Winter School on Knowledge Technologies for Complex Business Environments

Linked Data and APIs
Session 2: Linked APIs

Steffen Stadtmüller, AIFB, KSRI, Karlsruhe Institute of Technology, DE
Ljubljana, SLO, December 1st, 2011
Attribution

- Some Slides from
  - Andreas Harth, Barry Norton, Sebastian Speiser, Cedric Mesnage; *ESWC 2011 Linked Services Tutorial [Linked Data Services – LIDS]*; Crete, GR; May 30, 2011
  - Barry Norton, Denny Vrandecic; *IEEE SSSC 2011 Linked Data and Services*; Berkeley, US; August 8, 2011

- Thank you!
Agenda

- Motivation
- Preliminaries
  - Hyper Text Transfer Protocol
  - Representational State Transfer
- Linked APIs
  - Design Guidelines
  - API Descriptions
  - API Interaction
  - Linked APIs Semantics
  - Wrapping Techniques
  - Real World Example
- Questions
Why do we care?

MOTIVATION
Motivation: Linked Open Data
Motivation: Dynamic Data

- The Web today is not only about serving static data
- Data is often dynamically created as a result of some calculation carried out over input data

- Not all data sources will be published as fully materialized data sets:
  - Data is changing constantly (e.g., sensor data or stock quotes)
  - Data is calculated based on infinitely many inputs (e.g. route between two geographical locations)
  - Provider does not want arbitrary access (e.g., flight ticket prices, social networks)
Motivation: Triggering Functionalities

- The Web today offers functionalities that go beyond data and information:
  - Service endpoints, forms and APIs are used to trigger functionalities in the web and the real world as well (e.g., ordering a pizza or solving a re-captcha)
  - Programmableweb.com lists ~4300 APIs¹

¹http://programmableweb.com
Motivation: Triggering Functionalities

- The Web today offers functionalities that go beyond data and information:
  - Service endpoints, forms and APIs are used to trigger functionalities in the web and the real world as well (e.g., ordering a pizza or solving a re-captcha)
  - Programmableweb.com lists ~4300 APIs¹

- But: they use heterogenous formats, mostly only textually described

  Developers have to gain a deep understanding of every API and write individually tailored code to consume services in applications

¹http://programmableweb.com
Motivation

- Increased value comes from combinations of functionalities and APIs
  - 6300 Mashups on programmableweb.com
  - But a lot of manual effort is required for this compositions (glue code)
  - Structured API descriptions ease the composition process considerably
  - Semantic descriptions allow to execute several tasks automatically (e.g., data matching, discovery, repair)
Preliminaries

HTTP AND REST
HTTP Overview

- HTTP, by which all documents on the WWW are served, is a client server protocol
  - Every interaction is based on:
  - Request
    - Method
      - GET (retrieve entity identified by URI)
      - PUT (store entity in body under the given URI)
      - POST (submit the information in the body as a new subordinate of the resource URI)
      - DELETE (delete entity identified by URI)
    - URI
    - Header
    - [optional] Body (with POST, PUT)
HTTP Overview

- HTTP, by which all documents on the WWW are served, is a client server protocol
  - Every interaction is based on:
    - Request
      - Method
        - GET
        - PUT
        - POST
        - DELETE
      - URI
      - Header
      - [optional] Body (with POST, PUT)
    - Response
      - Response Code (Integer)
      - Header
      - [optional] Body
HTTP conneg and Linked Data URI Lookup

- A foundational issue in Linked Data was the distinction of URIs for real-world objects versus documents (e.g., RDF) that might describe them.
- This can be handled in the HTTP Header together with content negotiation:

  HTTP GET
  http://dbpedia.org/resource/India
  Accept: text/N3

  303 (see other)
  http://dbpedia.org/data/india.n3
HTTP conneg and Linked Data URI Lookup

- A foundational issue in Linked Data was the distinction of URIs for real-world objects versus documents (e.g., RDF) that might describe them.
- This can be handled in the HTTP Header together with content negotiation:

  HTTP GET
  http://dbpedia.org/resource/India
  Accept: text/html

  303 (see other)
  http://dbpedia.org/page/india
REST

- HTTP, maintained by IETF not W3C, is just one (primary) implementation of an architectural style called:
  - REST = REpresentational State Transfer

- REST Principles:
  - Application state and functionality is divided into resources
  - Every resource is uniquely addressable
  - All resources share a uniform interface:
    - A constrained set of well-defined operations
    - A constrained set of content types
LAPIS
DESIGN GUIDELINES
Linked APIs (LAPIS)

... aim to promote a scalable and efficient style of APIs, by bringing together:

- RESTful Web architecture
  - resource-oriented
  - manipulated with HTTP verbs
    - GET, PUT, POST, DELETE
  - Negotiate representations

- Linked data
  - Uniform use of URIs
  - Use of RDF and SPARQL
Linked APIs Guidelines

- Linked APIs come with a set of guiding principles:
  1. Describe APIs as LOD prosumers with input and output descriptions as SPARQL graph patterns
  2. Communicate RDF by RESTful content negotiation
  3. The output should make explicit its relation with the input

- Associated with the last principle is an optional fourth:
  4. When wrapping non-Linked APIs, make the lifting/mapping open as SPARQL CONSTRUCT queries and make implicit knowledge explicit.
Describe API input and output as SPARQL graph patterns
**Linked API Description**

- LAPIS are (URI identified) Web functionalities, that
  - Consume RDF data (input)
  - Carry out their function according to the input (e.g., retrieving dynamic data, trigger a real world effect…)
  - Produce RDF data (output)
- To describe the RDF data a LAPI expects as input and produces as output, we use the graph pattern formalism provided by the SPARQL query language.
Linked API resource URI

- As an example consider a API that serves the friends of a person in a social network:
  - This API is identified with the URI:
    
    http://lapis.org/sn/getFriends
Linked API I/O Description

- As an example consider a API that serves the friends of a person in a social network:
  - This API is identified with the URI:
    http://lapis.org/sn/getFriends

- Input and output description:

  **In:**
  - ?x a foaf:Person.
  - ?x sn:id ?id.

  **Out:**
  - ?y foaf:age ?age
Linked API Description

- Graph pattern can be included in RDF as (typed) literals.
  - This lapis is identified with the URI:
    - http://lapis.org/sn/getFriends
  - Input and output description:
    - In:
      - ?x a foaf:Person.
      - ?x sn:id ?id.
    - Out:
    - Embedded in RDF description
      - sn:getFriends rdf:type msm:Service;
        sawsdl:modelReference
        [a msm:Precondition;
         rdf:value "{"?x a foaf:Person ... }"],
        [a msm:Postcondition;
         rdf:value "{"?x foaf:knows ?y ... }"].

Pattern Match
RDF in
Linked
API
RDF out

Description (RDF+SPARQL)
Pattern
Pattern
Linked API Description

- RDF can be published, with the API resource URI referring to it
  - This API is identified with the URI:
    - HTTP GET
      - Accept: text/N3
      - 303* (see other)
  - Accessible under e.g.
    - HTTP GET
      - Accept: text/N3
      - 200 (ok)

*optional

```
http://lapis.org/sn/getFriends
```

```
sn:getFriends rdf:type msm:Service;
sawsdl:modelReference
  [a msm:Precondition;
    rdf:value "{?x a foaf:Person … }" ],
  [a msm:Postcondition;
    rdf:value "{?x foaf:knows ?y … }" ].
```
Communicate RDF by RESTful content negotiation

LAPIS
API INTERACTION

25
Linked API interaction

- API Execution via HTTP POST:
  - POST RDF data that matches the input pattern to the API resource
  - The API responds according to output pattern

```
?x a foaf:Person.
?x sn:id ?id.
```

```
ex:Steffen a foaf:Person.
ex:Steffen sn:id "abcde".
```

```
ex:Steffen foaf:knows ex:Günter.
ex:Günter foaf:age "29".
<http://...> foaf:depicts ex:Günter.
```

```
?y foaf:age ?age
?pic foaf:depicts ?y
```
Linked API Interaction

- API is executed for every “pattern hit” in the input graph:

```plaintext
http://lapis.org/sn/getFriends

HTTP POST

ex:Steffen a foaf:Person.
ex:Steffen sn:id "abcde".
ex:Steffen sn:id "12345".
ex:Marko a foaf:Person.
ex:Marko sn:id "xyz".

In:  ?x a foaf:Person.
     ?x sn:id ?id.
```

match
Linked API Interaction

- API is executed for every “pattern hit” in the input graph:

```
http://lapis.org/sn/getFriends
```

**HTTP POST**

```
ex:Steffen a foaf:Person.
ex:Steffen sn:id "abcde".
ex:Steffen sn:id "12345".
ex:Marko a foaf:Person.
ex:Marko sn:id "xyz".
```

**Response**

```
ex:Steffen foaf:knows ex:Günter.
ex:Günter foaf:age "29".
<http://...> foaf:depicts ex:Günter.
```

```
ex:Steffen foaf:knows ex:Roland.
ex:Roland foaf:age "27".
<http://...> foaf:depicts ex:Roland.
```

```
ex:Marko foaf:knows ex:Steffen.
```
Linked API Interaction

- The patterns are constraining, but not defining the input and output:

  http://lapis.org/sn/getFriends

  **HTTP POST**
  
  ex:Steffen a foaf:Person.

  **In:**
  
  ?x a foaf:Person. 
  ?x sn:id ?id.

  **no hit**
Linked API Interaction

- The patterns are constraining, but not defining the input and output:

```
http://lapis.org/sn/getFriends
```

**HTTP POST**

```
ex:Steffen a foaf:Person.
ex:Steffen foaf:age "27".
ex:Steffen sn:id "12345".
```

**In:**

```
?x a foaf:Person.
?x sn:id ?id.
```
Linked API Interaction

- The pattern are constraining, but not defining the input and output
- but they should be as precise as possible:

http://lapis.org/sn/getFriends

Out:
?

x foaf:knows ?y.
?y foaf:age ?age
?pic foaf:depicts ?y
OPTIONAL
(?y foaf:based_near ?loc)

Response

ex:Steffen foaf:knows ex:Günter.
ex:Günter foaf:age “29”.
<http://...> foaf:depicts ex:Günter.

ex:Steffen foaf:knows ex:Roland.
ex:Roland foaf:age “27”.
<http://...> foaf:depicts ex:Roland.
ex:Roland foaf:based_near “Karlsruhe”.
Interlinking Data Sets with dynamic LAPIS data

Use Cases:

- Processing of static data set, using new interlinked set for further applications
- Linked Data endpoint, enriching data before returning to client (server-side)
- Linked Data browser, enriching data after retrieving it from server (client-side)
Example: enriching CRM information

Customers

| #34 John Doe | Address |
| #35 Steffen Stadtmüller | Address |
| #36 Jane Doe | Address |
| #37 .... |

known relations:

| Roland Stühmer | #12 Günter Ladwig |

CRM Application

static data

dynamic data
LAPIS URIs to identify an API call

- To link to a dynamic output dataset we need a GETable URI to identify it.

```
http://lapis.org/sn/getFriends
```

- We extend the API URI with key/value pairs, where every key represents a variable in the input graph pattern

```
?x a foaf:Person.
?x sn:id ?id.
```
LAPIS URIs to identify an API call

- To link to a dynamic output dataset we need a GETable URI to identify it.

HTTP GET

http://lapis.org/sn/getFriends

When the URI is called (via HTTP GET), the values are bound to the corresponding variables in the pattern, and valid input is created.

In:

```plaintext
?x a foaf:Person.
?x sn:id ?id.
```

http://lapis.org/sn/getFriends?x=“ex:Steffen”&id=abcde
LAPIS URIs to identify an API call

- To link to a dynamic output dataset we need a GETable URI to identify it.

HTTP GET

http://lapis.org/sn/getFriends

In:

?x a foaf:Person.
?x sn:id "abcde".

extend with k/v

http://lapis.org/sn/getFriends?x="ex:Steffen"&id=abcde

ex:Steffen a foaf:Person.
ex:Steffen sn:id "abcde".
LAPIS URIs to identify an API call

- To link to a dynamic output dataset we need a GETable URI to identify it.

- Observation: For data-delivering APIs we can often identify a “central entity”, that is handled and described.

- In such cases we can use the API call URI (+hashtag) itself to identify this entity

http://lapis.org/sn/getFriends

ex:Steffen a foaf:Person.
ex:Steffen sn:id “abcde”.
ex:Steffen foaf:knows ex:Günter.
ex:Günter foaf:age “29”.
<http://...> foaf:depicts ex:Günter.
LAPIS URIs to identify an API call

- To link to a dynamic output dataset we need a GETable URI to identify it.

HTTP GET

http://lapis.org/sn/getFriends?id=abcde

In:

?x a foaf:Person.
?x sn:id ?id.
LAPIS URIs to identify an API call

- To link to a dynamic output dataset we need a GETable URI to identify it.

HTTP GET

```
http://lapis.org/sn/getFriends?id=abcde
```

Response

```
<http://lapis.org/sn/getFriends?id=abcde#person>
  foaf:knows ex:Günter.

ex:Günter foaf:age "29".
<http://...> foaf:depicts ex:Günter.
```
Example: enriching CRM information

Customers

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>John Doe</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Steffen Stadtmüller</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>Jane Doe</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>….</td>
<td></td>
</tr>
</tbody>
</table>

known relations:

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Günter Ladwig</td>
</tr>
</tbody>
</table>

static data

ex:Steffen a foaf:Person.
ex:Steffen sn:id "abcde".
ex:Steffen owl:sameAs

dynamic data

HTTP GET
LAPIS URIs to identify an API call

- Include this API call URI in the LAPIS output, when appropriate and another entity is used
  - owl:sameAs, rdfs:seeAlso

HTTP POST

http://lapis.org/sn/getFriends

Response

ex:Steffen foaf:knows ex:Günter.
ex:Günter foaf:age “29”.
ex:Steffen owl:sameAs

Out:

The output should make explicit its relation with the input.
Looking at semantics

- Approaches to Semantic Web Services (e.g., SAWSDL\(^1\)) focused on a class based annotations of service descriptions

- But what does the service do?
  - How is the age in the output related to person from the input?
  - Does the output talk about the same person than the input?

\(^1\)http://www.w3.org/2002/ws/sawsdl/
Looking at semantics

- To really understand the functionality we need to understand the relation between input and output:
  - Explain the semantics of the calculation carried out over the input.
  - With what information is the input going to be enriched (data delivering API)?
  - What functionality was triggered by invoking the API?
  - How does the input influence the output (functionality or data)?

- Linked APIs achieve this by implying the relation over the variables in the graph patterns:

  In: 
  ```
  ?x a foaf:Person.
  ?x sn:id ?id.
  ```

  Out: 
  ```
  ?y foaf:age ?age
  ?pic foaf:depicts ?y
  ```

  - **Safe Variables**
  - **Unsafe Variables**
Looking at semantics

- To really understand the functionality we need to understand the relation between input and output:
  - Explain the semantics of the calculation carried out over the input.
  - With what information is the input going to be enriched (data delivering API)?
  - What functionality was triggered by invoking the API?
  - How does the input influence the output (functionality or data)?

- Linked APIs achieve this by implying the relation over the variables in the graph patterns:

  **In:**
  
  \[
  \text{?x a foaf:Person.} \\
  \text{?x sn:id ?id.}
  \]

  **Out:**
  
  \[
  \text{?x foaf:knows ?y.} \\
  \text{?y foaf:age ?age} \\
  \text{?pic foaf:depicts ?y}
  \]

  "The API will provide age and picture of someone X knows"
Looking at semantics

- To really understand the functionality we need to understand the relation between input and output:
  - Explain the semantics of the calculation carried out over the input.
  - With what information is the input going to be enriched (data delivering API)?
  - What functionality was triggered by invoking the API?
  - How does the input influence the output (functionality or data)?

- Linked APIs achieve this by implying the relation over the variables in the graph patterns:

  In:  
  ```
  ?x a foaf:Person.  
  ?x sn:id ?id.
  ```

  Out:  
  ```
  ?y foaf:knows ?x.  
  ?y foaf:age ?age  
  ?pic foaf:depicts ?y
  ```

  „The API will provide age and picture of someone who knows X“
Looking at semantics

- To really understand the functionality we need to understand the relation between input and output:
  - Explain the semantics of the calculation carried out over the input.
  - With what information is the input going to be enriched (data delivering API)?
  - What functionality was triggered by invoking the API?
  - How does the input influence the output (functionality or data)?

- Linked APIs achieve this by implying the relation over the variables in the graph patterns:

  In: ?x a foaf:Person.
  ?x sn:id ?id.

  ?x foaf:age ?age
  ?pic foaf:depicts ?x

  „The API will provide age and picture of X and someone he knows“
Leveraging the cloud to understand the API

- The graph patterns are not to be seen isolated, but in the context of the LOD cloud
- An agent (human and machine alike) can gain an understanding of the API descriptions by resolving URIs in the pattern.
- Additional knowledge (i.e., schema information) can be retrieved
Make the **lifting/mapping open as SPARQL CONSTRUCT** queries

LAPIS
WRAPPING TECHNIQUES
Wrapping legacy APIs

- It is not reasonable to abandon all the Web APIs that already exist.
  - Introduction of semantics for Web APIs is based on a “network effect”:
  - The more LAPIS are publicly available, the more value can a provider gain from following the same standards

- When a developer wants to combine LAPIS with legacy services and APIs
  - Instead of writing glue code he can wrap the non-LAPI and use it for other combinations
  - If the API is free, he can offer its functionality as a third-party to other developers
Wrapping legacy APIs

- LAPIS specifically address the problem of leveraging existing functionalities and exposing them as LAPIS by recommending to
  - publish how the functionality is semantically interpreted,
  - using LOD technologies

HTTP GET / POST  http://lapis.org/sn/getFriends  Response

lowering  lifting  facebook
Calling a LAPIS wrapper

- A LAPIS that wraps an existing Web API acts as a semantic intermediary offering the original functions:
  - The RDF submitted to the LAPIS is used to trigger the legacy API
  - The extraction of parameters to call the API happens with a SPARQL SELECT
  - The process of constructing non-semantic data for API calls is referred to as "Lowering"

- We leverage, that all APIs and REST services follow a fixed structure:
  - Example:

```plaintext
http://legacy.org/API/getFriends/Karlsruhe/abcde/
```
Calling a LAPIS wrapper

- A LAPIS that wraps an existing Web API acts as a semantic intermediary offering the original functions:
  - The RDF submitted to the LAPIS is used to trigger the legacy API
  - The extraction of parameters to call the API happens with a SPARQL SELECT

```
ex:Steffen a foaf:Person.
ex:Steffen sn:id "abcde".
ex:Steffen foaf:based_near "Karlsruhe".
```

```
SELECT ?loc ?user
WHERE {
  ?so sn:id ?user
  ?so foaf:based_near ?loc }
```

HTTP POST

**API Endpoint URI**

+ Karlsruhe, abcde

```
http://legacy.org/API/getFriends/Karlsruhe/abcde/
```
Building the response

- Web APIs deliver structured (non semantic) data
  - Most use JSON\(^1\) or proprietary XML
  - Both are serializations of a tree structure

```
{
  "Person": {
    "firstName": "Günter",
    "lastName": "Ladwig",
    "Age": 29,
    "lives": "Karlsruhe"
  }
  "Person": {
    "firstName": "Roland",
    "lastName": "Stühmer",
    "Age": 27,
    "lives": "Karlsruhe"
  }
}
```

```
<xml>
  <Person>
    <Person>
      <firstName>Günter</firstName>
      <lastName>Ladwig</lastName>
      <Age>29</Age>
      <lives>Karlsruhe</lives>
    </Person>
  </Person>
  <Person>
    <firstName>Roland</firstName>
    <lastName>Stühmer</lastName>
    <Age>27</Age>
    <lives>Karlsruhe</lives>
  </Person>
</xml>
```

\(^1\)Javascript Object Notation: http://json.org
Building the response

- Web APIs deliver structured (non semantic) data
  - Most use JSON\(^1\) or proprietary XML
  - Both are serializations of a tree structure

A tree is also a graph!
Building the response

- First we change the syntax
  - keys are properties
  - values are objects (literals)

```json
{
    "Person": {
        "firstName": "Günter",
        "lastName": "Ladwig",
        "Age": 29,
        "lives": "Karlsruhe"
    }
    "Person": {
        "firstName": "Roland",
        "lastName": "Stühmer",
        "Age": 27,
        "lives": "Karlsruhe"
    }
}
```

```
@prefix temp: <http://lapis.org/dummy#>.
_:root temp:Person _:a.
_:a temp:firstName "Günter".
_:a temp:lastName "Ladwig".
_:a temp:Age "29".
_:a temp:lives "Karlsruhe".

_:root temp:Person _:b.
_:b temp:firstName "Roland".
_:b temp:lastName "Stühmer".
_:b temp:Age "27".
_:b temp:lives "Karlsruhe".
```
Building the response

- The so created RDF still misses any kind of semantic meaning
- To transform it to real Linked Data a SPARQL CONSTRUCT query is used
- The process of constructing semantic data out of a non semantic API or service response is referred to as „Lifting“
- To make the implicit knowledge (i.e., the relation between input and output) of the API/service explicit, we combine it with the LAPIS input
Building the response

- CONSTRUCTing real Linked Data out of the API reply tree:

```
_:root temp:Person _:a.
_:a temp:firstName „Günter“.
_:a temp:lastName „Ladwig“.
_:a temp:Age „29“.
_:a temp:lives „Karlsruhe“.

_:root temp:Person _:b.
_:b temp:firstName „Roland“.
_:b temp:lastName „Stühmer“.
_:b temp:Age „27“.
_:b temp:lives „Karlsruhe“.

ex:Steffen a foaf:Person.
ex:Steffen sn:id “abcde”.
ex:Steffen foaf:based_near „Karlsruhe“.
```

---

```
CONSTRUCT {
  ?x foaf:knows ?q.
  ?q foaf:givenName ?gn.
  ?q foaf:familyName ?fn.
  ?q foaf:based_near ?loc.
} WHERE {
  ?x a foaf:Person.
  ?x sn:id ?id.
  ?y temp:lastName ?fn.
  BIND(URI(CONCAT("http://...", ?gn)) AS ?q)
}
```
Building the response

- CONSTRUCTing real Linked Data out of the API reply tree:

CONSTRUCT {
  ?x foaf:knows ?q.
  ?q foaf:givenName ?gn.
  ?q foaf:familyName ?fn.
  ?q foaf:based_near ?loc.
}
WHERE {
  ?x a foaf:Person.
  ?x sn:id ?id.
  ?y temp:lastName ?fn.
  BIND(URI(CONCAT("http://example.org/people#", ?gn)) AS ?q)
  BIND(URI("http://example.org/people#Roland") AS ?q)
  BIND(<http://example.org/people#Roland> AS ?q)
}

- All JSON values are literals, but sometimes we need to build a URI

- SPARQL 1.1 offers functions to transform strings, convert them into a URI and bind it to a URI:
Building the response

- CONSTRUCTing real Linked Data out of the API reply tree:

```turtle
CONSTRUCT {
  ?x foaf:knows ?q.
  ?q foaf:givenName ?gn.
  ?q foaf:familyName ?fn.
  ?q foaf:based_near ?loc.
} WHERE {
  ?x a foaf:Person.
  ?x sn:id ?id.
  ?y temp:lastName ?fn.
  BIND(URI(CONCAT("http://...", ?gn)) AS ?q)
}
```

ex:Steffen foaf:knows ex:Günter.
ex:Steffen foaf:knows ex:Roland.

ex:Günter foaf:givenName „Günter“.
ex:Günter foaf:familyName „Ladwig“.
ex:Günter foaf:age „29“.
ex:Günter foaf:based_near „Karlsruhe“.

ex:Roland foaf:givenName „Roland“.
ex:Roland foaf:familyName „Stühmer“.
ex:Roland foaf:age „27“.
ex:Roland foaf:based_near „Karlsruhe“.
Publish Lowering and Lifting Information

- The lowering SELECT and the lifting CONSTRUCT query represent the intuition behind the semantic interpretation of the wrapped API/service.
- Therefore they should be published and a link included in the API description.

What is the semantic behind the parameters that have to be used to call the service/API?

What is the semantic behind the data the service/API replies to a request?
Publish Lowering and Lifting Information

- The lowering SELECT and the lifting CONSTRUCT query represent the intuition behind the semantic interpretation of the wrapped API/service.
- Therefore they should be published and a link included in the API description.

```xml
sn:getFriends a msm:Service;
sn:getFriends sawsdl:modelReference _:pre.
_:pre a msm:Precondition.
_:pre rdf:value "{?x a foaf:Person ... }".
_:post a msm:Postcondition;
_:post rdf:value "{?x foaf:knows ?y ... }".
sn:getFriends sawsdl:loweringSchemaMapping
sn:getFriends sawsdl:liftingSchemaMapping
```
Publish Lowering and Lifting Information

- The lowering SELECT and the lifting CONSTRUCT query represent the intuition behind the semantic interpretation of the wrapped API/service.
- Therefore they should be published and a link included in the API description.

```sparql
CONSTRUCT {  
  ?x foaf:knows ?q.  
  ?q foaf:givenName ?gn.  
  ?q foaf:familyName ?fn.  
  ?q foaf:based_near ?loc.  
} WHERE {  
  ?x a foaf:Person...  
}

SELECT ?loc ?user  
WHERE {  
  ?so sn:id ?user  
  ?so foaf:based_near ?loc }  
```

HTTP GET

sn:getFriends sawsdl:loweringSchemaMapping


sn:getFriends sawsdl:liftingSchemaMapping

Real World Examples

- **Linked Open Movies**
  - [http://linkedopenmovies.appspot.com](http://linkedopenmovies.appspot.com)
  - Search for movies via keyword string; provides rich information on the identified movies
  - Wraps API functionalities from themoviedb.org

- **vocab.cc**
  - [http://vocab.cc](http://vocab.cc)
  - RDF vocabulary search via keyword string; provides usage information of the identified classes and properties
  - Made from scratch

- **bing2RDF**
  - [http://bing2rdf.appspot.com](http://bing2rdf.appspot.com)
  - Picture search and meta information about the pictures
  - Wraps the Microsoft bing search API
Bing2RDF Web Form

POST new picture query (in RDF/XML)

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:so="http://purl.org/linkeddata/services/ontology/SearchOntology#">
  <so:Query>
    <so:queryString>bunny</so:queryString>
  </so:Query>
  <so:Query>
    <so:queryString>dcg</so:queryString>
  </so:Query>
</rdf:RDF>
```

You can POST queries to the service.
Remember conneg:
Web Browser always asks for HTML
Use of another client

```bash
> curl -X POST http://bing2rdf.appspot.com/Bing/getPictures
-T data.rdf -H "Accept: text/N3"
```
Related Work

- **OWL-S**: Ontology to describe Web Services
  - Message parts are mapped onto ontological concepts
  - Rules for decision points in processes
- **Web Service Modeling Ontology (WSMO)** [Roman et al.]
  - similar to OWL-S, but broader scoped
- **Minimal Service Model (MSM), iServe** [Pedrinaci et al.]
  - MSM a lightweight description with homogeneous view over WSDL and Web APIs
  - iServe: Centralized service repository
- **RESTdesc**: functional descriptions of REST APIs [Verborgh et al.]
  - Description in N3 (superset of RDF)
  - Also graph pattern based
- **hRESTs and SA-REST** [Kopecky]
  - Class based annotation of HTML pages as REST API description
Summary

- The Web today is not only about static data:
  - Data is created as a result of some calculation carried out over input data
  - Functionallities are offered that go beyond data and information
- To overcome problems arising out of heterogeneous formats Linked APIs as structured semantic descriptions offer a promising solution
  - Based on successful Linked Data Technologies
  - Leveraging lightweight RESTful architecture

- Concretely LAPIS come with a set of guiding principles:
  - Describe APIs as LOD prosumers with input and output descriptions as SPARQL graph patterns
  - Communicate RDF by RESTful content negotiation
  - The output should make explicit its relation with the input
  - When wrapping non-Linked APIs, make the lifting/mapping open as SPARQL CONSTRUCT queries and make implicit knowledge explicit.
Thank you
References


Appendix

DISCOVERY OF LAPIS
The Idea: Linked APIs in the Cloud

- API Descriptions are in the LOD cloud interlinked with other data
  - Descriptions of Linked APIs contain SPARQL Graph patterns to describe their functionality
- But how to match this functionality with user requirements and needs?

Input:
{?me a foaf:Person}

Output:
{?me foaf:knows ?x}
Agents looking for APIs can use API templates as formalizations of user requirements and constraints

- Syntactically equivalent to API descriptions (SPARQL graph patterns)
- What input can they provide for invocation? (input pattern)
- What do they expect as output? (output pattern)
Discovery as Graph Pattern Containment

Template Input $\supseteq$ Service Description Input

Template Output $\subseteq$ Service Description Output

- We define a number of metrics to provide for a continuum of matches and thus for ranking of APIs:

\[
\begin{align*}
\text{psr}_{\text{input}} &= \frac{\#(\{\text{predicates in template}\} \cap \{\text{predicates in service description}\})}{\#\{\text{predicates in service description}\}} \\
\text{psr}_{\text{output}} &= \frac{\#(\{\text{predicates in template}\} \cap \{\text{predicates in service description}\})}{\#\{\text{predicates in template}\}} \\
\text{rsr}_{\text{input}} &= \frac{\#(\{\text{resources in template}\} \cap \{\text{resources in service description}\})}{\#\{\text{predicates in service description}\}} \\
\text{rsr}_{\text{output}} &= \frac{\#(\{\text{resources in template}\} \cap \{\text{resources in service description}\})}{\#\{\text{resources in template}\}} 
\end{align*}
\]

- Direct Pattern Containment (dpc)
  - One pattern is ASKed against a skolemized version of another.
  - This results in a binary matching decision.
Discovery as Graph Pattern Containment

Template Input \supseteq Service Description Input
Template Output \subseteq Service Description Output

API Description:

Input:
{ ?me a foaf:Person ;
  sn:id ?uid . }

Output:
{ ?me foaf:knows
  [a foaf:Person;
   foaf:name ?name;
   foaf:age ?age;
   foaf:openid ?oid] . }

Template:

Input:
{?me a foaf:Person.}

Output:
{?me foaf:knows
  [a foaf:Person;
   foaf:name ?name;
   foaf:age ?age] . }

<table>
<thead>
<tr>
<th></th>
<th>psr</th>
<th>rsr</th>
<th>dpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0.5</td>
<td>1.0</td>
<td>NO</td>
</tr>
<tr>
<td>Output</td>
<td>1.0</td>
<td>1.0</td>
<td>YES</td>
</tr>
</tbody>
</table>
What about scalability?

- The metrics have to be calculated for every combination of subscribed templates and API descriptions in the cloud.

- Hadoop is the open source implementation of Google MapReduce.
  - Allows for parallel computation of the metrics on different nodes in a cluster (map),
  - combination of the metrics and ranking of the APIs (reduce)
  - on a distributed file system (HDFS).

1 https://hadoop.apache.org
To evaluate the scalability of the system for thousands of API descriptions and/or templates we generate random graph patterns.
Evaluation

- To evaluate the scalability of the system for thousands of API descriptions and/or templates we generate random graph patterns.

Settings:

- One template matched over 10 000 API descriptions,
- between 5 and 50 triple patterns per graph pattern,
- on one, two, five, eight and ten (work)nodes
- (additionally on every setup one machine as Namenode)
- with a blocksize of 1MB
- and a replication factor 3.
- two runs (to account for fluctuations in network traffic)

- 3 x 8.19MB of data on 3 x 9 blocks

9 map tasks are triggered
Evaluation

Nodes | sec | mean (sec) | standard deviation | standard error
--- | --- | --- | --- | ---
1 | 1. run | 394 | 
2. run | 395 | 394.5 | 1.0 | 0.7
2 | 1. run | 223 | 221 | 3.0 | 2.1
2. run | 219 | 
5 | 1. run | 120 | 122 | 2.4 | 1.7
2. run | 124 | 
8 | 1. run | 121 | 119 | 3.2 | 2.2
2. run | 117 | 
10 | 1. run | 81 | 81.5 | 0.5 | 0.4
2. run | 82 |
Evaluation

- 9 MapTasks can be executed
- Between 1 and 5 worknodes:
  - More MapTasks can be executed simultaneously by adding more nodes.
  - Almost every worknode has to execute at least two MapTasks.
- 5 and 8 worknodes:
  - Namenode loses the possibility for load balancing to some extend.
  - Load balancing takes into account that the worknodes contain non-disjunctive subsets of the overall input data (locality is preferred),
  - but Hadoop prefers parallel computation over load locality.
- 10 worknodes:
  - The system can (and must) decide to disregard the least useful worknode
Evaluation

- This effect can be avoided by allowing for more MapTasks (i.e., smaller block size or more input data).

- But the observation shows, that the improvement in terms of execution speed can not only be attributed to the increase of computation resources (i.e., adding additional CPUs and memory with every worknode), but also to the strategic distribution and execution of matching sub-tasks.