Crash Course RDF+SPARQL
RDF

- RDF is describing metadata per triples
- "simplest possible database"
- Abstract away from (relational, or tree-like) structure

**Triples:** Subject Predicate Object

- axel isA Person .
- axel knows gb .
- axel knows andreas .
- gb isSupervisorOf gennaro .
Resources in RDF

- Resources are identified by URIs (to encourage web-wide unique identifiers)

“axel isA Person”

```
<http://polleres.net/foaf.rdf#me>  
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> 
<http://xmlns.com/foaf/0.1/Person> .
```

Ugly to read… allow shortcuts:

```ruby
@prefix : http://polleres.net/foaf.rdf#
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
@prefix foaf: http://xmlns.com/foaf/0.1/

:me rdf:type foaf:Person .
```
Apart from URIs Literal values allowed for objects:

:me foaf:name "Axel Polleres" .
:me ex:age "33"^^xsd:integer .
RDF allows making statements about unknown resources:

- “axel knows someone called ‘David’ “

```html
:me foaf:knows _:x .
_:x foaf:name “David” .
```

_:x a bit like an existential variable…
_:x is a so-called “blank node” … why?
Sets of Triples are often viewed as a Graph:

```prolog
:me a Person.
:me foaf:name "Axel Polleres".
:me ex:age "33"^^xsd:integer.
:me foaf:knows _:x.
_:x foaf:name "David".
:me foaf:knows <http://www.gibbi.com/me>.
<http://www.gibbi.com/me> foaf:name "GB".
```

--> draw the graph
Syntaxes

- RDF/XML  ... barely readable for humans but good for exchange.
- Turtle  ... what we used so far, plus a few shortcuts, “Terse Rdf Language”
2 Example RDF graphs:

```
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

<ex.org/bob> foaf:maker _:a.
_:a a foaf:Person ; foaf:name "Bob";
    foaf:knows _:b.

_:b a foaf:Person ; foaf:nick "Alice".
<alice.org/> foaf:maker _:b
```

```
# Graph: alice.org
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

alice:me a foaf:Person ; foaf:name "Alice" ;
    foaf:knows _:c.

_:c a foaf:Person ; foaf:name "Bob" ;
    foaf:nick "Bobby".
```

Turtle shortcuts:

- ‘;’ groups predicate value pairs with common subject.
- ‘,’ groups object for the same predicate
- [ ] blank nodes can also be abbreviated with brackets.
SPARQL

• Simple Protocol and RDF Query Language
  – Basic Graph Patterns (Conjunctive queries)
  – UNIONS
  – GRAPH Patterns
  – OPTIONAL Patterns
  – FILTERs
SPARQL Queries

• 3 basic forms
  – SELECT
  – ASK
  – CONSTRUCT

• We start with SELECT:

```
SELECT Variables
FROM Dataset
WHERE Pattern
```
Basic Graph Patterns
(Conjunctive queries)

“select persons and their names”

SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }

<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:a</td>
<td>“Bob”</td>
</tr>
<tr>
<td>_:c</td>
<td>“Bob”</td>
</tr>
<tr>
<td>alice:me</td>
<td>“Alice”</td>
</tr>
</tbody>
</table>
UNIONs

“select Persons and their names or nicknames”

```
SELECT ?X ?Y
FROM ...
WHERE { { ?X foaf:name ?Y . }
  UNION { ?X foaf:nick ?Y .} }
```

<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:a</td>
<td>“Bob”</td>
</tr>
<tr>
<td>_:c</td>
<td>“Bob”</td>
</tr>
<tr>
<td>alice:me</td>
<td>“Alice”</td>
</tr>
<tr>
<td>_:b</td>
<td>“Alice”</td>
</tr>
<tr>
<td>_:c</td>
<td>“Bobby”</td>
</tr>
</tbody>
</table>
GRAPH patterns

"select creators of graphs and the persons they know"

SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
    GRAPH ?G { ?X foaf:knows ?Y. } }

<table>
<thead>
<tr>
<th>?X</th>
<th>?Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:a</td>
<td>_:b</td>
</tr>
</tbody>
</table>
**OPTIONAL**

- Leaves unmatchable variables unbound:

  "select all persons and optionally their names"

```sql
SELECT *
WHERE
{
  ?X a foaf:Person .
  OPTIONAL {?X foaf:name ?N }
}
```

<table>
<thead>
<tr>
<th>?X</th>
<th>?N</th>
</tr>
</thead>
<tbody>
<tr>
<td>_:a</td>
<td>“Bob”</td>
</tr>
<tr>
<td>_:b</td>
<td></td>
</tr>
<tr>
<td>_:c</td>
<td>“Bob”</td>
</tr>
<tr>
<td>alice:me</td>
<td>“Alice”</td>
</tr>
</tbody>
</table>
FILTERs

• By means of FILTERs, one can filter out undesired solutions, e.g.

“select persons older than 30”

```
SELECT ?X
WHERE { ?X a foaf:person .
  ?X ex:age ?Y .
  FILTER (?Y > 30)
}
```

– FILTERs can be complex boolean combinations ( &&, ||, !)
– Special FILTER functions allowed, e.g. “BOUND(Var)”
• FILTERs can be used to emulate set difference (or negation as failure):

“select all persons without an email address”

```
SELECT ?Name ?Email
WHERE
{
    ?X a ?Person
    OPTIONAL {?X :email ?Email }
    FILTER ( !bound( ?Email ) )
}
```

• FILTERs can NOT bind new variables!

```
SELECT ?X ?Y
    FILTER (?Y = ?Z + 1 ) }
```

will not produce results, since “unbound = 33+1” gives an error.
CONSTRUCT

• allows to create new triples …

CONSTRUCT{ :me foaf:knows ?X }
FROM <http://www.deri.ie/about/team>
WHERE { ?X a foaf:Person. }

• Tricky: blank nodes in CONSTRUCT

CONSTRUCT { :me foaf:knows _:x .
            _:x foaf:name ?X}
FROM <http://www.deri.ie/about/team>
WHERE { _:y foaf:name ?X . }
That’s all!

• Very simple, many useful extensions still missing, e.g.
  – calculating new bindings
  – aggregates