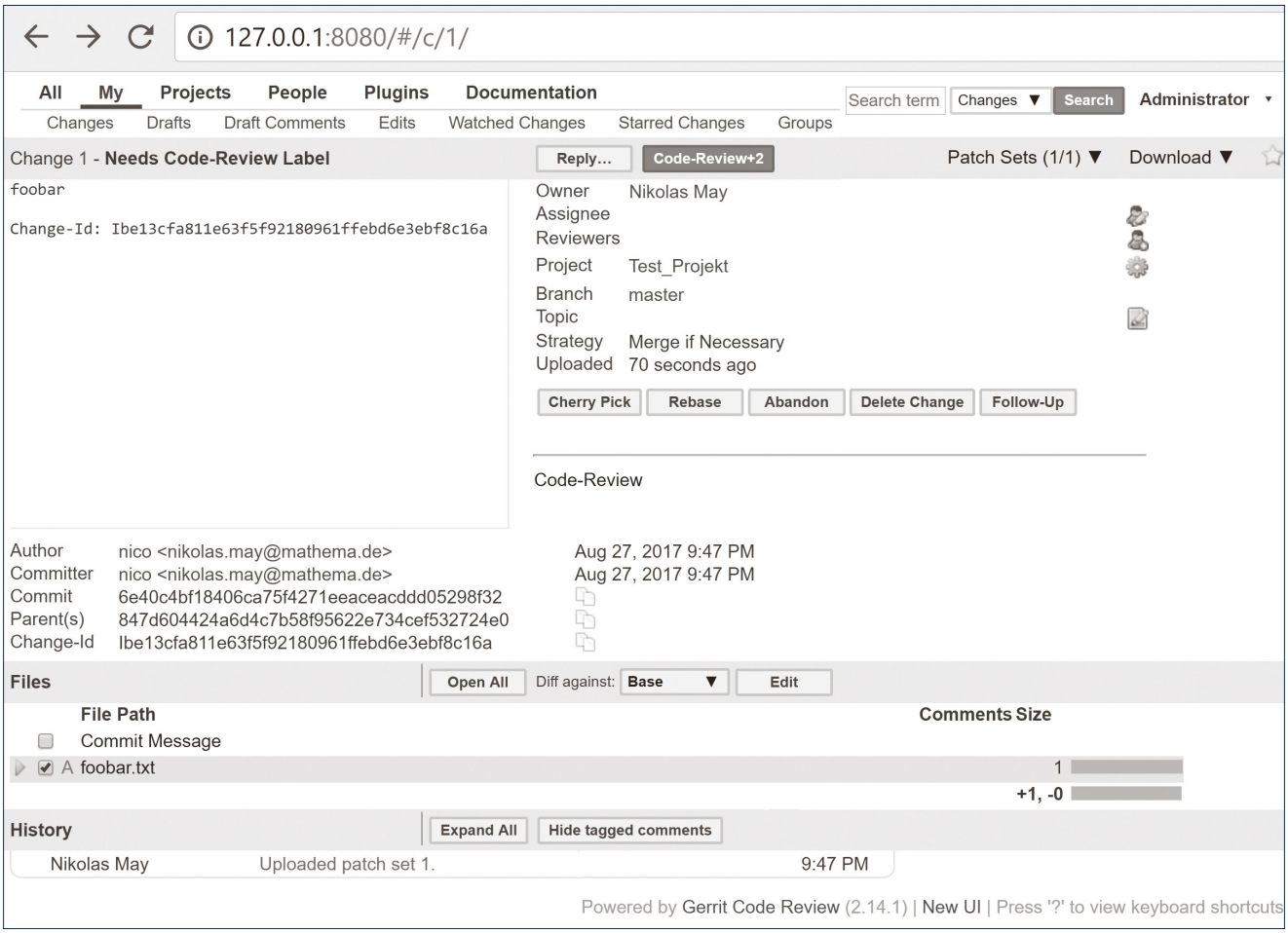


Abb. 6: Detailansicht

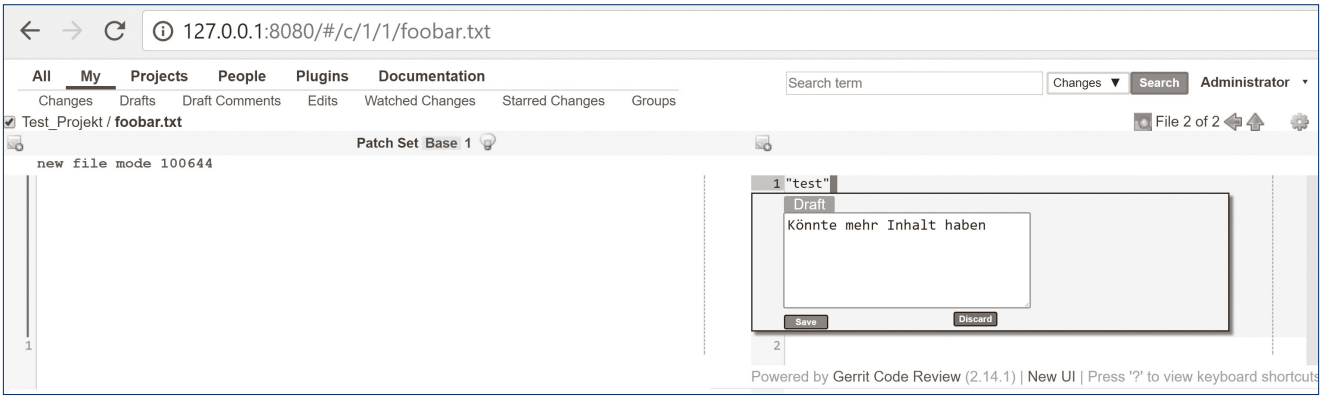


Abb. 7: Dateiänderungen

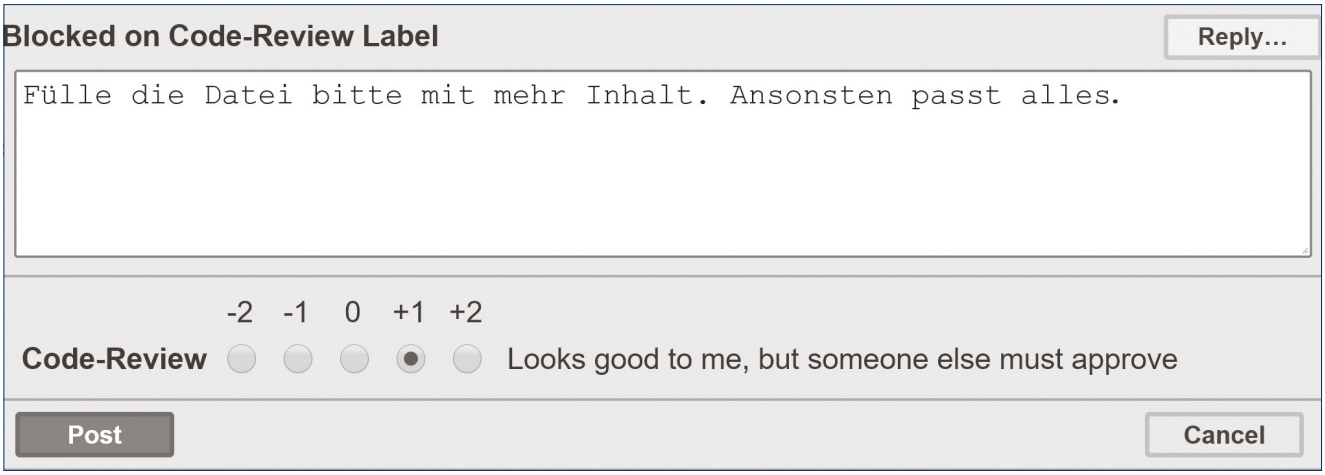


Abb. 8: Abschließende Bewertung

Rework

Da früher oder später ein Reviewer etwas zu beanstanden hat, kommt es dazu, dass man einen existierenden Commit anpassen muss. Bei Gerrit kann dies mit wenigen Zeilen durchgeführt werden:

```
git checkout <Commit der bandstandet wurde> //rework
git commit --amend
git push origin HEAD:refs/for/master
```

Der Reviewer kann die Änderungen wie im Teil „Codereview" prüfen und, wenn alle Mängel behoben wurden, freigeben.

Ausblick

Gerrit bietet neben dem Einbinden von Codereviews auch die Möglichkeit, in die CI-Strecke integriert zu werden. So kann man für jeden Commit die existierenden Tests ausführen und diesen Commit erst freigeben, wenn alle Tests erfolgreich durchgelaufen sind. Des Weiteren kann Gerrit um „Submit Rules" erweitert werden, um noch besser Projektspezifika zu erfüllen.

Kurzbiografie

Nikolas May ist als Entwickler und Consultant bei MATHEMA Software GmbH tätig. Seine Schwerpunkte liegen im Java Standard- und Enterprise-Bereich. Daneben beschäftigt er sich mit Java basierenden Web-Frameworks.

Mit der Schlange durch das Netz

Aufbereitung relationaler Daten für Neo4J mit Python-Mitteln

von Ilker Yümsek

Bei der Übernahme bestehender Daten aus einer relationalen Datenbank in eine Graph-Datenbank (z. B. Neo4J [1]) steht vor allem ein Punkt im Fokus: Die Struktur der Daten so umzugestalten, dass die Vorteile einer Graph-Datenbank zur Geltung kommen. Es gibt jedoch viel mehr Aspekte, die bei einem solchen Vorhaben berücksichtigt werden müssen. In diesem Artikel wird an einem Beispiel aus dem Bereich OpenData die Aufbereitung der Daten und die dafür verfügbaren Sprachmittel aus der Python-Welt vorgestellt.

Als Basis für den Artikel dient eine offene Datenquelle (zum Begriff 'offene Daten' siehe [2]), die durch die DB bereitgestellt wird [3]. Die Daten liegen als CSV-Datei vor und geben folgende Entitäten und Beziehungen wieder:

- Über unterschiedliche Stationen können Buchungen betätigt werden.
- Jede Station hat einen Betreiber.
- In einer Buchung wird ein Fahrzeug ausgeliehen.
- Das Fahrzeug wird von einer Station abgeholt und in einer Station abgegeben.
- Jede Buchung ist einer Tarifklasse zugeordnet.

Außer Buchungen haben alle Entitäten folgende Eigenschaften:

- Firmenname (COMPANY)
- Firmengruppe (COMPANY_GROUP)

Fahrzeuge haben folgende Eigenschaften:

- ID (VEHICLE_HAL_ID)
- Art (VEHICLE_MODEL_TYPE)
- Hersteller (VEHICLE_MANUFACTURER_NAME)
- Model (VEHICLE_MODEL_NAME)
- Detail-Infos (VEHICLE_TYPE_NAME)
- Fahrzeugidentifikationsnummer (VIN)
- Fahrzeugkennzeichen (REGISTRATION_PLATE)
- Seriennummer (SERIAL_NUMBER)
- Leistung (KW)
- Brennstoff (FUEL_TYPE_NAME)
- Beschaffungsart (OWNERSHIP_TYPE)
- Tank-Kapazität (CAPACITY_AMOUNT)
- Bordcomputertyp (ACCESS_CONTROL_COMPONENT_TYPE)

Stationen haben folgende Eigenschaften:

- ID – Teil 1 (RENTAL_ZONE_HAL_ID)
- ID – Teil 2 (RENTAL_ZONE_HAL_SRC)
- Name (NAME)
- Kurzname (CODE)
- Typ (TYPE)
- Stadt (CITY)
- Land (COUNTRY)
- Breitengrad (LATITUDE)
- Längengrad (LONGITUDE)
- Längengrad (LONGITUDE)
- Flag – Flughafen in der Nähe (POI_AIRPORT_X)
- Flag – Bahnhof in der Nähe (POI_LONG_DISTANCE_TRAINS_X)
- Flag – S-Bahn-Station in der Nähe (POI_SUBURBAN_TRAINS_X)
- Flag – U-Bahn in der Nähe (POI_UNDERGROUND_X)
- Flag – Aktiv (ACTIVE_X)

Tarifklassen haben folgende Eigenschaften:

- ID (HAL_ID)
- Bezeichnung (CATEGORY)

Buchungen haben folgende Eigenschaften:

- Buchung (BOOKING_HAL_ID)
- Verweis zur Tarifklasse (CATEGORY_HAL_ID)
- Verweis zum Fahrzeug (VEHICLE_HAL_ID)
- Verweis zum Kunden (CUSTOMER_HAL_ID)
- Buchungsdatum (DATE_BOOKING)
- Buchung vom (DATE_FROM)
- Buchung bis (DATE_UNTIL)
- Flag – Extra Buchungsgebühr notwendig (COMPUTE_EXTRA_BOOKING_FEE)
- Flag – Quernutzung (TRAVERSE_USE)
- Hinterlegte Streckenlänge im Rahmen der Buchung (DISTANCE)
- Name der Station, in der das Auto abgeholt wurde (START_RENTAL_ZONE)
- Verweis zur Station, in der das Auto abgeholt wurde (START_RENTAL_ZONE_HAL_ID)
- Name der Station, in der das Auto abgegeben wurde (END_RENTAL_ZONE)
- Verweis zur Station, in der das Auto abgegeben wurde (END_RENTAL_ZONE_HAL_ID)
- Eindeutigkeitsprefix zu den Stationsverweisen (RENTAL_ZONE_HAL_SRC)
- Stadt (CITY_RENTAL_ZONE)
- Technische Quelle der Buchung (TECHNICAL_INCOME_CHANNEL)

Um diese Daten in eine Neo4J-Datenbank zu importieren, müssen mehrere Punkte für jede Entität und deren Beziehungen analysiert und überprüft werden:

- Gibt es Spalten, die Null-Values enthalten?
- Welche Variabilität haben die Werte in den einzelnen Spalten, welchen Informationsmehrwert kommt dadurch zustande bzw. nicht zustande?
- Sind aus den Werten, die in der Spalte enthalten sind, gewisse Schlussfolgerungen in Bezug auf die Domäne ableitbar?
- Stimmen die Verweise auf andere Entitäten (relationale Integrität)?
- Welche Kategorisierungsmerkmale können aus den Spalten und Inhalten entnommen werden, um ein Label zu bilden?
- Wie können die Informationen in der Graph-DB abgelegt werden und welchen Zweck verfolgt man damit?

Auf Basis dieser Analyse entstehen dann Entscheidungen über die Art der Transformation, die für das Importieren in die Graph-DB notwendig ist.

Für die Analyse ist es sinnvoll, mit der einfachsten Identität anzufangen und die komplizierteste Entität erst am Ende zu betrachten. In diesem Sinne werden die einzelnen Entitäten im Folgenden analysiert.

An dieser Stelle gibt es noch einige Hinweise über die verwendeten Funktionalitäten/Bibliotheken:

Bei der Funktionalität kommen drei Python-Module und -Bibliotheken zum Einsatz: Pandas [7], Numpy [8] und Functools [9]. Diese werden folgendermaßen in den Code importiert:

```
import pandas as pd
import numpy as np
import functools as ft
```

Für das Laden der Dateien kommt immer derselbe Aufruf einer Methode im Pandas-Framework zum Einsatz. Die angesprochene Methode lädt die Daten in der angegebenen Datei und stellt sie als *DataFrame* zur Verfügung. Hier wird der Aufruf für die Tarifklassen als Beispiel dargestellt:

```
df_cat = pd.read_csv('./datasets/
OPENDATA_CATEGORY_CARSHARING.csv',
quotech ar='"',encoding ='utf-8', sep=';')
```

Der Aufruf unterscheidet sich unter den Entitäten nur durch den Dateinamen und wird deswegen im Folgenden nicht explizit erwähnt.

Für die Überprüfung der Spalten mit Null-Values gibt es zwei Möglichkeiten, die beiderseits zum Einsatz kommen:

1. Implizit über einen Info-Aufruf im Pandas-Framework:

```
df_rz.info(null_counts=True)
```

Das Ergebnis ist über die Anzahl der Zeilen in *DataFrame* und über die Anzahl der gefüllten Zeilen pro

Spalte implizit erkennbar. Die zugrundeliegende Anzahl der Zeilen liefert insgesamt die zweite Zeile in der Ausgabe:

```
RangeIndex: 548073 entries, 0 to 548072
```

Bei einer Spalte ohne Null-Values, ist die Anzahl der Non-Null-Zeilen gleich:

```
BOOKING_HAL_ID 548073 non-null int64
```

Bei einer Spalte mit Null-Values, weicht die Anzahl der Non-Null-Zeilen von dem Wert ab:

```
TECHNICAL_INCOME_CHANNEL 496097
non-null object
```

2. Über das Filtern der betroffenen Spalten mit Möglichkeiten aus dem Pandas-Framework [11]:

```
null_columns=df_lj_booking_rz.columns[
df_lj_booking_rz.isnull().any()
]
```

In der Ausgabe müssen dann diese Spalten angesprochen werden, um z. B. die Anzahl der Null-Zeilen auszugeben:

```
df_lj_booking_rz[null_columns].isnull().sum()
```

```
DISTANCE 201
TECHNICAL_INCOME_CHANNEL 51976
dtype: int64
```

Das Bilden der Label-Informationen und das Benennen der Beziehungstypen wurden auf eigene Funktionen ausgelagert. Im Folgenden werden nur Teile aus dieser Funktion dargestellt, wenn diese Themen behandelt werden. Hier erfolgt eine kurze (Pseudo-) Vorstellung der beiden Funktionen sowie die dazu gehörigen Erläuterungen:

Das Bilden der Label-Informationen geschieht in zwei Schritten: Über eine Map stellt man das Hauptlabel fest und ggf. ergänzt man dieses um weitere Labels auf Basis von Informationen aus bestimmten Spalten in der Zeile:

```
def getLabel(row, nodeName): switcher = {
  "ENTITY_1": "LABEL_FOR_ENTITY", ...
}
label = switcher.get(nodeName, 'OBJ')
if nodeName == 'ENTITY_1':
label += ";" + str(row["COL_OF_ENTITY_1"]) ...
return label
```

Danach wird nur erwähnt, wie das Hauptlabel lautet und ggf. anschließend der Code-Teil gezeigt, in dem die weiteren Labels generiert werden.

Das Benennen der Beziehungstypen erfolgt auf Basis einer Map-Verkettung:

```
def getRelationshipType(row, types):
  switcher = {
    "ENTITY_1": {
      "ENTITY_2": "RELATIONSHIP_NAME"
      },
    ...
  }
  relType = switcher.get(types[0], {}).get(
    types[1], 'UNDEFINED'
  )
  return relType
```

Dabei ist der Schlüssel der übergeordneten Map der Name der Entität, aus der die Beziehung herausgeht. Der Schlüssel der untergeordneten Map (als Wert innerhalb der ersten Map) ist der Name der Entität, die die Beziehung zeigt. Als Nächstes wird nur die Information erwähnt, welchen Typ eine Beziehung zwischen zwei Entitäten bekommt.

Das Schreiben der Dateien als Basis für das Importieren in die Neo4J-Datenbank erfolgt über folgende Funktion:

```
def writeDsvFile(
  df, typeName, delimiter, columnsList,
  headerList):
  filename = './output/' + typeName + '.dsv'
  df.to_csv(
    filename, index = False, sep = delimiter, columns =
    columnsList, header = headerList)
```

In dieser Methode kommt die DataFrame-Funktion to_csv zum Einsatz, wobei das Trennzeichen (',' als Default-Trennzeichen im Neo4J-Import-Tool) die auszugebenden Spalten und die Header-Spalten vorgibt. Die Header-Spalten spielen dabei eine Sonderrolle, weil das Import-Tool von Neo4J das Importieren der

Daten auf dieser Basis vornimmt. Informationen darüber sind unter [12] zu finden.

Die Spalten COMPANY und COMPANY_GROUP weisen in den unterschiedlichen Tabellen je einen einzigen Wert auf, weswegen sie nicht weiter behandelt werden.

Viele der Code-Beispiele enthalten Zeilen mit einem print(...)-Aufruf. Z. B. ist die zuvor dargestellte Zeile so zu finden:

```
print(df_lj_booking_rz[null_columns].isnull().sum())
```

Dies ist auf die Auslagerung der Behandlung einzelner Entitäten auf explizite Funktionen zurückzuführen. Ohne einen print(...)-Aufruf erzeugen diese Zeilen natürlich auch außerhalb einer Funktion (in einem Jupyter-Notebook) Ausgaben.

Der komplette Code ist unter [10] zu finden.

Die Tarifklassen

In den Tarifklassen gibt es keine Null-Value-Spalten:

```
print(df_cat[null_columns].isnull().sum())
```

```
Series([], dtype: float64)
```

Bei den Tarifklassen gibt es nur eine Spalte, die analysiert werden muss: Bezeichnung.

Wenn die in dieser Spalte vorkommenden Werte ausgegeben werden, wird deutlich, dass die Tarifklassen teilweise eine einfache Bezeichnung haben - aus einem Wort bestehend - und teilweise Spezialisierungen wiedergeben - aus mehreren Wörtern bestehend -, die zwei Informationen erbringen: Haupttarif-Klassifizierung (teilweise identisch mit einer anderen Tarifklasse mit einem Wort) und Detail-Klassifizierung:

```
print(sorted(df_cat['CATEGORY'].unique()))
```

```
'Kleinklasse (teilweise ohne Navi)',
'Komfortklasse',
'Kompaktklasse',
'Kompaktklasse Flughafen',
'Mini (teilweise ohne Navi)',
'Miniklasse Flughafen',
'Mittelklasse',
'Mittelklasse Flughafen',
'Oberklasse',
'Sonderklasse (Alfa MiTo / Citroën DS3)', '
'Sonderklasse (C-Klasse)',
'Sonderklasse (Fiat 500)',
'Sonderklasse (Ka)',
'Transport I',
'Transport II',
'Transport III', 'Van-/Busklasse',
'Versicherung pro Fahrt',
'Werbeklasse (mit Beklebung)',
'Zubehör'
```

Um eine saubere Wiederverwendbarkeit der Tarif-Informationen in anderen Knoten wie Buchungen zu ermöglichen, ist es sinnvoll Haupttarifklassen als zusätzliche Knoten zu bilden, die aus Tarifklassen heraus referenziert werden. Um das zu erreichen, müssen folgende Änderungen/Erweiterungen in den Daten vorgenommen werden:

- Haupttarifklassenbezeichung auf Basis des ersten Wortes bilden (Zusatzspalte 'PARENT_CATEGORY' in der DataFrame)
- Damit die Haupttarifklassen als Label nutzbar sind, das Zeichen '/' im ersten Wort entfernen
- Für die Tarifklassen, die nur aus einem Wort bestehen, den Prefix 'BASE' hinzufügen, damit die Bildung einer Haupttarifklasse möglich ist.
- Für die Haupttarifklassen eine eindeutige ID bilden. Dafür reicht es der bestehenden ID 10.000 zu addieren. Damit bleibt man im selben Nummernraum wie die bestehenden Tarifklassen.

Als Label wird den Tarifklassen die Bezeichnung TARIFF_CLASS und den Haupttarifklassen die Bezeichnung MAIN_TARIFF_CLASS (Zusatzspalten 'LABEL' und 'PARENT_LABEL' in der DataFrame) gegeben. Die Beziehung zwischen den beiden Knoten wird BELONGS_TO genannt (Zusatzspalte 'REL_TYPE_CAT_PCAT' in der DataFrame).

Der Code für diese Operationen sieht folgendermaßen aus:

```
df_cat['PARENT_CATEGORY'] = df_cat.apply({
  'CATEGORY' :
    lambda x: x.split()[ 0].replace('/', '').replace('-', '_')
})
```

```
df_cat['CATEGORY'] = df_cat.apply({
   'CATEGORY' : lambda x: x if (len(x.split ())>1)
      else (str(x) + " BASE")
})
df_cat['PARENT_ID'] = df_cat.apply({
   'HAL_ID' : lambda x: x + 10000
})
df_cat['LABEL'] = df_cat.apply(
   getLabel, axis=1, nodeName='CATEGORY'
)
df_cat['PARENT_LABEL'] = df_cat.apply(
   getLabel, axis=1, nodeName='PARENT_CA TEGORY'
)
 df_cat['REL_TYPE_CAT_PCAT'] = df_cat.apply(
   getRelationshipType, axis=1,
      types=["CATEGORY","PARENT_CATEGORY"]
)
```

Mit der DataFrame-Funktion apply(...) wird eine Funktion an jedem Glied (Zeile, wenn axis=1; Spalte, wenn axis=0) angewandt und das Ergebnis als eine neue DataFrame-oder Series-Instanz gespeichert.

Als Header wird derselbe Inhalt für beide Knotentypen (Haupttarifklasse und Tarifklasse) verwendet:

```
categoryID:ID(CATEGORY-ID),name,:LABEL
```

Als Ergebnis werden drei Import-Dateien erzeugt: Zwei für die Knoten, eine für die Beziehungen:

```
writeDsvFile(
   df_cat, 'categories', defaultCsvItemDelimiter, [
      'HAL_ID', 'CATEGORY', 'LABEL'
   ],
   ['categoryID:ID(CATEGORY-ID)','name', ':LABEL']
)
writeDsvFile(
   df_cat, 'parent_categories', defaultCsvItemDelimiter,
   ['PARENT_ID', 'PARENT_CATEGORY',
      'PARENT_LABEL'],
   ['categoryID:ID(CATEGORY-ID)','name', ':LABEL']
)
writeDsvFile(
   df_cat, 'rel_cat_pcat', defaultCsvItemDelimiter, [
   'HAL_ID', 'PARENT_ID', 'REL_TYPE_CAT_PCAT'
],
[':START_ID(CATEGORY-ID)',
   ':END_ID(CATEGORY-ID)', ':TYPE'])
```

Die Fahrzeug-Informationen

Der Nummernkreis bei den IDs der Fahrzeuge kann wie folgt festgestellt werden:

```
print(df_fz.VEHICLE_HAL_ID.agg(['min', 'max']))
```

```
min 143031
max 181564
Name: VEHICLE_HAL_ID, dtype: int64
```

Die kommenden Spalten beinhalten Null-Werte:

```
SERIAL_NUMBER 1138
CAPACITY_AMOUNT 95
ACCESS_CONTROL_COMPONENT_TYPE 1138
dtype: int64
```

Da die Spalten nicht solche sind, die sich als Label eignen (Klassifizierungsmerkmal), bedarf es keiner Anpassung. Als Label-Informationen werden drei Spalten fokussiert: Hersteller-, Typ- und Brennstoff-Informationen. Die Ausgabe der vorkommenden Werte bei diesen Spalten gibt folgende Werte zurück:

```
print('%s Unique values of column VEHICLE_
MANUFACTURER_NAME,
VEHICLE_MODEL_ TYPE and FUEL_TYPE_NAME %s'
%(seperatingLine, seperatingLine))
print(sorted(
df_fz['VEHICLE_MANUFACTURER_NAME'].unique()))
print(sorted(
df_fz['VEHICLE_MODEL_TYPE'].unique()))
print(sorted(
df_fz['FUEL_TYPE_NAME'].unique()))
```

```
Citroën Fiat Ford MCC Mercedes Mitsubishi Nissan
Opel Peugeot Renault Toyota VW
Auto
Diesel Erdgas (Nottank: Super) Plug-In (Strom,
Super) Strom Super (Benzin) Super E10
```

Außer den Brennstoff-Informationen sind die zwei Informationen als Label nutzbar. Bei den Brennstoff-Typen ist eine Überarbeitung notwendig, damit die Information als Label nutzbar ist. Auf Basis dieser Feststellungen sieht das Bilden der Label-Informationen so aus:

```
def getLabel(row, nodeName):
   switcher = {
      ...
      "VEHICLE": "VEHICLE",
      ...
   }
   ...
   elif nodeName == 'VEHICLE':
      fuelTypeOrig = str(row["FUEL_TYPE_NAME"])
```

```
        fuelTypeSwitcher = {
            "Diesel": "DIESEL",
            "Erdgas (Nottank: Super)": "ERDGAS",
            "Plug In (Strom, Super)": "PLUGIN",
        "Strom": "STROM",
        "Super (Benzin)": "SUPER",
        "Super E10": "E10"
    }
    fuelType = fuelTypeSwitcher.get(
        fuelTypeOrig, 'UNDEFINED'
    )
    label +=
        ";" + str(row[
            "VEHICLE_MODEL_TYPE"].upper()) +
        ";" + str(row[
            "VEHICLE_MANUFACTURER_NAME"].upper()) +
        ";" + fuelType;
    ...
```

Ob die Detailinformationen (Spalte "VEHICLE_TYPE_NAME") das Kommazeichen enthalten, findet man auf diese Weise heraus:

```
print(len(list(filter( \
    lambda x: "," in x \
    , df_fz['VEHICLE_TYPE_NAME'].unique() \
))))
```

Weil 53 Einträge das Kommazeichen enthalten, müssen sie von diesen bereinigt werden, damit sie nicht als Trennzeichen interpretiert werden:

```
df_fz['VEHICLE_TYPE_NAME'] =
df_fz.apply({'VEHICLE_TYPE_NAME' :
    lambda x: x .replace(',', '.')})
```

Der Header der Fahrzeuginformationen enthält außer den Detailinformationen über das Fahrzeug (als Attribut "modelDetails") auch die oben aufgeführten Spalten:

```
vehicleID:ID(VEHICLE- ID),modelName,
modelDetails,vin,registrationPlate,kw:long,
fuelType,ownershipType,:LABEL
```

Als Ergebnis gibt es nur eine Import-Datei für die Knoten:

```
writeDsvFile(
    df_fz, 'vehicles', defaultCsvItemDelimiter, [
        'VEHICLE_HAL_ID', 'VEHICLE_MODEL_NAME',
        'VEHICLE_TYPE_NAME', 'VIN',
        'REGISTRATION_PLATE', 'KW', 'FUEL_TYPE_NAME',
        'OWNERSHIP_TYPE', 'CAPACITY_AMOUNT',
        'ACCESS_CONTROL_COMPONENT_TYPE', 'LABEL'
    ],
    ['vehicleID:ID(VEHICLE-ID)','modelName',
        'modelDetails','vin','registrationPlate','kw:long',
        'fuelType','ownershipType','capacityAmount',
        'bordComputerType', ':LABEL'
    ]
)
```

Die Stationen

Bei den Stationen kommen zunächst die üblichen Feststellungen wie in den früheren Entitäten zum Tragen:

Nummernkreis der IDs:

```
min 38
max 406429
Name: RENTAL_ZONE_HAL_ID, dtype: int64
```

Falls es ein Interesse auf einen gemeinsamen Nummernkreis für alle Entitäten gegeben hätte, müsste an dieser Stelle überlegt werden, wie eine Eindeutigkeit gewährleistet werden kann. Da in diesem Fall einzelne Nummernkreise genommen werden, ist dies hier nicht relevant.

Spalten mit Null-Werten:

```
LATITUDE 12
LONGITUDE 12
dtype: int64
```

Da sich diese Spalten als Label nicht eignen, sind keine weiteren Überlegungen notwendig.

Nun sind einige Spalten daraufhin zu prüfen, ob sie bei der Transformation auf einen Graphen besonders berücksichtigt werden müssen:

Das "Eindeutigkeitsmerkmal" RENTAL_ZONE_HAL_SRC, das Land ("COUNTRY") und der Aktiv-Flag ("ACTIVE_X"):

```
print(df_rz['RENTAL_ZONE_HAL_SRC'].unique())
print(df_rz['COUNTRY'].unique())
print(df_rz['ACTIVE_X'].unique())
```

```
['Station']
['Deutschland']
['Nein' 'Ja']
```

Wie zu erkennen ist, besitzen die Spalten RENTAL_ZONE_HAL_SRC und COUNTRY keine Relevanz, da sie überall denselben Wert aufweisen. Der Flag für den Zustand der Station (aktiv oder nicht) kann jedoch als Label genommen werden.

Der nächste Punkt ist die Fragestellung, ob es ausschließlich nur Städte ohne Leerzeichen gibt:

```
print(ft.reduce(
   lambda x, y: x & y ,
      map(
         lambda x: len(x.split())==1
            , df_rz['CITY'].unique())
))
```

In diesem Code-Ausschnitt werden zunächst die Werte in der Spalte "CITY" darauf gemappt, ob die Anzahl der Wörter 1 ergibt. Anschließend werden die Elemente in der ganzen Liste mit & verknüpft. Das Ergebnis zeigt, dass es in der Spalte auch Städte gibt, die aus mehreren Wörtern bestehen:

```
False
```

Über fast denselben Weg kann man mit den Städten mit Leerzeichen verfahren:

```
print(list(filter(
   lambda x: len(x.split())>1
   , df_rz['CITY'].unique()
)))
```

```
['Frankfurt am Main', 'München (DB-intern)', 'Mannheim (BB)', 'Wiesbaden (BB)', 'Westerland auf Sylt', 'Flugh. Berlin', 'Flugh. München', 'Flugh. Frankfurt', 'Flugh. Stuttgart/Echterdingen', 'Au (Sieg)', 'Homburg / Saar', 'Mülheim / Ruhr', 'Losheim am See', 'St. Wendel', 'St. Ingbert', 'DB Ein-/Aussteuerung']
```

Aufgrund dieser Erkenntnisse würde sich der Aufwand nicht lohnen, aus dem Städtenamen ein Label zu bilden. Die Information über die Stadt fließt als normales Knoten-Attribut in die Graph-DB hinein.

Die letzte inhaltliche Frage ist, ob sich der Typ einer Station als Label eignet:

```
print(df_rz['TYPE'].unique())
```

```
['parkingarea' 'stationbased' 'freefloating']
```

Wie die Werte es verdeutlichen, kann die Information über den Typ der Station als Label genommen werden.

Zuletzt wird überprüft, ob die Längengrad- und Breitengrad-Informationen korrekt in die Graph-DB übertragbar sind. Dafür werden die letzten fünf Zeilen der beiden Spalten ausgegeben, um stichprobenweise Inhalte zu sehen. Dies geschieht mithilfe der Funktion iloc(...), die den Bereich in der DataFrame (hier die letzten fünf Zeilen [:5] und die Spalten 8 und 9 [7:9]) übergeben bekommt, die selektiert werden soll:

```
print(df_rz.iloc[:5, 7:9])
```

LATITUDE	LONGITUDE
52,491966685810670	13,437334746122360
52,509446616791216	13,433682918548584
54,077917752559770	12,132610380649567
53,633873801997470	11,406887769699097
49,981667892009890	9,144830703735351

Die Inhalte wären mit einem Komma als Seperator nicht übertragbar. Deswegen muss eine Umwandlung auf einen Float-Wert unternommen werden. Dafür ersetzen wir das Kommazeichen durch einen Punkt und wandeln den Typ der beiden Spalten auf Float um:

```
df_rz['LONGITUDE'] = df_rz.apply({
   'LONGITUDE' : lambda x: str(x).replace(', ', '.')
})
df_rz['LONGITUDE'] = df_rz['LONGITUDE'].astype(float)

df_rz['LATITUDE'] = df_rz.apply({
   'LATITUDE' : lambda x: str(x).replace(',', '.')
})
df_rz['LATITUDE'] = df_rz['LATITUDE'].astype(float)
```

LATITUDE	LONGITUDE
52.491967	13.437335
52.509447	13.433683
54.077918	12.132610
53.633874	11.406888
49.981668	99.144831

Beim Bilden des Labels nehmen wir zwei Spalten als zusätzliche Labels (TYPE und ACTIVE_X):

```
def getLabel(row, nodeName):
  switcher = {
    ...
    "RENTAL_ZONE": "STATION"
  }
  label = switcher.get(nodeName, 'OBJ')
  ...
  elif nodeName == 'RENTAL_ZONE':
    label += ";" + str(row["TYPE"].upper())
    label += ";ACTIVE" if str(
      row["ACTIVE_X"].upper()
    ) == "JA" else ";INACTIVE"
  return label
```

Der Header repräsentiert, die enthaltenen Informationen. Wichtig ist dabei, dass der Längengrad und der Breitengrad den Typ float in Neo4J deklariert bekommen (:float):

```
rentalZoneID:ID(RENTAL-ZONE-ID), name, code,
type, city, latitude:float, longtitude:float, poiAir-
port, poiLongDistanceTrains, poiSuburbanTrains,
poiUnderground, :LABEL
```

Als Ergebnis entsteht nur eine Datei für die Knoten-Erstellung:

```
writeDsvFile(df_rz, 'rentalZones', defaultCsvItemDelimiter,
['RENTAL_ZONE_HAL_ID', 'NAME', 'CODE', 'TYPE',
'CITY', 'LATITUDE','LONGITUDE', 'POI_AIRPORT_X',
'POI_LONG_DISTANCE_TRAINS_X', 'POI_SUBURBAN_
TRAINS_X','POI_UNDERGROUND_X', 'LABEL'],
['rentalZoneID:ID(RENTAL-ZONE-ID)', 'name', 'code',
'type', 'ci ty', 'latitude:float', 'longtitude:float', 'poiAirport',
'poiLongDistanceTrains', 'poiSuburbanTrains',
'poiUnderground', ':LABEL']
)
```

Die Buchungsinformationen

Bei den Buchungsinformationen gibt es zwei große Themenblöcke:

- Die Referenzen einer Buchung
- Die Eigenschaften einer Buchung

Bei den Referenzen einer Buchung stellt sich vor allem folgende Frage: Sind die Referenzen als Beziehung umzusetzen oder können sie auch anders aufgenommen werden? Aufgrund der Normalisierung der Daten ist es nicht direkt erkennbar, ob die referenzierte Entität für eine Buchung etwas "Auszeichnendes" hat. Wie bekannt geht es dabei, um die bisher behandelten Entitäten. Die Frage kann mit einem Fokus auf die Nutzung der Daten beantwortet werden: Was würde man gerne über den kurzen Weg erfahren wollen, wenn man eine einfache Auswertung über die Carsharing-Informationen erfahren will? Eine ziemlich zentral stehende Entität Buchung wäre durchaus dafür geeignet, solche Informationen als Label zu beinhalten. Auf Basis dieser Überlegungen kristallisieren sich zwei Informationen heraus: Wie bereits bei den Tarifklassen erwähnt, würde sich die als Meta-Information gebildete Haupttarifklasse als Merkmal für eine Buchung eignen. Bei den Fahrzeugen kämen sowohl die Eigenschaft Hersteller als auch die Information Brennstoff als Merkmal für eine Buchung infrage. Weil die Werte in der Spalte VEHICLE_MANUFACTURER_NAME stets aus einem Wort bestehen, wird aufgrund der Einfachheit die Hersteller-Information als Buchungsmerkmal genommen. Natürlich kann in anderen Fällen die Brennstoff-Information ebenfalls genommen werden.

Bei den Referenzen einer Buchung bleibt es aber nicht nur bei der Frage hinsichtlich der Merkmale, sondern es ergeben sich noch weitere Fragen:

- ob alle referenzierten Datensätze auch wirklich vorhanden sind,
- ob alle korrekten Referenzen auch zu einer Beziehung führen müssen,
- ob es nicht im Datenmodell als direkte Referenz nicht bestehende Zusammenhänge gibt, die sich durchaus als Beziehung in der Graph-Datenbank eignen.

Die letzten beiden Fragen sind als verwandt anzusehen. Auf sie kann man am besten eingehen, wenn man sich mit einer Referenz und deren Gegebenheit beschäftigt. Wie z. B. der Zusammenhang zwischen Buchung und Station: In der Buchung werden zwei Stationen referenziert: Die Anfangsstation (START_RENTAL_ZONE_HAL_ID) und die Endstation (END_RENTAL_ZONE_HAL_ID). Praktisch haben die Stationen jedoch eher etwas mit dem gebuchten Fahrzeug zu tun. Es geht nämlich nicht um die Station, in der die Buchung zustande kam, sondern um die Station, in der das Fahrzeug abgeholt wurde und um die Station, in der das Fahrzeug abgegeben wurde. Aus dem Grund eignen sich die Referenzen eher als Bindeglied zwischen Fahrzeug und Station. An dieser Stelle wird nur vermerkt, dass eine Beziehung zwischen Fahrzeug

und Station einer Beziehung zwischen Buchung und Station bevorzugt wird.

Bei der Untersuchung der Referenzen wird wie im Folgenden beschrieben vorgegangen, wobei die Vorgehensweise auch für das Holen der Label-Informationen aus anderen Entitäten wichtig ist: Die Buchungsinformationen werden mithilfe der bestehenden Foreign-Keys mit den anderen Entitäten schrittweise gejoint. Dabei werden die relevanten Informationen ebenfalls aus den anderen Entitäten geholt. Die Join-Operationen erfolgen in dieser Reihenfolge: [*[[Buchungen & Fahrzeuge] & Stationen]* & Tarifklassen].

Das Joinen findet durch die Funktion merge(...) statt, wobei die zu joinende Tabellen auf das Nötigste durch die Filter-Möglichkeit in DataFrames und durch die copy(...) Funktion beschränkt wird.

Beim Joinen der Buchungen und Fahrzeuge erweisen sich 1.608 Referenzen als ungültig:

```
df_lj_booking_vehicle = pd.merge(
   df_booking,
   df_fz.get(
      ['VEHICLE_MANUFACTURER_NAME',
         'VEHICLE_HAL_ID']
   ).copy(True),
   on='VEHICLE_HAL_ID', how='left')
   null_columns=df_lj_booking_vehicle.columns[
      df_lj_booking_vehicle.isnull().any()]
...
print(df_lj_booking_vehicle[null_columns].isnull().sum())
```

```
VEHICLE_MANUFACTURER_NAME 1608
dtype: int64
```

Es geht dabei um weniger als ein Dutzend Fahrzeuge, die fehlen:

```
print(sorted(
   df_lj_booking_vehicle[df_lj_booking_vehicle[
      "VEHICLE_MANUFACTURER_NAME"].isnull()
   ]
   ['VEHICLE_HAL_ID'].unique())
)
```

```
[157199, 160152, 160289, 172241, 172259,
172261, 172270, 172310, 172316, 172319,
173572, 173573, 173952]
```

Für die spätere Bearbeitung wird die fehlende Hersteller-Information als "UNKNOWN" umgewandelt und die existierenden Werte auf Upcase:

```
df_lj_booking_vehicle
['VEHICLE_MANUFACTURER_NAME']
   = df_lj_booking_vehicle.apply(
   {'VEHICLE_MANUFACTURER_NAME' : lambda x:
"UNKNOWN" if str(x) == 'nan ' else str(x).upper()}
   )
```

Für die Beziehung zwischen den Buchungen und Fahrzeugen kommt außerdem eine Spalte mit dem konstanten Wert "REFERS_TO" hinzu:

```
df_lj_booking_vehicle
['REL_TYPE_BOOKING_VEHICLE']
   = df_lj_booking_vehicle.apply(
   getRelationshipType, axis=1,
   types=["BOOKING","VEHICLE"]
   )
```

Im nächsten Schritt werden die Stationen in zwei Stufen gejoint:

```
df_lj_booking_rz = pd.merge(\
   pd.merge(\
      df_lj_booking_vehicle, \
      df_rz.get(['RENTAL_ZONE_HAL_ID']).copy(True), \
      left_on='START_RENTAL_ZONE_HAL_ID',
      right_on='RENTAL_ZONE_HAL_ID', how='left'),\
      df_rz.get(['RENTAL_ZONE_HAL_ID']).copy(True), \
      left_on='END_RENTAL_ZONE_HAL_ID',
      right_on='RENTAL_ZONE_HAL_ID', how='left', \
suffixes=('_LEFT', '_RIGHT'))
```

In der zweiten Merge-Operation bekommen die gleichnamigen Spalten aus den beiden DataFrames (in diesem Fall RENTAL_ZONE_HAL_ID) je einen Suffix (Parameter suffixes).

Als Ergebnis erweisen sich 31.863 Station-Referenzen als ungültig:

```
RENTAL_ZONE_HAL_ID_LEFT 31863
RENTAL_ZONE_HAL_ID_RIGHT 31863
dtype: int64
```

Im letzten Schritt kommen die Tarifklassen-Informationen dazu:

```
df_lj_booking_cat = pd.merge(
   df_lj_booking_rz,
   df_cat.get(['PARENT_CATEGORY', 'CATEGORY',
   'HAL_ID']).copy(True),
   left_on='CATEGORY_HAL_ID',
   right_on='HAL_ID',
   how='left')
```

Es stellt sich heraus, dass 3.500 referenzierte Tarifklassen nicht existieren:

```
RENTAL_ZONE_HAL_ID_LEFT 31863
RENTAL_ZONE_HAL_ID_RIGHT 31863
PARENT_CATEGORY 3500
CATEGORY 3500
HAL_ID 3500
dtype: int64
```

Dabei geht es um 42 Tarifklassen, die nicht gegeben sind:

```
categoryIdList = sorted(
   df_lj_booking_cat[df_lj_booking_cat["CATEGORY"] ==
  'UNDEFINED']\
   ['CATEGORY_HAL_ID'].unique()
)
corruptCategoryInfo = str(len(categoryIdList))
print('number of missing category ids:
   %s' %(corruptCategoryInfo))
```

```
number of missing category ids: 42
```

Für die spätere Verarbeitung werden die Tarifklassen-Informationen umgewandelt. Nicht existierende bekommen den Wert "UNDEFINED" und existierende werden Uppercase:

```
df_lj_booking_cat['PARENT_CATEGORY'] =
df_lj_booking_cat.apply(
   {'PARENT_CATEGORY' :
   lambda x: "UNDEFINED" if str(x) == 'nan'
   else str(x).upper()}
)
df_lj_booking_cat['CATEGORY'] =
df_lj_booking_cat.apply(
   {'CATEGORY' : lambda x: "UNDEFINED"
   if str(x) == 'nan' else str(x).upper()}
)
```

Für die Beziehung zwischen den Buchungen und Tarifklassen kommt außerdem eine Spalte mit dem konstanten Wert "ACCORDING_TO" hinzu:

```
df_lj_booking_cat['REL_TYPE_BOOKING_CATEGORY'] =
df_lj_booking_cat.apply(getRelationshipType, axis=1,
types=["BOOKING", "CATEGORY"])
```

Um die Beziehung zwischen Fahrzeug und Station herzustellen, wird das zuletzt zustande gekommene DataFrame (inkl. Fahrzeug- und Stationsinformationen) gefiltert (Zeilen mit ungültigen Verweisen fallen weg) und per groupby auf Basis der drei relevanten IDs gruppiert (die Anzahl per Gruppe findet sich dann in der Spalte count):

```
df_gb_vrz = df_lj_booking_cat
   [(df_lj_booking_cat
      .RENTAL_ZONE_HAL_ID_LEFT.notnull()) &
   (df_lj_booking_cat
      .RENTAL_ZONE_HAL_ID_RIGHT.notnull()) &
   (df_lj_booking_cat
      .VEHICLE_MANUFACTURER_NAME !=
      "UNKNOWN")]
.get(['VEHICLE_HAL_ID', 'RENTAL_ZONE_HAL_ID_
   LEFT', 'RENTAL_ZONE_HAL_ID_RIGHT',
      'VEHICLE_MANUFACTURER_NAME']).copy(True)
.groupby(['VEHICLE_HAL_ID', 'RENTAL_ZONE_HAL_
      ID_LEFT', 'RENTAL_ZONE_HAL_ID_RIGHT']).size()
.reset_index().rename(columns={0:'count'})
df_gb_vrz['RENTAL_ZONE_HAL_ID_LEFT']=
df_gb_vrz['RENTAL_ZONE_HAL_ID_LEFT']
   .astype(int)
df_gb_vrz['RENTAL_ZONE_HAL_ID_RIGHT']=
df_gb_vrz['RENTAL_ZONE_HAL_ID_RIGHT']
   .astype(int)
...
print(df_gb_vrz)
```

VEHICLE_ HAL_ID	RENTAL_ZONE_ HAL_ID_LEFT	RENTAL_ZONE_ HAL_ID_RIGHT	count
143031	400342	400342	5
146233	146233	404524	15

Die Suche nach Gruppen mit unterschiedlicher Anfang- und Endstation bleibt erfolglos:

```
print(df_gb_vrz
[df_gb_vrz.RENTAL_ZONE_HAL_ID_LEFT !=
   df_gb_vrz.RENTAL_ZONE_ HAL_ID_RIGHT])
```

```
Empty DataFrame
Columns: [VEHICLE_HAL_ID, RENTAL_ZONE_HAL_
ID_LEFT, RENTAL_ZONE_HAL_ID_RIGHT, count]
Index: []
```

Um weitere Erkenntnisse zu gewinnen, wird das DataFrame auf Basis des Fahrzeugs gruppiert, wodurch sich die Anzahl der unterschiedlichen Stationen pro Fahrzeug ergibt:

```
df_gb_vrz_counts =
   df_gb_vrz.groupby(['VEHICLE_HAL_ID']).size()
   .reset_index().rename(columns={0:'countOfRentalZones'})
print(df_gb_vrz_counts.sort_values('countOfRentalZones'))
```

VEHICLE_HAL_ID	countOfRentalZones
143031	1
161378	1
...	...
170607	4
148879	5
150420	5
...	...
161334	5
157805	6
148879	5
146243	6
151073	6
161595	7
...	...

Durch die Aggregierung der Inhalte wird deutlich, dass die überwiegende Mehrheit der Fahrzeuge nur mit einer Station in Beziehung steht:

```
print(df_gb_vrz_counts.countOfRentalZones.agg(
   ['min', 'max', 'median']
))
```

```
min 1.0
max 7.0
median 1.0
Name: countOfRentalZones, dtype: float64
```

Diese Erkenntnis kann auch durch das Gruppieren der Anzahl der unterschiedlichen Stationen gewonnen werden:

```
df_gb_vrz_diversity =
   df_gb_vrz_counts.groupby(['countOfRentalZones'])
      .size()
      .reset_index().rename(columns={0:'occurrence'})
print(df_gb_vrz_diversity.sort_values(
      'occurrence',ascending=False))
```

countOfRentalZones	occurrence
1	1201
2	337
3	87
4	25
5	10
6	3
7	2

Aufgrund dieser Erkentnisse wird eine einzige Beziehung ("WAS_BOOKED_IN") zwischen einem Fahrzeug und einer Station gebildet:

```
df_gb_vrz['REL_TYPE_VEHICLE_RENTAL_ZONE'] =
df_gb_vrz.apply(getRelationshipType, axis=1,
   types=["VEHICLE", "RENTAL_ZONE"])
```

Als Analyse-Ergebnis der Beziehungen kommen drei Dateien zustande (Beziehungen der Buchung zu den Fahrzeugen und Tarifklassen sowie die Beziehungen zwischen den Fahrzeugen und Stationen):

```
writeDsvFile(df_lj_booking_cat
   [df_lj_booking_cat
      ["VEHICLE_MANUFACTURER_NAME"]
         != "UNKNOWN"],
   'rel_booking_vehicle', defaultCsvItemDelimiter,
   ['BOOKING_HAL_ID', 'VEHICLE_HAL_ID',
      'REL_TYPE_BOOKING_VEHICLE'],
   [':START_ID(BOOKING-ID)',
      ':END_ID(VEHICLE-ID)', ':TYPE'])
writeDsvFile(df_lj_booking_cat[
   df_lj_booking_cat["CATEGORY"] != 'UNDEFINED'],
   'rel_booking_category', defaultCsvItemDelimiter,
   ['BOOKING_HAL_ID', 'CATEGORY_HAL_ID',
      'REL_TYPE_BOOKING_CATEGORY'],
   [':START_ID(BOOKING-ID)','
      :END_ID(CATEGORY-ID)', ':TYPE'])
writeDsvFile(df_gb_vrz,
   'rel_vehicle_rental_zone', defaultCsvItemDelimiter,
   ['VEHICLE_HAL_ID', 'count',
         'RENTAL_ZONE_HAL_ID_RIGHT',
         'REL_TYPE_VEHICLE_RENTAL_ZONE'],
   [':START_ID(VEHICLE-ID)', 'times',
         ':END_ID(RENTAL-ZONE-ID)', ':TYPE'])
```

Bei den Eigenschaften einer Buchung wird wie üblich zunächst nach den Spalten gesucht, die Null-Werte aufweisen:

```
DISTANCE 201
TECHNICAL_INCOME_CHANNEL 51976
dtype: int64
```

Für diese beide Spalten werden die Null-Werte nun mit Default-Werten belegt, damit sie für eine Label-Bildung verwendet werden können:

```
df_booking[['DISTANCE']] =
   df_booking[['DISTANCE']].fillna(value=0)
df_booking[['TECHNICAL_INCOME_CHANNEL']] =
df_booking[['TECHNICAL_INCOME_CHANNEL']]
      .fillna('UNDEFINED', axis=1)
```

Für das Bilden der Label-Informationen ist es noch relevant, was für Werte als technische Buchungsquellen vorkommen:

```
print(sorted(
    df_booking.TECHNICAL_INCOME_CHANNEL.unique())
)
```

['API', 'Bahn_de_2', 'Book-n-Drive Android', 'Book-n-Drive iPhone', 'Broker HAL', 'BwCarsharing Android', 'BwCarsharing WindowsPhone', 'BwCarsharing iPhone', 'BwFPS Dispo Testzugang', 'BwFPS Dispotool', 'BwFPS Portal Web', 'EMIL Carsharing', 'Flinkster - Mobility Map', 'Flinkster Android', 'Flinkster Carjump', 'Flinkster Connect', 'Flinkster Drive Carsharing', 'Flinkster E-Wald', 'Flinkster Windows', 'Flinkster iPhone', 'Ford Carsharing FordPass', 'Ford2Go Web', 'HALAPI Teilauto', 'ICS-Server', 'Internet', 'Multicity Android', 'Multicity iPhone', 'Onesto_Bahn', 'Scouter 255 Android 2Denker', 'Scouter 255 Web Praegnanz', 'Scouter 255 Windows 2Denker', 'Scouter 255 iOS 2Denker', 'Stuttgart Service Card', 'UNDEFINED', 'einfachMobil Android', 'einfachMobil iPhone', 'ford2go Android', 'ford2go iPhone', 'teilAuto']

Beim genauen Betrachten dieser Werte wird erkennbar, dass sie überwiegend aus zwei Informationen bestehen: Quelle und Interface der Quelle wie z. B. bei Flinkster - Mobility Map Flinkster als Quelle und Mobility Map als Interface zu Flinkster. In der zweiten Stufe der Auseinandersetzung mit diesen Informationen stellt man fest, dass die Interfaces zu den einzelnen Quellen ebenfalls eine gröbere Klassifizierung hergeben: Interfaces wie iPhone, Android oder WindowsPhone deuten auf eine App hin und andere wie Portal Web und E-Wald auf eine Website. Auf Basis dieser Erkentnisse werden zwei neue Eigenschaften einer Buchung gebildet: INCOME_CHANNEL_GROUP (oben erwähnt als Quelle) und INCOME_CHANNEL_TYPE (Klassifizierung des Interfaces):

```
def getTechnicalIncomeChannelType(row):
    channelToType = {
        "Internet": "Internet",
        ...
        "Stuttgart Service Card": "Mobility"
    }
    ctype = channelToType.get(
        str(row["TECHNICAL_INCOME_CHANNEL"]),
        "UNDEFINED").upper()
    return ctype
def getTechnicalIncomeChannelGroup(row):
    channelToGroup = {
        "API": "DB",
        ...
        "Book-n-Drive iPhone": "Book-n-Drive",
        "Broker HAL": "DB",
        "BwCarsharing Android": "BwCarsharing",
        ...
        "ford2go iPhone": "ford2go",
        "teilAuto": "Teilauto"
    }
    group = channelToGroup.get(
        str(row["TECHNICAL_INCOME_CHANNEL"]),
        "UNDEFINED").upper()
    return group
...
df_booking['INCOME_CHANNEL_TYPE'] =
    df_booking.apply(
        getTechnicalIncomeChannelType, axis=1)
df_booking['INCOME_CHANNEL_GROUP'] =
    df_booking.apply(
    getTechnicalIncomeChannelGroup, axis=1)
```

Aufgrund der vorkommenden Werte bei der Distanz und bei den Flags in einer Buchung sind ebenfalls Anpassungen notwendig, damit die Daten korrekt/geeignet übertragen werden:

```
print(df_booking.DISTANCE.unique())
print(sorted(
    df_booking.COMPUTE_EXTRA_BOOKING_FEE.unique()
))
print(sorted(df_booking.TRAVERSE_USE.unique()))
```

[14. 84. 1036. ..., 2057. 1686. 1737.] ['Ja', 'Nein']
['Ja', 'Nein']

Bei der Distanz wird eine Distanz-Kategorie gebildet, wodurch zwischen Kurz-, Mittel- und Langstrecken-Buchungen unterschieden werden soll:

```
def getDistanceCategory(row):
    distance = float(row["DISTANCE"])
    distanceCategory = 'SHORT_DISTANCE'
    if 500 > distance > 100:
        distanceCategory = 'MIDDLE_DISTANCE'
    elif distance > 500:
        distanceCategory = 'LONG_DISTANCE'
    return distanceCategory
...
df_booking['DISTANCE_CATEGORY'] = df_booking.
apply(getDistanceCategory, axis=1)
```

Außerdem werden die beiden Flags so umgewandelt, dass sie als Boolean übertragen werden können:

```
df_booking['TRAVERSE_USE'] =
    df_booking.apply(
    {'TRAVERSE_USE' : lambda x: "true" if str(x).upper() ==
        "JA" else "false"}
)
df_booking['COMPUTE_EXTRA_BOOKING_FEE'] =
    df_booking.apply(
    {'COMPUTE_EXTRA_BOOKING_FEE' : lambda x:
    "true" if str(x).upper() == "JA" else "false"}
)
```

Abschließend wird bei den Buchungen das Label auf Basis der ausgewählten Informationen aus der Buchung sowie aus anderen Entitäten gebildet:

```
def getLabel(row, nodeName):
    switcher = {
        ...
        "BOOKING": "BOOKING",
        ...
    }
    label = switcher.get(nodeName, 'OBJ')
    if nodeName == 'BOOKING':
        label += ";" + str(row["PARENT_CATEGORY"])
        label += ";" + str(row["INCOME_CHANNEL_TYPE"])
        label += ";" + str(
            row["INCOME_CHANNEL_GROUP"])
        label += ";" + str(
            row["VEHICLE_MANUFACTURER_NAME"])
        label += ";" + str(row["DISTANCE_CATEGORY"])
    ...
    return label
...
df_booking_final['LABEL'] =
    df_booking_final.apply(getLabel, axis=1,
        nodeName='BOOKING')
```

Gemeinsam mit anderen inhaltlichen Spalten werden dann die Label- und ID-Informationen als Import-Datei-Inhalt ausgegeben:

```
writeDsvFile(df_booking_final, 'bookings', ',',
    ['BOOKING_HAL_ID', 'DATE_BOOKING',
    'DATE_FROM', 'DATE_UNTIL', 'COMPUTE_EXTRA_
    BOOKING_FEE', 'TRAVERSE_USE', 'DISTANCE',
    'START_RENTAL_ZONE', 'END_RENTAL_ZONE',
    'CITY_RENTAL_ZONE', 'TECHNICAL_INCOME_
    CHANNEL', 'LABEL'],
    ['bookingID:ID(BOOKING-ID)','bookingDate','startDate',
    'endDate','computeExtraBookingFee:boolean',
    'traverseUse:boolean','distance:float','startRentalZone',
    'endRentalZone','cityRentalZone',
    'technicalIncomeChannel',':LABEL'])
```

Importieren

Auf Basis der entstandenen Daten ist es nun möglich - ohne referentiell- oder inhaltlich-bedingte Fehler - die Daten in die Graph-DB Neo4J zu importieren:

```
/Applications/neo4j-community-3.2.3/bin/neo4j-admin import
--mode csv --database bahnOpenDataCS.graph.db --nodes
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/bookings.dsv -- nodes
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/parent_categori es.dsv --nodes
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/categories.dsv --nodes
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rentalZones.dsv --nodes
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/vehicles.dsv --relationships
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_booking_vehicle.dsv --relationships
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_booking_cat egory.dsv --relationships
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_cat_pcat.dsv --relationships
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_vehicle_rental_zone.dsv --ignore-extra-
columns=true --ignore-duplicate-nodes=true --ignore-
missing-nodes=true --id-type=INTEGER
```

Ein wichtiger Hinweis ist hierbei: In dem Moment, wo in den folgenden Punkten eine von den Standardwerten abweichende Einstellung beim Importieren der Daten notwendig wird, ist es zum aktuellen Zeitpunkt erforderlich, auf das Skript neo4j-import (in Neo4J enthalten) umzusteigen:

- Anderes Trennzeichen als ','
- Anderes Trennzeichen als ';' bei Arrays
- Anderes Zeichen als '"' bei Text-Inhalten
- Limit-Vergabe für inkorrekte Einträge (ungültige Beziehungen, Duplikate)

Der Grund dafür ist, dass in der neuen Version von Neo4J (3.2.*), das alte Skript neo4j-import hinter der neuen Schnittstelle neo4j-admin zwar weiterhin verwendet wird, aber dabei die erweiterte Parametrisierung nicht berücksichtigt wird.

Wenn auf Basis des oben dargestellten Skript-Aufrufs das Importieren angestoßen wird, läuft das Skript erfolgreich durch und die Knoten und Beziehungen werden erfolgreich importiert:

```
Neo4j version: 3.2.3
Importing the contents of these files into /Applications/
   neo4j-community-3.
2.3/data/databases/bahnOpenDataCS.graph.db:
Nodes:
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/bookings.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/parent_categories.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/categories.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rentalZones.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/vehicles.dsv
Relationships:
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_booking_vehicle.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_booking_category.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_cat_pcat.dsv
/Users/ilker/Workspaces/jupyter/DB_OpenData_To_Neo4J/
output/rel_vehicle_rental_zone.dsv
...
Nodes, started 2017-09-10 14:19:50.903+0000 [>:21|NOD
E:7,63|*PROPERTIES(2)==================
=|LABEL SCAN------------ |v:40,03 MB/s(2)===] 551K Δ
376K
Done in 5s 612ms

Prepare node index, started 2017-09-10 14:19:56.584+0000
[*DETECT:14,98 MB-----------------------------------------------
---------- -------------------] 550K Δ 550K
Done in 638ms

Relationships, started 2017-09-10 14:19:57.233+0000
[>:??---------------|TYPE|*PREPARE(3)=============
=================|REC|CAL CU|PR|v:37,87 MB/s-]
1.09M Δ1.09M
Done in 1s 61ms

Relationship --> Relationship 1/4, started 2017-09-10
14:19:58.415+0000
 [>--------------------------------|*LINK----------------------------
--|v: ??(2)==============]1.09M Δ1.09M
Done in 285ms

...
Node --> Group, started 2017-09-10 14:20:04.093+0000
[>---------|FIRST-------------------------------
|*v:??------------------- -------------------]1.63K Δ1.63K
Done in 13ms

Node counts, started 2017-09-10 14:20:04.183+0000
[>(8)|*COUNT:76,29 MB-----------------------------------------
------------ -------------------] 560K Δ 560K
Done in 148ms

Relationship counts, started 2017-09-10
14:20:04.371+0000 [>----------------------------------|*COU
NT-----------------------------
-------------------] 1.1M
Done in 191ms
```

```
IMPORT DONE in 14s 352ms. Imported:
  550514 nodes
  1093376 relationships
  6054582 properties
Δ 1.1M
Peak memory usage: 531,22 MB
end of data load
Start neo4j
Active database: bahnOpenDataCS.graph.db
Directories in use:
home:
config:
logs:
plugins:
import:
data:
certificates: /Applications/neo4j-community-3.2.3/certificates run:
/Applications/neo4j-community-3.2.3/run
/Applications/neo4j-community-3.2.3
/Applications/neo4j-community-3.2.3/conf
/Applications/neo4j-community-3.2.3/logs
/Applications/neo4j-community-3.2.3/plugins
NOT SET
/Applications/neo4j-community-3.2.3/data
   Starting Neo4j.
```

Anschließend ist es z. B. möglich die unterschiedlichen Kombinationen zwischen Buchungs-, Fahrzeug- und Tarifvarianten mithilfe der APOC-Prozedur apoc.cypher.run [13] herauszufinden:

```
MATCH (b:BOOKING) with distinct labels(b) as
bookingLabels
WITH reduce(combinedLabels = '', n IN bookingLabels|
combinedLabels + ':' + n) AS fullLabel
CALL apoc.cypher.run("match ("+fullLabel+") return
count(*) as count", null ) yield value
return fullLabel, value.count as count order by count;

! "fullLabel" !
   !! "count" !!
! ":BOOKING:BWCARSHARING:VW:SHORT_
DISTANCE:KOMPAKTKLASSE:APP" !
   !! 1 !!

!":BOOKING:MIDDLE_
DISTANCE:MINI:UNDEFINED:OPEL"
   !! 2 !!

!":BOOKING:LONG_DISTANCE:UNDEFINED:OPEL:
TRANSPORT"
   !! 2 !!

!":BOOKING:MIDDLE_DISTANCE:MINI:UNDEFINED:
CITROËN"
   !! 2 !!
```

```
!":BOOKING:MIDDLE_DISTANCE:TRANSPORT:APP:
MERCEDES:BWCARSHARING"
   !! 3 !!

!":BOOKING:RENTALZONE:DB:SHORT_
DISTANCE:VAN_BUSKLASSE:MERCEDES"
   !! 3 !!

!":BOOKING:NISSAN:INTERNET:DB:MITTELKLASSE:
SHORT_DISTANCE"
   !! 874 !!

!":BOOKING:MINI:FIAT:SHORT_DISTANCE:APP:
FLINKSTER"
   !! 958 !!

!":BOOKING:MIDDLE_DISTANCE:KLEINKLASSE:
OPEL:APP:FLINKSTER"
   !! 1004 !!

!":BOOKING:NISSAN:INTERNET:DB:KLEINKLASSE:
SHORT_DISTANCE"
   !! 1023 !!

!":BOOKING:VW:INTERNET:DB:SHORT_DISTANCE:
TRANSPORT"
   !! 1580 !!

!":BOOKING:INTERNET:DB:KLEINKLASSE:SHORT_
DISTANCE:PEUGEOT"
   !! 1680 !!

!":BOOKING:OPEL:MIDDLE_DISTANCE:UNDEFINED"
   !! 1681 !!

!":BOOKING:SONDERKLASSE:CITROËN:SHORT_
DISTANCE:APP:FLINKSTER"
   !! 42220 !!

!":BOOKING:FLINKSTER:KLEINKLASSE:SHORT_
DISTANCE:APP:FORD"
   !! 48682 !!

!":BOOKING:INTERNET:DB:KLEINKLASSE:SHORT_
DISTANCE:FORD"
   !! 62416 !!

!":BOOKING:SONDERKLASSE:INTERNET:DB:
CITROËN:SHORT_DISTANCE"
   !! 92438 !!
```

Was bei den Ergebnissen besonders ins Auge sticht ist folgendes: Die Labels aus unterschiedlichen Quellen (Buchung/Fahrzeug/Tarfiklasse) werden nicht in der importierten Reihenfolge zurückgegeben. Woran liegt dies? Kann man an diesem Punkt etwas verbessern? Diese und andere Punkte wie Indizierung wären ein Thema für einen anderen Artikel.

Referenzen

[1] Neo4J,
http://neo4j.com

[2] OpenKnowledge,
https://okfn.org
https://okfn.de

[3] OpenData *Portal DB*,
http://data.deutschebahn.com
http://data.deutschebahn.com/dataset/data-flinkster,

[4] *Dokumentation Flinkster-Daten*,
http://download- data.deutschebahn.com/static/datasets/flinkster/20170516/20171605%20DokumentationDatenOpenData.pdf

[5] Jupyter,
http://jupyter.org

[6] Anaconda,
https://www.anaconda.com

[7] Pandas,
https://pandas.pydata.org

[8] NumPy,
http://www.numpy.org

[9] *Functools*,
https://docs.python.org/3/library/functools.html,

[10] GitHub *Github-Projekt*,
https://github.com/lkmsk/OpenDataToNeo4J/tree/master/DB_OpenData

[11] Blog Mark Needham,
http://www.markhneedham.com/blog/2017/07/05/pandas-find-rows-where-columnfield-is-null

[12] neo4j Import-Tool Neo4J,
https://neo4j.com/docs/operations-manual/current/tutorial/import-tool

[13] APOC,
https://neo4j-contrib.github.io/neo4j-apoc-procedures

Kurzbiografie

Ilker Yümsek arbeitet als Software-Architekt bei MATHEMA Software GmbH und unterstützt Kunden bei der Konzeption und Realisierung von Software-Lösungen im Enterprise-Umfeld. Sein Schwerpunkt liegt bei Java-basierten Lösungen im Bereich der Systemintegration und Enterprise-Anwendungen. Er legt großen Wert auf eine zielführende und konstruktive Kommunikation als Basis einer guten Teamarbeit sowie hochqualitativer Ergebnisse.

10 Jahre Herbstcampus

Ein Rückblick auf die Jubiläumsveranstaltung

von Oliver Klosa

Der Herbstcampus öffnete im September wieder seine Tore – mittlerweile tatsächlich schon zum zehnten Mal! Grund genug, auf die Jubiläumsveranstaltung vom 5. bis 7. September an der TH Nürnberg zurückzublicken.

Den Beginn machten am Dienstag die gutbesuchten Tutorien. Es gab sechs ganz unterschiedliche Themen, von denen jedes einzelne eine starke Resonanz erfuhr. Die Teilnehmer konnten sich nicht zuletzt anhand vieler praktischer Übungen in zwei verschiedene Java-Themen, Domain-Driven-Design, Docker, NativeScript und in das aktuelle Trendthema Deep Learning einarbeiten und weiterbilden.

Dass der Herbstcampus auch in diesem Jahr an den Konferenztagen wieder gut besucht war, zeigten u. a. die beiden Keynotes. Carola Lilienthal befasste sich mit den kognitiven Mechanismen unseres Gehirns im Zusammenhang mit dem Entwurf passender Software-Architekturen und Lina Böcker referierte über rechtliche Hintergründe, Lizenztypen und Compliance-Themen in Verbindung mit Open-Source- und freier Software.

Alle anderen Vorträge rund um die Themen Java, Programmiersprachen, Cloud, Sicherheit, Mobile

Entwicklung, Microservices und vielem mehr fanden interessiertes Publikum. Für jeden technologischen Geschmack war etwas dabei!

Guten Geschmack erwies ebenfalls das Catering-Team, das die unterschiedlichsten Köstlichkeiten in den Mittags- und Kaffeepausen servierte. Die Pausen, insbesondere die zwischen den Vorträgen, sind etwas länger als bei vielen anderen Konferenzen, und wurden von den Teilnehmern ausgiebig zum Netzwerken und zum Wissensaustausch genutzt. Dazu diente ebenfalls das abendliche Get-together. In gemütlicher Runde und bei heimischen Bierspezialitäten oder vorzüglichen Cocktails an der Fahrbar plauderten die noch Anwesenden bis in den späten Abend hinein, sowohl über technische als auch nicht technische Themen. Die Sponsoren und Aussteller wie adesso, XebiaLabs, codecentric, thecodecampus, ISO Gruppe und HANSER rundeten das Konferenzkonzept ab.

Die Veranstalter dpunkt.verlag, heise Developer, iX und MATHEMA waren auch in diesem Jahr sehr zufrieden mit der Organisation und dem Ablauf der Konferenz und blickten auf eine sehr gelungene Jubiläumsveranstaltung zurück, die ihren familiären Touch und Charme beibehalten hat und im Herbst 2018 vom 4. bis 6. September ihre Fortsetzung finden wird.

Um sich einen Eindruck von den diesjährigen Konferenztagen sowie dem Tutorientag zu verschaffen, können Sie sich die Bildergalerien auf der Flickr- oder der Facebook-Seite der MATHEMA Software GmbH ansehen.

- Flickr:
  https://www.flickr.com/photos/155286834@N04/sets/72157689074919095
- Facebook:
  https://www.facebook.com/pg/mathema.software.gmbh/photos/?tab=album&album_id=1938355119823506

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch? Dann geben Sie uns doch bitte Bescheid.
Bookware, Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0, Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
Kontakt: Oliver B. Fischer
(o.b.fischer@swe-blog.net)
http://jug-berlin-brandenburg.de/kontakt.html

Java UserGroup Bremen
Kontakt: Rabea Gransberger (rgransberger@gmx.de)
http://www.jugbremen.de

JUG DA
Java User Group Darmstadt
Kontakt: jug-da-orga@googlegroups.com
http://www.jug-da.de

Java User Group Saxony
Java User Group Dresden
Kontakt: Falk Hartmann
(falk.hartmann@jugsaxony.org)
http://www.jugsaxony.de

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
Kontakt: Heiko Sippel (info@rheinjug.de)
http://www.rheinjug.de

JUG Deutschland e.V.
Java User Group Deutschland e.V.
Kontakt: Stefan Koospal (office@java.de)
http://www.java.de

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
Kontakt: Heiko Sippel (heiko.sippel@ruhrjug.de)
http://www.ruhrjug.de

JUGF
Java User Group Frankfurt
Kontakt: Alexander Culum (alexander.culum@web.de)
http://www.jugf.de

JUG Görlitz
Java User Group Görlitz
Kontakt: (kontakt@jug-gr.de)
http://www.jug-gr.de

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
Kontakt: (jugkarlsruhe@gmail.com)
http://jug-karlsruhe.de

JUGC
Java User Group Köln
Kontakt: Michael Hüttermann
(michael@huettermann.net)
http://www.jugcologne.org

jugm
Java User Group München
Kontakt: Andreas Haug (ah@jugm.de)
http://www.jugm.de

JUG Münster
Java User Group für Münster und das Münsterland
Kontakt: Thomas Kruse (tkjugi@sforce.org)
http://www.jug-muenster.de

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
Kontakt: (info@jug-n.de)
http://www.jug-n.de

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V. , c/o Dr. Michael Paus
Kontakt: Dr. Micheal Paus (mp@jugs.org),
Hagen Stanek (hs@jugs.org),
Rainer Anglett (ra@jugs.org)
http://www.jugs.org

Schweiz

JUGS
Java User Group Switzerland
http://www.jug.ch (info@jug.ch)

.NET User Groups

Deutschland

.NET User Group Bonn
.NET User Group "Bonn-to-Code.Net"
Kontakt: Roland Weigelt (mail@bonn-to-code.net)
http://www.bonn-to-code.net

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
Kontakt: Udo Neßhöver, Ulrike Stirnweiß
(info@dodnedder.de)
http://www.dodnedder.de

.NET UserGroup Frankfurt
http://www.dotnet-usergroup.de

.NET User Group Friedrichshafen
Kontakt: Tobias Allweier (info@dotnet-fn.de)
http://www.dotnet-fn.de

.NET User Group Hannover
http://www.dnug-hannover.de
Kontakt: (dnug@indisoftware.de)

INdotNET
Ingolstädter .NET Developers Group
Kontakt: Gregor Biswanger
(gregor.biswanger@web-enliven.de)
http://www.indot.net

DNUG-Köln
DotNetUserGroup Köln
Kontakt: Albert Weinert (info@der-albert.com)
http://www.dnug-koeln.de

.NET User Group Leipzig
Kontakt: Alexander Groß (agross@dotnet-leipzig.de)
Torsten Weber (tweber@dotnet-leipzig.de)
http://www.dotnet-leipzig.de

.NET Developers Group München
Kontakt: Hardy Erlinger (hardy_erlinger@hotmail.com)
http://www.munichdot.net

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
Hilmar Bunjes (hilmar.bunjes@dotnet-oldenburg.de)
Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)
http://www.dotnet-oldenburg.de

.NET Developers Group Stuttgart
Kontakt: Michael Niethammer
(GroupLeader@devgroup-stuttgart.net)
http://www.devgroup-stuttgart.net

.NET Developer-Group Ulm
c/o artiso solutions GmbH
Kontakt: Thomas Schissler (tschissler@artiso.com)
http://www.dotnet-ulm.de

Österreich

.NET User Group Austria
c/o Global Knowledge Network GmbH,
Kontakt: Christian Nagel (ug@christiannagel.com)
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx

Software Craftsmanship Communities

Deutschland, Schweiz, Österreich

Softwerkskammer – Mehrere regionale Gruppen und Themengruppen unter einem Dach
Kontakt: Nicole Rauch (nicole.m@gmx.de)
http://www.softwerkskammer.org

MATHEMA

Global Day of Coderetreat

Am 18. November 2017 findet weltweit der Global Day of Coderetreat statt und wir machen wieder mit! Sei auch du dabei, wenn wir uns ohne Stress und mit viel Spaß auf die wesentlichen Dinge des Programmierens fokussieren!

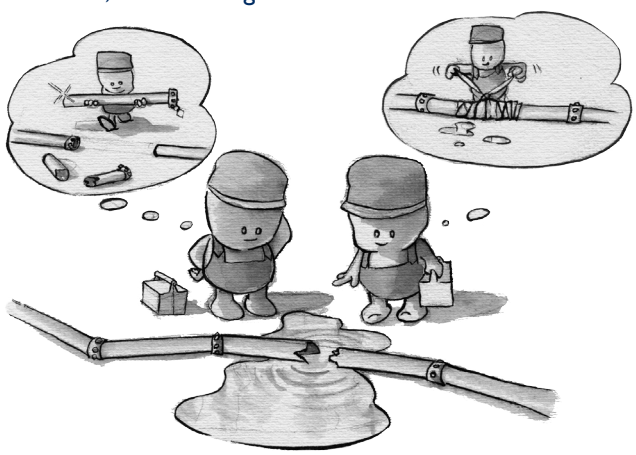
Die Agenda und alle weiteren Infos findet ihr auf unserer Website: https://www.mathema.de/veranstaltungen/global-day-of-coderetreat-2017.

Die Teilnahme ist kostenlos, aber nicht umsonst! Für das leibliche Wohl sorgt MATHEMA. Da die Teilnehmerzahl begrenzt ist, meldet euch bitte bald unter info@mathema.de an. First come, first served! Wir freuen uns auf euch!

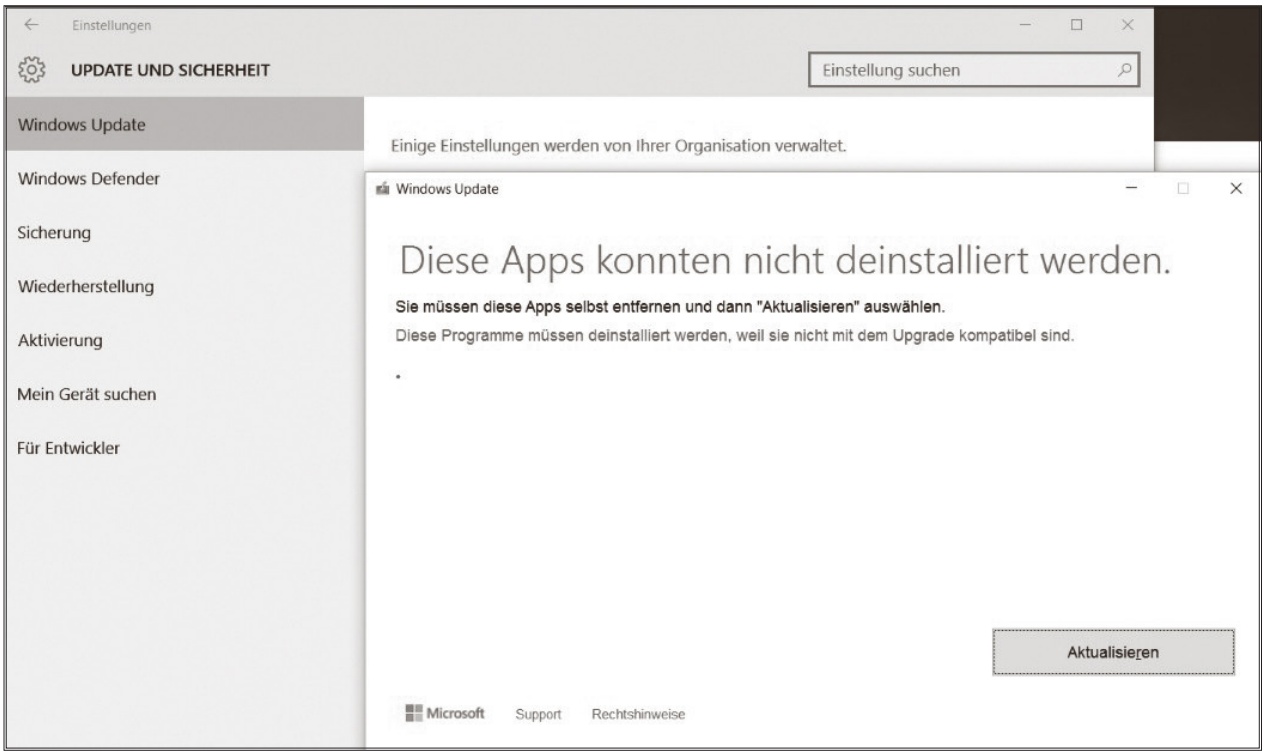
Hintergrundinformationen zum Global Day of Coderetreat findet ihr unter: http://globalday.coderetreat.org

Zeit und Ort

Samstag, 18. November 2017, 8:30 bis 17:00 Uhr,
MATHEMA Software GmbH, Haus 8,
Henkestraße 91, 91052 Erlangen



Das Allerletzte



Dies ist kein Scherz!
Diese Meldung wurde tatsächlich in der freien
Wildbahn angetroffen.
Ist Ihnen auch schon einmal ein Exemplar
dieser Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im Oktober.

Herbstcampus

Wissenstransfer par excellence

4. – 6. September 2018
in Nürnberg