MORE SPARQL AND JSON-LD

CSABA VERES

TESTING VALUES (1/2)

- Add age info:

<http://somewhere/RebeccaSmith/>

info:age "23"^^xsd:integer ;

vCard:FN "Becky Smith" ;

vCard:N [vCard:Family "Smith" ;

vCard:Given "Rebecca"] .

<http://somewhere/RebeccaSmith/>

info:age "25"^^xsd:integer ;

TESTING VALUES (2/2)

- PREFIX info: <http://somewhere/peopleInfo#>

  SELECT ?resource

  WHERE

  {

  ?resource info:age ?age .

  FILTER (?age >= 24)

  }

| resource |

==========

| <http://somewhere/JohnSmith/> |

Category	Functions / Operators	Examples
Logical & Comparisons	`!, &&, ||, =, !=, <, <=, >, >=, IN, NOT IN`	`?hasPermit || ?age < 25`
Conditionals *(SPARQL 1.1)*	`EXISTS, NOT EXISTS, IF, COALESCE`	`NOT EXISTS { ?p foaf:mbox ?email }`
Math	`+, -, *, /, abs, round, ceil, floor, RAND`	`?decimal * 10 > ?minPercent`
Strings *(SPARQL 1.1)*	`STRLEN, SUBSTR, UCASE, LCASE, STRSTARTS, CONCAT, STRENDS, CONTAINS, STRBEFORE, STRAFTER`	`STRLEN(?description) < 255`
Date/time *(SPARQL 1.1)*	`now, year, month, day, hours, minutes, seconds, timezone, tz`	`month(now()) < 4`
SPARQL tests	`isURI, isBlank, isLiteral, isNumeric, bound`	`isURI(?person) || !bound(?person)`
Constructors *(SPARQL 1.1)*	`URI, BNODE, STRDT, STRLANG, UUID, STRUUID`	`STRLANG(?text, "en") = "hello"@en`
Accessors	`str, lang, datatype`	`lang(?title) = "en"`
Hashing *(1.1)*	`MD5, SHA1, SHA256, SHA512`	`BIND(SHA256(?email) AS ?hash)`
Miscellaneous	`sameTerm, langMatches, regex, REPLACE`	`regex(?ssn, "\\d{3}-\\d{2}-\\d{4}")`

OPTIONALS (1/2)

- PREFIX info: <http://somewhere/peopleInfo#>

  PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

  SELECT ?name ?age

  WHERE

  {

  ?person vcard:FN ?name .

  OPTIONAL { ?person info:age ?age }

  }

  | name | age |

  ==========

  | "Becky Smith" | 23 |

  | "Sarah Jones" | |

  | "John Smith" | 25 |

  | "Matt Jones" | |

OPTIONALS (2/2)

- PREFIX info: <http://somewhere/peopleInfo#>

  PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

  SELECT ?name ?age

  WHERE

  {

    ?person vcard:FN ?name .

    ?person info:age ?age .

}

| name | age |

==========

| "Becky Smith" | 23 |

| "John Smith" | 25 |

UNION (1/3)

- Suppose you had the following data:

  @prefix foaf: <http://xmlns.com/foaf/0.1/> .

  @prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

  _:a foaf:name "Matt Jones" .

  _:b foaf:name "Sarah Jones" .

  _:c vcard:FN "Becky Smith" .

  _:d vcard:FN "John Smith" .

UNION (2/3)

- PREFIX foaf: <http://xmlns.com/foaf/0.1/>

  PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

  SELECT ?name

  WHERE

  {

    { [] foaf:name ?name } UNION { [] vCard:FN ?name }

  }

  | name |

  ===

  | "Matt Jones" |

  | "Sarah Jones" |

  | "Becky Smith" |

  | "John Smith" |

UNION (3/3)

- Remembering where the data comes from

  PREFIX foaf: <http://xmlns.com/foaf/0.1/>

  PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

  SELECT ?name1 ?name2

  WHERE

  {

    { [] foaf:name ?name1 } UNION { [] vCard:FN ?name2 }

  }

  | name1 | name2 |

  ==

  | "Matt Jones" | |

  | "Sarah Jones" | |

  | | "Becky Smith" |

  | | "John Smith" |

UNION AND OPTIONAL

- Can often give same results … but not always

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name1 ?name2
WHERE
{
  OPTIONAL { ?x  foaf:name  ?name1 }
  OPTIONAL { ?x  vCard:FN   ?name2 }
}
```

```
------------------------------
| name1                      | name2        |
==================================
| "Matt Jones"               |              |
| "Sarah Jones"              |              |
|                            | "Becky Smith" |
|                            | "John Smith"  |
```

Combining SPARQL Graph Patterns

*Consider **A** and **B** as graph patterns.*

A Basic Graph Pattern – one or more triple patterns

```
A . B
```

⇒ Conjunction. Join together the results of solving A and B by matching the values of any variables in common.

Optional Graph Patterns

```
A OPTIONAL { B }
```

⇒ Left join. Join together the results of solving A and B by matching the values of any variables in common, if possible. Keep all solutions from A whether or not there's a matching solution in B

Combining SPARQL Graph Patterns

*Consider **A** and **B** as graph patterns.*

Either-or Graph Patterns

```
{ A } UNION { B }
```

⟹ Disjunction. Include both the results of solving A and the results of solving B.

"Subtracted" Graph Patterns (SPARQL 1.1)

```
A MINUS { B }
```

⟹ Negation. Solve A. Solve B. Include only those results from solving A that are *not compatible* with any of the results from B.

Property Paths *(SPARQL 1.1)*

- Property paths allow triple patterns to match arbitrary-length paths through a graph
- Predicates are combined with regular-expression-like operators.
- Examples on https://jena.apache.org/documentation/query/property_paths.html

Construct	Meaning
`path1/path2`	*Forwards path (`path1` followed by `path2`)*
`^path1`	*Backwards path (object to subject)*
`path1|path2`	*Either `path1` or `path2`*
`path1*`	*`path1`, repeated zero or more times*
`path1+`	*`path1`, repeated one or more times*
`path1?`	*`path1`, optionally*
`!uri`	*Any predicate except `uri`*
`!^uri`	*Any backwards (object to subject) predicate except `uri`*

PROPERTY PATH EXAMPLES

Find the name of any people that Alice knows.

```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows/foaf:name ?name .
}
```

Find the names of people 2 " `foaf:knows` " links away.

```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows/foaf:knows/foaf:name ?name .
}
```

This is the same as the strict SPARQL query:

```
{
  ?x  foaf:mbox <mailto:alice@example> .
  ?x  foaf:knows [ foaf:knows [ foaf:name ?name ]].
}
```

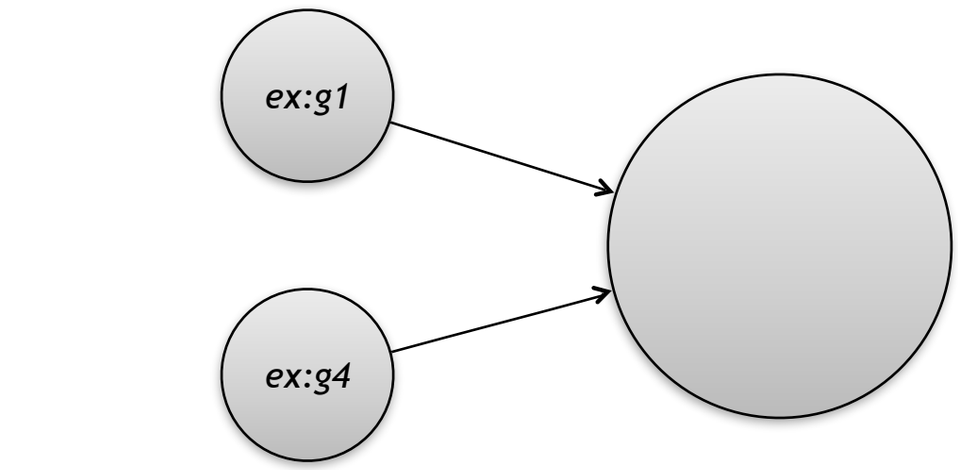
or, with explicit variables:

```
{
  ?x  foaf:mbox <mailto:alice@example> .
  ?x  foaf:knows ?a1 .
  ?a1 foaf:knows ?a2 .
  ?a2 foaf:name ?name .
}
```

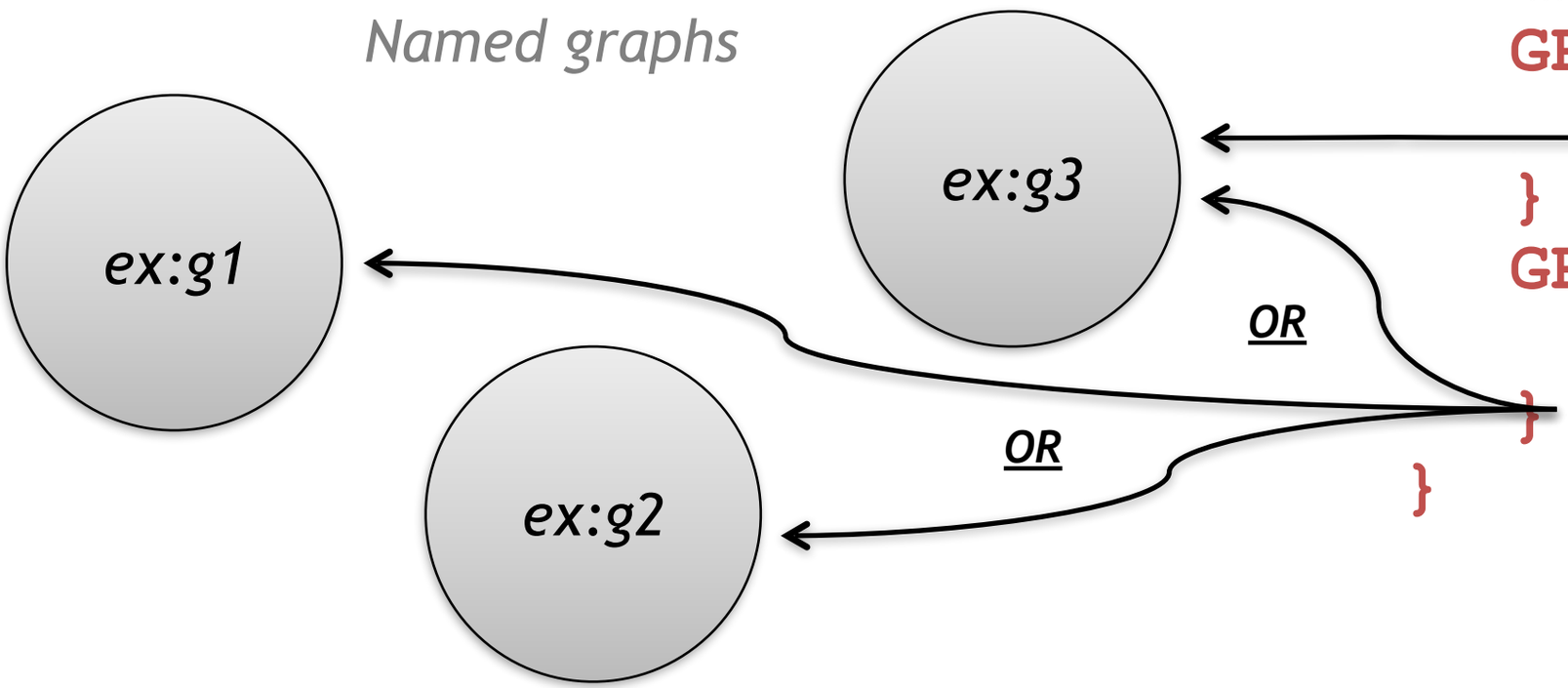
RDF Datasets

A SPARQL queries a *default graph* (normally) and zero or more *named graphs* (when inside a **GRAPH** clause).

Default graph
(the merge of zero or more graphs)



Named graphs




```
PREFIX ex: <…>
SELECT …
FROM ex:g1
FROM ex:g4
FROM NAMED ex:g1
FROM NAMED ex:g2
FROM NAMED ex:g3
WHERE {
   … A …
   GRAPH ex:g3 {
      … B …
   }
   GRAPH ?graph {
      … C …
   }
}
```

SPARQL Over HTTP (the SPARQL Protocol)

`http://host.domain.com/sparql/endpoint?`<*parameters*>

where <*parameters*> can include:

`query=`<*encoded query string*>

e.g. `SELECT+*%0DWHERE+{…`

`default-graph-uri=`<*encoded graph URI*>

e.g. `http%3A%2F%2Fexmaple.com%2Ffoo…`

n.b. zero of more occurrences of `default-graph-uri`

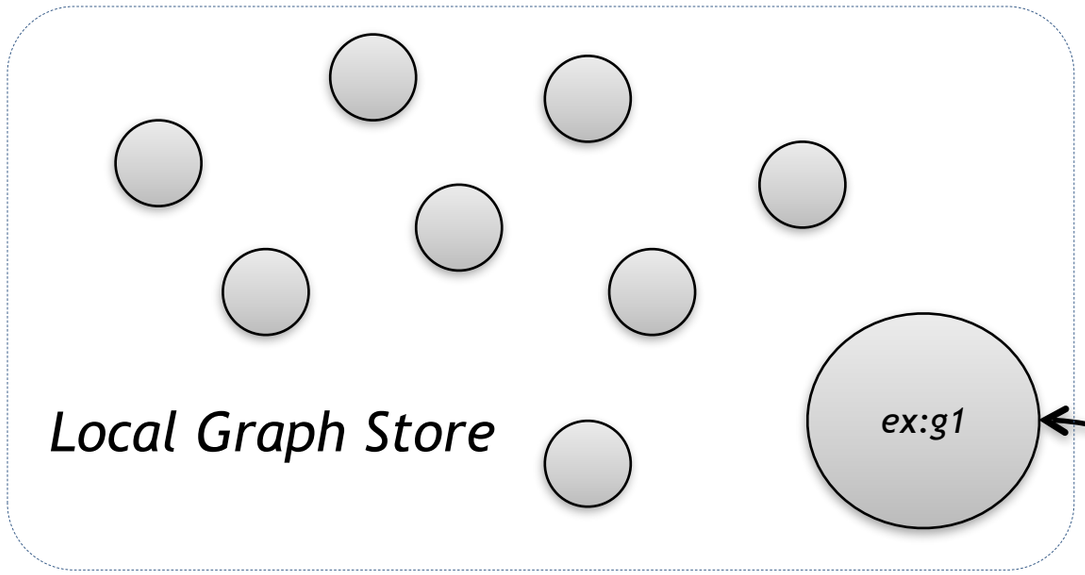
`named-graph-uri=`<*encoded graph URI*>

e.g. `http%3A%2F%2Fexmaple.com%2Fbar…`

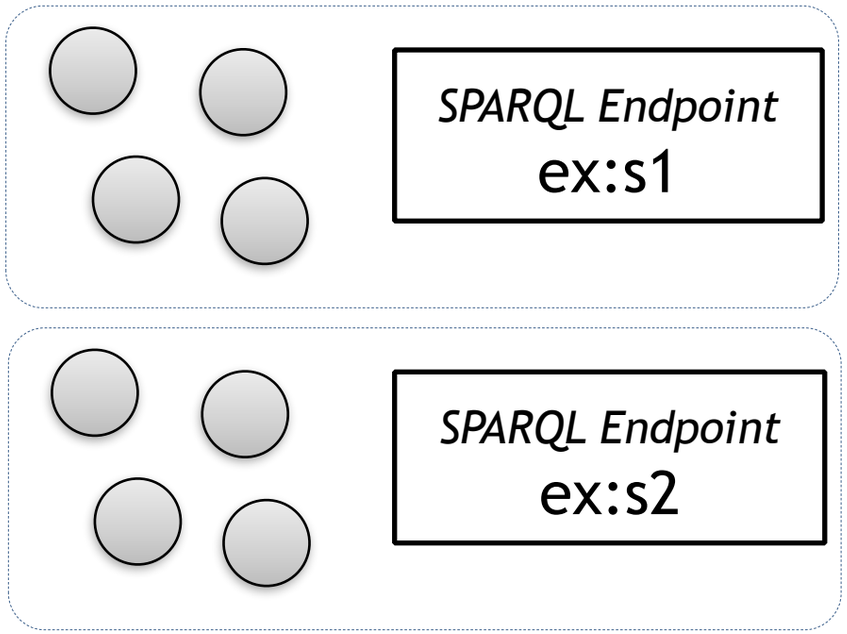
n.b. zero of more occurrences of `named-graph-uri`

HTTP `GET` or `POST`. Graphs given in the protocol override graphs given in the query.

Federated Query *(SPARQL 1.1)*




```
PREFIX ex: <…>
SELECT …
FROM ex:g1
WHERE {
    … A …
    SERVICE ex:s1 {
        … B …
    }
    SERVICE ex:s2 {
        … C …
    }
}
```



Some Public SPARQL Endpoints

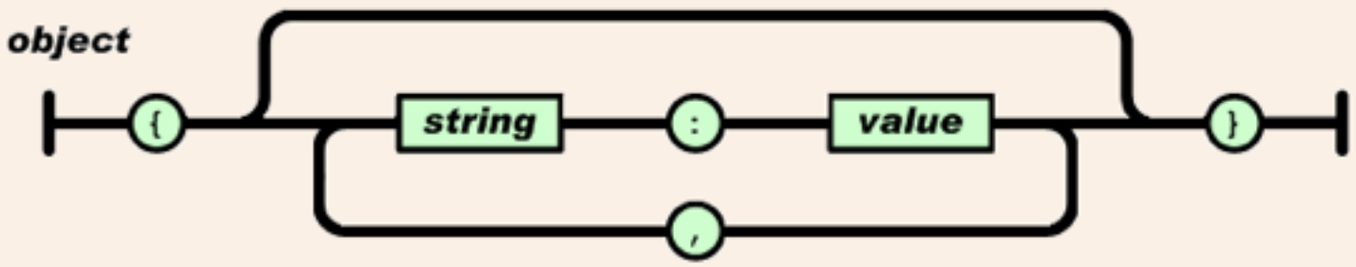
Name	URL	What's there?
SPARQLer	*http://sparql.org/sparql.html*	*General-purpose query endpoint for Web-accessible data*
DBPedia	*http://dbpedia.org/sparql*	*Extensive RDF data from Wikipedia*
DBLP	*http://www4.wiwiss.fu-berlin.de/dblp/snorql/*	*Bibliographic data from computer science journals and conferences*
LinkedMDB	*http://data.linkedmdb.org/sparql*	*Films, actors, directors, writers, producers, etc.*
World Factbook	*http://www4.wiwiss.fu-berlin.de/factbook/snorql/*	*Country statistics from the CIA World Factbook*
bio2rdf	*http://bio2rdf.org/sparql*	*Bioinformatics data from around 40 public databases*

JSON

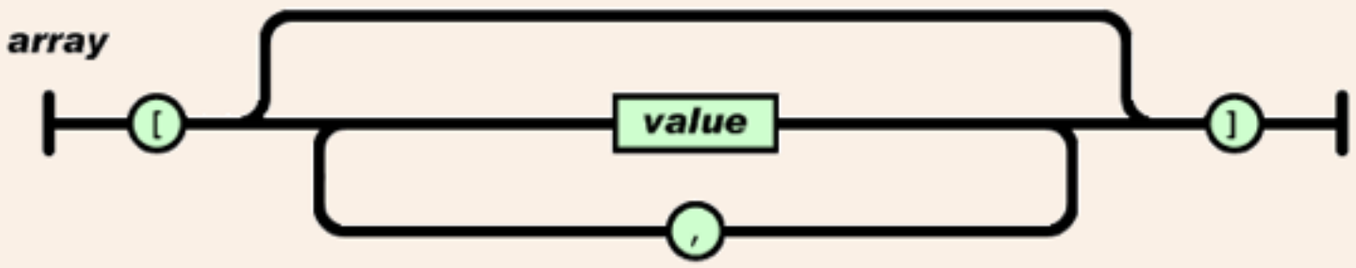
- JavaScript Object Notation
- A lightweight data-interchange format
- JSON is a text format that is completely language independent

JSON STRUCTURES

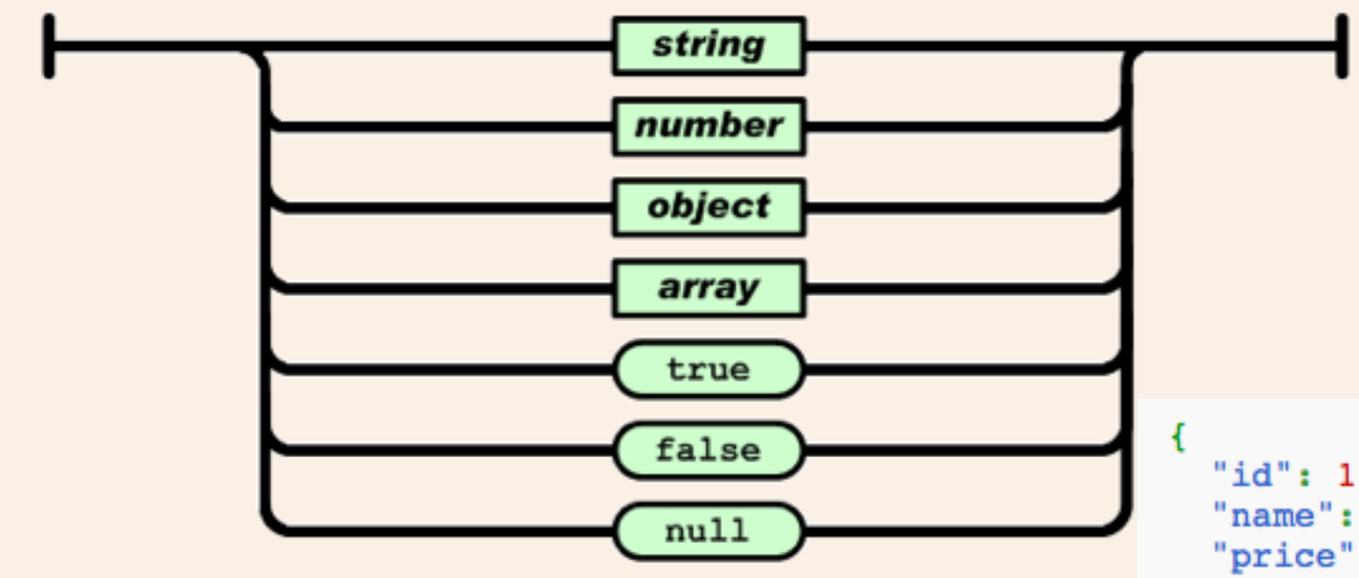
- A collection of name/value pairs
  - object, record, struct, dictionary, hash table, keyed list, or associative array
- An ordered list of values
  - array, vector, list, or sequence
- These are universal data structures. Virtually all modern programming languages support them in one form or another.



An *array* is an ordered collection of values. An array begins with `[` (left bracket) and ends with `]` (right bracket). Values are separated by `,` (comma).



A *value* can be a *string* in double quotes, or a *number*, or `true` or `false` or `null`, or an *object* or an *array*. These structures can be nested.




```
{
  "id": 1,
  "name": "Foo",
  "price": 123,
  "tags": [
    "Bar",
    "Eek"
  ],
  "stock": {
    "warehouse": 300,
    "retail": 20
  }
}
```

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

object

list

JSON-LD

- JSON-LD is a lightweight Linked Data format.
- It is easy for humans to read and write.
- A lightweight syntax to serialize Linked Data in JSON

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

KEY FEATURES

- a universal identifier mechanism for JSON objects via the use of IRIs,
- a way to disambiguate keys shared among different JSON documents by mapping them to IRIs via a context,
- a mechanism in which a value in a JSON object may refer to a JSON object on a different site on the Web,
- the ability to annotate strings with their language,
- a way to associate datatypes with values such as dates and times,
- and a facility to express one or more directed graphs, such as a social network, in a single document.

RDF COMPATIBILITY

- JSON-LD is designed to be usable directly as JSON, with no knowledge of RDF
- Also designed to be usable as RDF, if desired, for use with other Linked Data technologies
- It can be used as another RDF syntax, like Turtle
- Developers only need to know JSON and two keywords (@context and @id) to use the basic functionality in JSON-LD

PROBLEMS WITH SIMPLE JSON

- Difficult to integrate JSON from different sources as the data may contain keys that conflict with other data sources
- No built-in support for hyperlinks

```
{
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

```
{
  "http://schema.org/name": "Manu Sporny",
  "http://schema.org/url": { "@id": "http://manu.sporny.org/" },  ← The '@id' keyword means 'This value is an identifier that is an IRI'
  "http://schema.org/image": { "@id": "http://manu.sporny.org/images/manu.png" }
}
```

CONTEXT

- The previous example is a bit verbose
- "Context" allows the developer to use simple terms

EXAMPLE 3: Context for the sample document in the previous section

```
{
  "@context":
  {
    "name": "http://schema.org/name",  ← This means that 'name' is shorthand for 'http://schema.org/name'
    "image": {
      "@id": "http://schema.org/image",  ← This means that 'image' is shorthand for 'http://schema.org/image'
      "@type": "@id"  ← This means that a string value associated with 'image' should be interpreted as an identifier that is an IRI
    },
    "homepage": {
      "@id": "http://schema.org/url",  ← This means that 'homepage' is shorthand for 'http://schema.org/url'
      "@type": "@id"  ← This means that a string value associated with 'homepage' should be interpreted as an identifier that is an IRI
    }
  }
}
```

EXAMPLE 4: Referencing a JSON-LD context

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

MORE JSON-LD
HTTP://WWW.W3.ORG/TR/JSON-LD/

EXAMPLE 5: In-line context definition

```
{
  "@context":
  {
    "name": "http://schema.org/name",
    "image": {
      "@id": "http://schema.org/image",
      "@type": "@id"
    },
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  },
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

EXAMPLE 6: Values of @id are interpreted as IRI

```
{
...
  "homepage": { "@id": "http://example.com/" }
...
}
```

EXAMPLE 8: IRI as a key

```
{
...
  "http://schema.org/name": "Manu Sporny",
...
}
```

EXAMPLE 11: Identifying a node

```
{
  "@context":
  {
    ...
    "name": "http://schema.org/name"
  },
  "@id": "http://me.markus-lanthaler.com/",
  "name": "Markus Lanthaler",
  ...
}
```

EXAMPLE 9: Term expansion from context definition

```
{
  "@context":
  {
    "name": "http://schema.org/name"
  },
  "name": "Manu Sporny",
  "status": "trollin'"
}
```

JSON-LD TYPES

THE TYPE OF A PARTICULAR NODE CAN BE SPECIFIED USING THE @TYPE KEYWORD. IN LINKED DATA, TYPES ARE UNIQUELY IDENTIFIED WITH AN IRI.

EXAMPLE 12: Specifying the type for a node

```
{
...
  "@id": "http://example.org/places#BrewEats",
  "@type": "http://schema.org/Restaurant",
...
}
```

A node can be assigned more than one type by using an array:

EXAMPLE 13: Specifying multiple types for a node

```
{
...
  "@id": "http://example.org/places#BrewEats",
  "@type": [ "http://schema.org/Restaurant", "http://schema.org/Brewery" ],
...
}
```

The value of an `@type` key may also be a term defined in the active context:

EXAMPLE 14: Using a term to specify the type

```
{
  "@context": {
    ...
    "Restaurant": "http://schema.org/Restaurant",
    "Brewery": "http://schema.org/Brewery"
  }
  "@id": "http://example.org/places#BrewEats",
  "@type": [ "Restaurant", "Brewery" ],
  ...
}
```