

# A Vision on Algebraic Flows for Declarative Resource Descriptions

Jitse De Smet, Ruben Verborgh, Ruben Taelman

Ghent University – imec – IDLab, Belgium

Research Foundation - Flanders

# Overview

**Veterinarian use case**

**Related Work**

**Vision**

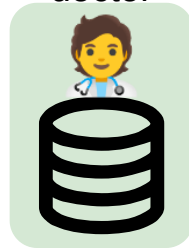
**Conclusion**

# Veterinarian use case

pet owner



regular  
doctor



other  
doctor



# Overview

Veterinarian use case

**Related Work**

Vision

**Conclusion**

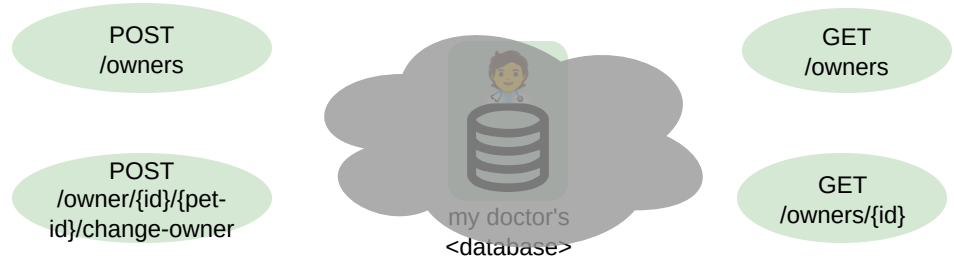
# RPC/ Syntactic descriptions fail to describe relationships

WADL

OpenAPI/ Swagger

AsyncAPI

Can be used as documentation and to generate skeleton of client side code.



# MCP introduces ambiguity

Standard for connecting AI applications to external systems

Builds on JSON-RPC spec

Example interface description:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "tools": [
      {
        "name": "get_owner_and_pets",
        "title": "Owner Information",
        "description": "Retrieves all information about an owner and their pets registered with 'heathy pet co.'",
        "inputSchema": {
          "type": "object",
          "properties": {
            "id": {
              "type": "string",
              "description": "The systems uuid-v4 used to represent the owner as registered in the system"
            }
          },
          "required": ["id"]
        }
      }
    ]
  }
} \(docs\)
```

create\_owner\_with\_pet

change\_owner\_of\_pet



get\_all\_owners

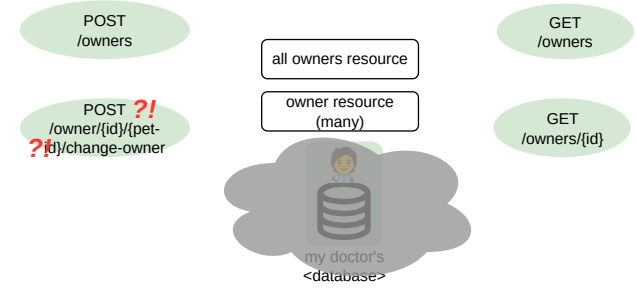
get\_owner\_and\_pet

# Semantic Web Descriptions cannot model underlying resources

Hydra, OWL-S, RESTdesc

Model RESTful interactions well  
(GET, PUT, PATCH, DELETE, POST)

Do not model underlying resources



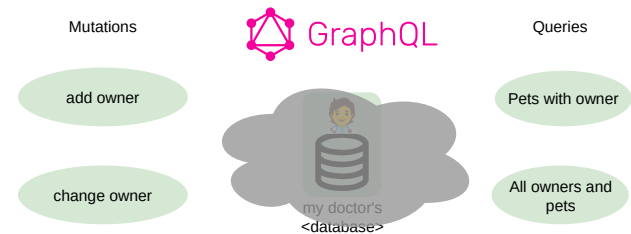
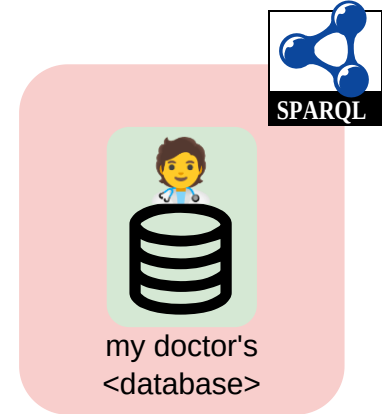
# Query-Based interfaces can be cost heavy, inherently symmetrical, or resemble RPC

SPARQL endpoint, GraphQL

SPARQL endpoint only models 1 resource, the data

GraphQL functions describes like RPC

Both tie into an 'expensive' execution environment



# Overview

Veterinarian use case

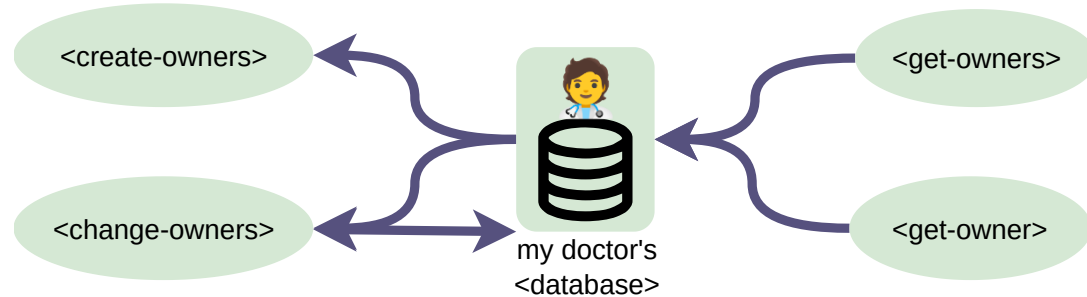
Related Work

**Vision**

**Conclusion**

# Goals

1. Execution environment independent
2. Models intermediate resources (asymmetric interfaces)
3. Deterministic description of relationship between resources
4. Standing on the shoulders of giants

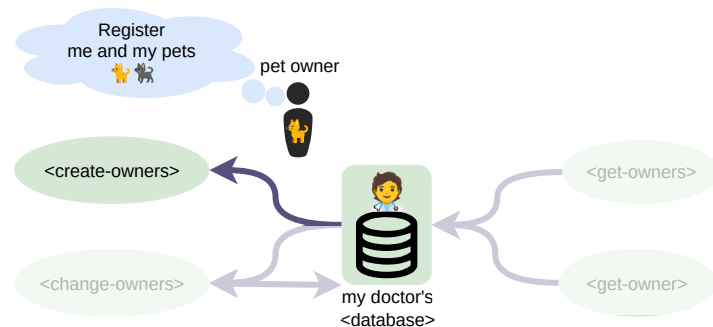


# Algebraic Descriptions over RDF data

Get Interface Description, discover <create-owner> with:

1. **Interaction method:** POST to /owners
2. **Resource representation:** Turtle
3. **Expected data shape:** SPARQL ASK
4. **Consequence of success:** SPARQL update on <database>

```
INSERT {  
  GRAPH <database> {  
    ?ownerBound a ex:owner ;  
      ex:name ?name ; ex:pet ?petBound .  
    ?petBound a ex:pet ;  
      ex:name ?petName ; ex:age ?age .  
  }  
} WHERE {  
  ?resource ex:name ?name ;  
  BIND( UUID() AS ?ownerUuid ) .  
  ?resource ex:pet ?pet .  
  BIND( UUID() AS ?petUuid ) .  
  ?pet ex:name ?petName ;  
    ex:age ?age ;  
  BIND( URI(CONCAT(ex:owners, '/', ?ownerUuid)) AS ?ownerBound ) .  
  BIND( URI(CONCAT(?ownerBound, '/', ?petUuid)) AS ?petBound ) .  
}
```

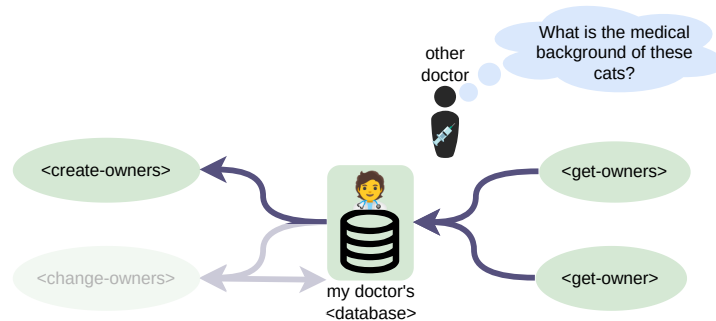


# Derivation over an underlying resource

Get Interface Description, discover <get - owners> and <get - owner> with:

1. Interaction method: GET to /owners and GET to /owner/{id}
2. Resource representation: Turtle
3. No expected data, but consequence of success a SPARQL CONSTRUCT

```
CONSTRUCT { [] ex:item ?owner . } WHERE {  
  GRAPH <database> {  
    ?owner a ex:owner .  
  }  
}  
  
CONSTRUCT {  
  ?id a ex:owner ;  
  ?ownerP ?owner0 ;  
  ?owner0 ?petP ?pet0 ;  
} WHERE {  
  GRAPH <database> {  
    ?id a ex:owner ;  
    ?ownerP ?owner0 ;  
    OPTIONAL { ?owner0 ?petP ?pet0 . }  
  }  
}
```

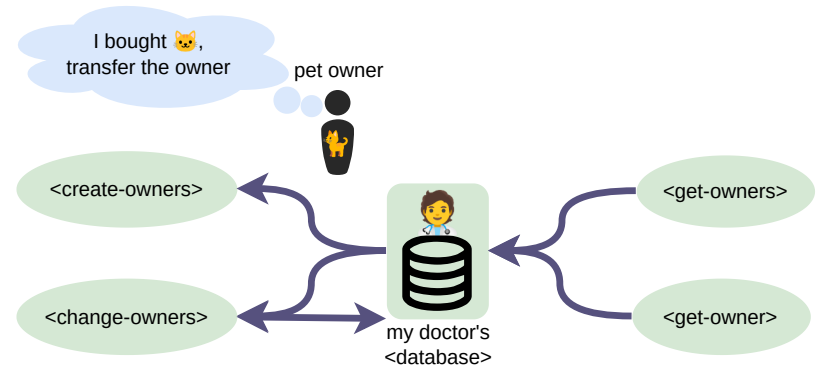


# Beyond REST: consistency boundary

Get Interface Description, discover <change - owner> using interaction method:

POST to /change-owners/{id}/{pet-id}

```
DELETE {
  GRAPH <database> {
    ?origOwner ex:pet ?orig-pet .
    ?origPet a ex:pet ;
    ex:age ?age ;
    ex:name ?name .
  }
} INSERT {
  GRAPH <database> {
    ?newOwner ex:pet ?movedIri .
    ?movedIri a ex:pet ;
    ex:age ?age
    ex:name ?name
  }
} WHERE {
  # From received data:
  ?o ex:new-owner ?newOwner .
  BIND( URI( CONCAT(ex:owners, '/', ?id) ) AS ?origOwner ) .
  BIND( URI( CONCAT(?origOwner, '/', ?petId) ) AS ?origPet ) .
  BIND( URI( CONCAT(?newOwner, '/', ?petId) ) AS ?movedIri ) .
  # Go look in the current state of the database for whether the owner and pet exist.
  GRAPH <database> {
    ?origOwner ex:pet ?origPet .
    ?origPet a ex:pet ;
    ex:age ?age ;
    ex:name ?name .
    ?newOwner a ex:owner ;
  }
}
```

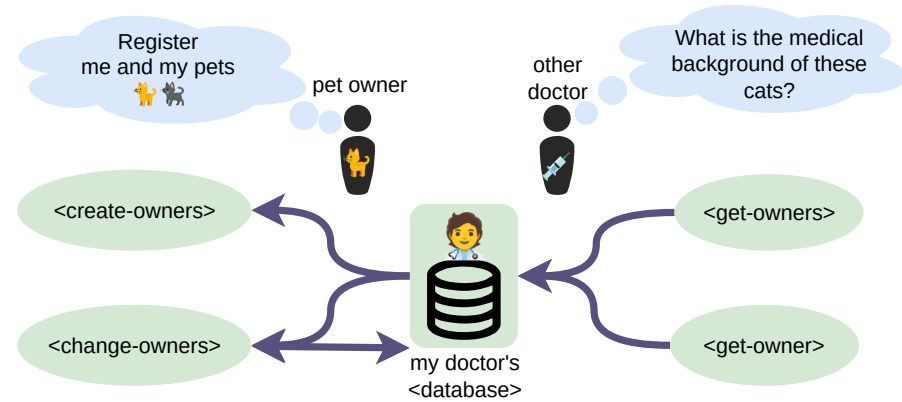


# Conclusion

1. Endpoint discovery
2. Infer downstream effect of modifications
3. Using existing query writing skills

## Future work

1. Formalization
2. Fine-grained, policy aware semantics
3. Proof of Concept



Additional

# Related Issues

By explicitly describing the algebraic mappings, you enter the database realm:  $D \xrightarrow{m} T$

1.  $Q(D)$ , written as  $Q(T)$
2. View selection (what to materialize)
3. Schema transformation/ migration (when original resource is not accessible)
4. Query rewriting using materialized views  
(targeting some resource, can you rewrite using views)
5. The View Update Problem  
(when you target non-materialized resources (might be easier because of blank nodes))
6. Incremental view maintenance (can be interesting for large views)
7. Access policy/ access control/ declarative access policy in relational databases?
8. New stuff too: what endpoints to call and how, given an update