CHAPTER 9

Use Cases

By reading the book, you learned how to write HTML5 Microdata and JSON-LD annotations in the markup, develop Semantic Web applications, describe web services in standardized languages, run powerful queries on Linked Open Data (LOD) datasets, and develop Semantic Web applications. Now that you have become familiar with Semantic Web technologies, let's analyze four complex examples, to get ready for real-life implementations!

RDB to RDF Direct Mapping

Using the R2RML language for expressing customized mappings from relational database (RDB) to Resource Description Framework (RDF) datasets, you refer to logical tables to retrieve data from an input database. A logical table can be either a base table, a view, or an SQL query [1]. Assume you have a relational database about the staff members in your enterprise and would like to map it to RDF. Each staff member is identified by a unique identifier (`ID`), which is used as the primary key (see Table 9-1).

Table 9-1. *The Employee Database Table*

ID	FirstName	LastName
`INTEGER`	`VARCHAR(50)`	`VARCHAR(50)`
`10`	`John`	`Smith`
`11`	`Sarah`	`Williams`
`12`	`Peter`	`Jones`

The employee projects are described by the employee identifiers (`ID_Employee`) and the project identifiers (`ID_Project`), both of which are primary foreign keys (see Table 9-2).

Table 9-2. *The `Employee_Project` Database Table*

ID_Employee	ID_Project
`INTEGER`	`INTEGER`
`10`	`110`
`11`	`111`
`11`	`112`
`12`	`111`

The projects use an integer identifier (ID), which is a primary key featured by the description of a maximum of 50 characters (Table 9-3).

Table 9-3. *The Project Database Table*

ID	Description
INTEGER	VARCHAR(50)
110	WebDesign
111	CloudComputing
112	DomainRegistration

The *direct mapping* defines an RDF graph representation of the data from the relational database [2]. The direct mapping takes the relational database (data and schema) as input, and generates an RDF graph called the *direct graph*. During the mapping process, the staff members, the projects, and the relationship between them are expressed in RDF using the Turtle syntax (see Listing 9-1). The rule for translating each row of a logical table to zero or more RDF triples is specified as a *triples map*. All RDF triples generated from one row in the logical table share the same subject. A triples map is represented by a resource that references other resources. Each triples map has exactly one `rr:logicalTable` property with a value representing the logical table that specifies a Structured Query Language (SQL) query result to be mapped to RDF triples. A triples map also has exactly one *subject map* that specifies the technique to use for generating a subject for each row of the logical table, using the `rr:subjectMap` property,[1] whose value is the subject map. The triples map may have optional `rr:predicateObjectMap` properties, whose values are *predicate-object maps* that create the predicate-object pairs for each logical table row of the logical table. The predicate map-object map pairs specified by predicate-object maps can be used together with the subjects generated by the subject map for creating RDF triples for each row.

Listing 9-1. Direct Mapping from RDB to RDF

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.com> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@base <http://example.com/base/> .

<TriplesMap1>
  a rr:TriplesMap;

  rr:logicalTable [ rr:tableName "\"Employee\""; ] ;

  rr:subjectMap [ rr:template "http://example.com/employee/{\"\ID\"}"; ];

  rr:predicateObjectMap
  [
    rr:preficate        ex:firstName ;
        rr:objectMap    [ rr:column "\"FirstName\"" ]
  ];
```

[1]Alternatively, the `rr:subject` constant shortcut property can also be used.

```
  rr:predicateObjectMap
  [
    rr:predicate        ex:lastName ;
        rr:objectMap    [  rr:column "\"LastName\"" ]
  ]
  .

<TripleMap2>
  a rr:TriplesMap;

  rr:logicalTable [ rr:tableName "\"Project\""; ] ;

  rr:subjectMap [ rr:template "http://example.com/project/{\"\ID\"}"; ];

  rr:predicateObjectMap
  [
    rr:preficate        ex:id ;
        rr:objectMap    [ rr:column "\"ID\"" ]
  ];

  rr:predicateObjectMap
  [
    rr:predicate        ex:description ;
        rr:objectMap    [  rr:column "\"Description\"" ]
  ]
  .

<linkMap_1_2>
  a rr:TriplesMap;

  rr:logicalTable [ rr:tableName "\"Employee_Project\""; ] ;

  rr:subjectMap [ rr:template "http://example.com/employee/{\"\ID_Employee\"}"; ];

  rr:predicateObjectMap
  [
    rr:preficate        ex:involvedIn ;
        rr:objectMap    [ rr:template "http://example.com/project/{\"ID_Project\"}" ];
  ] .
```

The R2RML mapping is supported by software tools such as the db2triples software library [3], OpenLink Virtuoso [4], RDF-RDB2RDF [5], morph [6], and Ultrawrap [7]. In this example, the result is a set of RDF triples that describe the staff members and the projects they are involved in (Table 9-4).

Table 9-4. *The RDF Triples of the Output*

Subject	Predicate	Object
`<http://example.com/employee/10>`	`<http://example.com/lastName>`	`"Smith"`
`<http://example.com/employee/10>`	`<http://example.com/firstName>`	`"John"`
`<http://example.com/employee/12>`	`<http://example.com/lastName>`	`"Jones"`
`<http://example.com/employee/12>`	`<http://example.com/firstName>`	`"Peter"`
`<http://example.com/employee/11>`	`<http://example.com/lastName>`	`"Williams"`
`<http://example.com/employee/11>`	`<http://example.com/firstName>`	`"Sarah"`
`<http://example.com/project/110>`	`<http://example.com/description>`	`"WebDesign"`
`<http://example.com/project/110>`	`<http://example.com/id>`	`"110"^^<http://www/w3/org/2001/XMLSchema#integer>`
`<http://example.com/project/111>`	`<http://example.com/description>`	`"CloudComputing"`
`<http://example.com/project/111>`	`<http://example.com/id>`	`"111"^^<http://www/w3/org/2001/XMLSchema#integer>`
`<http://example.com/project/112>`	`<http://example.com/description>`	`"DomainRegistration"`
`<http://example.com/project/112>`	`<http://example.com/id>`	`"112"^^<http://www/w3/org/2001/XMLSchema#integer>`
`<http://example.com/employee/10>`	`<http://example.com/involvedIn>`	`<http://example.com/project/110>`
`<http://example.com/employee/12>`	`<http://example.com/involvedIn>`	`<http://example.com/project/111>`
`<http://example.com/employee/11>`	`<http://example.com/involvedIn>`	`<http://example.com/project/112>`
`<http://example.com/employee/11>`	`<http://example.com/involvedIn>`	`<http://example.com/project/111>`

By default, all RDF triples are in the default graph of the output dataset. However, a triples map can contain graph maps that place some or all of the triples into named graphs instead.

A Semantic Web Service Process in OWL-S to Charge a Credit Card

Assume a Web service that charges a valid credit card. In OWL-S, Web services can be modeled as *processes* that specify how clients can interact with the service. There can be any number of *preconditions*, which must all hold in order for the process to be successfully invoked. A process has zero or more *inputs*, representing information required, under some *conditions*, for the performance of the process. A process might have any number of *outputs* that represent information provided by the process to the requester. *Effects* describe real-world conditions the process relies on. To describe the credit card charging process in OWL-S, we have to check whether the card is overdrawn or not, which can be defined as an *atomic process* (a description of a service that expects one message and returns one message in response). If the card is overdrawn, a failure should be displayed. Otherwise, if the card can be charged, the process has to be executed. Hence, the description of a process includes two `result` elements: one for charging the card and another for the error handler (see Listing 9-2).

Listing 9-2. OWL-S Description of Charging a Credit Card [8]

```
<process:AtomicProcess rdf:ID="Purchase">
   <process:hasInput>
      <process:Input rdf:ID="ObjectPurchased" />
   </process:hasInput>
   <process:hasInput>
      <process:Input rdf:ID="PurchaseAmt" />
   </process:hasInput>
   <process:hasInput>
      <process:Input rdf:ID="CreditCard" />
   </process:hasInput>
   <process:hasOutput>
      <process:Output rdf:ID="ConfirmationNum" />
   </process:hasOutput>
   <process:hasResult>
     <process:Result>
        <process:hasResultVar>
           <process:ResultVar rdf:ID="CreditLimH">
              <process:parameterType rdf:resource="&ecom;#Dollars" />
           </process:ResultVar>
        </process:hasResultVar>
        <process:inCondition>
          <expr:KIF-Condition>
            <expr:expressionBody>
              (and (current-value (credit-limit ?CreditCard)
                                  ?CreditLimH)
                   (>= ?CreditLimH ?purchaseAmt))
            </expr:expressionBody>
          </expr:KIF-Condition>
        </process:inCondition>
        <process:withOutput>
           <process:OutputBinding>
              <process:toParam rdf:resource="#ConfirmationNum" />
              <process:valueFunction rdf:parseType="Literal">
                 <cc:ConfirmationNum xsd:datatype="&xsd;#string" />
              </process:valueFunction>
           </process:OutputBinding>
        </process:withOutput>
        <process:hasEffect>
          <expr:KIF-Condition>
            <expr:expressionBody>
              (and (confirmed (purchase ?purchaseAmt) ?ConfirmationNum)
                   (own ?objectPurchased)
                   (decrease (credit-limit ?CreditCard)
                             ?purchaseAmt))
            </expr:expressionBody>
          </expr:KIF-Condition>
        </process:hasEffect>
     </process:Result>
     <process:Result>
        <process:hasResultVar>
           <process:ResultVar rdf:ID="CreditLimL">
              <process:parameterType rdf:resource="&ecom;#Dollars" />
           </process:ResultVar>
        </process:hasResultVar>
        <process:inCondition>
          <expr:KIF-Condition>
            <expr:expressionBody>
              (and (current-value (credit-limit ?CreditCard)
                                  ?CreditLimL)
                   (< ?CreditLimL ?purchaseAmt))
            </expr:expressionBody>
          </expr:KIF-Condition>
        </process:inCondition>
        <process:withOutput rdf:resource="&ecom;failureNotice" />
           <process:OutputBinding>
              <process:toParam rdf:resource="#ConfirmationNum" />
              <process:valueData rdf:parseType="Literal">
                 <drs:Literal>
                    <drs:litdefn xsd:datatype="&xsd;#string">00000000</drs:litdefn>
                 </drs:Literal>
              </process:valueData>
           </process:OutputBinding>
        </process:withOutput>
     </process:Result>
   </process:hasResult>
</process:AtomicProcess>
```

The data transformation produced by the process is specified by the inputs and outputs (hasInput, hasOutput). Atomic processes always receive the inputs specifying the information required for the execution of the process from the client. The result of the process execution is that the credit card is charged, and the money is withdrawn from the account. The effect in this example describes that the customer now owns the object (own ?objectPurchased) and that the amount of money in the credit card account has been reduced (decrease (credit-limit ?CreditCard) ?purchaseAmt). In real-life applications, such services typically send an invoice with or without a notification about the success of the transaction. Credit card transactions have two results: one for the case in which there was sufficient balance to pay the bill, and one for when there wasn't. Each result can be augmented with a further binding.

Modeling a Travel Agency Web Service with WSMO

Assume the following scenario. Leslie wants to book a flight and a hotel for a tropical holiday. The fictional Dream Holidays Travel Agency provides recreational and business travel services based on Semantic Web Service technologies. The travel agency arranges the flight booking and the hotel booking under the contract with the service providers (Figure 9-1).

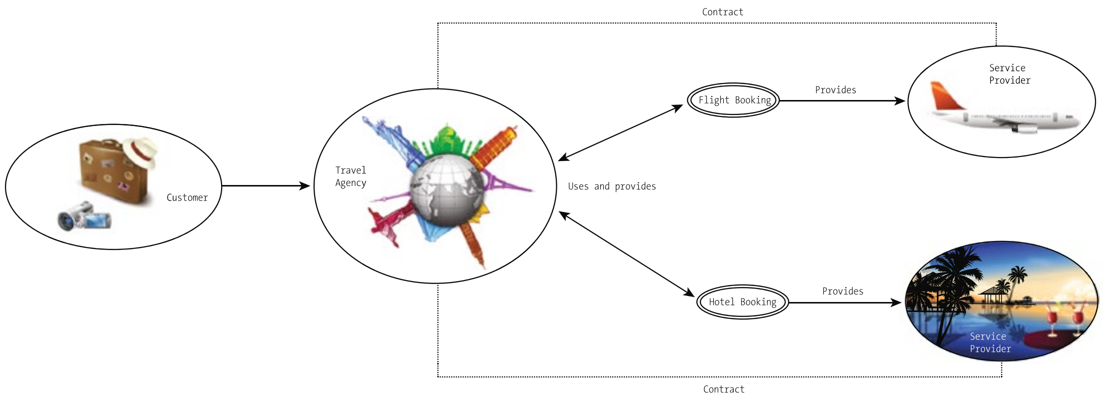

Figure 9-1. *Travel agency modeling*

The goal of the service can be described as "book a flight and hotel room for a tropical holiday for Leslie." The post-condition is to get a trip reservation by providing the current location, the destination, the payment method, and the name of the hotel (see Listing 9-3).

Listing 9-3. Defining the Service Goal

```
goal _"http://www.example.com/successfulBooking"
  capability
    postcondition
      definedBy
        ?tripReservation memberOf tr#reservation[
           customer hasValue fof#Leslie,
           origin hasValue loc#adelaide,
           destination hasValue loc#bali,
           travel hasValue ?flight,
           accommodation hasValue ?Hotel
           payment hasValue tr#creditcard
        ] and
        ?flight[airline hasValue tr#staralliance] memberOf tr#flight and
        ?hotel[name hasValue "Tropical Paradise Hotel"] memberOf tr#hotel .
```

The DREAMHOLIDAYS service description should contain the tickets, hotels, amenities, and so on. The pre-state capability description includes the reservation request and the prerequisites, such as a valid credit card (see Listing 9-4).

Listing 9-4. Pre-State Capability Description

```
capability DREAMHOLIDAYScapability
 sharedVariables {?creditCard, ?initialBalance, ?item, ?passenger}
 precondition
  definedBy
   ?reservationRequest[
        reservationItem hasValue ?item,
        passenger hasValue ?passenger,
        payment hasValue ?creditcard,
      ] memberOf tr#reservationRequest and
     ((?item memberOf tr#trip) or (?item memberOf tr#ticket)) and
     ?creditCard[balance hasValue ?initialBalance] memberOf po#creditCard.

 assumption
  definedBy
   po#validCreditCard(?creditCard) and
   (?creditCard[type hasValue po#visa] or ?creditCard[type hasValue po#mastercard]).
```

The post-state capability description includes the post-conditions, the reservation price, and the final value of the credit card (see Listing 9-5).

Listing 9-5. Post-State Capability Description

```
postcondition
       definedBy
          ?reservation[
               reservationItem hasValue ?item,
               customer hasValue ?passenger,
               payment hasValue ?creditcard
           ] memberOf tr#reservation .

    assumption
       definedBy
         reservationPrice(?reservation, "AUD", ?tripPrice) and
          ?finalBalance= (?initialBalance - ?ticketPrice) and
          ?creditCard[po#balance  hasValue  ?finalBalance] .
```

Querying DBpedia Using the RDF API of Jena

As mentioned earlier, Apache Jena uses the ARQ engine for the processing SPARQL queries. The ARQ API classes are found in `com.hp.hpl.jena.query`. The core classes of ARQ are `Query`, which represents a single SPARQL query; the `Dataset`, where the queries are executed; `QueryFactory`, used for generating `Query` objects from SPARQL strings; `QueryExecution`, which provides methods for query execution; `ResultSet`, which contains the results obtained from an executed query; and `QuerySolution`, which represents a row of query results. If there are multiple answers to the query, a `ResultSet` will be returned, containing the `QuerySolutions`.

To query DBpedia from Jena, you can use `QueryFactory` and `QueryExecutionFactory`. `QueryFactory` has `create()` methods to read a textual query and return a `Query` object with the parsed query. `QueryExecutionFactory` creates a `QueryExecution` to access a SPARQL service over HTTP in the form `QueryExecutionFactory.sparqlService(String service,Query query)`, where `service` is a string representing a SPARQL service. You can create a test connection to DBpedia's SPARQL service, as shown in Listing 9-6.

Listing 9-6. Test Connection to DBpedia's SPARQL Endpoint

```
import com.hp.hpl.jena.query.QueryExecution;
import com.hp.hpl.jena.query.QueryExecutionFactory;
import com.hp.hpl.jena.sparql.engine.http.QueryExceptionHTTP;

public class QueryTest {
  public static void main(String[] args) {
    String service = "http://dbpedia.org/sparql";
    String query = "ASK { }";
    QueryExecution qe = QueryExecutionFactory.sparqlService(service, query);
    try {
      if (qe.execAsk()) {
        System.out.println(service + " is UP");
        }
      } catch (QueryExceptionHTTP e) {
        System.out.println(service + " is DOWN");
      } finally {
      qe.close();
    }
  }
}
```

As you can see, the `service` string contains the SPARQL endpoint of DBpedia. Now, run a query to retrieve people who were born in Eisenach. To achieve this, you need a `SELECT` SPARQL query that searches the person objects for `dbo:birthPlace: Eisenach`, as shown in Listing 9-7.

Listing 9-7. A SPARQL Query to Run on DBpedia from Jena

```
String service="http://dbpedia.org/sparql";
String query="PREFIX dbo:<http://dbpedia.org/ontology/>"
 + "PREFIX : <http://dbpedia.org/resource/>"
 + "PREFIX foaf:<http://xmlns.com/foaf/0.1/>"
 + "select ?person ?name where {?person dbo:birthPlace : Eisenach."
 + "?person foaf:name ?name}";
QueryExecution qe=QueryExecutionFactory.sparqlService(service, query);
ResultSet rs=qe.execSelect();
while (rs.hasNext()){
  QuerySolution s=rs.nextSolution();
  Resource r=s.getResource("?person");
  Literal name=s.getLiteral("?name");
  System.out.println(s.getResource("?person").toString());
  System.out.println(s.getLiteral("?name").getString());
}
```

The result should contain the people who were born in Eisenach, such as Johann Sebastian Bach.

Summary

In this chapter, you analyzed four complex Semantic Web examples. You saw how to map a relational database table to RDF, describe a process in OWL-S, model a Semantic Web Service in WSMO, and, finally, query an LOD dataset programmatically from Apache Jena.

By having read this book, you now understand the core Semantic Web concepts and mathematical background based on graph theory and Knowledge Representation. You learned how to annotate your web site markup with machine-readable metadata from Schema.org, DBpedia, GeoNames, and Wikidata to boost site performance on Search Engine Result Pages. By now, you can write advanced machine-readable personal descriptions, using vCard-based hCard, FOAF, Schema.org, and DBpedia. You have seen how to publish your organization's data with semantics, to reach wider audiences, including HTML5 Microdata or JSON-LD annotations for your company, products, services, and events to be considered by Google for inclusion in the Knowledge Graph. You know how to serialize structured data as HTML5 Microdata, RDF/XML, Turtle, Notation3, and JSON-LD and create machine-readable vocabularies and ontologies in RDFS and OWL. You have learned to contribute to the Open Data and Open Knowledge initiatives and know how to publish your own LOD datasets. You know how to set up your programming environment for Semantic Web application development and write programs in Java, Ruby, and JavaScript using popular APIs and software libraries such as Apache Jena and Sesame. You also learned how to store and manipulate data in triplestores and quadstores and became familiar with the most popular graph databases, such as AllegroGraph and Neo4j. You are capable of describing and modeling Semantic Web Services with OWL-S, WSDL, WSML, and WS-BPEL. You can run complex SPARQL queries on large LOD datasets, such as DBpedia and Wikidata, and even encourage data reuse with your own easy-to-access OpenLink Virtuoso, Fuseki, or 4store SPARQL endpoint. Finally, you learned about Big Data applications leveraging Semantic Web technologies, such as the Google Knowledge Vault, the Facebook Social Graph, IBM Watson, and the Linked Data Service of the largest library in the world.

References

1. Das, S., Sundara, S., Cyganiak, R. (eds.) (2012) R2RML Processors and Mapping Documents. In: R2RML: RDB to RDF Mapping Language. www.w3.org/TR/r2rml/#dfn-r2rml-mapping. Accessed 1 May 2015.

2. Arenas, A., Bertails, A., Prud'hommeaux, E., Sequeda, J. (eds.) (2012) Direct Mapping of Relational Data to RDF. www.w3.org/TR/rdb-direct-mapping/. Accessed 1 May 2015.

3. Antidot (2015) db2triples. https://github.com/antidot/db2triples. Accessed 1 May 2015.

4. OpenLink Software (2015) Virtuoso Universal Server. http://virtuoso.openlinksw.com. Accessed 1 May 2015.

5. Inkster, T. (2015) RDF-RDB2RDF—map relational database to RDF declaratively. https://metacpan.org/release/RDF-RDB2RDF. Accessed 1 May 2015.

6. Calbimonte, J.-P. (2015) morph. https://github.com/jpcik/morph. Accessed 1 May 2015.

7. Capsenta (2015) Ultrawrap. http://capsenta.com. Accessed 1 May 2015.

8. Martin, D. et al. (2004) Service Profiles. In: OWL-S: Semantic Markup for Web Services. www.w3.org/Submission/OWL-S/. Accessed 1 May 2015.