050033 PR

Praktikum Medizinische Informationssysteme

*Benedek Gal – a1268262*

Universität Wien, Informatik

**Abstract.** Das Ziel von Praktikum ist die Erkenntnis von Fhir Open Server Implementationen, die Erstellung einer Fhir-Umgebung, das Testen der Umgebung mit Random-Daten am Windows 7.

**Keywords:** Fhirbase, Fhir Hapi, Server, Client, RESTful, API, Ressource, Patient, Encounter, Location

1. Einleitung

Fhir ist die nächste Generation von Standard Framework von HL7 Organisation. Es kombiniert der Beste Merkmalen von HL7 Version 2, Version 3 und CDA Produkte mit den neuesten Web Standards mit starkem Fokus an Umsetzbarkeit.

1. **Ursprüngliches Ziel**:

Populationsauswertungen machen (Aggregierte Information sammeln) mithilfe eine der Fhir-Implementierungen. Krankenhausaufenthalt der Bevölkerung zu visualisieren.

1. **Ziel des Praktikums:**

* eine der Fhir-Entwicklungsumgebungen wählen, aufsetzen, kennenlernen,
* FHIR-Server aufbauen, mit Testdaten befüllen,
* Ressourcen wählen: **Patient** (Altersgruppe: 0-14, 15-29, 30-44, 45-59, 60-74, 75-89, Geschlecht), Encounter, Krankenhaus, **Bundesland** (9), Bezirk

1. Fhirbase in Docker

Fhirbase ist an PostgreSQL aufgebaute Extension und benötigt PostgreSQL v9.4 oder höher. Erstes Mal beschäftigte ich mich mit Fhirbase unter Docker, leider konnte ich die Umgebung mit der Anleitung an Webseite dennoch nicht zum Laufen bringen.

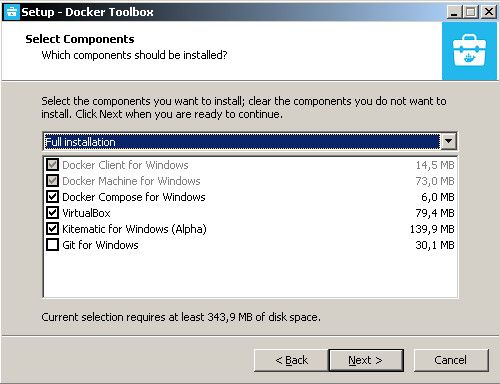
**2.1 Installation von Docker am Windows 7**

Ich versuchte die Umgebung durch Docker zu installieren. Das ist schon in Win7 bisschen kompliziert.

Die offizielle Anleitung sagt, dass man am Win7 Microsoft Hyper-V braucht. Falls man diese Richtung wählt dann gelangt man zu einem Streit ob Win7 tatsächlich Hyper-V nutzen oder nicht nutzen kann. Nach ausgiebigen Recherchen am Google entschied ich mich nicht an Hyper-V zu suchen, weil Docker Packet sowieso VirtualBox beinhaltet. Man arbeitet entweder mit Linux oder bleibt beim Win7 und sucht Umwege.

Docker ist nicht einfach am Windows zu nutzen. Docker ist an Linux geschrieben [[1]](#footnote-1). Im Endeffekt braucht es also ein Linux System zu laufen. Am Windows Systems geht es leicht mit Linux VM. Man meldet sich an dem Virtual Maschine an und nutzt Docker als man am Linux arbeitete.

**2.1 Installation von Docker Toolbox**

Die offizielle Anleitung für Win7 bietet ein Docker Toolbox an VirtualBox aber nachdem Hochfahren des Servers konnte ich nicht Zugang zu Sever aufbauen.

Die Anleitung von GitHub ist klar formuliert, wurde durchgemacht, aber in Praxis der Zugang zum Server ist nicht gelungen.

Ich versuchte mithilfe Kitematic, die am Docker Server stehende offizielle Images zu starten. Die funktionierten, aber den selbst aufgebaute Fhirbase Server konnte ich nicht kontaktieren.

Was den Aufbau eines Servers angeht, gibt es viele Wege, aber im Endeffekt ging was immer schief.

1. Fhirbase in Vagrant - Installation

Für Fhirbase aufzubauen bietet sich noch die Möglichkeit mithilfe Vagrant und Virtualbox zu starten. Um die Umgebung von Fhirbase installieren zu können, braucht man die folgenden Anwendungen zu installieren.:

* Git
* VirtualBox v5.1.4
* Vagrant v1.8.5
* PostgreSQL v9.5

Die Reihe der Installation ist nicht von Bedeutung. Alle obigen Anwendungen kommt mit ausführbare Installation-file. Die Installation deswegen schnell und einfach.

**3.1 Fhirbase-plv8 Settings-änderung**

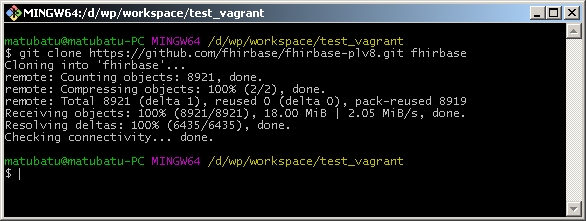
Die vollständigen Installationsschritte werden im nächsten Absatz detailliert geschrieben. Der Unterschied hier ist nur die Beschaffung von Fhirbase, die jetzt mithilfe Git geklont wird.

Git-Bash starten im Workspace wohin Fhirbase-plv8 herunterladen wird.

1. Fhirbase von GitHub klonen:

*git clone https://github.com/fhirbase/fhirbase-plv8.git fhirbase*

Oder einfach von GitHub als ZIP herunterladen.



1. In *fhirbase/vagrant/provision* Folder: beide Datei öffnen und die folgende Änderungen vornehmen:



* **provision-dependencies.sh bearbeiten:**

Die letzte Zeile: *pgxn install plv8 || exit 1* mit Folgenden ersetzen:

*pgxn install plv8=1.4.4 || exit 1*

* **provision-** **environment.sh bearbeiten:**

die folgenden Zeilen mit # auskommentieren:

*# echo "curl --location* [*https://github.com/fhirbase/fhirbase-plv8/releases/download/v1.4.0.0/fhirbase-*](https://github.com/fhirbase/fhirbase-plv8/releases/download/v1.4.0.0/fhirbase-)

*1.4.0.0.sql.zip| funzip | psql -d fhir" \*

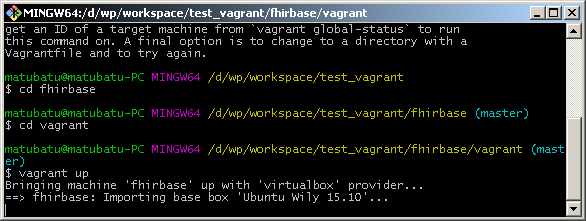
*# | sudo -u postgres sh || exit 1*

Falls man diese Bearbeitung nicht macht, kommt zu einem Error beim Starten des Virtual Machines.

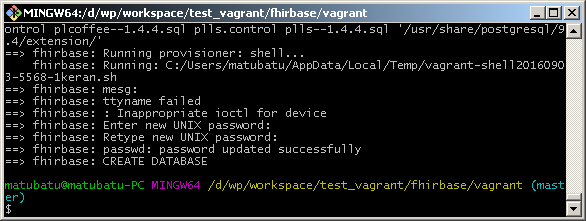
Das kann man leicht beheben nach der Änderung mit *vagrant provision* Befehl.

3. In Git-Konsol fhirbase/vagrant Folder öffnen, wo das Vagrantfile ist:

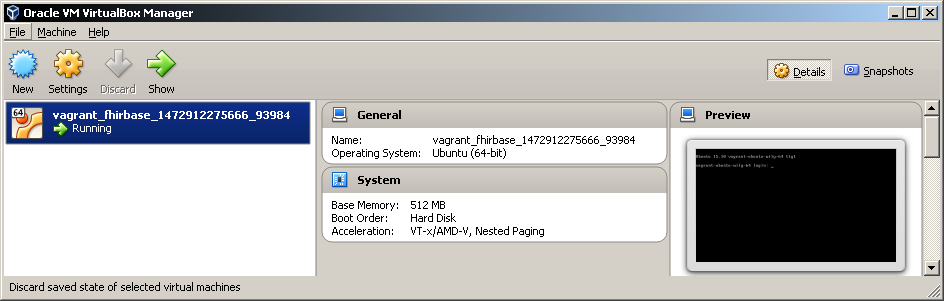
*vagrant up*



4. Eine Ubuntu Virtuelle Maschine wird in VirtualBox erzeugt, an dem ein PostrgreSQL Datenbank mit allen Abhängigkeiten zur Verfügung steht.

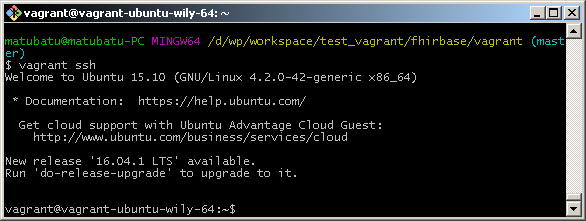


In VirtualBox Manager sieht man auch die Ubuntu Virtuelle Maschine:



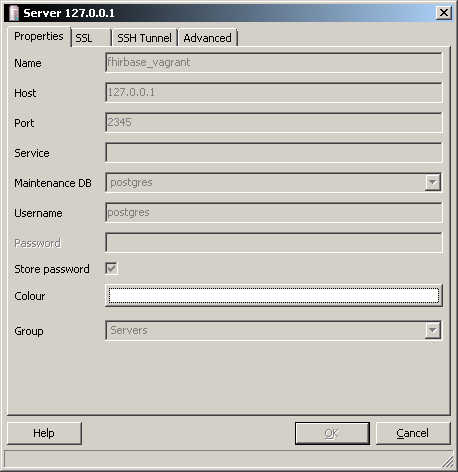
5. Mit *vagrant ssh Befehl:*

SSH-Konsol von Virtuelle-Maschine wird gestartet.



Fhirbase ist ein PostgreSQL Extension für Speicherung und Abfragung von FHIR-Ressourcen. Das Interaktion mit Fhirbase ist mit pgAdmin III der einfachste.

6. pgAdmin III öffnen und ein neuer Server schaffen:

Danach kann man den Zugang zu Server testen.

Die Datenbank aufzufüllen, nutzt man am besten ein Konsol Fester, wo die Tabellen von Fhir mithilfe Script-dateien eingelesen wird.

7. Die SQL-Script-Dateien können von https://mias.meduniwien.ac.at/owncloud/index.php/s/ict8kElMfxmC8pv heruntergeladen werden.   
Passwort: *fhirbase*.

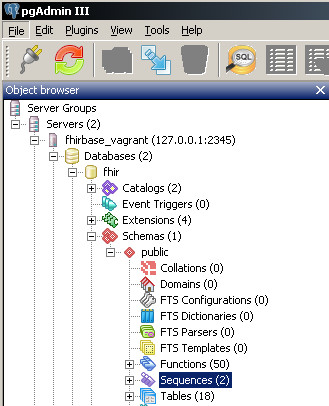
8. Konsol öffnen in *C:\Program Files\PostgreSQL\9.5\bin>*

Folgender Befehl eingeben:

*psql -d fhir -h 127.0.0.1 -p 2345 -U postgres*

Bekommt man jetzt die Fhir-DB Prompt wo der Path der jeweiligen SQL-Scriptfile eingegeben wird:

Die drei Script-Dateien wurden ins Scripts Ordner kopiert:

*\i 'c:/Program Files/PostgreSQL/9.5/scripts/fhirbase-1.4.0.0\_c\_utf\_1.sql'*

*\i 'c:/Program Files/PostgreSQL/9.5/scripts/fhirbase-1.4.0.0\_c\_utf\_2.sql'*

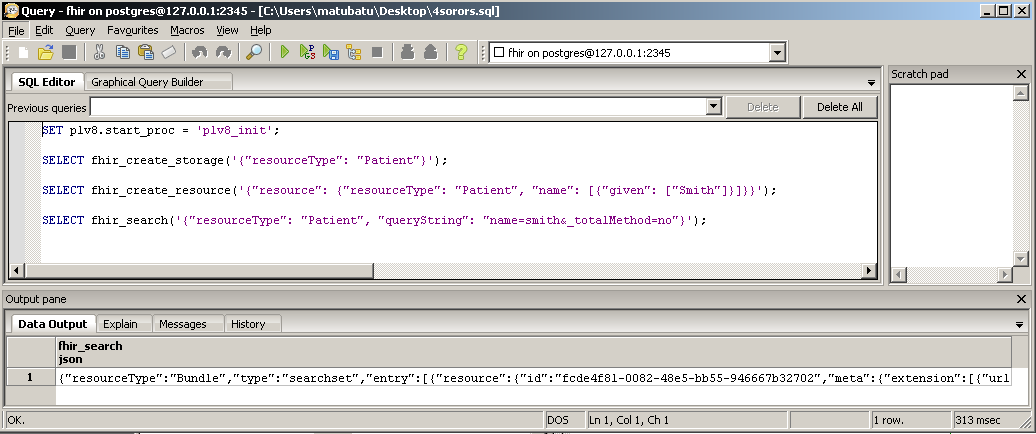
*\i 'c:/Program Files/PostgreSQL/9.5/scripts/fhirbase-1.4.0.0\_c\_utf\_3.sql'*

Das Ergebnis von den Scripten sind 50 Funktionen und 20 Tables:

1. Fhirbase in Vagrant - Einführung
2. **Erzeugen von Patient in pgAdmin III:**

Beim Einfügen von Ressourcen kann man nicht mit nativen SQL-Befeheln operieren. Es muss immer *Stored Procedures* genutzt werden: <http://fhirbase.github.io/demo/tutorial.html>.

Im Tutorial findet man gute Beispiele für Patient Erzeugung:



Um fhirbase-plv8 laufen zu können muss beim Zugang zum postgresql die erste Set Befehl angeben.

Andere alternative: in postgresql.conf File einmal angeben: *echo "plv8.start\_proc='plv8\_init'" >> c:\Program Files\PostgreSQL\9.5\data\postgresql.conf*

*SELECT fhir\_create\_storage('{"RessourceType": "Patient"}');*

Fhirbase schafft Patient Schema für alle Patienten

*SELECT fhir\_create\_Ressource('{"Ressource": {"RessourceType": "Patient", "name": [{"given": ["Smith"]}]}}');*

Patient Ressource wird gemacht mit Namen.

*SELECT fhir\_search('{"RessourceType": "Patient", "queryString": "name=smith&\_totalMethod=no"}');*

Einfacher Suche nach Patient mit dem Namen: *smith*

In Data Output-feld sieht man das Ergebnis von json File. Das ID von Patienten ist automatisch generiert.

Das Ziel wäre erstmal die Implementation von einem Patienten in Java Fhir-Hapi. Das erzeugt uns ein Json String. Json wird dann in einem anderen String mit Stored Procedure Syntax konkateniert und in Sql-Statement aufgerufen.

1. **Erzeugen von Patient mit Encounter in pgAdmin III:**

Das ist wieder einen kleinen Test, weil es in Fhirbase-Webseite nicht so klar formuliert wurde.

Erstellen wir ein Encounter Schema mithilfe *SELECT fhir\_create\_storage*

Um ein Encounter erzeugen zu können, muss der Patient in Encounter referenziert werden. Dazu braucht man das ID von Patient. (im Punkt 3. wurde es automatisch generiert).

Es ist auch möglich konkrete ID von Patienten zu erstellen mit *"allowId": true*

Das Erzeugen von Patient mit Encounter sieht folgendes aus:

*(c:\…\myProject\StoredProcedures\Patient\_Encounter\_Create.sql)*

SET plv8.start\_proc = 'plv8\_init';

SELECT fhir\_create\_Ressource ('{

"allowId": true,

"Ressource ": {

"Ressource Type": "Patient",

"id": "smith",

"name":[{"given":"Bruno"}]}}');

SELECT fhir\_create\_Ressource ('{

"allowId": true,

"Ressource ":

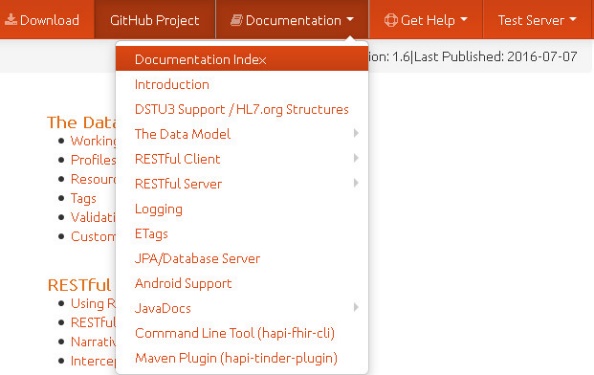
{"Ressource Type":"Encounter",

"status": "onleave",

"patient": {"reference": "Patient/smith"}}}');

*Encounter Data* *öffnen*, checken Ressource jsonb Feld:

{"id": "f58e58da-0ee8-4b00-828e-641c58881a41", "meta": {"extension": [{"url": "fhir-request-method", "valueString": "POST"}, {"url": "fhir-request-uri", "valueUri": "Encounter"}], "versionId": "c3f31a5f-ec6c-443f-a647-2e2fe210db49", "lastUpdated": "2016-09-17T09:36:38.288Z"}, "status": "onleave", "patient": {"reference": "Patient/smith"}, "Ressource Type": "Encounter"}

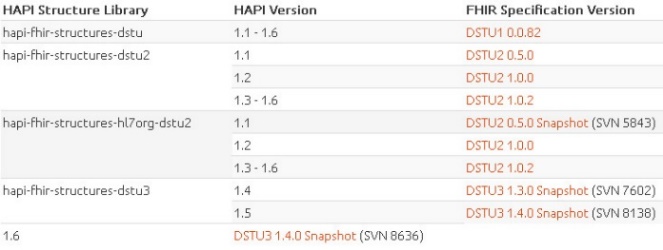
1. FHIR Hapi - Installation

**1.** **Installation**

Das Fhir-Hapi ist eine mächtige Bibliothek, damit Fhir-messaging in eigenen Anwendungen implementieren zu können. Das Fhir-Hapi Dokumentation-Verzeichnis findet man an <http://hapifhir.io/docindex.html> .

Die Fhir-Hapi Umgebung wird mithilfe Maven aufgebaut. Das Version von HAPI 1.5 wurde gewählt:

* **Structure Library:**   
  hapi-fhir-structures-dstu2
* DSTU2 1.0.2

****Die benötigten Bibliotheken und Dependencies wurde mithilfe Maven gebaut.

Das Project am GitHub:

<https://github.com/matubatu/Praktikum_FhirHapi>

**2.** **Einführung - erste Schritte mit Hapi:***myProject/ErsteSchritte/*

In der Anleitung gezeigten Beispielen waren leicht erzeugbar und gut annotiert:

<http://hapifhir.io/doc_intro.html>

*Parsing2String.java:*

Das Java-file demonstriert wie man vom String einen Patient Ressource mithilfe Parsing konvertiert.

*Ressource 2String.java:*

Das File erzeugt einen Patienten mit statischen Dateien und ausgibt in Json-Formate:

*GenericClient.java:*

Mithilfe Generic-Client kann man Zugang zum Testserver bekommen und bestehende Testdaten abfragen.

Link des Testservers: http://fhirtest.uhn.ca/

**3.** **Erzeugter Patient und Encounter Export zu Fhirbase mit Statische-daten:**   
myProject/Test/ PostgreConnectivityTest.java

Das ist einen Verbindungstest zu Fhirbase mit einem Patienten und ihm gehörendes Encounter. Der Test ist erst mit statischen Dateien befühlt. Ausführen in Eclipse und Testen in pgAdmin - *View the Data in selected Object* Icon.

**4. Erzeugen von Patient mit Random-daten mithilfe Hapi Objekt:***PatientErzeugen\_Hapi\_Test.java*

Das Erzeugen von Patient mit Random-daten mithilfe Dokumentation ist leicht nachvollziehbar.

Starten das Java File findet man die generierte Patient Dateien in Console-fenster. Das ursprüngliches Ziel wäre diese Random Dateien in Json File konvertieren, in Stored Procedure packen und in Hapi ausführen.

**5. Erzeugen von Patient mit Random-daten mithilfe Hapi Objekt:***PatEncounter\_Random\_Object.java*

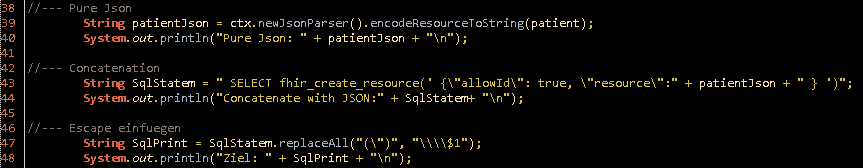
Das ist ein Versuch ein, mit randomisierten Dateien befühlte Hapi-Objekt mithilfe JsonParser in String zu konvertieren, zum Stored Procedure addieren, Syntax ändern und in Sql Statement ausführen.

In Konsole zeigt es das Ergebnis der drei Schritten:

1. Erzeugte *Pure Json*

2*.* Json addiert zum Stored Procedure: *Concatenate with Json*

3. Escape Zeichnen einfügen



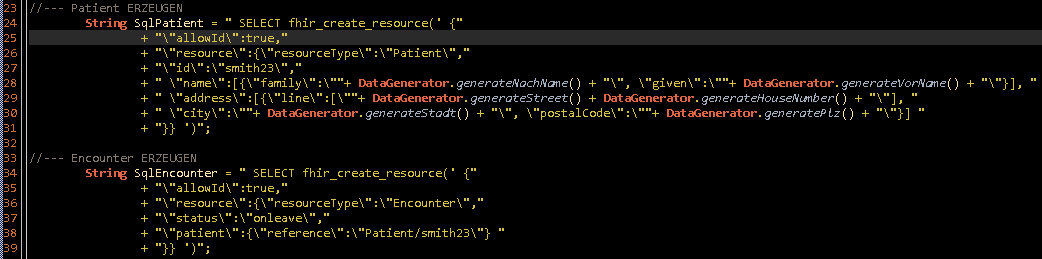
In den ersten und zweiten Punkten erstellten Befehlen funktionieren in pgAdmin (manuelle Prüfung). Leider nach dem Einfügen von „Escapes“ bekommnt man eine Fehlermeldung. Es wird in Weiteren getestet.

**6. Erzeugen von Patient mit Random-daten ohne Hapi Objekt:***PatEncounter\_Random\_NoObject.java*

Der andere und einfachere Weg wäre, dass es kein Hapi-Objekt erzeugt wird.

Um 1 Million Patient mit jeweils einem Encounter erzeugen zu können, kann es auch ohne Erstellung von Java Objekt bewerkstellig werden. So wird sowohl die Erzeugung von Objekt als auch die Befüllung erspart.

In diesem Beispiel wird ein Patient-Encounter Paar erstellt und in den String-Statements (SqlPatient / SqlEncounter) die randomisierten Dateien hinzugefügt.



*PatEncounter\_Random\_NoObject2.java*

Im Beispiel wurde die Erzeugung von Patient-Encounter mit For-Schleife und unique PatientID implementiert.

**7. Arbeiten mit Million Patienten:***c:\git\Praktikum\_FhirHapi\myProject\src\main\java\myProject\Database.java*

Der erste Test erfolgt mit 10.000 Patienten.

In allgemein kann man sagen, dass PostgreSQL v9.5.3 beim Löschen von Ressourcen in pgAdmin sehr instabil ist. Refresh Option zu drücken ist nicht empfehlenswert. Das Nuzten von Vacuum für Garbage zu sammeln ist empfohlen.

**7. Arbeiten mit echten Dateien von Österreich:***c:\git\Praktikum\_FhirHapi\myProject\src\main\java\myProject\Database\_AT.java*

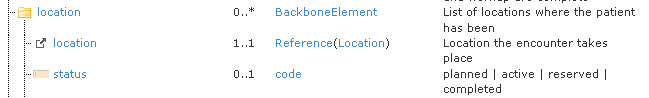
Der nächste Schritt wäre die Datenbank mit echten österreichischen Patient Namen, Adressen, den Encounters gehörende Krankenhäusern und …ausfühlen.

In HL7 Dokumentation wird es nachgeschaut, wie die Struktur von Encounter bezüglich Krankenhaus oder Location aufgebaut:

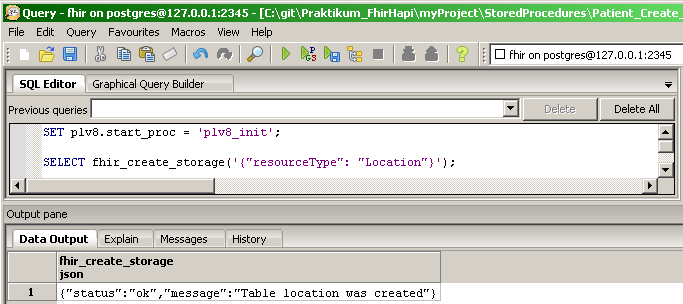
https://www.hl7.org/fhir/encounter.html

*In Resource Content* Json-Tab ist es gut sichtbar, dass ein Encounter-Ressource ein Location list beinhaltet.





Die Location ist wieder ein Referenzobjekt. Es wird ein Location Schema gemacht:



Es wird erst mit statischen Dateien getestet ob alles funktioniert:

*c:\git\Praktikum\_FhirHapi\myProject\src\main\java\myProject\Test/Postgre\_Pat\_Enc\_Location.java*

NEM KELL Testen den Zusammenhang der 3 Resorucen (*Patient-Encounter-Location*) mit random Dateien:

*c:\git\Praktikum\_FhirHapi\myProject\src\main\java\myProject\Database\_AT.java*

Die echten Postleitzahlen für jede Stadt erhaltete ich in Excel Document:

*..\myProject\Dokumente\RawDateien\ Plz-bezirk.xlsx*

Die Adresse und Name der Krankenhäuser in Österrecih erhaltete ich in Sql Stored Porcedure Form:

*..\myProject\Dokumente\RawDateien\* mapping\_kas.sql

In mapping\_kas Dokument gibt es aber mehr Information als ich für die Aufgabe benötigte. Es wird weitere Vorarbeit benötigt.

Da bei dem Randomisierten dummy Patienten (*Database.java*) erschaffte Nachnamen schon echte Namen waren, musste ich die folgende Randomisation ändern:

* Vornamen, um echte deutsche Namen bekommen zu können.
* Stadt-Postleitzal Paaren, die man von den oben genannte Plz-bezirk.xlsx bekommt.

Da die Stadt-Postleitzal Paaren sowieso in dieser Excel-Form kammen, würde es sinnvoll diese von Excel importieren und nutzen.

*..\myProject\src\main\java\myProject\src\main\java\Test\ReadExcelData\_PLZ\_Stadt.java*

Das File importiert plz-stadt Paaren von Excel Dokument.

Die Namen und Adresse von Krankenhäuser können dann nach einer bisschen Vorarbeit auch von Excel importieren:

*..\myProject\src\main\java\myProject\src\main\java\Test\ReadExcelData\_KH\_Address.java*

Der Test importiert mehrere Excel-Cellen in ein Record-Objekt, das in ein HashMap gepackt wird.

Mit diesen Erfahrungen wird in DataBase\_AT.java den ganzen Prozess implementiert:

*..\myProject\src\main\java\myProject\src\main\java\myProject\DataBase\_AT.java*

Im Laufe der Prozess wird erst alle 281 Location aus Excel: *KH\_AddressGenerator.java importiert.*

Dann werden die beliebige Anzahl von Patient Ressourcen gepaart mit Encounters erstellt. Die deutsche Vor- und Nachnamen werden aus vorgefertigten Listen generiert.

Die Adresse von Patienten sind randomisierte Dummy Dateien, aber die Plz-Stadt Paaren kommen aus Excel randomisiert (Echte Städte mit Postleitzahlen).

1. https://www.ibm.com/developerworks/community/blogs/jfp/entry/Using\_Docker\_Machine\_On\_Windows?lang=en [↑](#footnote-ref-1)