

# The Knowledge Graph Cookbook

## Recipes for Knowledge Graphs that Work:

- Learn why and how to build knowledge graphs that help enterprises use data to innovate, create value and increase revenue. This practical manual is full of recipes and knowledge on the subject.
- Learn more about the variety of applications based on knowledge graphs.
- Learn how to build working knowledge graphs and which technologies to use.
- See how knowledge graphs can benefit different parts of your organization.
- Get ready for the next generation of enterprise data management tools.

**Dr. Jans Aasman, CEO, Franz Inc. is interviewed in the Expert Opinion Section.**

**“KNOWLEDGE GRAPHS AREN’T WORTH THEIR NAME IF THEY DON’T ALSO LEARN AND BECOME SMARTER DAY BY DAY” – Dr. Aasman**

## INTERVIEWS

The creation of knowledge graphs is interdisciplinary. Good chefs regularly visit other restaurants for inspiration. We have asked experts working in the field of knowledge graphs and semantic data modelling to comment on their experience in this area. They have worked with various stakeholders in different industries, so that you, dear reader, may further develop your understanding of the topic.



**JANS AASMAN**

FRANZ

Dr. Jans Aasman is CEO at Franz Inc., a leading provider of Knowledge Graph Technologies (AllegroGraph) and AI-based Enterprise solutions. Dr. Aasman is a noted speaker, author, and industry evangelist on all things graph.

---

“KNOWLEDGE GRAPHS AREN'T WORTH THEIR NAME IF THEY DON'T ALSO LEARN AND BECOME SMARTER DAY BY DAY”

---

[Click here](#) to get the book as free PDF or Kindle version.

---

# Graphorum – Dr. Aasman Presenting

**Graph-Driven Event Processing for Intelligent Customer Operations**

Wednesday, October 16, 2019

10:15 AM – 11:15 AM

Level: Case Study



In the typical organization, the contents of the actual chat or voice conversation between agent and customer is a black hole. In the modern Intelligent Customer Operations center, the interactions between agent and customer are a source of rich information that helps agents to improve the quality of the interaction in real time, creates more sales, and provides far better analytics for management. The Intelligent Customer Operations center is enabled by a taxonomy of the products and services sold, speech recognition to turn conversations into text, a taxonomy-driven entity extractor to take the important concepts out of conversations, and machine learning to classify chats in various ways. All of this is stored in a real-time Knowledge Graph that also knows (and stores) everything about customers and agents and provides the raw data for machine learning to improve the agent/customer interaction.

In this presentation, we describe a real-world Intelligent Customer Organization that uses graph-based technology for taxonomy-driven entity extraction, speech recognition, machine learning, and predictive analytics to improve quality of conversations, increase sales, and improve business visibility.

<https://graphorum2019.dataversity.net/sessionPop.cfm?confid=132&proposalid=11010>

---

# The Importance of FAIR Data in Earth Science

Franz's CEO, Jans Aasman's recent [Marine Technology News](#):

Data's valuation as an enterprise asset is most acutely realized over time. When properly managed, the same dataset supports a plurality of use cases, becomes almost instantly available upon request, and is exchangeable between departments or organizations to systematically increase its yield with each deployment.



Dr. Jans Aasman, CEO, Franz Inc.

These boons of leveraging data as an enterprise asset are the foundation of GO FAIR's Findable Accessible Interoperable Reusable (FAIR) principles profoundly impacting the data management rigors of geological science. Numerous organizations in this space have embraced these tenets to swiftly share information among a diversity of disciplines to safely guide the stewardship of the earth.

According to Dr. Annie Burgess, Lab Director of Earth Science Information Partners (ESIP), the "most pressing global challenges cannot be solved by a single organization. Scientists require data collected across multiple disciplines, which are often managed by many different agencies and institutions." As numerous members of the earth science community are realizing, the most effectual means of managing those disparate data according to FAIR principles is by utilizing the semantic standards underpinning knowledge graphs.

Read the [full article at Marine Technology News](#)

---

# Harnessing the Internet of Things with JSON-LD



Franz's CEO, Jans Aasman's recent IoT Evolution Article:

Conceptually, the promise of the Internet of Things is almost halcyon. Its billions of sensors are all connected, continuously transmitting data to support tailored, cost-saving measures maximizing revenues in applications as diverse as smart cities, smart price tags, and predictive maintenance in the Industrial Internet.

Practically, the data management necessities of capitalizing on this promise by the outset of the next decade are daunting. The vast majority of these datasets are unstructured or semi-structured. The data modeling challenges of rectifying their schema for integration are considerable. The low latency action required to benefit from their data implies machine intelligence largely elusive to today's organizations.

.....

The self-describing, linked data approach upon which JSON-LD is founded excels at the low latent action resulting from machine to machine communication in the IoT. The nucleus of the linked data methodology—semantic statements and their unique Uniform Resource Identifiers (URIs)—are read and

understood by machines. This characteristic aids many of the IoT use cases requiring machine intelligence; by transmitting IoT data via the JSON-LD format organizations can maximize this boon. Smart cities provide particularly compelling examples of the machine intelligence fortified by this expression of semantic technology.

Read the [full article at IoT Evolution](#)

---

## SHACL – Shapes Constraint Language in AllegroGraph

[SHACL](#) is a SHApE Constraint Language. It specifies a vocabulary (using triples) to describe the shape that data should have. The *shape* specifies things like the following simple requirements:

- How many triples with a specified subject and predicate should be in the repository (e.g. at least 1, at most 1, exactly 1).
- What the nature of the object of a triple with a specified subject and predicate should be (e.g. a string, an integer, etc.)

See the [specification](#) for more examples.

SHACL allows you to validate that your data is conforming to desired requirements.

For a given validation, the shapes are in the *Shapes Graph* (where *graph* means a collection of triples) and the data to be validated is in the *Data Graph* (again, a collection of

triples). The SHACL vocabularly describes how a given shape is linked to *targets* in the data and also provides a way for a Data Graph to specify the Shapes Graph that should be used for validation. The result of a SHACL validation describes whether the Data Graph conforms to the Shapes Graph and, if it does not, describes each of the failures.

## Namespaces Used in this Document

Along with standard predefined namespaces (such as `rdf:` for `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>` and `rdfs:` for `<http://www.w3.org/2000/01/rdf-schema#>`), the following are used in code and examples below:

```
prefix fr: <https://franz.com#>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix franz: <https://franz.com/ns/allegrograph/6.6.0/>
```

## A Simple Example

Suppose we have a *Employee* class and for each *Employee* instance, there must be exactly one triple of the form

```
emp001 hasID "000-12-3456"
```

where the object is the employee's ID Number, which has the format is [3 digits]-[2 digits]-[4 digits].

This TriG file encapsulates the constraints above:

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

```
<https://franz.com#Shapes> {
  <https://franz.com#EmployeeShape>
    a sh:NodeShape ;
    sh:targetClass <https://franz.com#Employee> ;
    sh:property [
      sh:path <https://franz.com#hasID> ;
```

```

    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
        sh:pattern    "^[0-9][0-9][0-9]-[0-9][0-9]-
[0-9][0-9][0-9][0-9]$" ;
    ] .
}

```

It says that for instances of `fr:Employee` (`sh:targetClass <https://franz.com#Employee>`), there must be exactly 1 triple with predicate (path) `fr:hasID` and the object of that triple must be a string with pattern [3 digits]-[2 digits]-[4 digits] (`sh:pattern "^[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]$"`).

This TriG file defines the `Employee` class and some employee instances:

```

@prefix fr: <https://franz.com#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

```

```

{
  fr:Employee
    a rdfs:Class .
  fr:emp001
    a fr:Employee ;
    fr:hasID "000-12-3456" ;
    fr:hasID "000-77-3456" .
  fr:emp002
    a fr:Employee ;
    fr:hasID "00-56-3456" .
  fr:emp003
    a fr:Employee .
}

```

Recalling the requirements above, we immediately see these problems with these triples:

1. `emp001` has two `hasID` triples.
2. The value of `emp002`'s ID has the wrong format (two



leading digits rather than 3).

3. *emp003* does not have a *hasID* triple.

We load the two TriG files into our repository, and end up with the following triple set. Note that all the employee triples use the default graph and the SHACL-related triples use the graph <https://franz.com#Shapes> specified in the TriG file.

s	p	o	g
Employee	rdf:type	rdfs:Class	
emp001	rdf:type	Employee	
emp001	hasID	"000-12-3456"	
emp001	hasID	"000-77-3456"	
emp002	rdf:type	Employee	
emp002	hasID	"00-56-3456"	
emp003	rdf:type	Employee	
EmployeeShape	property	_:b7A1D241Ax1	Shapes
_:b7A1D241Ax1	pattern	"^[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9]\$"	Shapes
_:b7A1D241Ax1	datatype	xs:string	Shapes
_:b7A1D241Ax1	maxCount	"1"	Shapes
_:b7A1D241Ax1	minCount	"1"	Shapes
_:b7A1D241Ax1	path	hasID	Shapes
EmployeeShape	targetClass	Employee	Shapes
EmployeeShape	rdf:type	NodeShape	Shapes

Now we use **agtool shacl-validate** to validate our data:

```
bin/agtool shacl-validate --data-graph default --shapes-graph
https://franz.com#Shapes shacl-repo-1
```

```
Validation report:          Does not conform
Created:                    2019-06-27T10:24:10
Number of shapes graphs:   1
Number of data graphs:     1
Number of NodeShapes:      1
Number of focus nodes checked: 3
```

3 validation results:

Result:

```
Focus node:      <https://franz.com#emp001>
Path:           <https://franz.com#hasID>
Source Shape:   _:b7A1D241Ax1
                Constraint                               Component:
<http://www.w3.org/ns/shacl#MaxCountConstraintComponent>
Severity:       <http://www.w3.org/ns/shacl#Violation>
```

Result:

```
Focus node:      <https://franz.com#emp002>
Path:           <https://franz.com#hasID>
Value:         "00-56-3456"
Source Shape:   _:b7A1D241Ax1
                Constraint
                Component:
<http://www.w3.org/ns/shacl#PatternConstraintComponent>
Severity:      <http://www.w3.org/ns/shacl#Violation>
```

```
Result:
Focus node:     <https://franz.com#emp003>
Path:          <https://franz.com#hasID>
Source Shape:   _:b7A1D241Ax1
                Constraint
                Component:
<http://www.w3.org/ns/shacl#MinCountConstraintComponent>
Severity:      <http://www.w3.org/ns/shacl#Violation>
```

The validation fails with the problems listed above. The **Focus node** is the subject of a triple that did not conform. **Path** is the predicate or a property path (predicates in this example). **Value** is the offending value. **Source Shape** is the shape that established the constraint (you must look at the shape triples to see exactly what **Source Shape** is requiring).

We revise our employee data with the following SPARQL expression, deleting one of the emp001 triples, deleting the emp002 triple and adding a new one with the correct format, and adding an emp003 triple.

```
prefix fr: <https://franz.com#>
```

```
DELETE DATA {fr:emp002 fr:hasID "00-56-3456" } ;
```

```
INSERT DATA {fr:emp002 fr:hasID "000-14-1772" } ;
```

```
DELETE DATA {fr:emp001 fr:hasID "000-77-3456" } ;
```

```
INSERT DATA {fr:emp003 fr:hasID "000-54-9662" } ;
```

Now our employee triples are

s	p	o	g
emp002	hasID	"000-14-1772"	
emp003	hasID	"000-54-9662"	
Employee	rdf:type	rdfs:Class	
emp001	rdf:type	Employee	
emp001	hasID	"000-12-3456"	
emp002	rdf:type	Employee	
emp003	rdf:type	Employee	

We run the validation again and are told our data conforms:

```
% bin/agtool shacl-validate --data-graph default --shapes-graph https://franz.com#Shapes shacl-repo-1
Validation report: Conforms
Created: 2019-06-27T10:32:19
Number of shapes graphs: 1
Number of data graphs: 1
Number of NodeShapes: 1
Number of focus nodes checked: 3
```

When we refer to this example in the remainder of this document, it is to the un-updated (incorrect) triples.

## SHACL API

The example above illustrates the SHACL steps:

1. Have a data set with triples that should conform to a shape
2. Have SHACL triples that express the desired shape
3. Run SHACL validation to determine if the data conforms

Note that SHACL validation does not modify the data being validated. Once you have the conformance report, you must modify the data to fix the conformance problems and then rerun the validation test.

The main entry point to the API is **agtool shacl-validate**. It takes various options and has several output choices. Online help for **agtool shacl-validate** is displayed by running `agtool shacl-validate --help`.

In order to validate triples, the system must know:

1. What triples to examine
2. What rules (SHACL triples) to use
3. What to do with the results

## Specifying what triples to examine

Two arguments to **agtool shacl-validate** specify the triples to evaluate: `--data-graph` and `--focus-node`. Each can be specified multiple times.

- The `--data-graph` argument specifies the graph value for triples to be examined. Its value must be an IRI or default. Only triples in the specified graphs will be examined. `default` specifies the default graph. It is also the default value of the `--data-graph` argument. If no value is specified for `--data-graph`, only triples in the default graph will be examined. If a value for `--data-graph` is specified, triples in the default graph will only be examined if `--data-graph default` is also specified.
- The `--focus-node` argument specifies IRIs which are subjects of triples. If this argument is specified, only triples with these subjects will be examined. To be examined, triples must also have graph values specified by `--data-graph` arguments. `--focus-node` does not have a default value. If unspecified, all triples in the specified data graphs will be examined. This argument can be specified multiple times.

The `--data-graph` argument was used in the [simple example](#) above. Here is how the `--focus-node` argument can be used to restrict validation to triples with subjects `<https://franz.com#emp002>` and `<https://franz.com#emp003>` and to ignore triples with subject `<https://franz.com#emp001>` (applying **agtool shacl-validate** to the original non-conformant data):

```
% bin/agtool shacl-validate --data-graph default \
```

```
--shapes-graph https://franz.com#Shapes \  
--focus-node https://franz.com#emp003 \  
--focus-node https://franz.com#emp002 shacl-repo-1  
Validation report:          Does not conform  
Created:                   2019-06-27T11:37:49  
Number of shapes graphs:   1  
Number of data graphs:     1  
Number of NodeShapes:     1  
Number of focus nodes checked: 2
```

2 validation results:

Result:

```
Focus node:      <https://franz.com#emp003>  
Path:           <https://franz.com#hasID>  
Source Shape:   _:b7A1D241Ax2  
                Constraint                Component:  
<http://www.w3.org/ns/shacl#MinCountConstraintComponent>  
Severity:       <http://www.w3.org/ns/shacl#Violation>
```

Result:

```
Focus node:      <https://franz.com#emp002>  
Path:           <https://franz.com#hasID>  
Value:          "00-56-3456"  
Source Shape:   _:b7A1D241Ax2  
                Constraint                Component:  
<http://www.w3.org/ns/shacl#PatternConstraintComponent>  
Severity:       <http://www.w3.org/ns/shacl#Violation>
```

## Specifying What Shape Triples to Use

Two arguments to **agtool shacl-validate**, analogous to the two arguments for data described above, specify Shape triples to use. Further, following the SHACL spec, data triples with predicate `<http://www.w3.org/ns/shacl#shapeGraph>` also specify graphs containing Shape triples to be used.

The arguments to **agtool shacl-validate** are the following. Each may be specified multiple times.

- The `--shapes-graph` argument specifies the graph value

for shape triples to be used for SHACL validation. Its value must be an IRI or default. default specifies the default graph. The `--shapes-graph` argument has no default value. If unspecified, graphs specified by data triples with the `<http://www.w3.org/ns/shacl#shapeGraph>` predicate will be used (they are used whether or not `--shapes-graph` has a value). If `--shapes-graph` has no value and there are no data triples with the `<http://www.w3.org/ns/shacl#shapeGraph>` predicate, the data graphs are used for shape graphs. (Shape triples have a known format and so can be identified among the data triples.)

- The `--shape` argument specifies IRIs which are subjects of shape nodes. If this argument is specified, only shape triples with these subjects and subsidiary triples to these will be used for validation. To be included, the triples must also have graph values specified by the `--shapes-graph` arguments or specified by a data triple with the `<http://www.w3.org/ns/shacl#shapeGraph>` predicate. `--shape` does not have a default value. If unspecified, all shapes in the shapes graphs will be used.

## Other APIs

There is a lisp API using the function [validate-data-graph](#), defined next:

```
validate-data-graphdb &key data-graph-iri/s shapes-graph-iri/s shape/s focus-node/s verbose conformance-only?  
function
```

Perform SHACL validation and return a validation-report structure.

The validation uses `data-graph-iri/s` to construct the

dataGraph. This can be a single IRI, a list of IRIs or NIL, in which case the default graph will be used. The shapesGraph can be specified using the shapes-graph-iri/s parameter which can also be a single IRI or a list of IRIs. If shape-graph-iri/s is not specified, the SHACL processor will first look to create the shapesGraph by finding triples with the predicate sh:shapeGraph in the dataGraph. If there are no such triples, then the shapesGraph will be assumed to be the same as the dataGraph.

Validation can be restricted to particular shapes and focus nodes using the shape/s and focus-node/s parameters. Each of these can be an IRI or list of IRIs.

If conformance-only? is true, then validation will stop as soon as any validation failures are detected.

You can use [validation-report-conforms-p](#) to see whether or not the dataGraph conforms to the shapesGraph (possibly restricted to just particular shape/s and focus-node/s).

The function [validation-report-conforms-p](#) returns t or nil as the validation struct returned by [validate-data-graph](#) does or does not conform.

validation-report-conforms-preport  
function

Returns t or nil to indicate whether or not REPORT (a validation-report struct) indicates that validation conformed. There is also a REST API. See [HTTP reference](#).

## Validation Output

The [simple example](#) above and the [SHACL examples](#) below show output from **agtool validate-shacl**. There are various output formats, specified by the --output option. Those examples use the plain format, which means printing results descriptively.

Other choices include json, trig, trix, turtle, nquads, rdf-n3, rdf/xml, and ntriples. Here are the [simple example](#) (uncorrected) results using ntriples output:

```
% bin/agtool shacl-validate --output ntriples --data-graph
default --shapes-graph https://franz.com#Shapes shacl-repo-1
```

```
_ :b271983AAx1
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/ns/shacl#ValidationReport> .
_ :b271983AAx1      <http://www.w3.org/ns/shacl#conforms>
"false"^^<http://www.w3.org/2001/XMLSchema#boolean> .
_ :b271983AAx1      <http://purl.org/dc/terms/created>
"2019-07-01T18:26:03"^^<http://www.w3.org/2001/XMLSchema#dateT
ime> .
_ :b271983AAx1      <http://www.w3.org/ns/shacl#result>
_ :b271983AAx2 .
_ :b271983AAx2
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/ns/shacl#ValidationResult> .
_ :b271983AAx2      <http://www.w3.org/ns/shacl#focusNode>
<https://franz.com#emp001> .
_ :b271983AAx2      <http://www.w3.org/ns/shacl#resultPath>
<https://franz.com#hasID> .
_ :b271983AAx2      <http://www.w3.org/ns/shacl#resultSeverity>
<http://www.w3.org/ns/shacl#Violation> .
_ :b271983AAx2
<http://www.w3.org/ns/shacl#sourceConstraintComponent>
<http://www.w3.org/ns/shacl#MaxCountConstraintComponent> .
_ :b271983AAx2      <http://www.w3.org/ns/shacl#sourceShape>
_ :b271983AAx3 .
_ :b271983AAx1      <http://www.w3.org/ns/shacl#result>
_ :b271983AAx4 .
_ :b271983AAx4
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/ns/shacl#ValidationResult> .
_ :b271983AAx4      <http://www.w3.org/ns/shacl#focusNode>
<https://franz.com#emp002> .
_ :b271983AAx4      <http://www.w3.org/ns/shacl#resultPath>
<https://franz.com#hasID> .
_ :b271983AAx4      <http://www.w3.org/ns/shacl#resultSeverity>
```



```

<http://www.w3.org/ns/shacl#Violation> .
_:b271983AAx4
<http://www.w3.org/ns/shacl#sourceConstraintComponent>
<http://www.w3.org/ns/shacl#PatternConstraintComponent> .
_:b271983AAx4 <http://www.w3.org/ns/shacl#sourceShape>
_:b271983AAx3 .
_:b271983AAx4 <http://www.w3.org/ns/shacl#value> "00-56-3456"
.
_:b271983AAx1 <http://www.w3.org/ns/shacl#result>
_:b271983AAx5 .
_:b271983AAx5
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/ns/shacl#ValidationResult> .
_:b271983AAx5 <http://www.w3.org/ns/shacl#focusNode>
<https://franz.com#emp003> .
_:b271983AAx5 <http://www.w3.org/ns/shacl#resultPath>
<https://franz.com#hasID> .
_:b271983AAx5 <http://www.w3.org/ns/shacl#resultSeverity>
<http://www.w3.org/ns/shacl#Violation> .
_:b271983AAx5
<http://www.w3.org/ns/shacl#sourceConstraintComponent>
<http://www.w3.org/ns/shacl#MinCountConstraintComponent> .
_:b271983AAx5 <http://www.w3.org/ns/shacl#sourceShape>
_:b271983AAx3 .

```

You can have the triples added to the repository by specifying the `--add-to-repo` option true.

In the plain output information is provided about how many data graphs are examined, how many shape graphs were specified and node shapes are found, and how many focus nodes are checked. If zero focus nodes are checked, that is likely not what you want and something has gone wrong. Here we mis-spell the name of the shape graph (`https://franz.com#shapes` instead of `https://franz.com#Shapes`) and get 0 focus nodes checked:

```

% bin/agtool shacl-validate --data-graph default --shapes-graph https://franz.com#shapes shacl-repo-1
Validation report:           Conforms
Created:                     2019-06-28T10:34:22
Number of shapes graphs:     1

```

Number of data graphs: 1  
Number of NodeShapes: 0  
Number of focus nodes checked: 0

## SPARQL integration

There are two sets of magic properties defined: one checks for basic conformance and the other produces validation reports as triples:

- `?valid franz:shaclConforms ( ?dataGraph [ ?shapesGraph ] )`
- `?valid franz:shaclFocusNodeConforms1 ( ?dataGraph ?nodeOrNodeCollection )`
- `?valid franz:shaclFocusNodeConforms2 ( ?dataGraph ?shapesGraph ?nodeOrNodeCollection )`
- `?valid franz:shaclShapeConforms1 ( ?dataGraph ?shapeOrShapeCollection [ ?nodeOrNodeCollection ] )`
- `?valid franz:shaclShapeConforms2 ( ?dataGraph ?shapesGraph ?shapeOrShapeCollection [ ?nodeOrNodeCollection ] )`
- `(?s ?p ?o) franz:shaclValidationReport ( ?dataGraph [ ?shapesGraph ] )`
- `(?s ?p ?o) franz:shaclFocusNodeValidationReport1 ( ?dataGraph ?nodeOrNodeCollection ) .`
- `(?s ?p ?o) franz:shaclFocusNodeValidationReport2 ( ?dataGraph ?shapesGraph ?nodeOrNodeCollection ) .`
- `(?s ?p ?o) franz:shaclShapeValidationReport1 ( ?dataGraph ?shapeOrShapeCollection [ ?nodeOrNodeCollection ] ) .`
- `(?s ?p ?o) franz:shaclShapeValidationReport2 ( ?dataGraph ?shapesGraph ?shapeOrShapeCollection [ ?nodeOrNodeCollection ] ) .`

In all of the above `?dataGraph` and `?shapesGraph` can be IRIs, the literal 'default', or a variable that is bound to a SPARQL collection (list or set) that was previously created with a

function

like <https://franz.com/ns/allegrograph/6.5.0/fn#makeSPARQLList>  
or <https://franz.com/ns/allegrograph/6.5.0/fn#lookupRdfList>.

If a collection is used, then the SHACL processor will create a temporary RDF merge of all of the graphs in it to produce the data graph or the shapes graph.

Similarly, `?shapeOrShapeCollection` and `?nodeOrNodeCollection` can be bound to an IRI or a SPARQL collection. If a collection is used, then it must be bound to a list of IRIs. The SHACL processor will restrict validation to the shape(s) and focus node(s) (i.e. nodes that should be validated) specified.

The `shapesGraph` argument is optional in both of the `shaclConforms` and `shaclValidationReport` magic properties. If the `shapesGraph` is not specified, then the `shapesGraph` will be created by following triples in the `dataGraph` that use the `sh:shapesGraph` predicate. If there are no such triples, then the `shapesGraph` will be the same as the `dataGraph`.

For example, the following SPARQL expression

```
construct { ?s ?p ?o } where {  
  # form a collection of focusNodes  
  bind(<https://franz.com/ns/allegrograph/6.6.0/fn#makeSPARQLList>  
<http://Journal1/1942/Article25>,  
  <http://Journal1/1943>) as ?nodes)  
  (?s ?p ?o)  
<https://franz.com/ns/allegrograph/6.6.0/shaclShapeValidationReport1>  
  ('default' <ex://franz.com/documentShape1> ?nodes) .  
}
```

would use the default graph as the Data Graph and the Shapes Graph and then validate two focus nodes against the shape `<ex://franz.com/documentShape1>`.

# SHACL Example

We build on our [simple example](#) above. Start with a fresh repository so triples from the simple example do not interfere with this example.

We start with a TriG file with various shapes defined on some classes.

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix fr: <https://franz.com#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<https://franz.com#ShapesGraph> {
fr:EmployeeShape
  a sh:NodeShape ;
  sh:targetClass fr:Employee ;
  sh:property [
    ## Every employee must have exactly one ID
    sh:path fr:hasID ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
    sh:pattern "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]$" ;
  ] ;
  sh:property [
    ## Every employee is a manager or a worker
    sh:path fr:employeeType ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
    sh:in ("Manager" "Worker") ;
  ] ;
  sh:property [
    ## If birthyear supplied, must be 2001 or before
    sh:path fr:birthYear ;
    sh:maxInclusive 2001 ;
    sh:datatype xsd:integer ;
```

```

] ;
sh:property [
  ## Must have a title, may have more than one
  sh:path fr:hasTitle ;
  sh:datatype xsd:string ;
  sh:minCount 1 ;
] ;

sh:or (
  ## The President does not have a supervisor
  [
    sh:path fr:hasTitle ;
    sh:hasValue "President" ;
  ]
  [
    ## Must have a supervisor
    sh:path fr:hasSupervisor ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:class fr:Employee ;
  ]
) ;

sh:or (
  # Every employee must either have a wage or a salary
  [
    sh:path fr:hasSalary ;
    sh:datatype xsd:integer ;
    sh:minInclusive 3000 ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ]
  [
    sh:path fr:hasWage ;
    sh:datatype xsd:decimal ;
    sh:minExclusive 15.00 ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ]
)
.

```

```
}
```

This file says the following about instances of the class `fr:Employee`:

1. Every employee must have exactly one ID (object of `fr:hasID`), a string of the form `NNN-NN-NNNN` where the `Ns` are digits (this is the simple example requirement).
2. Every employee must have exactly one `fr:employeeType` triple with value either "Manager" or "Worker".
3. Employees may have a `fr:birthYear` triple, and if so, the value must be 2001 or earlier.
4. Employees must have a `fr:hasTitle` and may have more than one.
5. All employees except the one with title "President" must have a supervisor (specified with `fr:hasSupervisor`).
6. Every employee must either have a wage (a decimal specifying hourly pay, greater than 15.00) or a salary (an integer specifying monthly pay, greater than or equal to 3000).

Here is some employee data:

```
@prefix fr: <https://franz.com#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

```
{
  fr:Employee
    a rdfs:Class .

  fr:emp001
    a fr:Employee ;
    fr:hasID "000-12-3456" ;
    fr:hasTitle "President" ;
    fr:employeeType "Manager" ;
    fr:birthYear "1953"^^xsd:integer ;
```

fr:hasSalary "10000"^^xsd:integer .

fr:emp002

a fr:Employee ;  
fr:hasID "000-56-3456" ;  
fr:hasTitle "Foreman" ;  
fr:employeeType "Worker" ;  
fr:birthYear "1966"^^xsd:integer ;  
fr:hasSupervisor fr:emp003 ;  
fr:hasWage "20.20"^^xsd:decimal .

fr:emp003

a fr:Employee ;  
fr:hasID "000-77-3232" ;  
fr:hasTitle "Production Manager" ;  
fr:employeeType "Manager" ;  
fr:birthYear "1968"^^xsd:integer ;  
fr:hasSupervisor fr:emp001 ;  
fr:hasSalary "4000"^^xsd:integer .

fr:emp004

a fr:Employee ;  
fr:hasID "000-88-3456" ;  
fr:hasTitle "Fitter" ;  
fr:employeeType "Worker" ;  
fr:birthYear "1979"^^xsd:integer ;  
fr:hasSupervisor fr:emp002 ;  
fr:hasWage "17.20"^^xsd:decimal .

fr:emp005

a fr:Employee ;  
fr:hasID "000-99-3492" ;  
fr:hasTitle "Fitter" ;  
fr:employeeType "Worker" ;  
fr:birthYear "2000"^^xsd:integer ;  
fr:hasWage "17.20"^^xsd:decimal .

fr:emp006

a fr:Employee ;  
fr:hasID "000-78-5592" ;  
fr:hasTitle "Filer" ;

```

fr:employeeType "Intern" ;
fr:birthYear "2003"^^xsd:integer ;
fr:hasSupervisor fr:emp002 ;
fr:hasWage "14.20"^^xsd:decimal .

fr:emp007
a fr:Employee ;
fr:hasID "000-77-3232" ;
fr:hasTitle "Sales Manager" ;
fr:hasTitle "Vice President" ;
fr:employeeType "Manager" ;
fr:birthYear "1962"^^xsd:integer ;
fr:hasSupervisor fr:emp001 ;
fr:hasSalary "7000"^^xsd:integer .
}

```

Comparing these data with the requirements, we see these problems:

1. emp005 does not have a supervisor.
2. emp006 is pretty messed up, with (1) employeeType "Intern", not an allowed value, (2) a birthYear (2003) later than the required maximum of 2001, and (3) a wage (14.40) less than the minimum (15.00).

Otherwise the data seems OK.

We load these two TriG files into an empty repository (which we have named **shacl-repo-2**). We specify the default graph for the data and the <https://franz.com#ShapesGraph> for the shapes. (Though not required, it is a good idea to specify a graph for shape data as it makes it easy to delete and reload shapes while developing.) We have 101 triples, 49 data and 52 shape. Then we run **agtool shacl-validate**:

```

% bin/agtool shacl-validate --shapes-graph
https://franz.com#ShapesGraph --data-graph default shacl-
repo-2

```

There are four violations, as expected, one for emp005 and three for emp006.



Validation report: Does not conform  
Created: 2019-07-03T11:35:27  
Number of shapes graphs: 1  
Number of data graphs: 1  
Number of NodeShapes: 1  
Number of focus nodes checked: 7

#### 4 validation results:

##### Result:

Focus node: <https://franz.com#emp005>  
Value: <https://franz.com#emp005>  
Source Shape: <https://franz.com#EmployeeShape>  
Constraint Component:  
<https://www.w3.org/ns/shacl#OrConstraintComponent>  
Severity: <https://www.w3.org/ns/shacl#Violation>

##### Result:

Focus node: <https://franz.com#emp006>  
Path: <https://franz.com#employeeType>  
Value: "Intern"  
Source Shape: \_:b19D062B9x221  
Constraint Component:  
<http://www.w3.org/ns/shacl#InConstraintComponent>  
Severity: <http://www.w3.org/ns/shacl#Violation>

##### Result:

Focus node: <https://franz.com#emp006>  
Path: <https://franz.com#birthYear>  
Value:  
"2003"^^<http://www.w3.org/2001/XMLSchema#integer>  
Source Shape: \_:b19D062B9x225  
Constraint Component:  
<http://www.w3.org/ns/shacl#MaxInclusiveConstraintComponent>  
Severity: <http://www.w3.org/ns/shacl#Violation>

##### Result:

Focus node: <https://franz.com#emp006>  
Value: <https://franz.com#emp006>  
Source Shape: <https://franz.com#EmployeeShape>  
Constraint Component:  
<http://www.w3.org/ns/shacl#OrConstraintComponent>

Severity: `<http://www.w3.org/ns/shacl#Violation>`

Fixing the data is left as an exercise for the reader.

---

## Using JSON-LD in AllegroGraph – Python Example



The following is example #19 from our [AllegroGraph Python Tutorial](#).

JSON-LD is described pretty well at <https://json-ld.org/> and the specification can be found at <https://json-ld.org/latest/json-ld/>.

The website <https://json-ld.org/playground/> is also useful.

There are many reasons for working with JSON-LD. The major search engines such as Google require ecommerce companies to mark up their websites with a systematic description of their products and more and more companies use it as an easy serialization format to share data.

The benefit for your organization is that you can now combine your documents with graphs, graph search and graph algorithms. Normally when you store documents in a document store you set up your documents in such a way that it is optimized for direct retrieval queries. Doing complex joins for multiple types of documents or even doing a shortest path through a mass of object (types) is however very complicated. Storing JSON-LD objects in AllegroGraph gives you all the benefits of a document store *and* you can semantically link objects

together, do complex joins and even graph search.

A second benefit is that, as an application developer, you do not have to learn the entire semantic technology stack, especially the part where developers have to create individual triples or edges. You can work with the JSON data serialization format that application developers usually prefer.

In the following you will first learn about JSON-LD as a syntax for semantic graphs. After that we will talk more about using JSON-LD with AllegroGraph as a document-graph-store.

## Setup

You can use Python 2.6+ or Python 3.3+. There are small setup differences which are noted. You do need *agraph-python-101.0.1* or later.

Mimicking instructions in the Installation document, you should set up the [virtualenv](#) environment.

1. Create an environment named *jsonld*:

```
python3 -m venv jsonld
```

or

```
python2 -m virtualenv jsonld
```

2. Activate it:

Using the Bash shell:

```
source jsonld/bin/activate
```

Using the C shell:

```
source jsonld/bin/activate.csh
```

### 3. Install *agraph-python*:

```
pip install agraph-python
```

And start **python**:

```
python  
[various startup and copyright messages]  
>>>
```

We assume you have an AllegroGraph 6.5.0 server running. We call **ag\_connect**. Modify the *host*, *port*, *user*, and *password* in your call to their correct values:

```
from franz.openrdf.connect import ag_connect  
with ag_connect('repo', host='localhost', port='10035',  
               user='test', password='xyzzzy') as conn:  
    print (conn.size())
```

If the script runs successfully a new repository named *repo* will be created.

## JSON-LD setup

We next define some utility functions which are somewhat different from what we have used before in order to work better with JSON-LD. **createdb()** creates and opens a new repository and **opendb()** opens an existing repo (modify the values of *host*, *port*, *user*, and *password* arguments in the definitions if necessary). Both return repository connections which can be used to perform repository operations. **showtriples()** displays triples in a repository.

```
import os  
import json, requests, copy
```

```

from franz.openrdf.sail.allegrographserver import
AllegroGraphServer
from franz.openrdf.connect import ag_connect
from franz.openrdf.vocabulary.xmlschema import XMLSchema
from franz.openrdf.rio.rdfformat import RDFFormat

# Functions to create/open a repo and return a
RepositoryConnection
# Modify the values of HOST, PORT, USER, and PASSWORD if
necessary

def createdb(name):
    return
    ag_connect(name,host="localhost",port=10035,user="test",passwo
rd="xyzyzy",create=True,clear=True)

def opendb(name):
    return
    ag_connect(name,host="localhost",port=10035,user="test",passwo
rd="xyzyzy",create=False)

def showtriples(limit=100):
    statements = conn.getStatements(limit=limit)
    with statements:
        for statement in statements:
            print(statement)

```

Finally we call our **createdb** function to create a repository and return a *RepositoryConnection* to it:

```
conn=createdb('jsonplay')
```

## Some Examples of Using JSON-LD

In the following we try things out with some JSON-LD objects that are defined in json-ld playground: [jsonld](#)

The first object we will create is an *event dict*. Although it is a Python dict, it is also valid JSON notation. (But note that not all Python dictionaries are valid JSON. For example,

JSON uses null where Python would use None and there is no magic to automatically handle that.) This object has one key called @context which specifies how to translate keys and values into predicates and objects. The following @context says that every time you see ical: it should be replaced by http://www.w3.org/2002/12/cal/ical#, xsd: by http://www.w3.org/2001/XMLSchema#, and that if you see ical:dtstart as a key then the value should be treated as an xsd:dateTime.

```
event = {
  "@context": {
    "ical": "http://www.w3.org/2002/12/cal/ical#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ical:dtstart": { "@type": "xsd:dateTime" }
  },
  "ical:summary": "Lady Gaga Concert",
  "ical:location": "New Orleans Arena, New Orleans, Louisiana, USA",
  "ical:dtstart": "2011-04-09T20:00:00Z"
}
```

Let us try it out (the subjects are blank nodes so you will see different values):

```
>>> conn.addData(event)
>>> showtriples()
( _:b197D2E01x1, <http://www.w3.org/2002/12/cal/ical#summary>,
  "Lady Gaga Concert")
( _:b197D2E01x1, <http://www.w3.org/2002/12/cal/ical#location>,
  "New Orleans Arena, New Orleans, Louisiana, USA")
( _:b197D2E01x1, <http://www.w3.org/2002/12/cal/ical#dtstart>,
  "2011-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#dateTime>)
```

## Adding an @id and @type to Objects

In the above we see that the JSON-LD was correctly translated into triples but there are two immediate problems: first each

subject is a blank node, the use of which is problematic when linking across repositories; and second, the object does not have an RDF type. We solve these problems by adding an `@id` to provide an IRI as the subject and adding a `@type` for the object (those are at the lines just after the `@context` definition):

```
>>> event = {
  "@context": {
    "ical": "http://www.w3.org/2002/12/cal/ical#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ical:dtstart": { "@type": "xsd:dateTime" }
  },
  "@id": "ical:event-1",
  "@type": "ical:Event",
  "ical:summary": "Lady Gaga Concert",
  "ical:location": "New Orleans Arena, New Orleans,
Louisiana, USA",
  "ical:dtstart": "2011-04-09T20:00:00Z"
}
```

We also create a test function to test our JSON-LD objects. It is more powerful than needed right now (here we just need `conn.addData(event)` and `showTriples()` but `test` will be useful in most later examples. Note the `allow_external_references=True` argument to `addData()`. Again, not needed in this example but later examples use external contexts and so this argument is required for those.

```
def
test(object, json_ld_context=None, rdf_context=None, maxPrint=100
, conn=conn):
    conn.clear()
    conn.addData(object, allow_external_references=True)
    showtriples(limit=maxPrint)
```

```
>>> test(event)
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/2002/12/cal/ical#summary>, "Lady Gaga
```

```
Concert")
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/2002/12/cal/ical#location>, "New Orleans
Arena, New Orleans, Louisiana, USA")
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/2002/12/cal/ical#dtstart>,
"2011-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://www.w3.org/2002/12/cal/ical#Event>)
```

Note in the above that we now have a proper subject and a type.

## Referencing a External Context Via a URL

The next object we add to AllegroGraph is a person object. This time the @context is not specified as a JSON object but as a link to a context that is stored at <http://schema.org/>. Also in the definition of the function test above we had this parameter in `addData:allow_external_references=True`. Requiring that argument explicitly is a security feature. One should use external references only that context at that URL is trusted (as it is in this case).

```
person = {
  "@context": "http://schema.org/",
  "@type": "Person",
  "@id": "foaf:person-1",
  "name": "Jane Doe",
  "jobTitle": "Professor",
  "telephone": "(425) 123-4567",
  "url": "http://www.janedoe.com"
}
```

```
>>> test(person)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/name>, "Jane Doe")
```



```
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/jobTitle>, "Professor")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/telephone>, "(425) 123-4567")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/url>, <http://www.janedoe.com>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://schema.org/Person>)
```

## Improving Performance by Adding Lists

Adding one person at a time requires doing an interaction with the server for each person. It is much more efficient to add lists of objects all at once rather than one at a time. Note that `addData` will take a list of dicts and still do the right thing. So let us add a 1000 persons at the same time, each person being a copy of the above person but with a different `@id`. (The example code is repeated below for ease of copying.)

```
>>> x = [copy.deepcopy(person) for i in range(1000)]
>>> len(x)
1000
>>> c = 0
>>> for el in x:
    el['@id'] = "http://franz.com/person-" + str(c)
    c = c + 1
>>> test(x,maxPrint=10)
(<http://franz.com/person-0>, <http://schema.org/name>, "Jane
Doe")
(<http://franz.com/person-0>, <http://schema.org/jobTitle>,
"Professor")
(<http://franz.com/person-0>, <http://schema.org/telephone>,
"(425) 123-4567")
(<http://franz.com/person-0>, <http://schema.org/url>,
<http://www.janedoe.com>)
(<http://franz.com/person-0>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ,
```

```

<http://schema.org/Person>)
(<http://franz.com/person-1>, <http://schema.org/name>, "Jane
Doe")
(<http://franz.com/person-1>, <http://schema.org/jobTitle>,
"Professor")
(<http://franz.com/person-1>, <http://schema.org/telephone>,
"(425) 123-4567")
(<http://franz.com/person-1>, <http://schema.org/url>,
<http://www.janedoe.com>)
(<http://franz.com/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://schema.org/Person>)
>>> conn.size()
5000
>>>

```

```

x = [copy.deepcopy(person) for i in range(1000)]
len(x)

```

```

c = 0
for el in x:
    el['@id'] = "http://franz.com/person-" + str(c)
    c = c + 1

```

```

test(x,maxPrint=10)

```

```

conn.size()

```

## Adding a Context Directly to an Object

You can download a context directly in Python, modify it and then add it to the object you want to store. As an illustration we load a person context from [json-ld.org](http://json-ld.org) (actually a fragment of the schema.org context) and insert it in a person object. (We have broken and truncated some output lines for clarity and all the code executed is repeated below for ease of copying.)

```

>>>

```

```

context=requests.get("https://json-ld.org/contexts/person.json
ld").json()['@context']
>>> context
{'Person': 'http://xmlns.com/foaf/0.1/Person',
 'xsd': 'http://www.w3.org/2001/XMLSchema#',
 'name': 'http://xmlns.com/foaf/0.1/name',
 'jobTitle': 'http://xmlns.com/foaf/0.1/title',
 'telephone': 'http://schema.org/telephone',
 'nickname': 'http://xmlns.com/foaf/0.1/nick',
 'affiliation': 'http://schema.org/affiliation',
 'depiction': {'@id': 'http://xmlns.com/foaf/0.1/depiction',
 '@type': '@id'},
 'image': {'@id': 'http://xmlns.com/foaf/0.1/img', '@type':
 '@id'},
 'born': {'@id': 'http://schema.org/birthDate', '@type':
 'xsd:date'},
 ...}
>>> person = {
    "@context": context,
    "@type": "Person",
    "@id": "foaf:person-1",
    "name": "Jane Doe",
    "jobTitle": "Professor",
    "telephone": "(425) 123-4567",
}
>>> test(person)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/name>, "Jane Doe")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/title>, "Professor")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/telephone>, "(425) 123-4567")
(<http://xmlns.com/foaf/0.1/person-1>,
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
 <http://xmlns.com/foaf/0.1/Person>)
>>>

context=requests.get("https://json-ld.org/contexts/person.json
ld").json()['@context']
# The next produces lots of output, uncomment if desired
#context

```

```
person = {
  "@context": context,
  "@type": "Person",
  "@id": "foaf:person-1",
  "name": "Jane Doe",
  "jobTitle": "Professor",
  "telephone": "(425) 123-4567",
}
test(person)
```

## Building a Graph of Objects

We start by forcing a key's value to be stored as a resource. We saw above that we could specify the value of a key to be a date using the `xsd:dateTime` specification. We now do it again for `foaf:birthdate`. Then we created several linked objects and show the connections using Gruff.

```
context = { "foaf:child": {"@type":"@id"},
            "foaf:brotherOf": {"@type":"@id"},
            "foaf:birthdate": {"@type":"xsd:dateTime"}}
```

```
p1 = {
  "@context": context,
  "@type": "foaf:Person",
  "@id": "foaf:person-1",
  "foaf:birthdate": "1958-04-09T20:00:00Z",
  "foaf:child": ['foaf:person-2', 'foaf:person-3']
}
```

```
p2 = {
  "@context": context,
  "@type": "foaf:Person",
  "@id": "foaf:person-2",
  "foaf:brotherOf": "foaf:person-3",
  "foaf:birthdate": "1992-04-09T20:00:00Z",
}
```

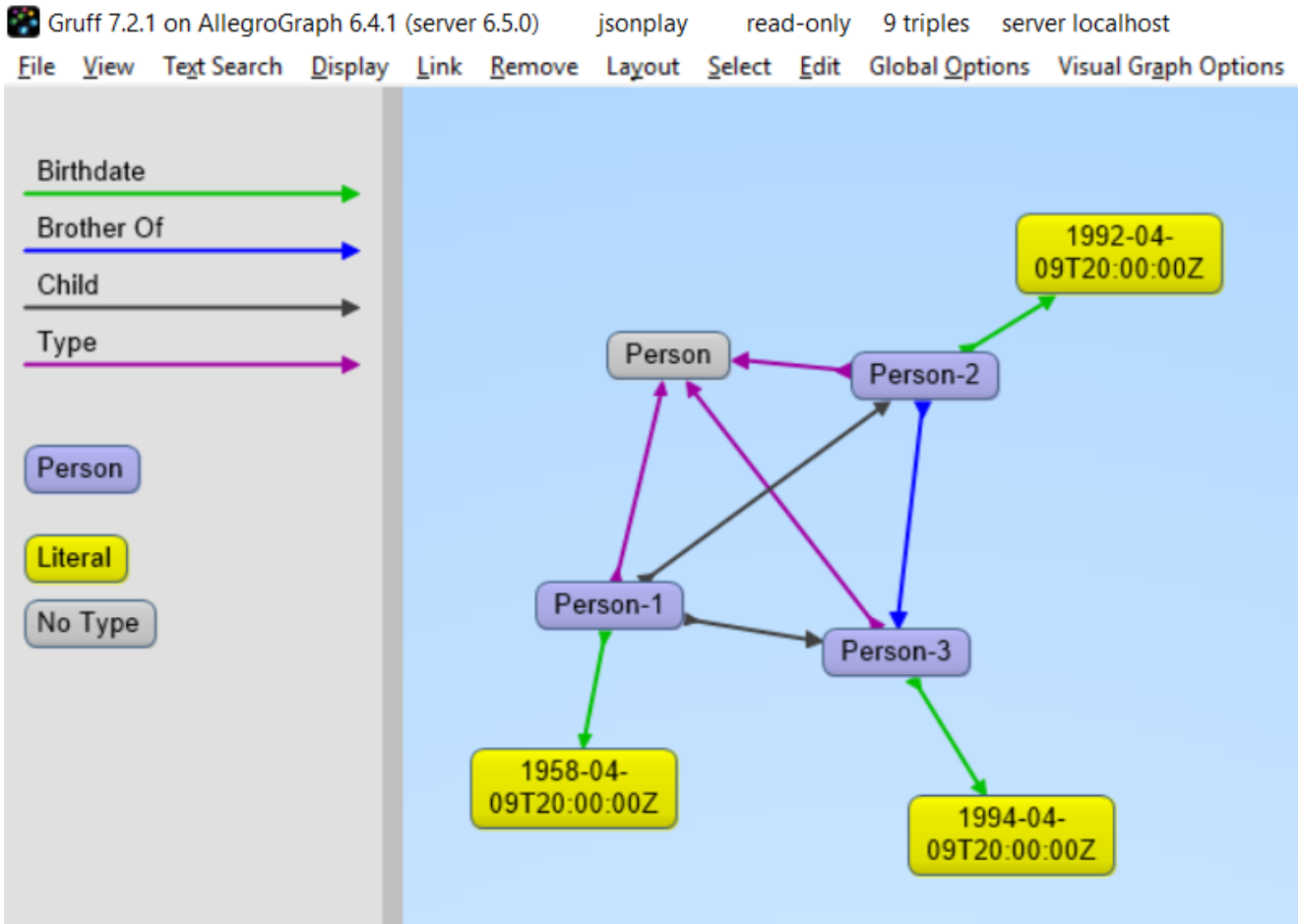
```
p3 = {"@context": context,
      "@type": "foaf:Person",
```

```
"@id": "foaf:person-3",
"foaf:birthdate": "1994-04-09T20:00:00Z",
}
```

```
test([p1,p2,p3])
```

```
>>> test([p1,p2,p3])
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/birthdate>,
"1958-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/child>,
<http://xmlns.com/foaf/0.1/person-2>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/child>,
<http://xmlns.com/foaf/0.1/person-3>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
(<http://xmlns.com/foaf/0.1/person-2>,
<http://xmlns.com/foaf/0.1/brotherOf>,
<http://xmlns.com/foaf/0.1/person-3>)
(<http://xmlns.com/foaf/0.1/person-2>,
<http://xmlns.com/foaf/0.1/birthdate>,
"1992-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://xmlns.com/foaf/0.1/person-2>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
(<http://xmlns.com/foaf/0.1/person-3>,
<http://xmlns.com/foaf/0.1/birthdate>,
"1994-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://xmlns.com/foaf/0.1/person-3>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
```

The following shows the graph that we created in Gruff. Note that this is what JSON-LD is all about: connecting objects together.



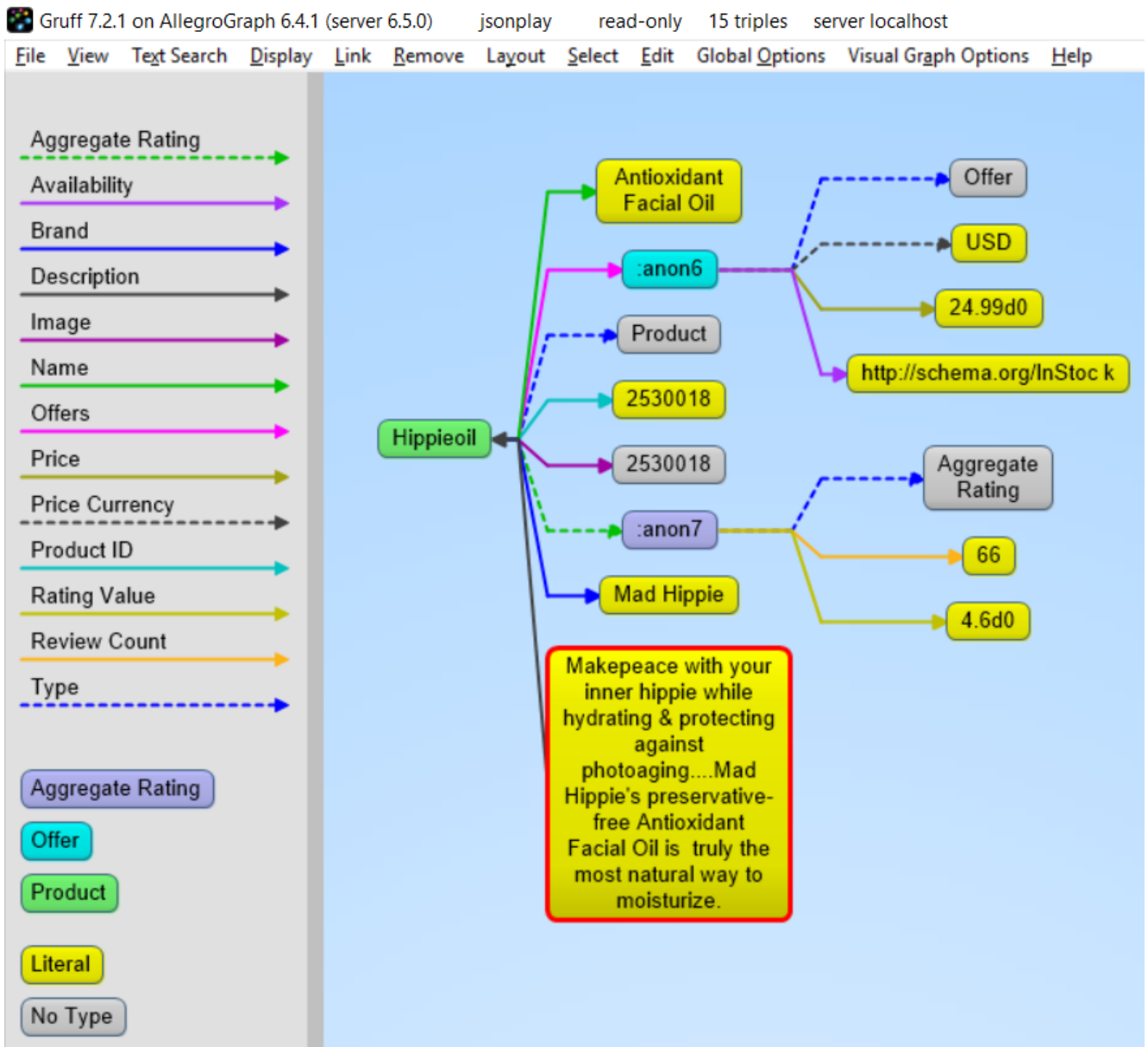
## JSON-LD Keyword Directives can be Added at any Level

Here is an example from the wild. The URL <https://www.ulta.com/antioxidant-facial-oil?productId=xlsImpprod18731241> goes to a web page advertising a facial oil. (We make no claims or recommendations about this product. We are simply showing how JSON-LD appears in many places.) Look at the source of the page and you'll find a JSON-LD object similar to the following. Note that @ directives go to any level. We added an @id key.

```
hippieoil = {"@context":"http://schema.org",
"@type":"Product",
"@id":"http://franz.com/hippieoil",
"aggregateRating":
  {"@type":"AggregateRating",
   "ratingValue":4.6,
```

```
"reviewCount":73},
  "description":"""Make peace with your inner hippie while
hydrating & protecting against photoaging....Mad Hippie's
preservative-free Antioxidant Facial Oil is truly the most
natural way to moisturize.""",
  "brand":"Mad Hippie",
  "name":"Antioxidant Facial Oil",
  "image":"https://images.ultra.com/is/image/Ulta/2530018",
  "productID":"2530018",
  "offers":
    {"@type":"Offer",
      "availability":"http://schema.org/InStock",
      "price":"24.99",
      "priceCurrency":"USD"}}
```

```
test(hippieoil)
```



## JSON-LD @graphs

One can put one or more JSON-LD objects in an RDF named graph. This means that the fourth element of each triple generated from a JSON-LD object will have the specified graph name. Let's show in an example.

```
context = {
  "name": "http://schema.org/name",
  "description": "http://schema.org/description",
  "image": {
    "@id": "http://schema.org/image", "@type": "@id"
  },
}
```



```

    "geo": "http://schema.org/geo",
    "latitude": {
        "@id": "http://schema.org/latitude", "@type":
"xsd:float" },
    "longitude": {
        "@id": "http://schema.org/longitude", "@type":
"xsd:float" },
    "xsd": "http://www.w3.org/2001/XMLSchema#"
}

```

```

place = {
    "@context": context,
    "@id": "http://franz.com/place1",
    "@graph": {
        "@id": "http://franz.com/place1",
        "@type": "http://franz.com/Place",
        "name": "The Empire State Building",
        "description": "The Empire State Building is a 102-
story landmark in New York City.",
        "image":
"http://www.civil.usherbrooke.ca/cours/gci215a/empire-state-bu
ilding.jpg",
        "geo": {
            "latitude": "40.75",
            "longitude": "73.98" }
    }}

```

and here is the result:

```

>>> test(place, maxPrint=3)
(<http://franz.com/place1>, <http://schema.org/name>, "The
Empire State Building", <http://franz.com/place1>)
(<http://franz.com/place1>, <http://schema.org/description>,
"The Empire State Building is a 102-story landmark in New York
City.", <http://franz.com/place1>)
(<http://franz.com/place1>, <http://schema.org/image>,
<http://www.civil.usherbrooke.ca/cours/gci215a/empire-state-bu
ilding.jpg>, <http://franz.com/place1>)
>>>

```

Note that the fourth element (graph) of each of the triples is

<http://franz.com/placel>. If you don't add the @id the triples will be put in the default graph.

Here a slightly more complex example:

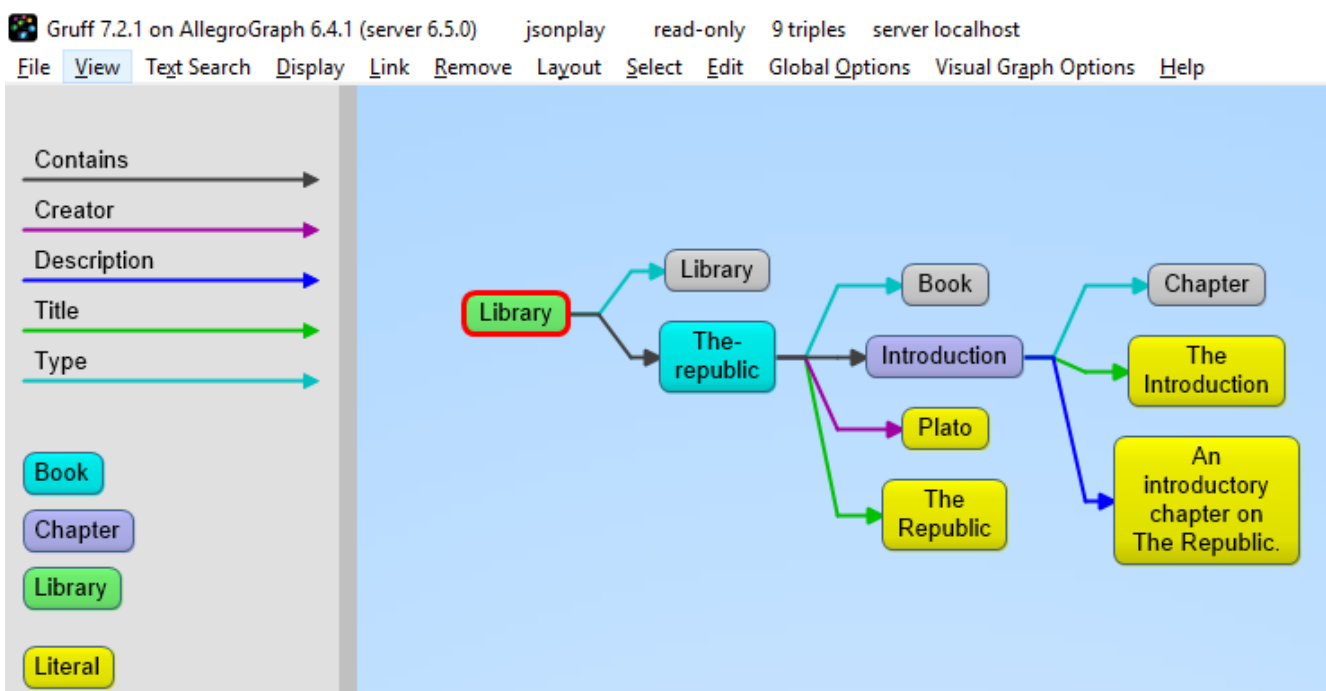
```
library = {
  "@context": {
    "dc": "http://purl.org/dc/elements/1.1/",
    "ex": "http://example.org/vocab#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ex:contains": {
      "@type": "@id"
    }
  },
  "@id": "http://franz.com/mygraph1",
  "@graph": [
    {
      "@id": "http://example.org/library",
      "@type": "ex:Library",
      "ex:contains": "http://example.org/library/the-republic"
    },
    {
      "@id": "http://example.org/library/the-republic",
      "@type": "ex:Book",
      "dc:creator": "Plato",
      "dc:title": "The Republic",
      "ex:contains":
"http://example.org/library/the-republic#introduction"
    },
    {
      "@id":
"http://example.org/library/the-republic#introduction",
      "@type": "ex:Chapter",
      "dc:description": "An introductory chapter on The
Republic.",
      "dc:title": "The Introduction"
    }
  ]
}
```

With the result:

```

>>> test(library, maxPrint=3)
(<http://example.org/library>,
<http://example.org/vocab#contains>,
<http://example.org/library/the-republic>,
<http://franz.com/mygraph1>) (<http://example.org/library>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://example.org/vocab#Library>,
<http://franz.com/mygraph1>)
(<http://example.org/library/the-republic>,
<http://purl.org/dc/elements/1.1/creator>,
"Plato", <http://franz.com/mygraph1>)
>>>

```



## JSON-LD as a Document Store

So far we have treated JSON-LD as a syntax to create triples. Now let us look at the way we can start using AllegroGraph as a combination of a document store and graph database at the same time. And also keep in mind that we want to do it in such a way that you as a Python developer can add documents such as dictionaries and also retrieve values or documents as dictionaries.

## Setup

The [Python source file jsonld\\_tutorial\\_helper.py](#) contains various definitions useful for the remainder of this example. Once it is downloaded, do the following (after adding the path to the filename):

```
conn=createdb("docugraph")
from jsonld_tutorial_helper import *
addNamespace(conn,"jsonldmeta","http://franz.com/ns/allegrograph/6.4/load-meta#")
addNamespace(conn,"ical","http://www.w3.org/2002/12/cal/ical#"
)
```

Let's use our event structure again and see how we can store this JSON document in the store as a document. Note that the `addData` call includes the keyword: `json_ld_store_source=True`.

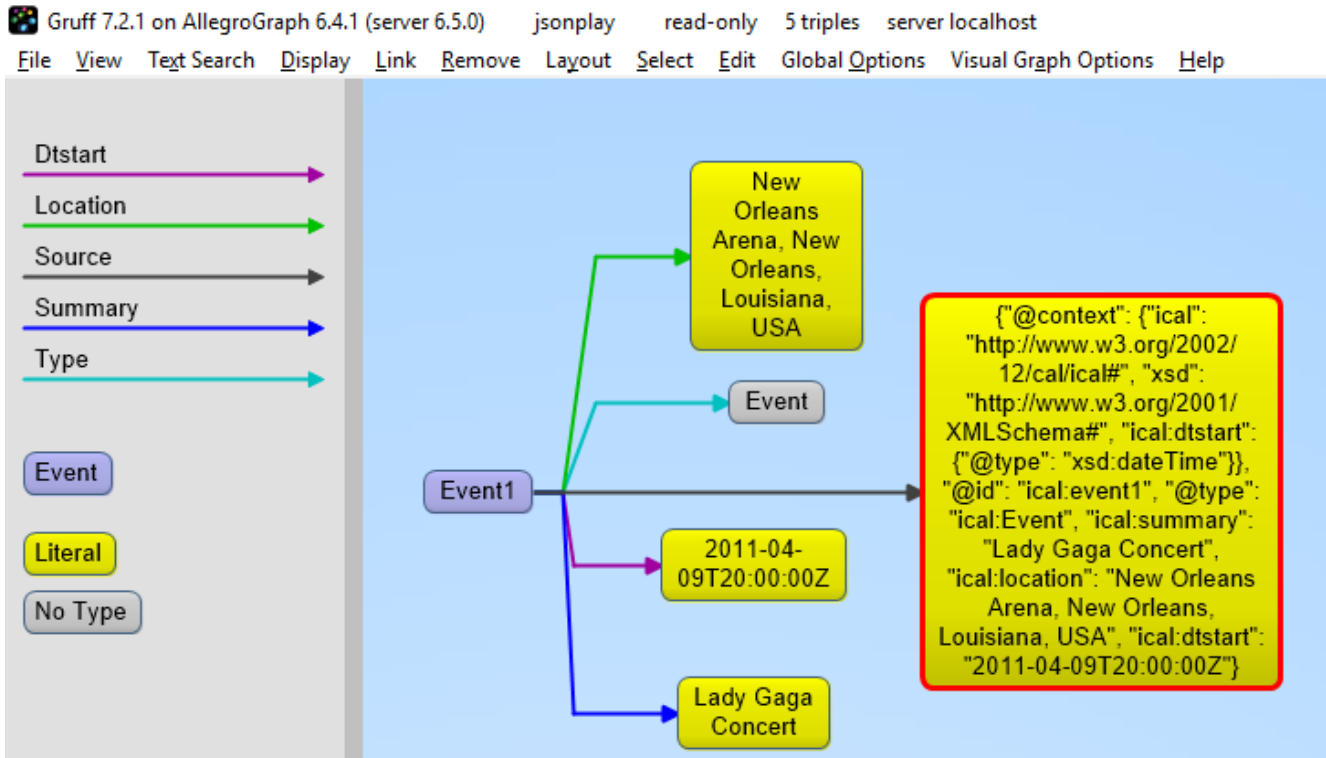
```
event = {
  "@context": {
    "@id": "ical:event1",
    "@type": "ical:Event",
    "ical": "http://www.w3.org/2002/12/cal/ical#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ical:dtstart": { "@type": "xsd:dateTime" }
  },
  "ical:summary": "Lady Gaga Concert",
  "ical:location":
  "New Orleans Arena, New Orleans, Louisiana, USA",
  "ical:dtstart": "2011-04-09T20:00:00Z"
}
```

```
>>> conn.addData(event,
allow_external_references=True,json_ld_store_source=True)
```

The `jsonld_tutorial_helper.py` file defines the function `store` as simple wrapper around `addData` that always saves the JSON source. For experimentation reasons it also has a parameter `fresh` to clear out the repository first.

```
>>> store(conn,event, fresh=True)
```

If we look at the triples in Gruff we see that the JSON source is stored as well, on the root (top-level `@id`) of the JSON object.



For the following part of the tutorial we want a little bit more data in our repository so please look at the helper file `jsonld_tutorial_helper.py` where you will see that at the end we have a dictionary named `obs` with about 9 diverse objects, mostly borrowed from the `json-ld.org` site: a person, an event, a place, a recipe, a group of persons, a product, and our hippieoil.

First let us store all the objects in a fresh repository. Then we check the size of the repo. Finally, we create a freetext index for the JSON sources.

```
>>> store(conn,[v for k,v in obs.items()], fresh=True)
>>> conn.size()
86
>>>
conn.createFreeTextIndex("source",['<http://franz.com/ns/alleg
```

```
rograph/6.4/load-meta#source>']])
```

```
>>>
```

## Retrieving values with SPARQL

To simply retrieve values in objects but not the objects themselves, regular SPARQL queries will suffice. But because we want to make sure that Python developers only need to deal with regular Python structures as lists and dictionaries, we created a simple wrapper around SPARQL (see helper file). The name of the wrapper is `runSparql`.

Here is an example. Let us find all the roots (top-level *@ids*) of objects and their types. Some objects do not have roots, so `None` stands for a blank node.

```
>>> pprint(runSparql(conn,"select ?s ?type { ?s a ?type }"))
[{'s': 'cocktail1', 'type': 'Cocktail'},
 {'s': None, 'type': 'Individual'},
 {'s': None, 'type': 'Vehicle'},
 {'s': 'tesla', 'type': 'Offering'},
 {'s': 'place1', 'type': 'Place'},
 {'s': None, 'type': 'Offer'},
 {'s': None, 'type': 'AggregateRating'},
 {'s': 'hippieoil', 'type': 'Product'},
 {'s': 'person-3', 'type': 'Person'},
 {'s': 'person-2', 'type': 'Person'},
 {'s': 'person-1', 'type': 'Person'},
 {'s': 'person-1000', 'type': 'Person'},
 {'s': 'event1', 'type': 'Event'}]
>>>
```

We do not see the full URIs for `?s` and `?type`. You can see them by adding an appropriate *format* argument to `runSparql`, but the default is terse.

```
>>> pprint(runSparql(conn,"select ?s ?type { ?s a ?type }
limit 2",format='ntriples'))
[{'s': '<http://franz.com/cocktail1>', 'type':
```

```
'<http://franz.com/Cocktail>'},
      {'s': None, 'type':
'<http://purl.org/goodrelations/v1#Individual>'}]}
>>>
```

## Retrieving a Dictionary or Object

`retrieve` is another function defined (in `jsonld_tutorial_helper.py`) for this tutorial. It is a wrapper around SPARQL to help extract objects. Here we see how we can use it. The sole purpose of `retrieve` is to retrieve the JSON-LD/dictionary based on a SPARQL pattern.

```
>>> retrieve(conn,"{?this a ical:Event}")
[{'@type': 'ical:Event', 'ical:location': 'New Orleans Arena,
New Orleans, Louisiana, USA', 'ical:summary': 'Lady Gaga
Concert', '@id': 'ical:event1', '@context': {'xsd':
'http://www.w3.org/2001/XMLSchema#', 'ical':
'http://www.w3.org/2002/12/cal/ical#', 'ical:dtstart':
{'@type': 'xsd:dateTime'}}, 'ical:dtstart':
'2011-04-09T20:00:00Z'}]}
>>>
```

Ok, for a final fun (if you like expensive cars) example: Let us find a thing that is “fast and furious”, that is worth more than \$80,000 and that we can pay for in cash:

```
>>>
addNamespace(conn,"gr","http://purl.org/goodrelations/v1#")
>>> x = retrieve(conn, """"{ ?this fti:match 'fast furious*';
      gr:acceptedPaymentMethods gr:Cash ;
      gr:hasPriceSpecification ?price .
      ?price gr:hasCurrencyValue ?value ;
      gr:hasCurrency "USD" .
      filter ( ?value > 80000.0 ) }""")
>>> pprint(x)
[{'@context': {'foaf': 'http://xmlns.com/foaf/0.1/',
'foaf:page': {'@type': '@id'},
'gr': 'http://purl.org/goodrelations/v1#',
```

```

        'gr:acceptedPaymentMethods': {'@type': '@id'},
        'gr:hasBusinessFunction': {'@type': '@id'},
        'gr:hasCurrencyValue': {'@type': 'xsd:float'},
        'pto': 'http://www.productontology.org/id/',
        'xsd': 'http://www.w3.org/2001/XMLSchema#'},
'id': 'http://example.org/cars/for-sale#tesla',
'type': 'gr:Offering',
'gr:acceptedPaymentMethods': 'gr:Cash',
'gr:description': 'Need to sell fast and furiously',
'gr:hasBusinessFunction': 'gr:Sell',
'gr:hasPriceSpecification': {'gr:hasCurrency': 'USD',
                              'gr:hasCurrencyValue':
'85000'},
'gr:includes': {'@type': ['gr:Individual', 'pto:Vehicle'],
                 'foaf:page':
'http://www.teslamotors.com/roadster',
                 'gr:name': 'Tesla Roadster'},
'gr:name': 'Used Tesla Roadster']]
>>> x[0]['@id']
'http://example.org/cars/for-sale#tesla'

```

---

# New!!! AllegroGraph v6.5 – Multi-model Semantic Graph and Document Database

Download – [AllegroGraph v6.5](#) and [Gruff v7.3](#)

AllegroGraph – [Documentation](#)

Gruff – [Documentation](#)

## Adding JSON/JSON-LD Documents to a Graph Database

Traditional document databases (e.g. MongoDB) have excelled at storing documents at scale, but are not designed for linking



data to other documents in the same database or in different databases. AllegroGraph 6.5 delivers the unique power to define many different types of documents that can all point to each other using standards-based semantic linking and then run SPARQL queries, conduct graph searches, execute complex joins and even apply Prolog AI rules directly on a diverse sea of objects.

AllegroGraph 6.5 provides free text indexes of JSON documents for retrieval of information about entities, similar to document databases. But unlike document databases, which only link data objects within documents in a single database, AllegroGraph 6.5 moves the needle forward in data analytics by semantically linking data objects across multiple JSON document stores, RDF databases and CSV files. Users can run a single SPARQL query that results in a combination of structured data and unstructured information inside documents and CSV files. AllegroGraph 6.5 also enables retrieval of entire documents.

There are many reasons for working with JSON-LD. The big search engines force ecommerce companies to mark up their webpages with a systematic description of their products and more and more companies use it as an easy serialization format to share data.

A direct benefit for companies using AllegroGraph is that they now can combine their documents with graphs, graph search and graph algorithms. Normally when you store documents in a document database you set up your documents in such a way that it is optimized for certain direct retrieval queries. Performing complex joins for multiple types of documents or even performing a shortest path through a mass of object (types) is too complicated. Storing JSON-LD objects in AllegroGraph gives users all the benefits of a document database AND the ability to semantically link objects together, run complex joins, and perform graph search queries.

Another key benefit for companies is that your application developers don't have to learn the entire semantic technology stack, especially the part where developers have to create individual RDF triples or edges. Application developers love to work with JSON data as serialization for objects. In JavaScript the JSON format is syntactically identical to the code for creating JavaScript objects and in Python the most import data structure is the 'dictionary' which is also near identical to JSON.

### **Key AllegroGraph v6.5 Features:**

- Support for loading JSON-LD and also some non-RDF data files, that is files which are not already organized into triples or quads. See [Loading non-RDF data](#) section in the [Data Loading](#) document for more information on loading non-RDF data files. Loading JSON-LD files is described along with other RDF formats in the [Data Loading](#) document. The section [Supported RDF formats](#) lists all supported RDF formats.
- Support for two phase commits (2PC), which allows AllegroGraph to participate in distributed transactions compromising a number of AllegroGraph and non-AllegroGraph databases (e.g. MongoDB, Solr, etc), and to ensure that the work of a transaction must either be committed on all participants or be rolled back on all participants. Two-phase commit is described in the [Two-phase commit](#) document.
- An event scheduler: Users can schedule events in the future. The event specifies a script to run. It can run once or repeatedly on a regular schedule. See the [Event Scheduler](#) document for more information.

- AllegroGraph is 100 percent ACID, supporting Transactions: Commit, Rollback, and Checkpointing. Full and Fast Recoverability. Multi-Master Replication
- Triple Attributes – Quads/Triples can now have attributes which can provide fine access control.
- Data Science – Anaconda, R Studio
- 3D and multi-dimensional geospatial functionality
- SPARQL v1.1 Support for Geospatial, Temporal, Social Networking Analytics, Hetero Federations
- Cloudera, Solr, and MongoDB integration
- JavaScript stored procedures
- RDF4J Friendly, Java Connection Pooling
- Graphical Query Builder for SPARQL and Prolog – Gruff
- SHACL (Beta) and SPIN Support (SPARQL Inferencing Notation)
- AGWebView – Visual Graph Search, Query Interface, and DB Management
- Transactional Duplicate triple/quad deletion and suppression
- Advanced Auditing Support
- Dynamic RDFS++ Reasoning and OWL2 RL Materializer
- AGLoad with Parallel loader optimized for both traditional spinning media and SSDs.

Numerous other optimizations, features, and enhancements.

Read the release notes – <https://franz.com/agraph/support/documentation/current/release-notes.html>