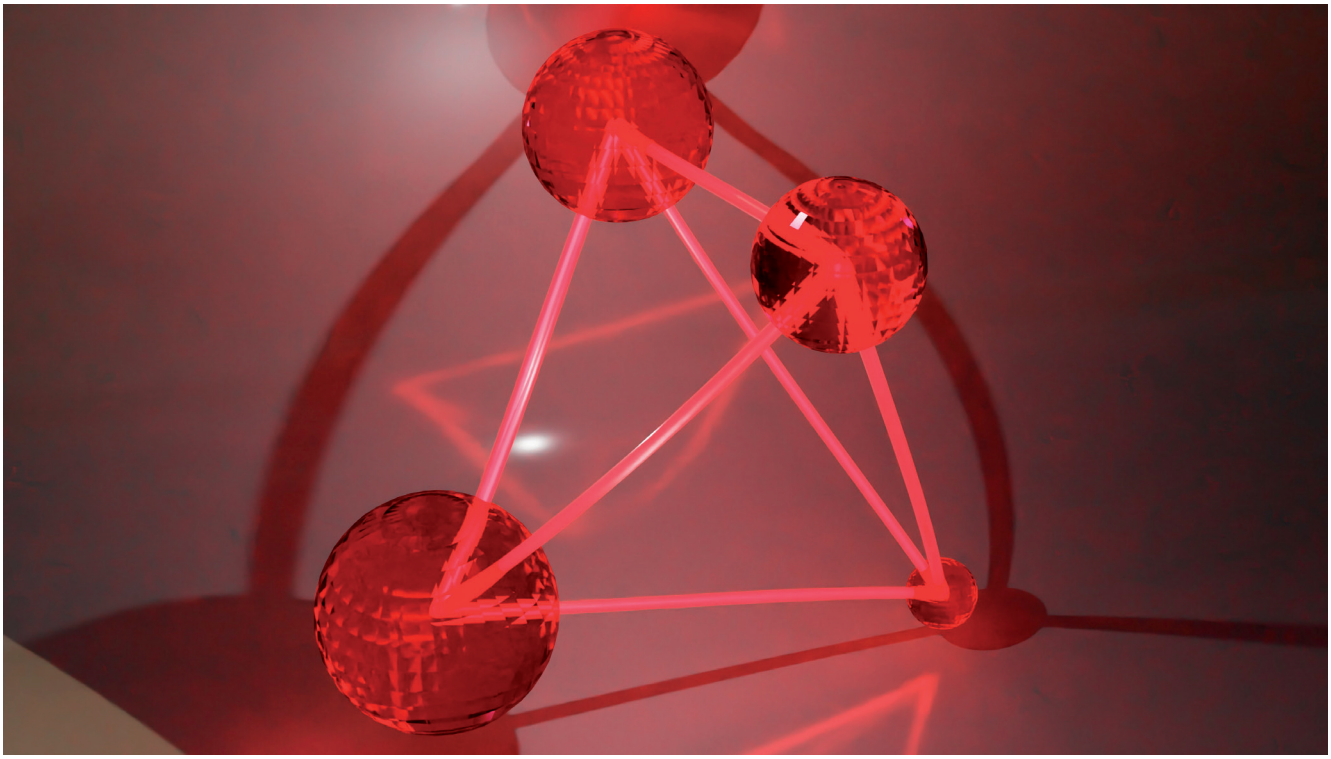


SEMANTISCHES WEB

Knowledge-Graphen im Praxiseinsatz

Dieser Artikel zeigt, wie Sie semantische Datenmodelle und Graphdatenbanken einfach in bestehende Applikationen integrieren – mögen die Hürden auch zunächst hoch erscheinen.

Knowledge-Graphen versprechen Daten aus den vielen heutzutage in heterogenen IT-Landschaften eingesetzten Anwendungen zusammenzuführen, Informationen zu harmonisieren, mit Metadaten und Beziehungen daraus Wissen zu generieren und dieses dann für Analyse- und Empfehlungssysteme, für smarte digitale Assistenten oder künstliche Intelligenz bereitzustellen.

Als Teil von Digitalisierungsstrategien sind Knowledge-Graphen nicht nur ein willkommenes Instrument für den Aufbau von Wissen als internes Enterprise-Asset, sondern auch für innovative, datengetriebene Geschäftsmodelle. Dabei sind die zugrunde liegenden semantischen Technologien nicht nur technisch spannend, sondern durch den stetig wachsenden Hunger der Anwender nach mehr intelligenten Lösungen für uns Entwickler auch ein nachhaltig lukratives Thema.

Klingt die Vision doch attraktiv, bestehende Daten zu semantifizieren, Informationen eine Bedeutung zu geben und menschliches Wissen zu digitalisieren, sodass dieses anschließend unabhängig von Zeit und Ort von Menschen und Maschinen wiederverwendet werden kann. Basiswissen mit Fachwissen unterschiedlicher Domänen nachvollziehbar in Knowledge-Graphen zu kombinieren, statt es in Code zu verstecken, aus dem es schwerlich wieder extrahierbar ist. Letztlich Best Practices nur an einer Stelle zu entwickeln oder Fehler nur an einer Stelle zu beheben und diese Errungenschaften dann mithilfe intelligenter Agenten effizient in Code und Daten für unsere Apps umsetzen zu lassen.

Wären da nicht die gefühlt so umfänglichen Einstiegshürden mit neuen Akronymen wie RDF, RDFS oder OWL, Sprachen wie SPARQL, SHACL oder RML, Strukturen wie Taxonomien oder Ontologien, Linked-Open-Data (LOD), Triple- und Quad-Stores, die sich so gänzlich unterscheiden von den bekannten relationalen Datenbanksystemen. Ganz zu schweigen von der Abkehr von den bekannten Tabellen und Kollektionen aus den traditionellen SQL- und NoSQL-Datenbanken oder den bekannten Konzepten der Data-Stores wie

Redis und Elasticsearch hin zu Aussagen (Statements), Behauptungen (Assertions) und Fakten (Axioms), die wieder einmal unsere Lernkurven zu erhöhen drohen.

Wenngleich die Erfindung des semantischen Webs durch Tim Berners-Lee schon mehr als 20 Jahre zurückliegt und dieses längst durch das W3C standardisiert ist [1], liegt noch immer dieser wissenschaftliche Schleier über Begriffen wie Semantik und Graphen-Theorie oder Technologien wie Inferenz und Reasonern. Dies lässt Fragen nach dem Nutzen und Implementierungsaufwand für Knowledge-Graphen ebenso aufkommen wie Zweifel am Reifegrad und an der Praxistauglichkeit aktueller Werkzeuge.

Was sind eigentlich Knowledge-Graphen?

In diesem Artikel wollen wir genau diese Zweifel ausräumen, indem wir zunächst einen Überblick über die diversen Aspekte von Knowledge-Graphen geben und anschließend einige komfortable Tools vorstellen, die den Einstieg in die Entwicklung semantischer Applikationen schmackhaft machen.

Knowledge-Graphen werden im Sinne des Linked-Open-Data-Konzepts verstanden als Integration verteilter Wissensquellen, technisch meist betrieben in Form einer oder mehrerer föderierter Graphdatenbanken. Diese enthalten Modelle, die Klassen-Bäume, auch als Taxonomien oder T-Box bezeichnet, oder Ontologien, wiederum bestehend aus einer T-Box und aus einer A-Box, die letztlich die Individuals, die eigentlichen Datensätze, die Instanzen eines Knowledge-Graphen enthält.

Gängige Praxis ist dabei, dass Taxonomien und Ontologien durch Modellierungswerkzeuge wie zum Beispiel das Open-Source-Tool Protégé [2] serialisiert in Dateiform bereitgestellt und dann in die Graphdatenbank importiert werden. Dort werden dann die Individuals via SPARQL [3] erstellt und gemanagt oder via RML [4] aus bestehenden Quellen wie CSV, JSON oder XML transformiert und via SHACL [5] als Validierungssprache geprüft.

Per se sind Graphdatenbanken erst einmal schemalos. Das bedeutet, dass Individuals beliebige Properties enthalten können und diese wiederum beliebige Werte oder Referenzen. Das semantische Web und damit auch die semantischen Graphdatenbanken wurden unter anderem mit dem Ziel designt, Objekte der realen Welt und deren Beziehungen untereinander zu repräsentieren.

Entsprechend gibt es auch keine Tabellen oder Kollektionen, sondern vielmehr Konzepte, Klassen und Individuals. Hinzu kommen Properties: Data-Properties zur Speicherung von Literalen, also konkreten Werten, und Object-Properties zur Verwaltung von Relationen. Annotations bieten die Möglichkeit, Zusatzinformationen und Metadaten an beliebige Ressourcen anzuhängen.

All diese Ressourcen werden auch als Entitäten bezeichnet. Da Knowledge-Graphen einem verteilten Konzept (LOD) folgen, müssen alle diese Entitäten weltweit eindeutig identifizierbar sein. Hierzu erhält jede Entität einen sogenannten Internationalized Resource Identifier, den IRI. IRIs sind zusammengesetzt aus einem Namespace, einem Separator und einem Local Identifier. Für Letzteren wird häufig eine UUID verwendet, zur besseren Lesbarkeit gegebenenfalls mit einem vorangestellten Typbezeichner. Ein typisches Beispiel ist:

```
http://ont.enapso.com/dotnetpro#
  Person_8e7980b9_bec9_4e39_af50_a8f3e6c0d349
```

IRIs sind immer Primärschlüssel. Die aus den relationalen Datenbanksystemen bekannten Sekundär- oder Fremdschlüssel gibt es in Graphdatenbanken ebenso wenig wie zusammengesetzte Schlüssel aus mehreren Feldern. Demzufolge ist es weder nötig noch sinnvoll, redundante Tripel, also solche mit gleicher Kombination von Subjekt, Prädikat und Objekt, zu verwalten. Der Versuch, solche einzufügen, wird ohne eine Fehlermeldung schlicht ignoriert.

Ähnlich wie bei XML wird für die Namespaces ein Format wie zum Beispiel *http://ont.enapso.com/dotnetpro* verwendet. Das Schema *http* ist dabei nicht mit einem Web-URL zu verwechseln und kann auch andere Werte erhalten. Die Verwendung von *https* führt dabei nicht zu einer Verschlüsselung. Zur besseren Lesbarkeit serialisierter Tripel- oder Quad-Dateien können Namespaces durch sogenannte Präfixe als eine Art Alias ersetzt werden, im folgenden Beispiel *dnp*. Hier ist der Einsatz des Doppelpunkts als Separator obligatorisch:

```
@prefix dnp: <http://ont.enapso.com/dotnetpro#> .
dnp:Person_8e7980b9_bec9_4e39_af50_a8f3e6c0d349
```

Präfixe helfen dabei, die Visualisierung von Tripeln und Quads – und damit auch spätere SPARQL-Abfragen – zu vereinfachen, denn sie sind einfacher lesbar und wartbar. Wichtig dabei ist, dass IRIs in Graphdatenbanken intern in ihrer vollen Notierung, also unabhängig von Präfixen, abgebildet sind und daher ein Austausch von Daten – auch zwischen verschiedenen Datenbanksystemen – meist unproblematisch ist. ▶

Listing 1: Enapso-GraphDB-Client konfigurieren

```
const { EnapsoGraphDBClient } =
  require('@innotrade/enapso-graphdb-client');

let graphDBEndpoint =
    new EnapsoGraphDBClient.Endpoint({
  baseURL: 'http://localhost:7200',
  repository: 'Test',
  triplestore: 'graphdb', //
    'graphdb'|'fuseki'|'stardog'
  prefixes: [
    { prefix: 'entest',
      iri: 'http://ont.enapso.com/test#'
    }
  ],
  transform: 'toCSV'
  }
);
```

Präfixe hingegen sind lokal. Graphdatenbanken verwalten individuelle Listen von Präfixen, die beim Import und Export serialisierter Tripel- und Quad-Files herangezogen werden. Entsprechend ist ein konzertiertes Präfix-Management erforderlich. Insbesondere wenn in SPARQL-Kommandos im Applikationscode Präfixe verwendet werden, müssen diese synchron mit der Präfix-Konfiguration der Datenbank gehalten werden.

Auf der untersten Ebene basiert ein Knowledge-Graph auf RDF, dem Resource Description Framework [6]. Dabei handelt es sich um ein vom W3C standardisiertes Modellierungskonzept und Vokabularium des semantischen Webs, das mithilfe von Tripeln aus Subjekt, Prädikat und Objekt (S-P-O) einfache logische Aussagen formuliert, die leicht von Maschinen gelesen und verstanden werden können – eine Methode zur Beschreibung von Wissen durch die Definition von Beziehungen zwischen Datenobjekten. Ein Graph kann auch verstanden werden als eine umfängliche Menge logisch konsistent verlinkter Tripel.

Das folgende Beispiel zeigt zwei solche Aussagen, die erste für einen Zahlenwert mit einer Data-Property und die zweite für eine Relation zwischen zwei Individuals mit einer Object-Property, wobei *Josef* und *Maria* hier Platzhalter für die jeweiligen Individual-IRIs sind.

```
[Josef] prefix:hasAge "32"  xsd:int
[Josef] prefix:hasSpouse [Maria]
```

Auf RDF baut RDF-Schema (RDFS) [7] auf, eine Erweiterung des RDF-Sprachschatzes, die unter anderem Sub-Classes und Sub-Properties sowie Domain- und Range-Axiome unterstützt. Letztere sind Mechanismen zur automatischen Klassifizierung von Individuals mittels logischer Schlussfolgerungen. Diese sogenannte Inferenz, die von in der Datenbank laufenden Reasonern ausgeführt wird, generiert neue mit SPARQL in Echtzeit abfragbare virtuelle Tripel und kann daher auch als eine Art maschinellen Lernens verstanden werden. Ein umfängliches Thema, auf das wir in einem Folgeartikel genauer eingehen werden.

Ein weiteres wichtiges Konzept des W3C-Semantic-Web-Technologie-Stacks ist OWL, die Web Ontology Language [8], eine für das semantische Web designte Sprache für das Wissensmanagement. Neben Standards zum Beispiel zur Versionierung von Ontologien, zu Import-Direktiven oder globalen Class-Restrictions führt OWL insbesondere neue semantische Konstrukte ein.

Hierzu gehören beispielsweise Ausdrücke, die festlegen, ob Object-Properties symmetrisch, transitiv oder invers sind, welche Kardinalitäten für Data-Properties zugelassen sind oder ob Klassen äquivalent oder disjunkt sind. Ebenfalls Stoff für einen Folgeartikel. Beginnen wir zunächst mit dem Blick aus der Vogelperspektive.

Listing 2: Knowledge-Graphen via SPARQL abfragen

```
graphDBEndpoint.query(
  'select *
  where{
    ?ind rdf:type dnp:Person.
    ?ind dnp:surname ?surname.
    ?ind dnp:givenname ?givenname
  }',
  { transform: 'toJSON' }
).then((result) => {
  console.log(
    'Read the Person class individuals:\n' +
     JSON.stringify(result, null, 2));
}).catch((err) => {
    console.log(err);
});
```

Knowledge-Graphen strukturieren

Eine Instanz einer Graphdatenbank kann mehrere Repositories enthalten, die bezüglich ihrer Konfiguration, ihrer Benutzer und deren Zugriffsrechten strikt voneinander isoliert sind – ein wichtiger Sicherheitsaspekt. In der traditionellen Datenbankterminologie wird ein Schema als Sammlung von Tabellen beziehungsweise Kollektionen und eine Datenbank als Sammlung von Schemata verstanden. Insofern ist ein Repository einer Graphdatenbank mit einem Datenbankschema vergleichbar.

Bei Graphdatenbanken kommt mit den sogenannten Named Graphs ein weiteres Organisationsinstrument ins Spiel. Sie ermöglichen es, ein Repository in mehrere logische Einheiten, sogenannte Kontexte, zu strukturieren. Gleiche Aussagen können in mehreren Named Graphs verwaltet werden. Technisch erhält jedes Tripel hierzu die Information zum Kontext, in dem es gültig ist, was die Tripel so zu Quads macht. In anderen Worten: Ein Quad ist ein Tripel mit zusätzlicher Kontextinformation in einem Repository einer Graphdatenbank.

Knowledge-Graphen versus relationale Datenbanken

Schemata in traditionellen SQL- und NoSQL-Datenbanken enthalten, wie man es auch aus der objektorientierten Programmierung (OOP) kennt, individuelle Felder pro Klasse. Zwar können die Felder der einfacheren Lesbarkeit wegen technisch gleiche Namen tragen, semantisch aber eine ganz andere Bedeutung haben. Beispiel: *Gewicht* als *float*, einmal in *g* und einmal in *kg*. Eine gewünschte Aggregation erfordert damit eine Berücksichtigung der unterschiedlichen Maßeinheiten im Code.

Das Wissen steckt hier also – selbst von Menschen nur schwer wiederverwendbar – in der App und nicht einfach maschinenlesbar in der Datenbank. Kommt eine neue Klasse, Tabelle oder Kollektion mit einem Gewicht, zum Beispiel in *to*, dazu, entsteht Aufwand: Die App muss manuell angepasst, neu getestet und deployt werden.

Semantische Graphdatenbanken verfolgen demgegenüber einen anderen Ansatz, denn Daten- und Object-Proper-

Listing 3: Ergebnis einer SPARQL-Abfrage

```
{ "head": {
    "vars": [ "ubd", "surname", "givenname" ]
  },
  "results": {
    "bindings": [
      { "ind": {
        "type": "uri",
        "value": "http://ont.enapso.com/dotnetpro# \
          Person_8e7980b9_bec9_4e39_af50_a8f3e6c0d349"
        },
        "surname": {
          "type": "literal",
          "value": "Schulze"
        },
        "givenname": {
          "type": "literal",
          "value": "Alexander"
        }
      },
      :
    ]
  },
  "success": true
}
```

ties werden getrennt von den Klassen definiert und dann lediglich referenziert, zum Beispiel in Form von Class-Restrictions. Dabei wird die Bedeutung einer Property *Gewicht* über deren Annotationen wie Labels und Kommentare im Modell menschlich verständlich definiert und die gleiche Eigenschaft von mehreren Klassen wiederverwendet. Kommt dann eine neue Klasse mit der gleichen Eigenschaft hinzu, so muss für eine erweiterte Aggregation nicht einmal die App angepasst werden.

Eine von vielen weiteren semantischen Möglichkeiten ist, die Kardinalität für Eigenschaften festzulegen, also dass mit

```
prefix:Person hasSpouse max 1 prefix:Person
```

beispielsweise Personen maximal eine(n) Ehepartner(in) haben dürfen, so zumindest im europäischen Kulturkreis. Die Property *hasSpouse* kann auch als symmetrisch definiert werden. Wird die Aussage dann explizit für *Josef* getroffen, so ist sie automatisch implizit auch für *Maria* gültig. Diese sogenannte Inferenz wird vom Reasoner der Graphdatenbank übernommen.

All dies sind einige praktische Beispiele dafür, Wissen statt im Code in einem Knowledge-Graphen – einer semantischen Single Source of Truth (SSOT) – zu verwalten, Wissen dort zu harmonisieren und bereitzustellen, um damit so sowohl Effizienz als auch Qualität von Software-, Produktions- und Betriebsprozessen zu verbessern.

Knowledge-Graphen lesen und managen

W3C-konforme semantische Graphdatenbanken werden über die an SQL angelehnte Sprache SPARQL angesprochen. Während SPARQL 1.0 mit dem *select*-Kommando rein für Abfragen ausgelegt war, steht seit SPARQL 1.1 mit *insert* und *delete* eine Erweiterung für Manipulationen zur Verfügung. In SPARQL existiert jedoch kein *update*. Für Änderungen werden bestehende Tripel in einem Kommando gelöscht und neu eingefügt.

Da in RDF, RDFS und OWL alle Aussagen als Tripel beziehungsweise Quads abgebildet werden, können demnach nicht nur Daten, also die Individuals, sondern ebenso Klassen, Properties und Annotations über SPARQL abgefragt und verwaltet werden.

Knowledge-Graphen in Apps integrieren

Die meisten Graphdatenbank-Hersteller stellen zwar Client-Bibliotheken für verschiedene Programmiersprachen bereit, für Node.js-Applikationen zum Beispiel graphdb.js [9] von Ontotext oder stardog.js [10] von Stardog. Allen gemeinsam ist jedoch, dass diese recht hersteller- und systemnah ausgelegt sind. An generischer Treibertechnologie oder leistungsfähigen Abstraktions- und ORM-Tools seitens der Hersteller mangelt es derzeit noch.

Für eigene Applikationen bedeutet das trotz propagierter Interoperabilität – einer der Vorteile der W3C-konformen Semantic-Web-Technologie – immer noch eine unerwünschte Herstellerbindung. Tatsächlich lassen sich zwar technisch die Tripel- oder Quad-Stores leicht zwischen verschiedenen Systemen austauschen. Als Entwickler müssen Sie aber nicht nur ein solides Verständnis der komplexen Knowledge-Graph-Konzepte mitbringen, sondern sich zusätzlich noch mit den Eigenheiten der jeweiligen Datenbanken auseinandersetzen – einer der Gründe, warum sich die semantischen Technologien bisher eher nur schleppend durchgesetzt haben.

Eines der Werkzeuge, die diese Lücke füllen, ist der in GitHub frei verfügbare Enapso Graph Database Client [11] für Node.js. Der Client unterstützt und abstrahiert die Open-Source-Graphdatenbanken Apache Jena Fuseki [12], GraphDB von Ontotext [13] und Stardog [14], weitere sind in ▶

Listing 4: Aufbereitetes SPARQL-Ergebnis

```
{ "total": 2,
  "success": true,
  "records": [
    { "ind": http://ont.enapso.com/dotnetpro# \
        Person_8e7980b9_bec9_4e39_af50_a8f3e6c0d349,
      "surname": "Schulze",
      "givenname": "Alexander"
    },
    :
  ]
}
```

Arbeit. Nach kurzer Konfiguration, die sich im Wesentlichen auf einen Datenbank-URL, ein Repository sowie einen Benutzernamen nebst zugehörigem Kennwort beschränkt, authentifiziert sich die Applikation mit wenigen Zeilen an der Datenbank und führt erste SPARQL-Abfragen und -Updates gegen den Knowledge-Graphen aus.

Listing 1 zeigt den initialen Verbindungsaufbau zur Datenbank. Details zum Client und zur Fehlerbehandlung sind auf GitHub dokumentiert [11].

Als Nächstes folgt die Authentifizierung gegen die Datenbank beziehungsweise das Repository. Mit Mechanismen wie Basic-Auth oder der Verwaltung von JSON-Web-Tokens (JWT) braucht man sich dabei nicht auseinanderzusetzen, dies übernimmt die Library.

```
graphDBEndpoint.login('admin','root').then((result) => {
  console.log(result);
}).catch((err) => {
  console.log(err);
});
```

Wer das erste Mal mit SPARQL arbeitet, sieht sich neben RDF und OWL als Sprachen, die auf den Tripel-/Quad-Stores aufsetzen, noch mit einigen weiteren Aspekten konfrontiert, zum Beispiel mit der Verwaltung der Präfixe, den Aliasen für die Namespaces in den IRIs, der Konvertierung der XSD-Datentypen in die jeweilige Programmiersprache oder mit der Umwandlung der SPARQL-Ergebnisse in komfortabel in JavaScript oder anderen Programmiersprachen verarbeitbare JSON- oder CSV/TSV-Formate.

Mit dem Enapso Graph Database Client geschieht all dies mit einem einzelnen Kommando, wie in Listing 2 gezeigt.

Im Original sehen die Ergebnisse der SPARQL-Abfrage, die sogenannten *bindings* [15], im Rohformat zunächst so aus wie in Listing 3. Der integrierte JSON-Converter erkennt die zurückgelieferten RDF/OWL-Datentypen und wandelt alle String-Werte in den SPARQL-Bindings in ihre jeweilige JSON-Repräsentation um. Entsprechend aufbereitet sieht das Abfrage-Ergebnis dann so aus wie in Listing 4 gezeigt.

Wenn individuelle Ergebnis-Repräsentationen erforderlich sind, kann neben den mitgelieferten JSON-, CSV- und TSV-Konvertern auch ein eigener Konverter für die SPARQL-Bindings mittels eines einfachen Callbacks integriert werden.

Listing 5: Export von Tripeln aus dem Graphen

```
graphDBEndpoint.downloadToFile({
  format: EnapsoGraphDBClient.FORMAT_TURTLE.type,
  filename: '../ontologies/test.ttl',
  context: 'http://ont.enapso.com/dotnetpro'
}).then((result) => {
  console.log(result);
}).catch((err) => {
  console.log(err);
});
```

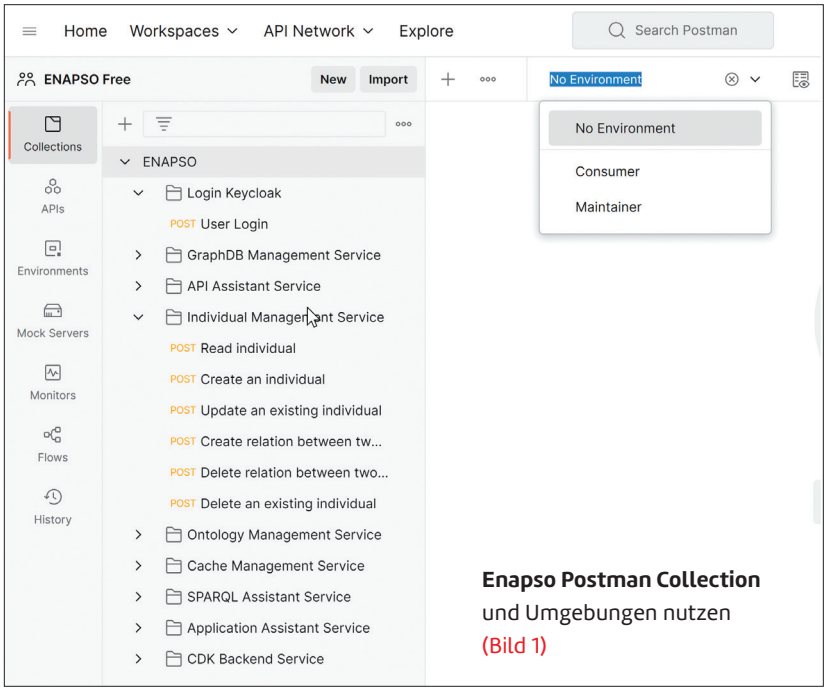
Knowledge-Graphen administrieren

Neben dem Management von Taxonomien und Individuals in einem Knowledge-Graphen mittels SPARQL ist es für die Anwender meist komfortabler, administrative Aufgaben nicht über verschiedene, verteilte Admin-Tools der Hersteller zu erledigen, sondern lieber mittels eines zentralen UI mit konsistenter UX mithilfe eines API in der eigenen App.

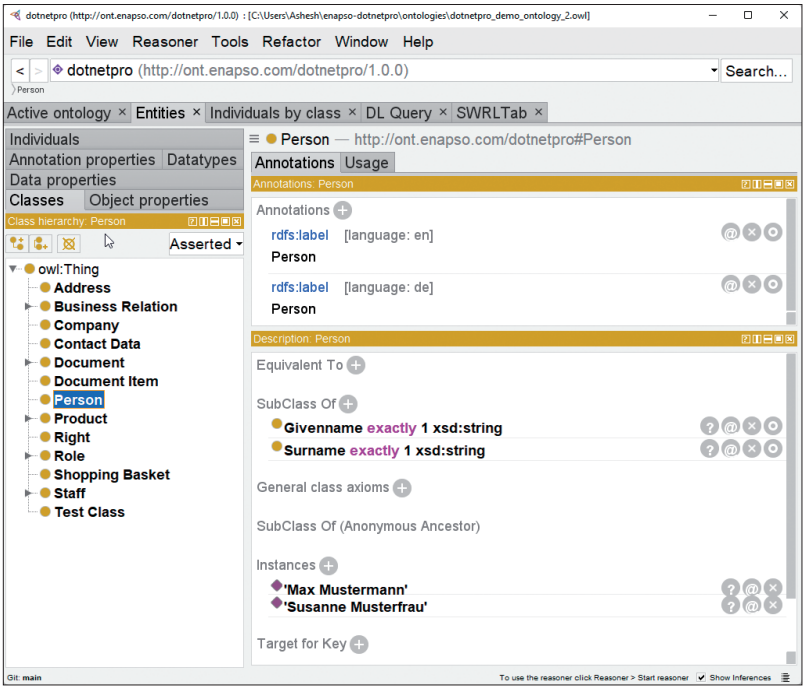
Im- und Export von RDF(S)-Stores oder OWL-Ontologien in und aus Repositories oder Named Graphs sind hier Beispiele, wahlweise über Dateien, URLs im Web oder direkt über die Daten in der App – idealerweise in allen gängigen Serialisierungen wie Turtle, JSON-LD oder RDF/XML, mit dem Ziel möglichst hoher Interoperabilität. Wollen Sie den stabilen Betrieb Ihrer Umgebung sicherstellen, sind zudem ein Monitoring von CPU-Last, Speicher- und Harddisk-Nutzung oder eine explizit auslösbare Garbage-Collection willkommene Features.

Das in GitHub als Open Source verfügbare Enapso Graph Database Admin Package [16] für Node.js bietet Funktionen für genau diese Aufgaben. Listing 5 und Listing 6 zeigen exemplarisch den Export von Tripeln im Turtle-Format (*.ttl*), das verhältnismäßig kompakt und von Menschen wie von Maschinen einfach zu lesen ist.

Einige Datenbankhersteller liefern noch weit umfangreichere Schnittstellen: neben der Benutzer-, Rechte- und Rollenverwaltung zum Beispiel für die Verwaltung von Remote-Locations oder für das Cluster-Ma-



Enapso Postman Collection und Umgebungen nutzen (Bild 1)



Die dotnetpro-Demo-Ontology in Protégé (Bild 2)

nagement. Diese sind jedoch überwiegend herstellerspezifisch und aufgrund der intern unterschiedlichen Architekturen auch schwer zu abstrahieren. Hierzu sei daher auf die Dokumentation in GitHub [16] beziehungsweise der Hersteller verwiesen.

Knowledge-Graphen skripten

Sobald der erste Knowledge-Graph angelegt und erfolgreich in die erste Applikation integriert ist, stellen sich umgehend weitere Fragen, zum Beispiel zu Betriebsthemen und Automationsoptionen: Wie können Backups zeitgesteuert automatisiert erstellt, Repositories und deren Benutzer, Rollen und Rechte verwaltet, Tests und CI/CD-Operationen, aber auch einfach nur Status- oder Datenbankabfragen auf Kommandozeilenebene ausgeführt werden?

Hier hilft das ebenfalls auf GitHub frei verfügbare Open-Source-Tool Enapso Graph Database Command Line Interface (CLI) [17]. Es unterstützt den Import und Export ganzer Repositories oder einzelner Named Graphs in allen gängigen Serialisierungen und Formaten, das Anlegen, Ändern und Löschen von Benutzern inklusive deren Rollen und Rechten sowie das Absetzen von SPARQL-Abfragen und -Updates direkt von der Windows-, Linux- oder macOS-Kommandozeile aus.

Installiert wird das Enapso Graph Database CLI über npm mit einem einzelnen Shell-Kommando:

```
npm i -g @innotrade/enapso-graphdb-cli
```

Im Anschluss steht das Enapso CLI unter *enapsogdb* in der Shell bereit. Das Kommando in Listing 7 zeigt exemplarisch den Import einer Ontologie im RDF/XML-Format in den Named Graphs <*http://ont.enapso.com/test*> des Repositorys *Test*.

Eine ausführliche Dokumentation aller Shell-Funktionen ist verfügbar unter [17]. Derzeit werden die W3C-konformen Graphdatenbanken Ontotext GraphDB, Apache Jena Fuseki und Stardog unterstützt. Im Rahmen des gestarteten Community-Prozesses ist die Anbindung weiterer Datenbanken geplant; funktionale Erweiterungen sind bereits angekündigt.

Knowledge-Graphen absichern

Die Möglichkeit, Informationen – sogar aus verschiedenen verteilten Quellen – fast beliebig zu Wissen zu verbinden, ist einer der zentralen Treiber für den Einsatz von Knowledge-Graphen. Entsprechend der Freude von Entwicklern und Nutzern über Freiheiten und Flexibilität von Tripel-/Quad-Stores und SPARQL gestalten sich in diesem Szenario parallel die Kopfschmerzen der Sicherheitsbeauftragten. Es liegen eben keine starren Schemata, Tabellen oder Kollektionen mehr vor, für deren Zugriff man Benutzer einfach mal eben autorisieren kann.

Wir hatten gezeigt, dass Knowledge-Graphen aus vielen, theoretisch und praktisch weltweit verteilten Datenbankinstanzen, diese wiederum aus vielen Repositories und diese wieder aus vielen Named Graphs zusammengesetzt sein können, wobei diese nicht zuletzt auch noch Tripel aus individuellen Namespaces enthalten können. ▶

Listing 6: Repräsentation von Tripeln im Turtle-Format

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/
  rdf-schema#> .
@prefix dnp: <http://ont.enapso.com/dotnetpro#> .
dnp:Person_8e7980b9_bec9_4e39_af50_a8f3e6c0d349 a
    owl:NamedIndividual, dnp:Person;
  rdfs:label "Alexander Schulze";
  dnp:givenname "Alexander";
  dnp:surname "Schulze" .
```

Listing 7: Ontologie-Import mit dem Enapso CLI

```
enapsogdb import
  --dburl http://localhost:7200
  --repository "Test"
  --context http://ont.enapso.com/test
  --baseiri http://ont.enapso.com/test#
  --sourcefile "imports/dotnetpro_demo_ontology_2.owl"
  --username "[username]"
  --password "[password]"
  --format "application/rdf+xml"
```

Entsprechend der offensichtlichen Problematik, klassische, fein granulierte und gar zentral verwaltete Permission-Konzepte auf ein solches Szenario anzuwenden, beschränken sich die Hersteller meist neben der Authentifizierung gegenüber den Datenbankinstanzen auf die Autorisierung über Rollen für den Lese- beziehungsweise Schreibzugriff auf bestimmte Repositories – oder bestenfalls Named Graphs. Lesezugriff bedeutet, SPARQL-*select*-Operationen ausführen zu dürfen. Ein Schreibzugriff berechtigt für *insert*- und *delete*-Operationen auf die entsprechend freigegebenen Repositories – innerhalb derer dann aber mit allen Freiheiten, ohne Beschränkung auf bestimmte Klassen oder Individuals.

Ein Medikament gegen sicherheitsbedingte Kopfschmerzen sind Views. Diese existieren nicht per se im SPARQL-Standard, doch in ihrer Form als Liste von SPARQL-Templates mit ihren jeweiligen Variablen können sie leicht mit einer Rolle assoziiert und Konsumenten so berechtigt werden, nur bestimmte Views und damit eben auch nur autorisiertes Wissen aus dem Knowledge-Graphen zu ziehen. Der Unterschied für uns Entwickler ist, dass wir keine komplexen SPARQL-Queries konstruieren und absetzen müssen, sondern einfach eine View auf dem Server ausführen. Ein viel einfacher zu kontrollierender Remote Procedure Call – idealerweise über ein einfaches Standard-REST-API, das von allen gängigen Programmiersprachen unterstützt ist.

Tabelle 1: Enapso-Knowledge-Graph-Dienste

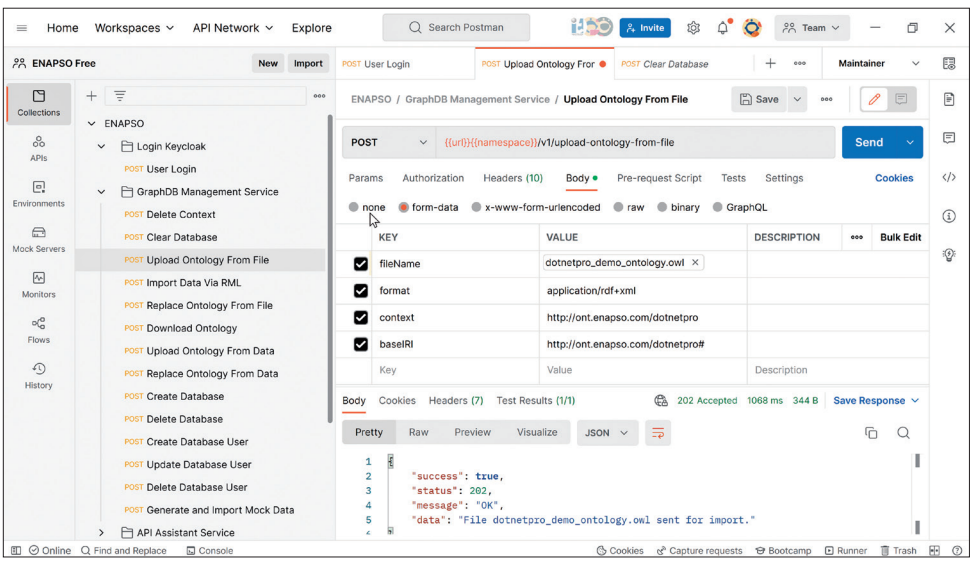
Microservice	Beschreibung
Individual Management	Anlegen, Lesen, Suchen, Ändern, Löschen (CRUD) von Individuals und deren Relationen im Knowledge-Graphen.
Ontology Management	Verwalten von Klassen, Properties und Annotations wie Labels und Comments, Taxonomien, Relations und Restrictions.
Graph Database Management	Verwalten von Repositories, Named Graphs, Benutzern, Rechten und Rollen, Hoch- und Herunterladen von Ontologien.
SPARQL Assistant	Generator für die automatische Erzeugung komplexer SPARQL-Abfrage- und Update-Kommandos.
API Assistant	Verwalten der High-Level-Routen für das REST-API (CRUD-Endpoints) für Klassen und Individuals und OpenAPI-Generierung.
Cache Management	Klassen-Cache über mehrere Named Graphs zur Beschleunigung von Query-Generatoren und Abfragen, Cache-Distribution.

Knowledge-Graphen betreiben

Bevor wir nun mit der Knowledge-Graph-Programmierung beginnen können: Welche Grundlagen sind dafür nötig? Zuunterst erst einmal eine W3C-konforme Graphdatenbank. Wie berichtet kommen hier etwa Apache Jena Fuseki oder Stardog infrage, für die Beispiele in diesem Artikel setzen wir die GraphDB Free Version von Ontotext [13] ein. Diese ist nur dahingehend beschränkt, maximal zwei Anfragen gleichzeitig zu bedienen, bietet aber ansonsten volle Funktionalität.

Im Node.js-Backend können die zuvor beschriebenen Enapso-Open-Source-Tools eingesetzt werden, für die Security freie Identity- und Access-Management-Systeme (IAM) wie zum Beispiel Keycloak [18] und für das Session-Management zum Beispiel das bekannte Redis [19]. Trotz großem Komfort und starkem Tool-Support aber noch immer ein aufwendiger Weg zum ersten Erfolgserlebnis.

Wer mit geringem Aufwand erste Erfahrungen mit Knowledge-Graphen sammeln möchte, kann sich die kostenlose Enapso Free Edition als Image herunterladen, direkt mit der Programmierung loslegen und sich im Anschluss tiefer in die zugrunde liegenden Technologien einarbeiten. Enapso ist eine Microservice-basierte SaaS-Plattform für Knowledge-Graphen. Eine bestehende Docker-Installation vorausgesetzt, genügen zwei einfache Kommandos, und die Plattform ist auf dem lokalen Host verfügbar. Eine detaillierte Dokumentation findet sich unter [20].



Hochladen einer Ontologie auf die Enapso-Plattform (Bild 3)

Durch die Virtualisierung ist ein Betrieb in einer privaten wie einer öffentlichen Cloud nutzbar. Eine Demo, betrieben unter AWS, steht online zum Ausprobieren bereit. Die URLs zu den API-Services und zur Open-API-Dokumentation sind in GitHub unter [20] verfügbar.

Für einen lokalen Test genügt die Light-Konfiguration enapso-free-edition-light. Diese nutzt Apache Jena Fuseki und eine einfache JSON-Datei zur Konfiguration von Benutzern und deren Berechtigungen. Sie kommt mit 4 GB Speicher aus. Die Developer-Konfiguration enapso-free-edition-

dev mit integrierter GraphDB Free und KeyCloak benötigt minimal 8, besser 16 GB RAM, wenn man mit größeren Datenmengen arbeiten möchte.

Enapso stellt eine Vielzahl von Knowledge-Graph-Diensten über ein Standard-REST-API bereit. Tabelle 1 gewährt einen ersten Einblick.

Erste Schritte zum eigenen Knowledge-Graphen

Sobald das Enapso-Docker-Image ausgeführt wird, steht die Plattform auf der lokalen Maschine für erste Tests zur Verfügung. Für den leichten Einstieg in die Entwicklung eigener semantischer Applikationen steht in GitHub unter [20] eine kostenlose Postman-Collection mit allen API-Calls zum Download bereit [21]. Darüber hinaus ist das REST-API auch mit OpenAPI dokumentiert. Auch über dessen UI können direkt API-Calls abgesetzt werden.

Zusätzlich zur Collection mit den API-Calls stehen eine Consumer- und eine Maintainer-Umgebung für Postman bereit. Die Consumer-Umgebung konfiguriert einen Benutzer mit reinen Leserechten, die Maintainer-Umgebung einen Benutzer mit Rollen für Schreiboperationen und administrative Funktionen. Bild 1 zeigt die nach Bereichen strukturierten API-Calls und die Auswahl der Umgebung nach deren Import.

Authentifizierung und Autorisierung

Vor der Ausführung des ersten Requests ist eine Authentifizierung erforderlich. In Postman wählt man hierzu zunächst die Umgebung, Consumer oder Maintainer, und führt dann den Login-Call aus. Das in der Antwort zurückgelieferte Token wird automatisch in einer Umgebungsvariablen gespeichert und Enapso ist bereit, erste Anfragen aus Postman zu bearbeiten.

Ontologien hochladen

Um mit erstem Übungsmaterial zu starten, kann man über den Graph Database Management Service eine Ontologie auf die Plattform hochladen. Im Enapso-Community-Repository auf GitHub steht hierzu die *dotnetpro-demo-pontology.ttl* zum Import bereit. Bild 2 zeigt die Ontologie exemplarisch in Protégé, einem ausgereiften semantischen Open-Source-Ontologie-Modellierungstool. Natürlich können Sie alternativ jedes W3C-konforme Modellierungswerkzeug einsetzen, eigene oder auch fremde Ontologien hochladen.

Bild 3 zeigt, wie eine Ontologie auf die Enapso-Plattform hochgeladen wird. Neben der Datei enthält der Request das Format, den Named Graphs (*context*) und den Default- beziehungsweise Base-IRI für Tripel ohne Präfix-Angabe.

CRUD-Operationen für Individuals

Nachdem das erste Model erfolgreich auf die Plattform hochgeladen ist, lassen Sie uns abschließend die ersten Datensätze anlegen, lesen, ändern und löschen. Listing 8 zeigt den Body des REST-*POST*-Requests, um ein neues Individual anzulegen.

Übergeben werden der IRI der Klasse *http://ont.enapso.com/dotnetpro#Person*, der Base-IRI *http://ont.enapso.com/dotnetpro#* und im Array *records* die Felder von einem oder auch gleich mehreren Datensätzen.

Gelesen werden die Individuals REST-konform über den *GET*-Request. Listing 9 zeigt ein entsprechendes Beispiel. Geändert wird ein Datensatz über den *UPDATE*-Request wie in Listing 10 gezeigt, gelöscht per *DELETE*-Request (Listing 11).

Die nötigen SPARQL-Kommandos für alle CRUD-Operationen werden innerhalb der Enapso-Plattform basierend auf Ihrer Klassen-Definition automatisch generiert. ▶

Listing 8: Ontologie-Import mit dem Enapso CLI

```
{ "cls": "http://ont.enapso.com/dotnetpro#Person",
  "baseiri": "http://ont.enapso.com/dotnetpro#",
  "records": [
    { "givenname": "Ashesh",
      "surname": "Goplani"
    }
  ]
}
Response:
{ "status": 200,
  "count": 1,
  "records": [
    { "givenname": "Ashesh",
      "surname": "Goplani",
      "iri": "http://ont.enapso.com/dotnetpro#
        Person_ef07dbe7-a9d9-4ff8-b611-28e9c4611ba2"
    }
  ],
  "message": "OK",
  "success": true
}
```

Listing 9: Lesen von Individuals

```
{ "cls": "http://ont.enapso.com/dotnetpro#Person"
}
Response:
{ "status": 200, "message": "Ok", "success": true,
  "count": 4, "total": 4,
  "offset": 0, "limit": 0,
  "records": [
    { "iri": "http://ont.enapso.com/dotnetpro#
        Person_8e7980b9_bec9_4e39_af50_a8f3e6c0d349",
      "givenname": "Max",
      "surname": "Mustermann",
      "hasRole": "http://ont.enapso.com/dotnetpro#
        Role_139a90cb_f8ed_48e5_afc5_7094ae132651"
    },
    :
  ]
}
```

Fazit

Natürlich ist dies erst der Anfang. Die Verwaltung von Klassen, Properties und Annotations, von Relationen und Joins, Views und Validierungen sind weitere Features.

Bis hierher haben wir gezeigt, dass schon wenige Handgriffe ausreichen, um einen ersten Knowledge-Graphen aufzusetzen und damit zu entwickeln. Eine Graphdatenbank ist mit wenigen Kommandos installiert, ob lokal oder virtualisiert oder in der Cloud. Ein erstes Repository ist schnell eingerichtet, und entsprechende SPARQL-Endpoints stehen mit Default-Konfigurationen zur Verfügung. Mit wenigen Zeilen Code sind Ontologien aus freien Modellierungstools wie Protégé in einen Named Graph hochgeladen, und mit einigen weiteren Zeilen werden Individuals über ein Standard-REST-API darin verwaltet. Administrative Werkzeuge und operative Tools runden die Palette ab. Die Grundlagen für Aufbau, Wartung und Betrieb eines Knowledge-Graphen sind damit gelegt.

Knowledge-Graphen sind jedoch weit mehr als nur eine alternative Datenbanktechnologie. In einer der nächsten Ausgaben der dotnetpro erfahren Sie mehr über semantische Modellierungen, wie Sie Knowledge-Graphen zur Integration und Harmonisierung heterogener Datenquellen einsetzen, mit Views auch komplexe Abfragen sicher ausführen, wie Sie mit Machine Teaching Wissen für künstliche Intelligenz und für Matching- sowie Empfehlungssysteme bereitstellen und wie Sie dieses Wissen für digitale Assistenzsysteme nutzen. Es bleibt spannend, bleiben Sie also dran. ■

Listing 10: Aktualisieren eines Individuals

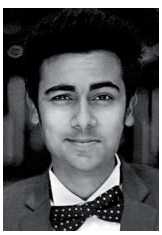
```
{ "cls": "http://ont.enapso.com/dotnetpro#Person",
  "records": [
    { "surname": "Jackson",
      "iri": "http://ont.enapso.com/dotnetpro#
        Person_f59376c1-1a63-4d27-a33b-ab356c380a02"
    }
  ]
}
Response:
{ "status": 200, "message": "OK",
  "success": true, "count": 1
}
```

Listing 11: Löschen eines Individuals

```
{ "cls": "http: http://ont.enapso.com/dotnetpro#
    Person",
  "records": [
    { "iri": "http://ont.enapso.com/dotnetpro#
        Person_f59376c1-1a63-4d27-a33b-ab356c380a02"
    }
  ]
}
Response:
{ "status": 200, "message": "OK",
  "success": true, "count": 1
}
```


[1] W3C, Semantic Web, https://www.w3.org/standards/semanticweb/
[2] Protégé, https://protege.stanford.edu/
[3] W3C, SPARQL, https://www.w3.org/TR/sparql11-query/
[4] RDF Mapping Language (RML), https://rml.io/specs/rml/
[5] W3C, Shapes Constraint Language, https://www.w3.org/TR/shacl/
[6] W3C, Resource Description Framework, https://www.w3.org/RDF/
[7] W3C, RDF Schema, https://www.w3.org/TR/rdf-schema/
[8] W3C, Web Ontology Language, https://www.w3.org/OWL/
[9] graphdb.js auf GitHub, https://github.com/Ontotext-AD/graphdb.js
[10] stardog.js auf GitHub, https://github.com/stardog-union/stardog.js
[11] enapso-graphdb-client auf GitHub, https://github.com/innotrade/enapso-graphdb-client
[12] Apache Jena, https://jena.apache.org/
[13] GraphDB, https://graphdb.ontotext.com/
[14] Stardog, https://www.stardog.com/get-started/
[15] W3C, SPARQL 1.1 Query Results JSON Format, https://www.w3.org/TR/sparql11-results-json/
[16] Enapso Graph Database Admin Package auf GitHub, https://github.com/innotrade/enapso-graphdb-admin
[17] Enapso Graph Database CLI auf GitHub, https://github.com/innotrade/enapso-graphdb-cli
[18] Keycloak, https://www.keycloak.org/
[19] Redis, https://redis.io
[20] Enapso-Community, https://github.com/innotrade/enapso-community
[21] Postman, https://www.postman.com

Alexander Schulze
ist Geschäftsführer der Innotrade GmbH und Experte für semantisches Datenmanagement und digitale Geschäftsmodelle. Er berichtet regelmäßig über KI und Wissensmanagement und deren konkreten Nutzen.
alexander.schulze@innotrade.de



Ashesh Goplani
ist Fullstack Semantic Software Developer bei der Innotrade GmbH, spricht und schreibt über Ontologie-getriebene Softwareassistenzsysteme und ist Experte für die Verbesserung von Softwareentwicklungsprozessen.
ashesh.goplani@innotrade.de

dnpCode A2304Enapso