

How-To CentraXX: FHIR Custom Export



NOW IS THE TIME.

Titel	CentraXX FHIR Custom Export
Ersteller	Jonas Küttner, Mike Wähnert
Datum	Mär 30, 2021 10:47
Version	1.12
Status	Final
Verbleib	Zu Schulungszwecken externe Weitergabe erlaubt.

Inhaltsverzeichnis

- 1 [Inhaltsverzeichnis](#)
- 2 [Abkürzungsverzeichnis](#)
- 3 [Einleitung](#)
 - 3.1 [Inhalt des Dokuments](#)
 - 3.2 [Zielgruppe](#)
 - 3.3 [Überblick über die Schnittstelle](#)
 - 3.3.1 [Scheduled Export](#)
 - 3.3.2 [Inkrementeller Export](#)
 - 3.3.3 [Manueller Export per REST-Call](#)
- 4 [Einrichtung in CentraXX](#)
 - 4.1 [Konfiguration der centraxx-dev.properties](#)
 - 4.2 [Einrichtung eines Projekts am Beispiel einer Ausleitung für das DKT](#)
 - 4.2.1 [ProjectConfig.json](#)
 - 4.2.2 [ExportResourceMappingConfig.json](#)
 - 4.2.3 [ResourceBundleExportConfig.json](#)
- 5 [Nutzung der Groovy-DSL-Bibliothek](#)
 - 5.1 [Einleitung/Überblick](#)
 - 5.2 [Voraussetzungen zur Nutzung der Kairos-FHIR-DSL Bibliothek und des Beispielprojekts](#)
 - 5.3 [Installation des FHIR-Beispielprojekts](#)
 - 5.4 [Installation der FHIR-DSL-Bibliothek](#)
 - 5.5 [Verwendung der DSL-Bibliothek zur Erstellung von Transformations-Skripten](#)
 - 5.5.1 [Allgemeines](#)
 - 5.5.2 [Zusammenbau der FHIR Nachricht mit Groovy Closures](#)
 - 5.5.3 [CXX JPA Meta Model und die Verwendung von context.source mit der JPA-Navigation](#)
 - 5.5.3.1 [JPA Meta Model](#)
 - 5.5.3.2 [Navigation mittels verketteter Methodenaufrufe](#)
 - 5.5.3.3 [Verwendung von Konstanten in Listen-Closures](#)
 - 5.5.4 [Definition eigener Methoden, um CentraXX-Entitäten optimal abzubilden](#)
 - 5.5.5 [Abfrage von Messwerten mittels JPA-Navigation](#)
 - 5.5.6 [Weitere Selektion mittels bedingtem Return](#)
- 6 [FHIR-Export in Zielsysteme](#)
 - 6.1 [Ausleitung in einen Blaze Store](#)
 - 6.2 [Ausleitung ins Dateisystem](#)
- 7 [Anhang](#)
 - 7.1 [Beispiel für ausgeleitetes FHIR-Bundle für die FHIR-Ressource "Patient"](#)

Abkürzungsverzeichnis

CXX	CentraXX, siehe https://www.kairos.de
FHIR	Fast Healthcare Interoperability Resources, siehe https://hl7.org/FHIR/
DSL	Domain-specific language
JPA	Java Persistence API
DKTK	Deutsches Konsortium für Translationale Krebsforschung (https://dktk.dkfz.de/de/home / https:////simplifier.net/oncology)
BBMRI RI	Biobanking and Biomolecular Resources Research Infrastructure, siehe https://www.bbmri.de/ueber-gbn/bbmri-eric/
GECCO	German Corona Consensus (https://www.netzwerk-universitaetsmedizin.de/covid-19-datensatz)

Einleitung

Inhalt des Dokuments

Die vorliegende Dokumentation des CentraXX-FHIR-Custom-Exports umfasst folgende Punkte:

- initiale Einrichtung der Schnittstelle für den CentraXX-FHIR-Custom-Export
- Konfiguration des Exports für spezifische FHIR-Projekte
- Einrichtung einer Entwicklungsumgebung und Installation einer Groovy-basierten Kairos-FHIR-DSL-Bibliothek
- Exemplarische Einführung in die Funktionalität der DSL als Grundlage zur Erarbeitung eigener Transformationsskripte.

Zur Einführung in die Nutzung der DSL werden Code-Beispiele aus bereits existierenden Beispiel-Transformationsskripten herangezogen. Dabei wird kontextuell auf die Programmiersprache 'Groovy' eingegangen, das vorliegende Dokument dient jedoch nicht dazu Grundlagen von Groovy, Prinzipien der Implementation der DSL oder die Spezifikation des HL7-FHIR-Standard zu vermitteln. An geeigneten Stellen werden weiterführende Links zu den eben genannten Themen referenziert. Weiterhin beinhaltet dieses Dokument keine umfassende Dokumentation der Kairos-FHIR-DSL selbst.

Voraussetzung für die Nutzung

Es wird eine konfigurierte und parametrierte CentraXX-Instanz vorausgesetzt. Folgende Programm-Versionen wurden zur Erarbeitung der Dokumentation verwendet:

Programm	Version
CentraXX	3.18.0
Kairos-FHIR-DSL-Bibliothek	1.7.0
HL7 FHIR	4.0.1
IntelliJ	2020.3.3
Blaze	0.10.3
Blazectl	0.2.0

Zielgruppe

Dieses Dokument richtet sich in erster Linie an CentraXX Nutzer mit IT-Verständnis, welche:

- die Schnittstelle verwalten
- den FHIR-Export für Projekte konfigurieren
- Transformations-Skripte erarbeiten oder bearbeiten.

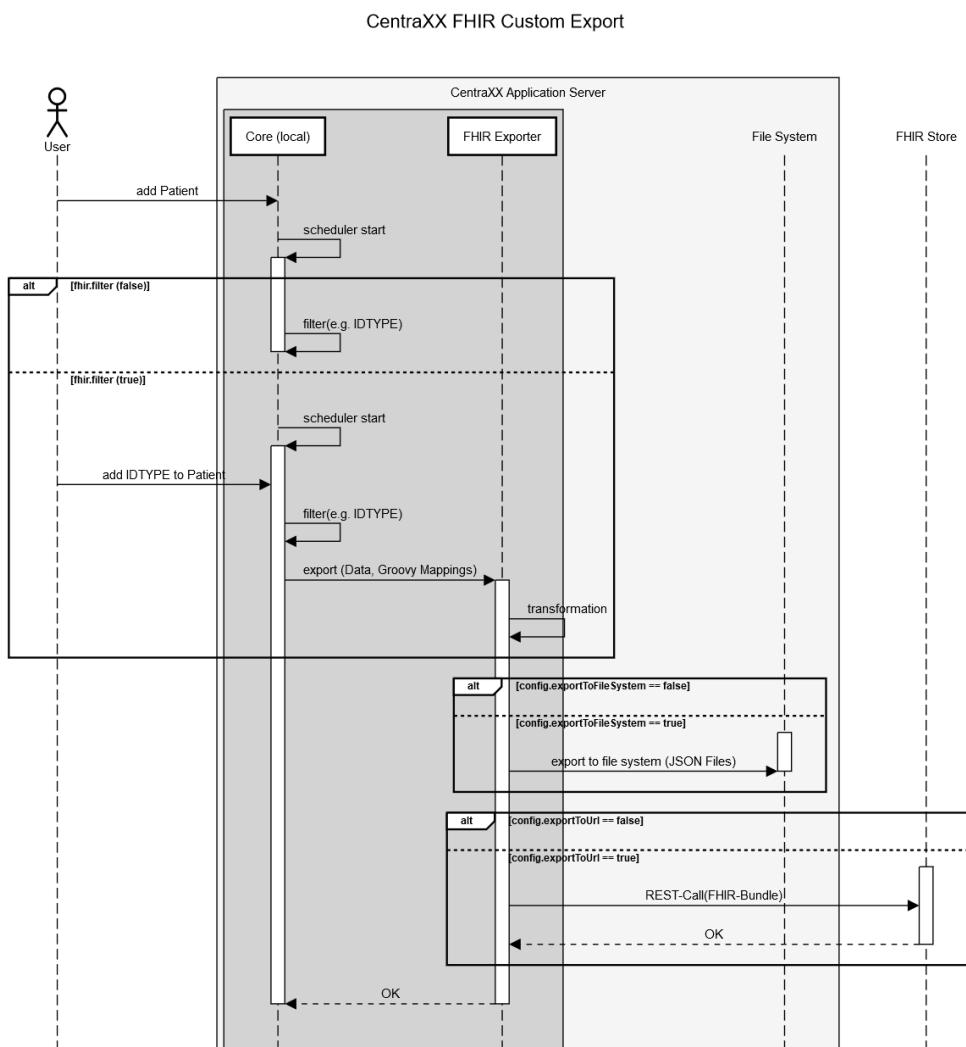
Grundlegende Erfahrungen in Java und/oder Groovy werden ebenso vorausgesetzt wie ein Basisverständnis über den HL7-FHIR-Standard, da dieser die Grundlage für die hier dokumentierte Schnittstelle ist.

Überblick über die Schnittstelle

Der CentraXX-FHIR-Custom-Export stellt primär einen projektspezifisch konfigurierten FHIR Export dar. Im Rahmen der Modulation des Exportverhaltens über ein Groovy Script werden lokal definierte CentraXX Entitäten auf die verschiedenen Felder einer FHIR Nachricht gemappt. Neben der vollen Kontrolle über die zu exportierenden Inhalte erlaubt die Schnittstelle ebenfalls eine Filterung der relevanten Patienten/Probanden. Die entstehenden FHIR Bundles können dann entweder im Filesystem persistiert werden oder an eine Ziel URL eines RESTful Service übertragen werden. Derzeit werden zwei Export-Mechanismen unterstützt:

Scheduled Export

Hierbei wird durch eine zeitliche Planung der Export für **alle** gefilterten Patienten gestartet. Der Prozess für den Scheduled Export wird im folgenden UML-Diagramm dargestellt:



Inkrementeller Export

Der andere Exportmechanismus wird durch eine Datenveränderung und nicht durch einen zeitlichen Trigger gestartet. Konkret bedeutet dies, dass hier nur die Patienten exportiert werden, die tatsächlich geändert wurden.

Sobald Daten eines Patienten in CentraXX geändert werden oder ein neuer Patient angelegt wird, wird der Export für diesen Patienten gestartet. Der Patient muss immer noch dem projektspezifischen Patientenfilter genügen. Der Export wird für alle assoziierten Entitäten (Patient, Diagnosen, Verläufe, etc.) vorgenommen. Um die Integrität der Referenzierung nicht zu gefährden, werden bei einer Änderung der Patientendaten immer alle Entitäten transformiert und exportiert. Der inkrementelle Export bietet somit eine effiziente Möglichkeit, um die Daten in einem FHIR-Store mit denen in CentraXX zu synchronisieren, ohne gleichbleibende Daten periodisch immer wieder zu übertragen.

Manueller Export per REST-Call

Der FHIR-Export per REST-Call für eine CentraXX-Entität gestartet werden. Der Aufbau des REST-Requests wird weiter unten beschrieben. Im Gegensatz zu den automatisierten Exporten, können mit dieser Methode Daten auch nicht-pseudonymisiert exportiert werden. Dazu muss der Nutzer, der den REST-Call tätigt die entsprechenden Berechtigungen besitzen. Die automatisierten Exportmechanismen nutzen einen System-Benutzer. Dieser besitzt derzeit keine Berechtigungen für nicht-pseudonymisierte Exporte. In den exportierten FHIR-Resource werden Datumsangaben auf den 01.01. des entsprechenden Jahres gesetzt. Diese Exportmethode bietet die Möglichkeit Daten nicht-pseudonymisiert zu exportieren. Dazu muss der Nutzer, mit dessen Credentials der Request gegen CentraXX authentifiziert wird, entsprechende Berechtigungen besitzen. Für die Authentifizierung wird [BasicAuthentication](#) verwendet.

Der Export von nicht-pseudonymisierten Daten ist derzeit nur über den Export per REST-Call möglich.

Folgende Query-Parameter sind für den Request zu spezifizieren:

Parameter	Typ	Beschreibung	Default-Wert
cxxEntity	String	CentraXX-Entität	-
fhirResource	String	Ziel-FHIR-Bündel-Ressource	-
project	String	Export-Projekt, das Transformations-Skript enthält	-
template	String	Groovy-Transformationsskript	-
fileExport	boolean	Export ins Datei-System	false
restExport	boolean	Export in Ziel-System (CentraXX oder Blaze-Store) per REST-Call	false
validateExport	boolean	Evaluation des Import der einzelnen FHIR-Ressourcen im FHIR-Bündel bei Exporten per REST-Call	false

Die CentraXX-Entität wird über das angegebene Transformation-Skript (template) transformiert und in der spezifizierten FHIR-Ressource exportiert. Der Ziel-Pfad des Exports ins Datei-System, bzw. die Ziel-URL des Exports per REST-Call, sind im ProjectConfig.json konfiguriert.

Einrichtung in CentraXX

Konfiguration der centraxx-dev.properties

Zur Nutzung des FHIR-Custom-Exports müssen die Export-Funktionen allgemein aktiviert werden. Zusätzlich muss das übergeordnete Export-Mapping-Verzeichnis spezifiziert werden, in welchem die Unterordner der Projekte mit den spezifischen Konfigurations- und Mapping-Dateien liegen. Die Properties in der centraxx-dev.properties beziehen sich immer auf den Applikationsserver und etwaige Pfade müssen auf dem Server vorhanden sein.

```
interfaces.fhir.custom.export.scheduled.enable=<true|false>
interfaces.fhir.custom.export.incremental.enable=<true|false>
interfaces.fhir.custom.mapping.dir=C:/applications/centraxx-home/fhir-custom-mappings
```

Die Änderungen an den centraxx-dev.properties erfordert einen Neustart von CentraXX.

Einrichtung eines Projekts am Beispiel einer Ausleitung für das DTKK

Die FHIR-Ressourcen sind projektspezifisch definiert. Bei der Einrichtung von Export-Projekten für verschiedene FHIR-Projekte werden für dieselbe CentraXX-Entität demnach auch projektspezifische Transformations-Skripte benötigt. Weswegen die Transformationsskripte und Konfigurationsdateien für jedes Projekt in einem separaten Unterverzeichnis bereitgestellt werden. Als Beispiel hierzu soll ein Projekt zur Transformation der CentraXX-Entitäten in FHIR-Profile des DTKK-Projekts angelegt werden. Die Profilierung der Ressourcen wurde durch das DTKK umgesetzt und wird von CXX lediglich abgebildet. Die FHIR Ressourcen sind unter <https://simplifier.net/oncology> definiert und dokumentiert. Dafür ist ein Unterverzeichnis **/dtkk** unter interfaces.fhir.custom.mapping.dir anzulegen. Auch wenn die Benamung des Unterverzeichnisses frei ist, sollten hier sprechende Namen verwendet werden. In diesem Verzeichnis sind die Groovy-Transformations-Skripte für dieses Projekt abzulegen. Der resultierende Pfad aus dem Beispiel oben würde sich somit zu: "C:/applications/centraxx-home/fhir-custom-mappings/dtkk" ergeben.

Nach dem Anlegen eines neuen Projektverzeichnisses und einem Neustart von CentraXX wird automatisiert eine Projektkonfigurationsdatei (ProjectConfig.json) im Projektverzeichnis erzeugt. Wenn die Bedingungen des ersten Synchronisationslaufes erfüllt sind und es zu einem ersten Durchlauf der Exportroutinen kommt, wird CX zwei weitere Konfigurationsdateien erzeugen. Somit wurden insgesamt folgende drei Konfigurationsdateien erzeugt:

- ProjectConfig.json
- ExportResourceMethodConfig.json
- BundleRequestMethodConfig.json.

ProjectConfig.json

Die **ProjectConfig.json** ist die Basis-Konfigurationsdatei für den FHIR-Export eines Export-Projekts. Diese Konfiguration kann nicht während der Laufzeit geändert werden. CentraXX muss nach Änderungen dieser Datei neu gestartet werden. Die Konfigurationsdatei enthält für jeden Konfigurationsparameter ein Feld „description“, welches die Funktion des Parameters in englischer Sprache erklärt. Dieses Feld dient nur zur Information des Nutzers. In den folgenden Beispielen werden sehr lange Description-Strings - zur vereinfachten Darstellung - mit "..." abgekürzt.

Scheduled Export: Mit dieser Config wird der Scheduled Export für das Projekt aktiviert. Alle gefilterten CentraXX Patienten-Daten werden zum spezifizierten Zeitpunkt exportiert.

```
"scheduledExportEnable": {
    "description": "If true, this project will export all filtered CXX patient data with the configured FHIR resources by the specified exportInterval.",
    "value": true
},
```

Inkrementeller Export: Aktiviert den inkrementellen Export. Wenn Patienten-Daten in CentraXX geändert werden, wird der Export für diesen Patienten gestartet. Es werden nur Patienten berücksichtigt, die dem allgemeinen Patientenfilter genügen.

```
"incrementalExportEnable": {
  "description": "If true, this project will export filtered CXX patient data with the configured FHIR resources by a detected change after a centraxx transaction.",
  "value": false
},
```

Patientenfilter: Dieser Filter spezifiziert, welche Patientendaten innerhalb des FHIR-Exports eingeschlossen werden. Die folgende Filtertypen stehen zur Auswahl:

Filter	Beschreibung
PATIENTSTUDY	Patient gehört zu einer bestimmten Studie im Studienregister. Die Studie wird folgendermaßen spezifiziert: "{studyCode}\${studyProfileCode}", z.B. COVID19\$BASIC.
CONSENTTYPE	Patient besitzt eine Einverständniserklärung eines bestimmten Typs (Code).
ORGUNIT	Patient gehört zu einer bestimmten CentraXX- Organisationseinheit (Code).
FLEXIFLAGDEF	Patient besitzt ein Universalattribut eines bestimmten Universalattribut-Typs (Code).
IDTYPE	Patient besitzt ID eines bestimmten ID-Typs (Code).

Es können mehrere Filterwerte angegeben werden. Es werden alle Patienten in den Export eingeschlossen, welche mindestens einem dieser Filter genügen. Die Filterwerte sind in Form einer Liste anzugeben. Wird keine oder eine leere Liste angegeben, so wird kein Patient exportiert.

Es folgt ein Beispiel zur Konfiguration eines Filters, welcher alle Patienten einschließt, die eine ID des Types „DKTK“ oder „MPI“ besitzen:

```
"patientFilterType": {
  "description": "...",
  "value": "IDTYPE"
},
"patientFilterValues": {
  "description": "...",
  "value": [ "DKTK", "MPI" ]
},
```

Seitengröße: Bestimmt die Größe einer CentraXX-Abfrage und des dazugehörigen FHIR-Bundles. Es wird empfohlen einen kleineren Wert zu wählen, wenn das Zielsystem ein Transaktions-System besitzt (z.B. 10 für CentraXX). Ist dies nicht der Fall, oder im Falle, dass ins Filesystem exportiert werden soll, so kann ein größerer Wert gewählt werden (z.B. 500).

```
"pageSize": {
  "description": "...",
  "value": 10
},
```

Exportzeitraum: Legt den Zeitraum des FHIR-Exportvorgangs fest. Die Zeitangabe erfolgt mittels eines sechsstelligen Chron-Ausdrucks. Mit der Standardkonfiguration erfolgt der Export jeden Tag um 12 Uhr. Hierbei sind [Cron-Ausdrücke von Spring](#) zu verwenden.

Als Beispiel die Standardkonfiguration:

```
"exportInterval": {
  "description": "Sets the export cron interval. Default is once a day.",
  "value": "0 0 12 * * *"
},
```

Export ins Filesystem: Die FHIR-Daten können in das lokale Filesystem exportiert werden. Dazu muss der Export mit folgender Property aktiviert werden:

```
"exportToFileSystem": {  
    "description": "If true, the scheduler will export to the specified file system exportFolder.",  
    "value": true  
},
```

Zudem muss das Zielverzeichnis für den Export angegeben werden. Existiert das angegebene Verzeichnis nicht, wird es von CentraXX neu angelegt:

```
"exportFolder": {  
    "value": "C:/centraxx-home/fhir-custom-export"  
},
```

Export an URL: Die FHIR-Ressourcen können per HTTP an externe Zielsysteme transferiert werden. Diese Export-Funktion wird mit folgender Property aktiviert:

```
"uploadToUrl": {  
    "description": "If true, the scheduler will export to the specified uploadUrl REST endpoint.",  
    "value": false  
},
```

Des Weiteren muss die URL des Zielsystems angegeben werden. Diese ist der Endpunkt einer Bundle-Ressource in einem FHIR-Store, wie Blaze oder CentraXX:

```
"uploadUrl": {  
    "value": "http://localhost:9090/fhir"  
},
```

Zur HTTP-Authentifizierung am Ziel-Server wird **Basic Authentication** verwendet. Nutzername und Passwort werden folgendermaßen konfiguriert:

```
"uploadUser": {  
    "description": "BasicAuth User name in the target system.",  
    "value": "admin"  
},  
"uploadPassword": {  
    "description": "BasicAuth Password in the target system.",  
    "value": "admin"  
},
```

Verlangt das System keine Authentifizierung, werden die angegeben Daten für Nutzername und Passwort ignoriert.

Weiterführend kann im Falle von "**uploadToUrl = true**" die Nutzung einer Validierung der HTTPS Zieladresse konfiguriert werden. Mit folgender Konfiguration wird spezifiziert, ob allen Zertifikaten vertraut werden kann:

```
"sslTrustAllEnable": {  
    "description": "If true, all ssl/tls certificates are trusted, otherwise the default java store is used for  
SSL/TLS verification.",  
    "value": false  
},
```

Ist die Option auf "false" gesetzt, wird der Standard-Truststore (Java TrustStore) verwendet. Im Falle, dass auch beispielsweise system-spezifischen Zertifikaten vertraut werden kann, so können diese in einem benutzerdefinierten Truststore abgelegt werden. Als Truststore ist eine **Java KeyStore**-Datei (.jks) zu verwenden. Zudem sind der Pfad der Truststore-Datei im Dateisystem und das Passwort zu konfigurieren:

```
"trustStoreEnable": {  
    "description": "...",  
    "value": false  
},
```

```

"trustStore": {
  "value": "C:/centraxx-home/fhir-custom-truststore.jks"
},
"trustStorePassword": {
  "value": "changeit"
},

```

Für den FHIR-Export können außerdem das Login für die Anfragen und Antworten der HTTP-Requests konfiguriert werden. Das kann hilfreich zur Initialisierung und zum Testen sein. Wird bspw. ein neuer Export konfiguriert, welcher per HTTP an einen FHIR-Store ausgeweitet wird, so können die gesendeten FHIR-Ressourcen und die HTTP-Antworten des Zielservers eingesehen werden. Die Messages werden standardmäßig innerhalb des Serverlogs geloggt. Mit folgender Option kann die HTTP-Message-Body, sprich das gesendete FHIR-Bundle, geloggt werden. Die Konfiguration "prettyPrintEnable" bietet hierbei die Option diese Nachrichten für den Nutzer leserlich strukturiert zu loggen. Hierbei gilt es zu beachten, dass die Aktivierung dieser Optionen bei einer hohen Anzahl von auszuleitenden Entitäten, den Serverlog schnell füllen werden.

```

"prettyPrintEnable": {
  "description": "Print style the HTTP message body.",
  "value": false
},
"fhirMessageLoggingEnable": {
  "description": "If true, HTTP message requests are logged in CentraXX separately.",
  "value": false
},

```

Die HTTP-Antwort des Zielservers kann ebenfalls geloggt werden. Dazu muss folgende Option aktiviert sein:

```

"fhirResponseLoggingEnable": {
  "description": "If true, HTTP message responses are logged in CentraXX separately.",
  "value": false
},

```

FHIR- Ressourcen werden Bundle-weise exportiert und in das Zielsystem eingeschleust. Ein solches Bundle kann bspw. Daten von fünf Exemplaren der FHIR-Ressource 'Patient' enthalten. Bei der Verarbeitung des HTTP-Request könnten z.B. vier der Patienten aus der Bundle-Ressource erfolgreich importiert werden, jedoch der Import für einen fehlschlagen. In diesem Fall wird jedoch trotzdem eine positive HTTP-Antwort (200 OK) erzeugt. Um den Import jeder einzelnen im Bundle enthaltenen FHIR-Ressource zu validieren, muss somit die Antwort geparsst und jeder Eintrag gegen die zugehörige FHIR-Ressource validiert werden. Diese Validierung wird mit folgender Konfiguration aktiviert:

```

"fhirResponseValidationEnable": {
  "description": "...",
  "value": false
}

```

Bei Verwendung des Scheduled Export ist nach Änderungen der ProjectConfig.json ein Neustart von CentraXX erforderlich. Der Scheduler muss mit der aktuellen Konfiguration initialisiert werden, damit Änderungen an den Filtern und dem Scheduling wirksam werden. Dies geschieht ausschließlich nach einem Neustart.

ExportResourceMappingConfig.json

Diese Konfiguration verknüpft eine CentraXX-Entität mit einer FHIR-Ressource über ein Groovy-Transformations-Skript (groovy-Mapping). Die gleiche CentraXX-Entität kann so konfiguriert werden, dass sie über verschiedene Transformations-Skripte auf die gleiche FHIR-Ressource abgebildet werden. Diese Konfiguration kann während der Laufzeit geändert werden und benötigt keinen Neustart von CentraXX. Die Reihenfolge der Mappings ist wichtig, wenn das Zielsystem die referentielle Integrität prüft (z.B. Blaze Store). Folgend ein Beispiel für einen Eintrag in ExportResourceMappingConfig.json:

```
{
  "selectFromCxxEntity": "ORGANIZATION_UNIT",
  "transformByTemplate": "organisation",
  "exportToFhirResource": "Organization"
},
```

Hier werden demnach Daten der CentraXX-Entität "ORGANIZATION_UNIT" mittels der Transformationsvorlage "organisation" auf die FHIR-Ressource "Organization" abgebildet. Das korrespondierende Transformations-Skript ist dann das Groovy-Skript das nach dem oben eingeführten Beispiel unter "C:\applications\centraxx-home\fhir-custom-mappings\dktk\organisation.groovy" liegt.

Hinweis: Die Einträge in der Standard-Version (bei Projekt-Erstellung von CentraXX erzeugt), spezifiziert zugleich auch alle CentraXX-Entitäten für die derzeit Ausleitungen möglich sind!

ResourceBundleExportConfig.json

Für den Export der FHIR-Ressourcen an eine Ziel-URL, können verschiedene HTTP-Methoden für unterschiedliche FHIR-Ressourcen-Typen verwendet werden. Welche HTTP-Methoden für welchen Ressourcen-Typ zu verwenden ist, hängt vom Zielsystem ab. In der ResourceBundleExportConfig.json Konfigurationsdatei wird den einzelnen FHIR-Ressourcen-Typen eine HTTP-Methode zugeordnet. Diese Konfiguration kann während der Laufzeit geändert werden.

Die Konfiguration könnte wie folgt aussehen:

```
{
  "description": "...",
  "resourceTypeToHttpMethod": {
    "Patient": "PUT",
    "Specimen": "PUT",
    "Condition": "PUT",
    "Encounter": "PUT",
    "Observation": "PUT",
    "Procedure": "PUT",
    "Organization": "PUT",
    "MedicationStatement": "PUT",
    "ClinicalImpression": "PUT"
  }
}
```

Ist ein Ressourcen-Typ nicht konfiguriert, so wird "PUT" als Voreinstellung verwendet. CentraXX nutzt "PUT" zum Anlegen, oder Updaten über die "logical id" einer FHIR-Ressource. "POST" wird zum Anlegen, oder Updaten über den "natural identifier" (nur Patient, Specimen) genutzt.

Nutzung der Groovy-DSL-Bibliothek

Einleitung/Überblick

DSL steht für "domain specific language". Die DSL stellt eine intuitive benutzbare Sprache dar, um Probleme einer spezifischen Domäne lösen zu können. Die Kairos-FHIR-DSL wurde mit dem Ziel entwickelt, möglichst einfach Transformation-Skripte zu schreiben, mit denen CentraXX-Entitäten auf benutzerspezifische FHIR-Profile abgebildet werden können. Somit wird ein standardisierter Datenaustausch zwischen CentraXX und anderen FHIR-basierten Systemen ermöglicht. Die Kairos-FHIR-DSL ist in der Programmiersprache [Groovy](#) realisiert. Dementsprechend sind auch die Transformation-Skripte in Groovy zu schreiben. Um mit der DSL arbeiten zu können sind grundlegende Kenntnisse von Groovy/Java vorauszusetzen.

Beispiel-Skripte für die Transformation von CentraXX-Entitäten zu FHIR-Profilen können in Form eines Beispielprojekte heruntergeladen werden. Ausschnitte aus diesen Skripten werden im weiteren Verlauf der vorliegenden Dokumentation exemplarisch erläutert.

Für folgende FHIR-Projekte werden Beispiel-Skripte zur Verfügung gestellt:

- [DKTK](#)
- [BBMRI](#)
- CentraXX zu CentraXX
- [GECCO](#)

Die Kairos-FHIR-DSL-Bibliothek ist im Beispielprojekt bereits eingebunden. Selbstverständlich kann sie aber auch in eigene Groovy-Projekte eingebunden werden. Zum Erstellen und Editieren der Groovy-Skripte unter zur Hilfenahme von Codecompletion und Syntaxhighlighting werden entsprechende IDE Umgebungen benötigt. Der Download und die Installation aller benötigten Programme werden folgend beschrieben.

Voraussetzungen zur Nutzung der Kairos-FHIR-DSL Bibliothek und des Beispielprojekts

- IDE mit Maven und Groovy Support, z.B. IDEA IntelliJ, Eclipse, etc.
- JDK ab 1.8
- Maven

IntelliJ: Um die Groovy-Transformationsskripte zu erarbeiten, wird eine Entwicklungsumgebung genutzt. Im Folgenden wird die Einrichtung mit IDEA IntelliJ beschrieben. Die derzeit kostenfreie Community-Edition ist hierfür ausreichend.

Den Installer können Sie auf der IntelliJ Webseite herunterladen (<https://www.jetbrains.com/idea/>)

JDK: Das Beispielprojekt wurde entwickelt und getestet mit Adopt OpenJDK 11 und Compiler Level 1.8

Maven: IntelliJ selbst kommt mit einem vorinstallierten Maven. Damit Maven über die Kommandozeile genutzt werden kann, was zur Installation der Kairos-FHIR-DSL Bibliothek für eigene Groovy-Projekte sehr hilfreich ist, muss es zum "Path" hinzugefügt werden. Fügen Sie dazu der Systemvariable "Path" den Pfad "[IntelliJ]\plugins\maven\lib\maven3\bin" hinzu. Es kann auch ein eigenes Maven installiert werden. Maven kann unter dem folgenden Link heruntergeladen werden: <https://maven.apache.org/download.cgi>. Maven wird nach folgender Anleitung installiert: <https://maven.apache.org/install.html>.

Konfiguration eines benutzerspezifischen Maven:

Im Falle, dass nicht das Maven verwendet werden soll, welches mit IntelliJ vorinstalliert wird, kann dieses in IntelliJ einfach geändert werden. Dies lässt sich unter **File Settings Build, Execution, Deployment Build Tools Maven**, konfigurieren. An dieser Stelle muss der "maven home path" auf "[custom-maven]\bin" gesetzt werden.

Konfiguration des JDK 8 als Projekt-JDK:

Um das benötigte JDK 8 für das Projekt zu nutzen, muss dieses in IntelliJ konfiguriert werden. Gehen Sie dazu unter **File Project Structure... Project** und setzen Sie hier unter "Project SDK" den Pfad auf das Adopt-JDK 8. Wenn der Standard-Pfad bei der Installation nicht geändert wurde, dann sollte "C:\Program Files\AdoptOpenJDK\jdk-8.0.275.1-hotspot" gewählt werden.

Installation des FHIR-Beispielprojekts

Es existiert ein Beispiel-Projekt, welches Groovy-Mappings für verschiedene Projekte bereithält. Diese umfassen momentan Mappings für die Projekte DTK, BBMRI (GBA/GBN) und CXX (CentraXX zu CentraXX-Synchronisation per FHIR). Dieses Projekt wird auf [Github](#) verwaltet. Der aktuelle Release kann ebenfalls von Github heruntergeladen werden. Gehen Sie dazu auf folgende [Seite](#). Hier sehen Sie eine Liste mit allen Releases. Downloaden Sie den Source Code des neusten Releases und entpacken Sie die heruntergeladene Zip-Datei in das Verzeichnis, in dem Ihr Projekt liegen soll. Nun kann das Projekt in IntelliJ geöffnet werden. Gehen Sie dazu unter **File New Project from Existing Sources...** und geben Sie den Pfad [kairos-fhir-mapping-example]/pom.xml an und erstellen Sie das Projekt. Damit Maven die Kairos-FHIR-DSL-Bibliothek, die auch auf Github bereitsteht, ins Projekt einbeziehen kann, muss die "settings.xml" Datei konfiguriert werden. Dazu gehen Sie unter **File Settings Build, Execution, Deployment Build Tools** und klicken "Maven". Im geöffneten Fenster setzen Sie bei "User settings file" im Kästchen "Override" den Haken und wählen dort folgenden Pfad aus: [Projekt-Root]/settings.xml. Falls Sie ein eigenes Maven im Projekt verwenden wollen, konfigurieren Sie dieses im Feld "Maven home path". Klicken Sie "Apply" und dann "Ok". Gehen Sie ins in Maven-Fenster (rechts) unter **[Projektname] Lifecycle Install** oder führen Sie **\$ mvn install -s <path to settings.xml>** im Terminal aus. Nachdem dieser Vorgang abgeschlossen ist, klicken Sie im Maven-Fenster oben links den "Reload All Maven Projects"-Button, damit maven alle notwendigen Plugins lädt. Gegebenenfalls muss noch das Projekt-JDK, wie oben beschrieben, konfiguriert werden.

Installation der FHIR-DSL-Bibliothek

Die Kairos-FHIR-DSL-Bibliothek wird mittels Maven über Github installiert. Erstellen Sie dazu ein neues Projekt in IntelliJ. Im Dialog muss links Groovy ausgewählt werden. Im nächsten Fenster wird das Projektverzeichnis konfiguriert. Nachdem das Projekt initialisiert wurde, muss Maven als Framework hinzugefügt werden. Dazu rechtsklicken Sie im Projekt-Fenster das Projektverzeichnis-Root-Element und wählen "Add Framework Support...". Im neuen Fenster setzen Sie den Haken bei "Maven" und bestätigen die Eingabe mit "OK". Es wird eine "pom.xml" Datei erstellt. Innerhalb dieser müssen folgende Einträge hinzugefügt werden, sodass die Kairos-FHIR-DSL-Bibliothek in das Projekt eingebunden werden kann:

```
<dependencies>
  <dependency>
    <groupId>de.kairos</groupId>
    <artifactId>kairos-fhir-dsl</artifactId>
    <version>1.6.0</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>github-kairos-fhir</id>
    <name>GitHub kairos-fhir Apache Maven Packages</name>
    <url>https://maven.pkg.github.com/kairos-fhir/kairos-fhir-dsl-mapping-example</url>
  </repository>
</repositories>
```

Der obere Eintrag spezifiziert die Dependency und der untere das Repository, wo diese zu finden ist. Github erfordert bislang für den automatischen Dependency-Download per Maven immer eine Authentifizierung über einen Access-Token. Um die Arbeit zu erleichtern wurde ein allgemeiner Access-Token im [Github-Projekt](#) als "settings.xml" mit veröffentlicht. Legen Sie sich in Ihrem IntelliJ-Projekt ebenfalls eine eigene Settings-XML-Datei an und fügen Sie den Datei-Inhalt aus Github ein. Gehen Sie nun unter [File Settings Build, Execution, Deployment Build Tools](#) und klicken "Maven". Im geöffneten Fenster setzen Sie bei "User settings file" im Kästchen "Override" den Haken und wählen den Pfad zu der eben erstellten XML-Datei aus. Falls Sie ein eigenes Maven im Projekt verwenden wollen, konfigurieren Sie dieses im Feld "Maven home path". Klicken Sie "Apply" und dann "Ok". Nun gehen Sie rechts im Maven-Fenster und wählen [\[Projektnamen\] Lifecycle Install](#) oder führen sie `$ mvn install -s <path to custom settings xml>` im IntelliJ-Terminal aus.

Verwendung der DSL-Bibliothek zur Erstellung von Transformations-Skripten

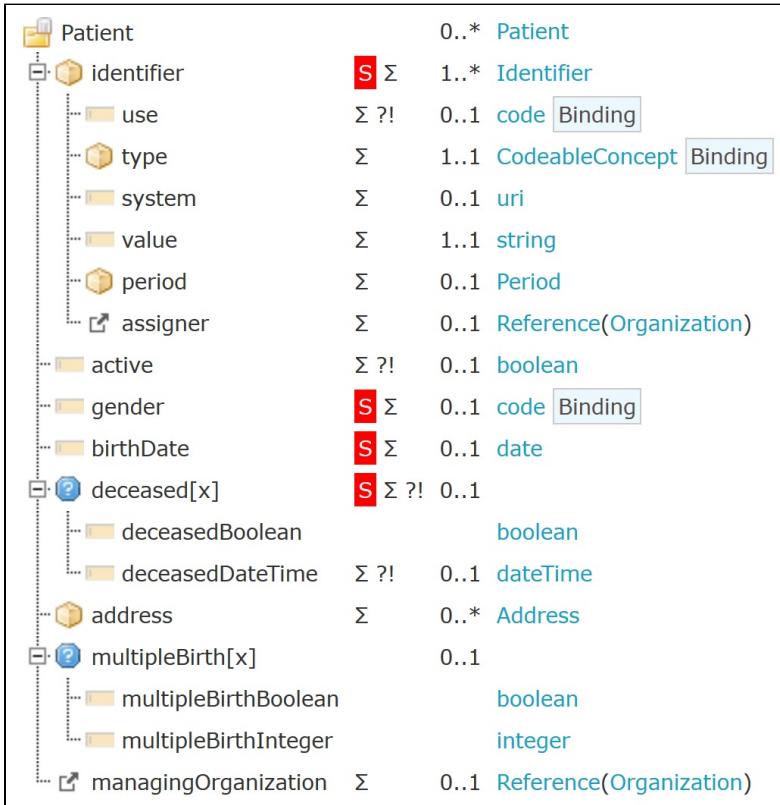
Allgemeines

Groovy bietet mit Closures ein Sprachelement, dass für die Entwicklung einer DSL maßgeblich ist. Closures sind Code-Blöcke, die in Groovy als Klasse behandelt werden und damit auch als Argumente an Methoden übergeben werden können. In Closures selbst können wiederum Methoden aufgerufen werden. Dies wird genutzt, um eine intuitive Syntax zur Abbildung von FHIR-Profilen zu bieten. Um bei Interesse Groovy-Closures und -Delegations besser verstehen zu können, wird die [Groovy-Dokumentation](#) des Apache Projekts empfohlen.

Über die Delegations-Strategie kann bestimmt werden, welche Methoden in einem Closure aufgerufen werden können. Beim Aufruf einer solchen Methode, wird zur Laufzeit versucht die entsprechenden Implementationen der im Closure aufgerufenen Methode über den Delegations-Mechanismus zu finden. Um als Nutzer die DSL verwenden zu können, wird die Information benötigt, welche Methoden in welchen Closures aufgerufen werden können. Dies wird bei der Entwicklung der DSL in Groovy, mittels Annotationen realisiert. Dadurch wird eine Auto-Code-Vervollständigung in der Entwicklungsumgebung ermöglicht.

Zusammenbau der FHIR Nachricht mit Groovy Closures

Im Folgenden soll das Prinzip des Closures und weiterer Groovy-Spezifitäten aufgeführt werden, die zur Erstellung von Transformations-Skripten genutzt werden können. Dazu wird das Groovy Skript "patient.groovy" betrachtet, dass im Beispielprojekt unter \src\main\java\projects\dktk\v2\patient.groovy vorzufinden ist. Dieses Skript bildet die CentraXX-Entität PatientMasterDataAnonymous auf das [DKTK FHIR-Profil "patient"](#) ab.



(Screengrab von <https://simplifier.net/oncology/patientenpseudonym>)

Die Kairos-FHIR-DSL repräsentiert die Struktur des FHIR-Profiles. Die in der Bibliothek bereitgestellten Methoden tragen die Namen der zugehörigen Elemente eines FHIR-Profiles. Es wird daher zunächst die äußere Methode "patient" aufgerufen. Diese akzeptiert ein Closure als Argument. Hier werden zuerst zwei generelle Elemente einer FHIR-Ressource festgelegt. Für das FHIR-Profil werden die Elemente "id" und "meta" gesetzt und deshalb können in diesem Closure die Methoden "id" und "meta" aufgerufen werden. Die **Basis-Definition einer FHIR-Resource** sieht folgendermaßen aus:

Name	Flags	Card.	Type	Description & Constraints	?
Resource	N	n/a		Base Resource	
id	Σ	0..1	id	Logical id of this artifact	
meta	Σ	0..1	Meta	Metadata about the resource	
implicitRules	?!	Σ	0..1 uri	A set of rules under which this content was created	
language		0..1	code	Language of the resource content Common Languages (Preferred but limited to AllLanguages)	

(Screengrab von <http://www.hl7.org/fhir/resource.html>)

Das Element "id" ist ein "Simple Type", es besitzt demnach einen Wert. Das Element "meta" ist wiederum ein "Complex Type" und kann folglich verschiedene Unterelemente besitzen. Im Detail sieht die Definition des komplexen Elements "meta" wie folgt aus:

Name	Flags	Card.	Type	Description & Constraints	?
Meta	Σ N		Element	Metadata about a resource Elements defined in Ancestors: id , extension	
versionId	Σ	0..1	id	Version specific identifier	
lastUpdated	Σ	0..1	instant	When the resource version last changed	
source	Σ	0..1	uri	Identifies where the resource comes from	
profile	Σ	0..*	canonical(StructureDefinition)	Profiles this resource claims to conform to	
security	Σ	0..*	Coding	Security Labels applied to this resource SecurityLabels (Extensible)	
tag	Σ	0..*	Coding	Tags applied to this resource Common Tags (Example)	

(Screengrab von <http://www.hl7.org/fhir/resource.html#Meta>)

Damit werden auch die Methodenaufrufe im Groovy-Skript ersichtlicher. Prinzipiell repräsentieren diese, die Struktur des FHIR-Profil.

```
patient {
    id = "Patient/" + context.source[patientMasterDataAnonymous().patientContainer().id()]
    meta {
        profile "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym"
    }
    ...
}
```

Im Closure, welches die Methode "patient" als Argument akzeptiert, dürfen die Methoden "id", "meta", "implicitRules" und "language" aufgerufen werden, da diese entsprechend in der Struktur-Definition des FHIR-Profil für eine Resource definiert sind. In IntelliJ kann das überprüft werden, indem ein gewünschter Element-Namen eingegeben wird. Dieser wird darauffolgend direkt von der Auto-Code-Vervollständigung angezeigt.

Im Beispiel wird außerdem dem Feld "id" ein String zugewiesen. In Groovy können Klassen sogenannte Properties besitzen, welchen mit dieser Syntax direkt ein Wert zugewiesen werden kann. In Java entspricht eine Property einem "private" Feld einer Klasse. Diese Zuweisung ist somit analog zum Aufruf der zugehörigen Setter-Methode. Groovy bietet für Properties von Klassen direkt Setter- und Getter-Methoden an, ohne das diese implementiert werden müssen. Die oben gezeigte Zuweisung kann alternativ wie folgt geschrieben werden:

```
patient {
    setId "Patient/" + context.source[patientMasterDataAnonymous().patientContainer().id()]
    ...
}
```

FHIR-Ressourcen, denen keine logische FHIR-Id zugewiesen wird, werden nicht der Bündel-Ressource hinzugefügt.

Der Ausdruck "context.source" wird weiter unten ausführlicher erläutert.

Das Element "meta" kann weitere Unterelemente besitzen und daher nimmt die zugehörige Methode wiederum ein Closure als Argument. In diesem Closure können nun laut FHIR-Profil-Definition die Methoden "versionId", "lastUpdated", "source", "profile", "security" und "tag" aufgerufen werden. Die Auto-Vervollständigung sollte in IntelliJ folgendermaßen aussehen:

```

16 patient {
17
18     id = "Patient/" + context.source[patientMasterDataAnonymous().patientContainer().id()]
19
20     meta {
21         profile profile: "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym"
22         ver|
23         P versionId String
24         m every() boolean
25         m every(Closure closure) booleanonymous().patientContainer().idContainer()].find {
26         m getVersionId() String 3: site specific
27         m setVersionId(String s) voidainer().idContainerType().code()
28     }
29
30     if (localId) {
31         identifier {
32             patient {
33
34                 id = "Patient/" + context.source[patientMasterDataAnonymous().patientContainer().id()]
35
36                 meta {
37                     profile profile: "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym"
38                     sec|
39                     m security(Closure<?> closure) void
40                     P security List<Coding>
41                     m setCommunication(List<PatientCommunicationComponent> list) voidus().patientContainer().idContainer()].find {
42                     m setContact(List<ContactComponent> list) voidte specific
43                     m setContained(List<Resource> list) void
44                     m setMetaClass(MetaClass metaClass) void
45                     m getSecurity() List<Coding>
46                     m setSecurity(List<Coding> list) void
47                 }
48             }
49         }
50     }
51 }

```

Prinzipiell werden die Methoden entsprechend des zu kreierenden FHIR-Profil ineinander geschachtelt. Methoden, die einem FHIR-Element von komplexem Typ entsprechen akzeptieren ein Closure als Argument, in welchem genau die korrespondierenden Methoden für die Unterelemente aufgerufen werden können. In der Abbildung oben rechts sehen Sie das Beispiel der Code-Vervollständigung für die Methode "security". Hat ein Element im FHIR-Profil nur einen Wert, so kann dieser direkt zugewiesen werden. Der entsprechende Aufruf ist in IntelliJ mit einem violett hinterlegten „p“ gekennzeichnet (z.B.: "versionId" in linker Abbildung). Im Falle, dass unter "meta" das Feld "versionId" mit "EXAMPLE-ID" festgelegt werden soll, so sieht der Code wie folgt aus:

```

patient {
    ...
    meta {
        profile "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym"
        versionId = "EXAMPLE-ID"
    }
}

```

Soll nun das komplexe Element "security" gesetzt werden, so muss ein Closure aufgerufen werden. "meta.security" ist vom Typ Coding. Die Struktur für ein "Coding" ist hier definiert. Des Weiteren ist für "Security" ein ValueSet definiert. Mit dieser Information kann nun die DSL genutzt werden, um das entsprechende Closure bspw. wie folgt aufzurufen:

```

patient {
    ...
    meta {
        profile "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym"
        versionId = "EXAMPLE-ID"
        security {
            system = "http://terminology.hl7.org/CodeSystem/v3-Confidentiality"
            code = "U"
        }
    }
}

```

Dies ist nur ein Beispiel, um den Methodenaufruf mit einem Closure zu demonstrieren. Im eigentlichen Anwendungsfall können selbstverständlich die zugewiesenen Werte von den Daten der CentraXX-Entität abhängen.

CXX JPA Meta Model und die Verwendung von context.source mit der JPA-Navigation

JPA Meta Model

Das Ziel der Mappings ist, die Daten einer CentraXX-Entität auf ein FHIR-Profil abzubilden. Dafür müssen die Daten vom Datenmodell der zu transformierenden CentraXX-Entität abgerufen werden können. Dazu wird der Ausdruck "context.source" verwendet. Das Objekt "context.source" ist eine Repräsentation der CentraXX-Entität in Form geschachtelter Hash-Maps. Eine Hash-Map ordnet einem Schlüssel (Key) einen Wert (Value) zu. Als Schlüssel wird jeweils der Name der Java JPA Property verwendet. Value ist entweder ein Simple Type (beispielsweise ein String), oder wiederum eine Hash-Map. Diese Struktur ermöglicht es in Groovy, Elemente in Form von geketteten Aufrufen, wie context.source[propertyX.propertyY.propertyZ] abzurufen. Im ersten Beispiel oben ist dieser Ausdruck bereits vorgekommen.

Navigation mittels verketteter Methodenaufrufe

Eine effiziente Nutzung der Datenabfrage über context.source ist nur möglich, wenn die Struktur des JPA-Meta-Models bekannt ist und man somit weiß, welche Keys die CentraXX-Entität überhaupt besitzt. Dies wird in der Kairos-FHIR-DSL durch die Implementation hierarchische Klassen realisiert, welche die Struktur des JPA-Meta-Models repräsentieren. Diese Repräsentation basiert auf einem String-Builder, welcher Methoden zu den möglichen JPA-Keys einer CentraXX-Entität anbietet. Somit wird eine Navigationsmöglichkeit bereitgestellt, um passende Keys für die jeweiligen Entitäten zu generieren und dafür Code-Vervollständigung zu nutzen. Die oberste Klasse "RootEntities" bietet hierzu statische Methoden zu allen verfügbaren CentraXX-Entitäten mit korrespondierendem Namen an. Bei der folgenden Zuweisung der Property "id", wird das Element "id" von context.source abgerufen.

```
id = "Condition/" + context.source["id"]
```

Mit Hilfe der JPA-Navigation kann dies folgendermaßen geschrieben werden:

```
id = "Condition/" + context.source[RootEntities.diagnosis().id()]
```

Mit "RootEntities.diagnosis()" werden die Navigationsmöglichkeiten für die CentraXX-Entität "Diagnosis" ausgewählt. Dieser Ausdruck kann weiter vereinfacht werden. In einem Groovy-Skript wird nur eine CentraXX-Entität transformiert und somit werden immer nur Werte von derselben Entität abgerufen. "RootEntities.diagnosis" kann somit als statischer Import hinzugefügt werden. Dazu rechtsklicken Sie in IntelliJ die Methode "diagnosis()" und gehen auf **Show Context Actions Add static import**. Somit kann der Ausdruck einfacher genutzt werden:

```
id = "Condition/" + context.source[diagnosis().id()]
```

Oder hier ein weiteres Beispiel:

```
subject {  
    reference = "Patient/" + context.source[diagnosis().patientContainer().id()]  
}
```

Es ist wichtig, sich klar zu machen, dass das "RootEntities" lediglich ein String-Builder ist, der je nach Aufruf seiner Methoden einen geketteten JPA-Schlüssel als String zurückgibt. Die Daten der CentraXX-Entität sind in context.source enthalten. Mit den Strings, können die entsprechenden Elemente von context.source abgerufen werden. Führt man folgendes Skript aus,

```

import de.kairos.fhir.centraxx.metamodel.RootEntities

println RootEntities.abstractSample().active().path()
println RootEntities.patientMasterDataAnonymous().patientContainer().idContainer().creator().username().path()

```

dann erhält man folgenden Output:

```

"active"
"patientcontainer.idContainer.creator.username"

```

Die resultierenden Strings enthalten nicht den Namen der Centraxx-Entität selbst (AbstractSample, PatientMasterDataAnonymous), da diese Information durch das Ressourcen-Mapping in context.source enthalten ist. Die Methode "path()" ist die Builder-Methode, die den String zurückgibt. Diese Methode muss nicht explizit aufgerufen werden, wenn der Ausdruck innerhalb des Subscript-Operators von context.source aufgerufen wird, da context.source den Aufruf der "path"-Methode selbst implementiert.

Verwendung von Konstanten in Listen-Closures

In manchen Fällen werden in Transformations-Skripten Listen von Elementen verarbeitet. Groovy-Listen bieten dafür verschiedene Funktionen an, welche ein Closure als Argument akzeptieren. Im folgenden Beispiel aus dem Skript /dktk/v2/specimen.groovy soll für jeden Eintrag im IdContainer ein FHIR-Element "identifier" festgelegt werden. Hierzu wird die Methode "each" verwendet:

```

context.source[abstractSample().idContainer()].each { final idc ->
    identifier {
        value = idc[PSN]
        type {
            coding {
                system = "urn:centraxx"
                code = idc[ID_CONTAINER_TYPE]?.getAt(CODE)
            }
        }
        system = "urn:centraxx"
    }
}

```

Zunächst wird der IdContainer abgefragt und mittels des ["?-Operators](#) geprüft, ob dieser überhaupt vorhanden ist. Beim Iterieren über die Elemente des IdContainers wird einer lokalen variable "idc" jeweils das Element zugewiesen. Im Closure der "each"-Methode werden nun wiederum Felder dieses Elements abgerufen. Jetzt befindet man sich allerdings nicht mehr in context.source, sondern greift jetzt auf die Felder des neuen Objekts "idc" zu. Das Objekt "idc" ist wiederum eine geschachtelte Map. Um zu wissen, welche Keys diese Map besitzen kann, muss man wissen, welcher Entität des CentraXX JPA-Model dieses Objekt entspricht. Dies findet man schnell heraus, wenn man sich anschaut, welchen Typ die Methode "idContainer()" zurückgibt. Dazu reicht es in IntelliJ den Cursor auf der entsprechenden Methode zu platzieren. Die Methode "idContainer()" gibt einen "SampleIdContainer" zurück. Jede Klasse der JPA-Repräsentation bietet die zugehörigen Keys als Konstanten an. Damit können nun die Felder von "idc" mit den passenden Keys abgerufen werden. Das Beispiel oben ist bereits mit statischen Importen vereinfacht. Die Aufrufe, die im Closure der "identifier"-Methode getätigten werden, sehen unvereinfacht wie folgt aus:

```

identifier {
    value = idc[SampleIdContainer.PSN]
    type {
        coding {
            system = "urn:centraxx"
            code = idc[SampleIdContainer.ID_CONTAINER_TYPE]?.getAt(IdContainerType.CODE)
        }
    }
    system = "urn:centraxx"
}

```

Auf diese Weise funktioniert auch die zweite Zuweisung. Dort muss man wissen, dass "idc [ID_CONTAINER_TYPE]" ein Objekt vom Typ "IdContainerType" zurückgibt. Hier stößt die Code-Vervollständigung allerdings an ihre Grenzen. Jedoch ist aus der Benamung der Klassen leicht zu erkennen, welcher Typ zurückgegeben wird. Alternativ kann man beim Skripten auch testweise den linearen Ausdruck schreiben und mittels der Code-Vervollständigung in IntelliJ den Return-Type herausfinden.

The screenshot shows a code editor with Java code. Line 46 contains the method call `abstractSample().idContainer().idContainerType()`. A tooltip appears over the method name, showing the full signature: `de.kairos.fhir.centraxx.metamodel.AbstractIdContainer public de.kairos.fhir.centraxx.metamodel.IdContainerType idContainerType()`. Below the tooltip, there is a Maven dependency entry: `Maven: de.kairos:kairos-fhir-dsl:1.6.0-SNAPSHOT`.

```

45 }
46 abstractSample().idContainer().idContainerType()
47 status = context.source[abstra
48 specimen]

```

Definition eigener Methoden, um CentraXX-Entitäten optimal abzubilden

Beispiel 1: Abbildung des CentraXX Gender auf das Gender des DKT-FHIR-Profil

Es treten Fälle auf, in denen die Daten einer CentraXX-Identität anders strukturiert sind als im zugehörigen FHIR-Profil. Dies hängt bspw. davon ab, welcher Satz an Werten zur Beschreibung eines bestimmten Patienten-Stammdatenpunkts in einer CentraXX-Instanz definiert ist. Betrachten wir z.B. das Element "Gender" im FHIR-Profil im Vergleich zur Definition in CentraXX. Für das FHIR-Profil ist folgender [ValueSet](#) definiert. Nachfolgend wird ein Beispiel für die Dokumentation des Geschlechts eines Patienten in CentraXX und in der FHIR-Ressource "Patient" des DKT-Projekts aufgeführt:

CentraXX	DKT-FHIR Resource "Patient"
männlich	male
weiblich	female
hermaphroditisch	other
unbekannt	unknown
unbestimmt	
sonstiges	
divers	

Um die Werte optimal zu ordnen zu können, kann eine eigene Methode definiert werden, um das CentraXX-Gender abzubilden. Folgend ein Ausschnitt aus /dktk/v2/patient.groovy:

```

patient {
    ...
    if (context.source[patientMasterDataAnonymous().genderType()]) {
        gender = mapGender(context.source[patientMasterDataAnonymous().genderType()])
    } as GenderType
}

```

Mittels if-Statement wird geprüft, ob für den zu transformierenden Patienten-Datensatz überhaupt ein Gender angegeben ist. Dazu wird "context.source" genutzt, um dieses von der CentraXX-Entität abzufragen. Ist das Gender definiert wird der Property "gender" der Wert mit der eigenen Methode "mapGender()" zugewiesen. Diese Methode ist folgendermaßen definiert:

```

static AdministrativeGender mapGender(final GenderType genderType) {
    switch (genderType) {
        case GenderType.MALE:
            return AdministrativeGender.MALE
        case GenderType.FEMALE:
            return AdministrativeGender.FEMALE
        case GenderType.UNKNOWN:
            return AdministrativeGender.UNKNOWN
        default:
            return AdministrativeGender.OTHER
    }
}

```

Mit einem Switch-Statement können die entsprechenden Fälle zugeordnet werden. Da die zusätzlichen Gender, wie bspw. "hermaphroditisch" nicht im FHIR ValueSet definiert sind, werden diese im Beispiel auf "Other" abgebildet. In diesem Fall bietet die FHIR-Bibliothek ein Enum "AdministrativeGender" an, in welchem die konstanten Werte des ValueSets definiert sind.

Beispiel 2: Setzen der "identifier" für DTK FHIR-Profil Patient:

Das Element "identifier" des FHIR-Profils wird folgendermaßen festgelegt. Der folgende Code ist erneut ein Ausschnitt aus dem Beispielskript /dtk/v2/patient.groovy:

```

patient {
    ...
    final def globalId = context.source[patientMasterDataAnonymous().patientContainer().idContainer()]??.find {
        "DKTK" == it[ID_CONTAINER_TYPE]?.getAt(CODE)
    }
    if (globalId) {
        identifier {
            value = globalId[PSN]
            type {
                coding {
                    system = "http://dtk.dkfz.de/fhir/onco/core/CodeSystem/PseudonymArtCS"
                    code = "Global"
                }
            }
        }
    }
    ...
}

```

Zuerst wird einer Variable "globalId" ein Wert zugewiesen. Groovy unterstützt sowohl statische als auch dynamische Typisierung. In diesem Fall wird "globalId" dynamisch typisiert. Sie nimmt also den Typ an, der ihr zu Laufzeit zugewiesen wird. In Groovy wird dazu das Keyword "def" verwendet. Mittels des "?"-Operator wird überprüft, ob überhaupt eine Property "patientcontainer.idContainer" vorhanden ist. Im Falle, dass diese vorhanden ist, so generiert context.source["patientContainer.idContainer"] eine Liste zurück. Diese Liste enthält wiederum geschachtelte Maps. In dieser Liste wird nach einem Element (also einer Map) gesucht, welches als Key die Property "idContainerType" besitzt. Da in CentraXX ein Patient prinzipiell mehrere IdContainer besitzen kann, wird das Ergebnis im .find Ausdruck weiter nach dem jeweiligen idContainertype mit dem "code"="DKTK" gefiltert, um zu einem eindeutigen Ergebnis zu gelangen. Ist dies wiederum der Fall, so wird die Variable "globalId" mit dem Ergebnis, der Map, welche den IdContainer representiert, belegt. Wird keine passende Map gefunden bleibt "globalId" gleich null.

Der zweite Code-Abschnitt wird nur ausgeführt, wenn „globalId“ ungleich null ist, sprich eine entsprechende Map existiert. Hierbei wird die Methode "identifier" des FHIR-Modells für den Zusammenbau eines Patienten aufgerufen. Dem Element "value" in "identifier" wird nun der Value aus der Map zum Key der Property "psn", der vorher definierten Variablen "globalId" zugewiesen.

Abfrage von Messwerten mittels JPA-Navigation

In CentraXX verwaltete Messbefunde können transformiert und exportiert werden. Folgendes Beispiel stammt aus dem Skript /bbmri/observationBodyHeight.groovy:

```
final def bodyHeightLfLv = context.source[laborMapping().laborFinding().laborFindingLaborValues()].find {
    "BodyHeightValue" == it[LaborFindingLaborValue.LABOR_VALUE]?.getAt(LaborValue.CODE)
}

if (bodyHeightLfLv) {
    valueQuantity {
        value = bodyHeightLfLv[LaborFindingLaborValue.NUMERIC_VALUE]
        unit = "cm"
        system = "http://unitsofmeasure.org"
        code = "cm"
    }
}
```

Dem Feld "valueQuantity" der FHIR-Ressource wird die Körpergröße des Patienten zugewiesen. Hierzu wird die Patientenakte für einen passenden Laborwert (LaborFindingLaborValue) durchsucht. Eine Patientenakte kann verschiedene Labormesswerte enthalten (z.B. Körpergewicht, Körpergröße, Blutdruck, etc.). Ein solcher Messwert ist im JPA-Meta-Model wiederum eine Hashmap. Diese besteht aus dem Messwert selbst und dem assoziierten Messparameter. Oben wird der Messwert per JPA-Navigation mittels "LaborFindingLaborValue.NUMERIC_VALUE" abgerufen. Es muss beim Abrufen des Messwertes die zum Datentyp passende Konstante wählen. Die Definition des Messparameters in CentraXX muss daher bekannt sein. Die weiteren Unterelemente von "valueQuantity" werden in diesem Beispiel explizit gesetzt. Dazu kann aber auch die JPA-Navigation abgerufen werden:

```
if (bodyHeightLfLv) {
    valueQuantity {
        value = bodyHeightLfLv[LaborFindingLaborValue.NUMERIC_VALUE]
        unit = bodyHeightLfLv[LaborFindingLaborValue.LABOR_VALUE]?.getAt(LaborValueInteger.UNIT)
        system = "http://unitsofmeasure.org"
        code = bodyHeightLfLv[LaborFindingLaborValue.LABOR_VALUE]?.getAt(LaborValueInteger.CODE)
    }
}
```

Das Element "laborMapping().laborFinding().laborFindingLaborValues().laborValue()" enthält den Messparameter. Die Messparameter können abhängig vom Typen verschiedene Eigenschaften besitzen. Der hier abgerufene Messparameter Körpergröße ist in CentraXX als Ganzzahl definiert. Daher besitzt dieser Messwert eine Einheit und kann darüber hinaus noch eine Ober- und Untergrenze besitzen. Andere Messparameter können andere Messwert-Datentypen (Boolean, Dezimalzahl, Enumeration, String) und besitzen teils andere Eigenschaften. Ein Messparameter mit dem Type Boolean hat beispielsweise keine Einheit und auch keine Ober-und-Untergrenze. Um per JPA-Navigation die spezifischen Attribute gezielt abrufen zu können, werden daher entsprechende JPA-Klassen (LaborValueBoolean, LaborValueDecimal, LaborValueEnumeration, LaborValueString) angeboten. Diese Klassen sind aus der Oberklasse "LaborValue" angeboten. Im Beispiel wird daher in der "getAt"-Funktion "LaborValueInteger" als Basis-JPA-Element verwendet. Sollen nur Elemente abgerufen werden, die alle Messparameter besitzen, kann man natürlich auch weiter auf der Oberklasse arbeiten. Jeder Messparameter besitzt einen Code:

```
code = bodyHeightLfLv[LaborFindingLaborValue.LABOR_VALUE]?.getAt(LaborValue.CODE)
```

Weitere Selektion mittels bedingtem Return

In CentraXX werden alle Daten patientenorientiert verwaltet. Messwerte sind dabei unter der Entität LABOR_MAPPING organisiert. In folgendem Skript-Ausschnitt (/projects/gecco /observationBloodPressure) wird eine FHIR-Ressource für den Messwert Blutdruck angelegt. Dieser Messwert ist in CentraXX Teil eines Messprofils (LaborMethod). Es ist sinnvoll, das Skript nur dann auszuführen, wenn das entsprechende Messprofil tatsächlich vorhanden ist.

```

observation {

    if (context.source[laborMapping()].laborFinding().laborMethod().code() != "BLOODGASPROFILE") {
        return // no export
    }

    id = "OxygenSaturation/" + context.source[laborMapping()].laborFinding().id()

    meta {
        profile "https://www.netzwerk-universitaetsmedizin.de/fhir/StructureDefinition/oxygen-saturation"
    }
}
...
}

```

Existiert das Messprofil mit dem Code "BLOODGASPROFILE" nicht, dann bricht das Skript ab und es erfolgt kein Export.

Das gezeigte Skript ist ein Arbeitsbeispiel. Die Codes müssen gemäß der Definitionen von Messprofilen, Messparametern etc. in Ihrer CentraXX-Instanz gewählt werden.

FHIR-Export in Zielsysteme

Für die folgenden Beispiele wurde ein Projekt-Verzeichnis unter "[centraxx-home]\fhir-custom-mappings\dktk" angelegt. Innerhalb dieses wurden alle Groovy-Skripte aus dem Beispielprojekt (s. oben) abgelegt.

<input type="checkbox"/>	Name	Größe	Dateierweiterung
<input checked="" type="checkbox"/>	tnmp	5 KB	.groovy
<input checked="" type="checkbox"/>	specimen	5 KB	.groovy
<input checked="" type="checkbox"/>	ExportResourceMappingConfig	4 KB	.json
<input checked="" type="checkbox"/>	ProjectConfig	4 KB	.json
<input checked="" type="checkbox"/>	verlauf	3 KB	.groovy
<input checked="" type="checkbox"/>	vitalstatus	3 KB	.groovy
<input checked="" type="checkbox"/>	systemtherapie	3 KB	.groovy
<input checked="" type="checkbox"/>	patient	3 KB	.groovy
<input checked="" type="checkbox"/>	grading	3 KB	.groovy
<input checked="" type="checkbox"/>	histologie	2 KB	.groovy
<input checked="" type="checkbox"/>	fermnetastasen	2 KB	.groovy
<input checked="" type="checkbox"/>	tumorstatusMetas	2 KB	.groovy
<input checked="" type="checkbox"/>	tumorstatusGesamt	2 KB	.groovy
<input checked="" type="checkbox"/>	tumorstatusLymph	2 KB	.groovy
<input checked="" type="checkbox"/>	tumorstatusLokal	2 KB	.groovy
<input checked="" type="checkbox"/>	fall	2 KB	.groovy
<input checked="" type="checkbox"/>	operation	2 KB	.groovy
<input checked="" type="checkbox"/>	residualstatusLokal	2 KB	.groovy
<input checked="" type="checkbox"/>	residualstatusGlobal	2 KB	.groovy
<input checked="" type="checkbox"/>	condition	2 KB	.groovy
<input checked="" type="checkbox"/>	strahlentherapie	2 KB	.groovy
<input checked="" type="checkbox"/>	BundleRequestMethodConfig	1 KB	.json
<input checked="" type="checkbox"/>	organisation	1 KB	.groovy

Die Konfigurationsdatei ExportResourceMappingConfig.json wurde für dieses Beispiel auf dem Standard-Zustand belassen (so wie sie nach dem ersten Start von CentraXX nach Anlegen des Projekt-Ordners von CentraXX erstellt wurde).

Es wurden drei Testpatienten in CentraXX angelegt. Des Weiteren wurde die Patienten-ID mit dem Code "DKTK" in CentraXX definiert. Diesen drei Testpatienten wurde eine solche "DKTK"-ID zugewiesen. Somit kann ein passender Patienten-Filter konfiguriert werden:

```
"patientFilterType": {
  "description": "...",
  "value": "IDTYPE"
},
"patientFilterValues": {
  "description": "The filter value.",
  "value": [ "DKTK" ]
},
```

Zum Testen ist es hilfreich, wenn Ausleitungen zeitnah stattfinden. Dazu wird für die folgenden Beispiele ein Exportintervall auf alle fünf Minuten konfiguriert:

```
"exportInterval": {
  "description": "Sets the export cron interval. Default is once a day.",
  "value": "0 */5 * * *"
},
```

Ein Beispiel für ein ausgeleitetes FHIR-Bundle der FHIR-Ressource Patient finden Sie im Anhang.

Ausleitung in einen Blaze Store

FHIR-Ressourcen können in einen Blaze-Store ausgeleitet werden. Für dieses Beispiel wird das "standalone JAR"-Blaze verwendet. Hinweise zur Installation finden Sie unter <https://alexanderkiel.gitbook.io/blaze/> und unter <https://github.com/samplify/blaze>. Zusätzlich wird ein weiteres Command-Line-Tool verwendet, mit dem im nachfolgenden Beispiel die Anzahl der Ressourcen im Blaze-Store abgefragt werden können. Hinweise zur Installation und Nutzung von Blaze-CTL finden Sie unter <https://github.com/samplify/blazectl>.

Blaze bietet eine REST-full API standardmäßig unter `http://localhost:8080/fhir` an. Für einen Export in den Blaze-Store müssen die Konfigurationen in ProjectConfig.json (spezifische Erläuterung im Abschnitt "Einrichtung eines Projekts") angepasst werden. Der Export per HTTP muss aktiviert und die Adresse des Zielsystems festgelegt werden.

```
"uploadToUrl": {
  "description": "If true, the scheduler will export to the specified uploadUrl REST endpoint.",
  "value": true
},
"uploadUrl": {
  "value": "http://localhost:8080/fhir"
},
```

Außerdem werden für diesen Test alle Zertifikate zugelassen:

```
"sslTrustAllEnable": {
  "description": "If true, all ssl/tls certificates are trusted, otherwise the default java store is used for SSL/TLS verification.",
  "value": true
},
```

Für erste Testzwecke ist es sinnvoll das Login der HTTP-Antwort des Zielservers zu aktivieren:

```
"prettyPrintEnable": {
  "description": "Print style the HTTP message body.",
  "value": true
},
"fhirResponseValidationEnable": {
  "description": "If true, the HTTP message responses is parsed and each response entry is validated against its FHIR response status. Even if the HTTP response status is 200 ok, the status of a single entry might be 400 (bad request). If the status is not 200 or 201, an error is logged.",
  "value": true
}
```

Die hier nicht aufgeführten Konfigurationen werden bei dem Standard-Werten belassen. Der Patientenfilter und das Ausleitungsintervall sind, wie oben beschrieben, konfiguriert.

Die Konfigurationsdatei BundleRequestMethodConfig.json kann auf dem Standard-Zustand belassen werden. Standardisiert werden für alle FHIR-Ressourcen-Typen die PUT-Methode verwendet. Da Blaze ausschließlich PUT akzeptiert, bedarf es hier keiner Modifizierung.

CentraXX muss nun neu gestartet werden, um den neu konfigurierten Export zu aktivieren. Davor kann bereits Blaze selbst gestartet und eine erste Abfrage über die Anzahl der Ressourcen in Blaze getätigt werden. Dazu muss der folgende Befehl in einer Windows-PowerShell ausgeführt werden:

```
PS C:\Users\          > blazectl --server http://localhost:8080/fhir count-resources
Count all resources on http://localhost:8080/fhir ...
-----
total : 0
```

Wie erwartet sind noch keine FHIR-Ressourcen in Blaze vorhanden. Nachdem CentraXX wieder gestartet ist, sollten bei einer erfolgreichen Konfiguration Einträge zum Export im Serverlog erhalten werden. Als Beispiel werden die Einträge für einen Export eines CentraXX-Patienten aufgeführt:

```
2021-01-14 20:25:01,503 INFO [de.kairos.centraxx.timer.NfnExportScheduler] (cxxScheduledTask-19) Select CXX
EntityType 'PATIENT_MASTER' transform by template 'vitalstatus' export to FHIR resource 'Observation'
2021-01-14 20:25:01,565 INFO [de.kairos.centraxx.fhir.r4.nfn.NfnExportManager] (cxxScheduledTask-19) Export
bundle with '3' resources to target system 'http://localhost:8080/fhir'
2021-01-14 20:25:01,568 INFO [de.kairos.centraxx.fhir.r4.nfn.NfnClient] (cxxScheduledTask-19) Cxx POST request: http://localhost:8080/fhir
2021-01-14 20:25:01,608 INFO [de.kairos.centraxx.fhir.r4.nfn.NfnClient] (cxxScheduledTask-19) Cxx response: 200
```

Für diesen Test sind in CentraXX genau drei Patienten vorhanden. Es existieren noch weitere Entitäten, welche exportiert wurden. Eine erneute Abfrage der Anzahl von FHIR-Ressourcen in Blaze ergibt das folgende Ergebnis:

```
PS C:\Users\          > blazectl --server http://localhost:8080/fhir count-resources
Count all resources on http://localhost:8080/fhir ...
-----
Condition : 1
Encounter : 1
Observation : 3
Organization : 1
Patient : 3
-----
total : 9
```

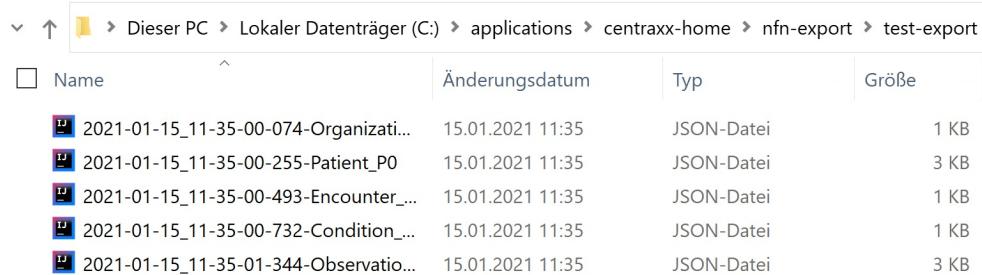
Ausleitung ins Dateisystem

Die aus dem Transformations-Vorgang entsprungenen FHIR-Ressourcen können in Form von JSON-Dateien in das lokale Dateisystem ausgeleitet werden. Dazu müssen verschiedene Konfigurationen in der ProjectConfig.json vorgenommen werden. Der Export muss aktiviert und ein Zielverzeichnis festgelegt werden. Dieses Zielverzeichnis wird neu angelegt, im Falle, dass es nicht bereits existiert:

```
"exportToFileSystem": {
  "description": "If true, the scheduler will export to the specified file system exportFolder.",
  "value": true
},
"exportFolder": {
  "value": "C:/applications/centraxx-home/fhir-custom-export/test-export"
},
```

Darüber hinaus relevante Konfiguration wie "patientFilterType", "patientFilterValue", "exportInterval" sind, wie oben beschrieben, vorgenommen worden.

Nach Neustart von CentraXX werden die JSON-Dateien im angegebenen Verzeichnis angezeigt:



The screenshot shows a Windows File Explorer window with the following path: Dieser PC > Lokaler Datenträger (C:) > applications > centraxx-home > nfn-export > test-export. The table lists five JSON files with their names, last modified date, type, and size.

<input type="checkbox"/>	Name	Änderungsdatum	Typ	Größe
	2021-01-15_11-35-00-074-Organizati...	15.01.2021 11:35	JSON-Datei	1 KB
	2021-01-15_11-35-00-255-Patient_P0	15.01.2021 11:35	JSON-Datei	3 KB
	2021-01-15_11-35-00-493-Encounter_...	15.01.2021 11:35	JSON-Datei	1 KB
	2021-01-15_11-35-00-732-Condition_...	15.01.2021 11:35	JSON-Datei	1 KB
	2021-01-15_11-35-01-344-Observatio...	15.01.2021 11:35	JSON-Datei	3 KB

Anhang

Beispiel für ausgeleitetes FHIR-Bundle für die FHIR-Ressource "Patient"

```
{
  "resourceType": "Bundle",
  "type": "transaction",
  "total": 3,
  "entry": [
    {
      "fullUrl": "Patient/1",
      "resource": {
        "resourceType": "Patient",
        "id": "1",
        "meta": {
          "versionId": "EXAMPLE-ID",
          "profile": [ "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym" ],
          "security": [ {
            "system": "http://terminology.hl7.org/CodeSystem/v3-Confidentiality",
            "code": "U"
          } ]
        },
        "identifier": [ {
          "type": {
            "coding": [ {
              "system": "http://dktk.dkfz.de/fhir/onco/core/CodeSystem/PseudonymArtCS",
              "code": "Global"
            } ]
          },
          "value": "DKTK-43134"
        }],
        "gender": "female",
        "birthDate": "2020-01-01"
      },
      "request": {
        "method": "PUT",
        "url": "Patient/1"
      }
    },
    {
      "fullUrl": "Patient/2",
      "resource": {
        "resourceType": "Patient",
        "id": "2",
        "meta": {
          "versionId": "EXAMPLE-ID",
          "profile": [ "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym" ],
          "security": [ {
            "system": "http://terminology.hl7.org/CodeSystem/v3-Confidentiality",
            "code": "U"
          } ]
        },
        "identifier": [ {
          "type": {
            "coding": [ {
              "system": "http://dktk.dkfz.de/fhir/onco/core/CodeSystem/PseudonymArtCS",
              "code": "Global"
            } ]
          },
          "value": "DKTK-98521"
        }],
        "gender": "male",
        "birthDate": "1956-01-01"
      },
      "request": {
        "method": "PUT",
        "url": "Patient/2"
      }
    },
    {
      "fullUrl": "Patient/11",
      "resource": {
        "resourceType": "Patient",
        "id": "11",
        "meta": {
          "versionId": "EXAMPLE-ID",
          "profile": [ "http://dktk.dkfz.de/fhir/StructureDefinition/onco-core-Patient-Pseudonym" ],
          "security": [ {
            "system": "http://terminology.hl7.org/CodeSystem/v3-Confidentiality",
            "code": "U"
          } ]
        }
      }
    }
  ]
}
```

```
},
"identifier": [ {
  "type": {
    "coding": [ {
      "system": "http://dktk.dkfz.de/fhir/onco/core/CodeSystem/PseudonymArtCS",
      "code": "Global"
    } ]
  },
  "value": "DKTK-42313"
} ],
"gender": "other",
"birthDate": "1993-01-01",
"deceasedDateTime": "2021-01-01"
},
"request": {
  "method": "PUT",
  "url": "Patient/11"
}
} ]
}
```