DEMIS-Importer

Abkündigung zum 31.12.2022

Nach dem 31.12.2022 wird es keinen Support mehr für den DEMIS-Importer geben. Eine einwandfreie Funktionsweise über diesen Zeitpunkt hinaus kann nicht garantiert werden.

Der DEMIS-Importer stellte eine Übergangslösung dar. Softwarehersteller sind daher angehalten, direkt die FHIR Schnittstelle von DEMIS zu verwenden. Zur Integration in Ihre Software stellen wir Ihnen auf Anfrage gern die Sourcen des DEMIS-Importers zur Verfügung.

Die aktuellste Dokumentation befindet sich immer in der README des DEMIS Importer Projekts. Die hier enthaltenen Informationen werden in regelmäßigen Abständen aktualisiert.

Java Projekt zur Umsetzung einer Komponente, die Meldungen für die jeweiligen Gesundheitsämter importiert.

Hinweis: Bei einigen JDKs kann es zu Problemen bei der Entschlüsselung kommen (HmacPBESHA256). Getestet wurden folgende JDKs (höhere Versionen sollten ebenfalls funktionieren):

- Oracle JDK Version 13.0.2
- OpenJdk Version 11
- AdoptOpenJdk 11.0.7.10

Nutzung

Dieses Kapitel beschreibt wie der DEMIS Importer als Kommandozeilenwerkzeug verwendet werden kann.

Kompilieren und Starten

Der DEMIS Importer ist als Java-Projekt umgesetzt, das mit Hilfe von Maven kompiliert wird (`mvn clean install`). Maven erstellt eine Jar-Datei welche im Verzeichnis `./target/demis-importer-0.1.jar` gefunden werden kann. `./` bezeichnet hier das Verzeichnis in dem das DEMIS Importer Projekt liegt.

Über die Kommandozeile kann die Jar-Datei mit Hilfe von Java ausgeführt werden. Wenn man sich im selben Verzeichnis wie die Jar-Datei befindet, lässt sich diese wie folgt ausführen:

```
java -jar demis-importer-0.1.jar
```

Beim ersten Ausführen wird wahrscheinlich der Fehler `MissingConfigurationFileException` auf der Kommandozeile ausgegeben. Das liegt daran, dass der DEMIS Importer bestimmte Parameter benötigt, um erfolgreich ausgeführt werden zu können. Der nächste Teil beschreibt die benötigte Konfigurationsdatei, um diese Parameter bereit zu stellen.

Konfigurationsdatei

Die Konfigurationsdatei enthält notwendige Parameter für die Ausführung. Folgende Angaben werden in der Konfigurationsdatei benötigt, wobei die mit einem $-Zeichen markierten Variablen durch richtige Werte angepasst werden müssen:

```
debuginfo.enabled=false
fhir.basepath=https://demis.rki.de/notification-clearing-api/fhir/
idp.tokenendpoint=https://demis.rki.de/auth/realms/OEGD/protocol/openid-connect/token
idp.oegd.clientid=demis-importer
idp.oegd.secret=secret_client_secret

idp.oegd.username=$USERNAME
idp.oegd.truststore=$PATH_TO_TRUSTORE_FILE
idp.oegd.truststorepassword=$PASSWORD_FOR_TRUSTORE_FILE
idp.oegd.authcertkeystore=$PATH_TO_KEY_STORE_FILE
idp.oegd.authcertpassword=$PASSWORD_FOR_KEY_STORE_FILE
idp.oegd.authcertalias=$ALIAS

# Define an alternative keystore & password (optional)
idp.oegd.outdated.authcertkeystore=
idp.oegd.outdated.authcertpassword=

# Define custom values for timeouts in milliseconds (optional) - defaults are 2000ms, 2000ms, 20000ms
respectively
connect.timeout.ms=
connection.request.timeout.ms=
socket.timeout.ms=
```

Die Datei-Pfade in der Konfigurationsdatei (z.B. Parameter `$PATH_TO_TRUSTORE_FILE` und `$PATH_TO_KEY_STORE_FILE`) sind entweder absolut oder relativ zum Arbeitsverzeichnis
zu setzen (z.B. `idp.oegd.truststore=C:/demis/demis-importer/certs/truststore.p12` oder `idp.oegd.truststore=certs/truststore.p12`).

Alternativer Keystore

`idp.oegd.outdated.authcertkeystore` und `idp.oegd.outdated.authcertpassword` sind optionale Parameter der Konfigurationsdatei, die den Pfad zu einem alternativen Keystore angeben können. Die Zertifikate zum Ver- und Entschlüsseln von Meldungen laufen nach einiger Zeit ab und müssen durch neue ersetzt werden. Beim Austausch der Zertifikate könnte es vorkommen, dass der Demis-Importer bereits
das neue Zertifikat hat, aber noch Meldungen in der notification-clearing-api existieren, die über das alte Zertifikat verschlüsselt sind. Durch die Angabe des alten Keystores unter den beiden Parametern kann der Demis-Importer das für die Meldung passende Zertifikat auswählen und entschlüsseln.

Laden der Konfigurationsdatei

Damit der DEMIS Importer die Konfigurationsdatei lesen kann muss diese ihm übergeben werden. Das geht auf zwei Arten:

1. Die Konfigurationsdatei wird unter dem Name `app.properties` im selben Verzeichnis gespeichert von wo man auch die Jar-Datei ausführt. Der Demis-Importer wird anschließend in genau diesem Verzeichnis nach einer Datei mit dem Namen `app.properties` suchen. Wenn er sie findet, wird er die Datei laden und die darin enthaltenen Parameter nutzen.
2. Man übergibt die Konfigurationsdatei mit dem Kommandozeilenparameter `-config`. Z.b. wenn man die Konfigurationsdatei unter "C:/demis/configuration.txt" gespeichert hat, kann man diese Datei an den DEMIS Importer wie folgt übergeben `java -jar demis-importer-0.1.jar -config C:/demis/configuration.txt`

Wenn die Parameter in der Konfigurationsdatei richtig sind, sollte der DEMIS Importer sämtliche Meldungen von der notification-clearing-api herunterladen und unter dem Verzeichnis `.\results` als JSON-Dateien und als XML-Dateien speichern.

Zeit-Konfigurationsdatei

Bei der ersten Ausführung des DEMIS Importers wird die "Zeit"-Konfigurationsdatei `app.time-properties` erstellt:

```
#Updating query dates in format yyyy-MM-dd HH:mm:ss:SSS (For explanation of abbreviations see: <https://docs.
oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html)>
#Wed Aug 26 12:59:58 CEST 2020
notification.last.update.date=2020-08-26 10:41:05:654
```

Diese Datei enthält den Parameter `notification.last.update.date`, welcher verwendet wird, um sich den aktuellsten Zeitstempel der bereits heruntergeladenen Meldungen über mehrere Ausführungen des DEMIS Importers zu merken. Beim Herunterladen neuer Meldungen wird der Parameter für die Unterscheidung zwischen alten und neuen Meldungen auf dem Server verwendet. Nur Meldugen mit einem `_lastUpdated` Feld, dessen Wert größer oder gleich (= aktuelleres oder zeitgleiches Datum) als der Parameter ist, werden
vom Importer heruntergeladen. Das Datum des `_lastUpdated` Felds erfolgreich heruntergeladener Meldungen wird anschließend für die Aktualisierung des Parameters genutzt.

Das Datum kann manuell auf die Millisekunde genau angepasst werden und muss lediglich dem Format `yyyy-MM-dd HH:mm:ss:SSS` (hier beschrieben) entsprechen.

Der oben beschriebene Vorgang nur aktuelle Meldungen (= neuer als der Zeitstempel in der `app.time-properties`) herunterzuladen ist das Standardverhalten des DEMIS Importers. Dieses Verhalten kann mit dem Kommandozeilenparameter `-all` umgangen werden. Fügt man `-all` hinzu werden sämtliche Meldungen heruntergeladen.

Kommandozeilenparameter

Der DEMIS Importer bietet einige Parameter, um dessen Funktionsweise anzupassen. Über die Hilfe-Option `-help` können diese auf der Kommandozeile mit englischer Beschreibung ausgegeben werden. Im folgenden werden die existierenden Parameter erklärt und einige Anwendungsbeispiele gegeben:

Optionen:

- `nobinary` Die Meldungen werden ohne Binärdateien heruntergeladen und enthalten nur eine Zusammenfassung des Inhalts der ursprünglichen Labormeldungen. Durch die kleinere Dateigröße sollten Meldungen schneller geladen werden können. Das interne Datum in `app.time-properties`
  wird durch Aufrufe mit diesem Parameter nicht aktualisiert.
- `all` Lade sämtliche Meldungen vom Server unabhängig von ihrem `_lastUpdate` Wert und dem internen in der `app.time-properties` gespeicherten Datum. Das interne Datum in `app.time-properties`
  wird durch Aufrufe mit diesem Parameter nicht aktualisiert.
- `timedir <arg>` Diese Option erlaubt es ein alternatives Verzeichnis für die Datei `app.time-properties` anzugeben. Als Standardwert wird die Datei immer im Arbeitsverzeichnis gespeichert. Sollte `app.time-properties` in dem angegebenen Verzeichnis noch nicht vorhanden sein wird die Datei einfach neu erstellt.
- `timereset` Löscht die gespeicherten Daten in der geladenen `app.time-properties`.
- `perpetual` Der DEMIS Importer kann mit dieser Option in einer Dauerschleife ausgeführt werden, um ständig Meldungen abzurufen.
- `frequency <arg>` Mit Hilfe dieser Option kann die Frequenz in Minuten bestimmt werden, in dem im "Schleifen-Modus" (aktiviert über die Option `perpetual`) die Meldungen abgerufen werden. Der Standardwert für das Abrufen liegt bei 20 Minuten.
- `config <arg>` Übergabe der Konfigurationsdatei an den DEMIS Importer.
- `dir <arg>` Festlegen eines anderen Verzeichnisses zum Abspeichern der Meldungen. Die Standardeinstellung ist `./results`.
- `help` Gebe Informationen über die Kommandozeilenparameter aus.
- `verbose` Aktiviert die zusätzliche Ausgabe von Informationen über die getätigten HTTP Anfragen und Antworten.
- `json` Speichert die abgerufenen Meldungen als JSON Dateien. In der Standardeinstellung werden die Meldungen doppelt, als JSON und XML abgespeichert.
- `xml` Speichert die abgerufenen Meldungen als XML Dateien.

Beispiele:

- Rufe alle Meldungen als XML-Dateien ab: `java -jar demis-importer-0.1.jar -all -xml`
- Rufe alle 10 Minuten sämtliche neue Meldungen ab: `java -jar demis-importer-0.1.jar -perpetual -frequency 10`
- Ändere die Verzeichnisse, aus denen die Konfigurationsdatei gelesen und die Meldungen abgelegt werden und rufe alle neuen Meldugen ab: `java -jar demis-importer-0.1.jar -config C:/demis/app.properties -dir C:/results`

Hintergrundinformationen

Dieses Kapitel beschreibt die Technischen Hintergründe und Umsetzungen.

Grundlagen des HAPI FHIR Client

Der DEMIS Importer verwendet zum Abrufen der Meldungen den Client von HAPI FHIR. Die Dokumentation von HAPI FHIR zum Client kann man hier finden.

Im Folgenden werden ein paar Grundlagen zur Verwendung des Client aufgelistet.

Der Client von Hapi Fhir kann wie folgt erstellt werden:

```
// Erstelle einen Client mit dem FhirContext Version R4
String serverBase = "https://demis.rki.de/notification-clearing-api/fhir/"
IGenericClient client = FhirContext.forR4().newRestfulGenericClient(serverBase);

// Erstelle und registriere einen Interceptor, der einen validen Token zur Anfragen hinzufügt
// Der Token wird durch die Klasse de.rki.demis.importer.mtls.TLSManager bereitgestellt
BearerTokenAuthInterceptor authInterceptor = new BearerTokenAuthInterceptor(token);
client.registerInterceptor(authInterceptor);

// Erstelle und registriere einen Interceptor zum Loggen
LoggingInterceptor loggingInterceptor = new LoggingInterceptor();
loggingInterceptor.setLogRequestSummary(true);
loggingInterceptor.setLogRequestBody(true);
client.registerInterceptor(loggingInterceptor);
```

In folgender Weise können mit dem Client Meldungen für ein bestimmtes Gesundheitsamt abgerufen werden:

```
String TAG_NAME = "https://demis.rki.de/fhir/CodeSystem/ResponsibleDepartment";
String exampleID = "1.0.01.1."
Bundle responseBundle = client.search()
                .forResource(Bundle.class);
                .withTag(TAG_NAME, exampleID)
                .encodedJson()
                .returnBundle(Bundle.class)
                .execute();
```

Der Server sucht nach Meldungen, welche mit dem passenden Tag versehen sind. Anschließend gibt er die Klasse `Bundle` zurück, welche die gefundenen Meldungen enthält.

Die Anzahl der zurückgegebenen Meldungen bei einer Anfrage ist begrenzt. Sollten mehr Meldungen zur Anfrage gefunden worden sein, als auf einer einzelnen Seite zurückgegeben werden können, wird ein Link gesetzt, der auf eine andere Seite mit weiteren Meldungen verweist. Dieses Verhalten wird "paging" genannt. Die verlinkte Seite kann wiederum einen weiteren Link auf die nächste Seite beinhalten und so fort. Folgender Code lädt eine Seite mit weiteren Meldungen , wenn der Link im responseBundle gesetzt wurde:

```
Bundle nextPageResponseBundle = null;
if (responseBundle.getLink(IBaseBundle.LINK_NEXT) != null) {
        nextPageResponseBundle = client.loadPage().next(responseBundle).execute();
}
```

Die Methode `fetchResourcesFromServer` in der Klasse `de.rki.demis.importer.QueryExecutor` setzt das "paging" um.

Die Meldungen liegen als Typ `Bundle` im responseBundle unter `entries` vor. Folgendes Beispiel iteriert durch alle Einträge und gibt den Geburtstag der Patienten aus:

```
        // Iterieren der Einträge des ResponseBundles, dass die Meldungen enthält
        for (BundleEntryComponent entry : responseBundle.getEntry()) {
                // Hier erhalten wir die eigentliche Meldungen
                Bundle notification = (Bundle) entry.getResource();
                // Iterieren der Elemente der Meldungen (Komposition, Organization, ...)
                for (BundleEntryComponent entry : responseBundle.getEntry()) {
                        if (entry instanceof Patient) {
                                Patient patient = (Patient) patient;
                                System.out.println(patient.get
                        }
                }
        }
```

Erweiterte notwendige Umsetzung

Das zuvor erklärte "paging" ist nur begrenzt verwendbar für große Datenmengen, da der Server die Seiten extra bereitstellen muss. Aus diesem Grund ist die Anzahl an Meldungen , die über das "paging" abgerufen werden können, ebenfalls begrenzt (der Default-Wert sollte bei etwa 250 Meldungen liegen). Um dennoch mehr Meldungen abrufen zu können nutzt der DEMIS Importer die Eigenschaft der notification-clearing-api die Meldungen sortiert nach dem `_lastUpdate` Feld zurückzugeben. Jede Anfrage wird um ein `_lastUpdate` Feld erweitert, dass anhand der heruntergeladenen Meldungen aktualisiert wird. Zusammen mit dem "paging" können so sehr große Mengen an Meldungen abgerufen werden. Die Methoden `fetchAndStoreAllNotifications` und `fetchAndStoreNotificatons` im `de.rki.demis.importer.DemisImporter` setzen dieses Verhalten um. Hilfklassen für das Verwalten des internen `_lastUpdate` Feldes sind die Klassen `de.rki.demis.importer.DateHandler` und `de.rki.demis.importer.helper.QueryDates`.