

SAMPLE CHAPTER

Developing RESTful web APIs in Java



# Restlet

## IN ACTION

Jérôme Louvel  
Thierry Templier  
Thierry Boileau

FOREWORD BY Brian Sletten



***Restlet in Action***

by Jérôme Louvel  
Thierry Templier  
Thierry Boileau

**Chapter 10**

# *brief contents*

---

## **PART 1 GETTING STARTED .....1**

- 1 ■ Introducing the Restlet Framework 3
- 2 ■ Beginning a Restlet application 13
- 3 ■ Deploying a Restlet application 46

## **PART 2 GETTING READY TO ROLL OUT .....79**

- 4 ■ Producing and consuming Restlet representations 81
- 5 ■ Securing a Restlet application 121
- 6 ■ Documenting and versioning a Restlet application 151
- 7 ■ Enhancing a Restlet application with recipes  
and best practices 165

## **PART 3 FURTHER USE POSSIBILITIES .....201**

- 8 ■ Using Restlet with cloud platforms 203
- 9 ■ Using Restlet in browsers and mobile devices 242
- 10 ■ Embracing hypermedia and the Semantic Web 274
- 11 ■ The future of Restlet 294

# 10

## *Embracing hypermedia and the Semantic Web*

---

### ***This chapter covers***

- Hypermedia and why it's important for RESTful web APIs
- Hypertext and hyperdata support in Restlet to drive applications
- The relationship between REST and the Semantic Web
- How Restlet can expose and consume linked data in RDF

As you've seen in the previous chapters, the Restlet Framework was designed on top of REST, the architectural style of the web. One of the benefits of that is that Restlet can be used to build all kinds of web applications, classic websites (Web 1.0), RIAs and web APIs (Web 2.0), and now even Semantic Web Services (Web 3.0).

In this chapter we cover in detail one of the core REST principles: *hypermedia as the engine of application state* (HATEOAS). This principle is important for designing RESTful web APIs but difficult to understand and implement correctly and pervasively. We'll explain ways to support this principle and rely on our RESTful mail application to illustrate design options.

We'll explain how Restlet can support the new hyperdata trend, describing its support for the RDF media type via a dedicated extension, on both the client and server sides. REST and the Semantic Web are a perfect match, and a concrete example of this is Restlet's ability to expose and consume linked data in RDF. As an illustration you'll "semantify" the mail application by adding support for semantic contacts using the FOAF standard format.

Let's get started with a short introduction to hypermedia and its relation to the web and to REST.

## 10.1 Hypermedia as the engine of RESTful web APIs

The most challenging principle of the REST architecture style to understand and apply correctly is that your application should be driven by hypermedia. This may at first seem cryptic and less important than other REST principles, such as the identification of resources using URIs or the interaction with resources via a uniform interface, but it's the last link that closes the REST design loop.

In this section we'll first describe the HATEOAS principle before defining hypermedia and two of its common specializations, hypertext and hyperdata (and explaining along the way how the Restlet Framework can help you deal with them).

### 10.1.1 The HATEOAS principle

REST was designed to reduce coupling issues between clients and servers and to allow their independent evolution. This remarkable capability, exploited daily by web browsers that don't need to be recompiled for each website you navigate, relies on the power of hypermedia to progressively discover the next state and available actions of a web application, relying on hyperlinks, embedded scripts, and web forms. This principle is often called HATEOAS, based on chapter 5 of the dissertation that defined REST [12].

HATEOAS means that when accessing a RESTful web API, programmatic clients should be able to dynamically navigate its resources, just as a web browser does with websites, progressively discovering the supported methods, related resources, and so on, thanks to the use of hypermedia. URIs should have no special meaning to clients, even though servers will likely organize them in a specific structure to facilitate implementation of resources, and even though they should be stable to facilitate bookmarking and replaying actions just as with websites.

In general, web API designers prefer to build this knowledge inside API client kits and developer documentation, using custom XML or JSON media types and versioned URIs, as you saw in chapter 6 when we talked about documenting and versioning Restlet applications. One of the reasons is that even though HATEOAS is a core REST principle, designing hypermedia media types isn't a simple task. Let's look closer at this issue, reminding ourselves first what *hypermedia* and *hypertext* mean.

### 10.1.2 What are hypermedia and hypertext?

Before becoming such a widespread concept, the idea of hypermedia was first envisioned in 1945 by Vannevar Bush in his famous article “As We May Think” [13], describing Memex, a system that would help humans to artificially reproduce their mental associative thinking process.

In 1963 Ted Nelson invented the terms *hypermedia* and *hypertext* and tried to illustrate how they could work in an ambitious project named Xanadu, described in an extensive article in *Wired* [14]. Here’s his original definition from his book *Literary Machines*: “By ‘hypertext’ I mean nonsequential writing—text that branches and allows choice to the reader, best read at an interactive screen.”

Even though Xanadu never became complete or usable, it was a great source of inspiration for products such as Hypercard, first published by Apple in 1987, and later the World Wide Web, with HTML as a simplified variant.

Combined with HTTP, HTML found a sweet spot inside the emerging internet and became a central aspect of the REST architecture style. The original WWW document from Tim Berners-Lee and Robert Cailliau in 1990 was titled “WorldWideWeb: Proposal for a HyperText Project,” underlying how critical hypertext and hypermedia were to the web. Here’s the first sentence of their founding email: “HyperText is a way to link and access information of various kinds as a web of nodes in which the user can browse at will. It provides a single user-interface to large classes of information (reports, notes, data-bases, computer documentation, and on-line help).” Over the years, several other formats added hypermedia capabilities, such as:

- PDF, Word, PowerPoint, Excel, and similar office productivity tools that allow inclusion of hyperlinks to web documents, potentially in the same format
- SVG vector image, which embed XLinks to other web documents
- Atom feeds, including links to other blog posts or resources

All those popular formats required significant effort and time to be designed and correctly implemented, but none is as popular as HTML. HTML is lightweight, human-readable, extremely versatile, and interoperable. In addition, it has a built-in forms feature that complements hyperlinks as a way to animate the application, discover next available transitions that a user can follow to navigate existing resources, change their state, create new ones, and so on.

### 10.1.3 Hypertext support in Restlet

As you’ve seen in previous chapters, there are several ways to produce HTML representations using the Restlet Framework. Here are the main options to remember:

- Use a template representation from the FreeMarker or the Velocity extensions, which can be combined with a data model to produce an HTML page to be rendered by a browser. This is similar to the JSP approach.

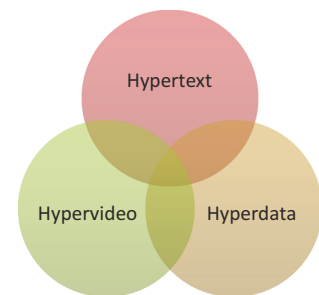
- Use an XML representation, built using either SAX or DOM APIs or retrieved via another means, such as a third-party web API, and wrap it with an XSLT representation as explained in section 4.2, in order to produce HTML output.
- Build the HTML document programmatically by appending elements, attributes, and content to an `AppendableRepresentation` and sending it back to a browser. This works well for short HTML documents that can be held in memory.
- Build the HTML document programmatically by extending the `write(Writer)` method of a `WriterRepresentation` as illustrated in section 8.4.1 and sending it back to a browser. This is a little bit harder to program than the previous option, but can produce an HTML document of any length without requiring large amounts of memory, because the content is progressively streamed to the browser. This is perfect when you convert large amounts of data retrieved from a database on the fly to HTML.
- Serve static HTML documents from disk using a `FileRepresentation`, or even serve static directories like a regular web server, using the `Directory` class as explained in section 8.1.3.

In addition, Restlet can add forms to HTML documents using the previous options and process posted form data sets using either the built-in `org.restlet.data.Form` class for simple URL encoded forms, as explained in section 8.1.1, or the Apache File-Upload extension, illustrated in section 8.1.5 for multipart postings, typically including a mix of regular fields and uploaded files.

Turning to the client side, the typical HTML client is a web browser, but sometimes you need to do a form posting programmatically. For simple forms, the `Form` class will work fine. But to send multipart forms you'll need to use the new `org.restlet.ext.html` extension introduced in version 2.1 of Restlet, including a `FormDataSet` class.

In our context, the problem with HTML is that it was primarily intended for hypertext representations displayed in web browsers, with a human interacting with the application. Most web APIs are used by programmatic clients that need to exchange what are typically stable data structures. This is clearly the main difficulty faced by web APIs that want to be RESTful. As noted by Roy T. Fielding in his blog [15], if you don't respect this principle, you shouldn't mark your web API as RESTful.

As a compromise, you could say it's *inspired by* REST, or *REST-like*, but there's clearly a need to mix structured data exchanges with hypertext and hypermedia in a way that's usable by programmatic clients and not only humans. This new trend is often called *hyperdata*, illustrated in figure 10.1. It's the topic of our next section.



**Figure 10.1** Hyperdata and other types of hypermedia

### 10.1.4 *The new hyperdata trend*

When you imagine your web API, you probably first visualize it from the server point of view, in terms of resource classes, hierarchies of URIs, or XML- or JSON-based data structures exchanged with clients. For a more guided approach, the ROA/D methodology described in appendix D proposes precise steps to develop your web API.

To make it truly RESTful you also need to put yourself in your clients' shoes and make sure that their coupling with your server is minimal, ideally only consisting of some base URIs, knowledge of the uniform interface used (typically HTTP standard methods), and finally knowledge of the media type. Let's focus here on the last point and see more concretely how you can make the XML- or JSON-based media types suitable for REST.

In the next subsections, you'll see three main approaches you can follow, not necessarily in an exclusive way.

#### USING STANDARD REPRESENTATIONS

The easiest approach is to rely on a standard media type such as Atom and its sister AtomPub standards to define how to exchange structured content such as blog entries, eventually extending it, as GData from Google and OData from Microsoft are doing.

Even though Atom and AtomPub are based on XML, retrieving the content in JSON form is generally possible as an alternative. The main benefit from this approach is that you increase the interoperability of your web API, because more clients will be able to access and understand its language. The main drawback is that you need to adapt or wrap your domain data inside those standard structures, which leads to additional complexity. Unless you're building a blog or a notification application, you can end up using the Atom structure as an unnecessary envelope for your own data.

#### MIXING MICROFORMATS AND HTML REPRESENTATIONS

The second possibility is to use HTML and insert structured data directly inside it, such as business cards, calendar events, social connections, and so on. This approach has been pioneered by the Microformats community [16] and is interesting because search engines like Google and Bing will detect the additional data and use it to enrich their search results.

But for programmatic clients the use bar is high, because parsing HTML documents is already a complex task, and parsing additional embedded data is even harder. Although it's a great idea to expose your API resources in HTML with embedded data, you should consider this as one option among other media types by supporting HTTP content negotiation, as explained in section 5.5.

#### CREATING CUSTOM REPRESENTATIONS

The remaining common possibility is to create your own XML- or JSON-based media types from scratch and add hypermedia capabilities to them. The classic web of HTML pages gains a lot of its power from the inclusion of hyperlinks to other pages. This gives modularity, reusability, and navigability of content. In your own custom XML or JSON media types, you can apply the same idea and use the ability to embed links to



other resources in the data you exchange between client and servers, just as you're used to doing with HTML documents.

In the email system you expose resources such as mails, accounts, and so forth. In addition to HTML representations aimed at HTML clients, these resources will provide XML representations for programmatic clients that are interested only in raw data. For example, Mr. Homer Simpson, one of the users of the system, has an account at [www.rmep.org/accounts/chunkylover53/](http://www.rmep.org/accounts/chunkylover53/). The representation of this resource (the data clients and server exchange when describing this user account) could be some XML containing the name of the user, its login, the name used when sending a message using this account, and so on:

```
<account>
  <firstName>Homer</firstName>
  <lastName>Simpson</lastName>
  <login>chunkylover53</login>
  <nickname>Personal mailbox of Homer</nickname>
  <senderName>Homer</senderName>
  ...
</account>
```

Likewise, the representation for a resource exposing a mail item could be some XML including its subject line, its content, its status (draft, sent, received), and so on. Each mail item is also stored in a particular account. In the representation of the mail resource you can refer to this account by using the URI as a hyperlink, as shown in the following snippet, where the account in question is none other than Mr. Simpson's:

```
<mail>
  <status>received</status>
  <subject>Message to self</subject>
  <content>Doh!</content>
  <accountRef>
    http://www.rmep.org/accounts/chunkylover53/
  </accountRef>
  ...
</mail>
```

A programmatic client that is handed the representation of a given mail item can then, if needed, interact with the account in which the mail is stored using the URI reference. The client application can get the representation of the account using the GET method and modify it with other methods such as PUT, POST, and DELETE, if allowed.

Using URIs like this to refer to other resources is easy and straightforward and will make your web API closer to being RESTful, with reasonable design and implementation effort.

But a limitation of this custom media type approach is that it restricts the clients that can use your web API. If you don't have the ability to develop and distribute client kits or libraries for popular platforms such as Java, .NET, Python, PHP, Android, or iOS, your clients will first need to develop parsers and formatters for your media types, which is a barrier for use and interoperability.

Whichever option you choose for your representations, you should expect to spend as much time on their design as you do for other web API aspects, including URI namespace structure.

As you've seen, interoperability between RESTful clients and servers based on standard formats is important and is a key factor in the success of the classic web based on the HTML, HTTP, and URI trio. But it's possible to go beyond this when developing hyperdata formats. The idea is to add a level of interoperability among the data itself, using the Semantic Web and its RDF standard format in a lightweight and pragmatic way. This approach is often called *Linked Data* and is the topic of the next section.

## 10.2 The Semantic Web with Linked Data

As you've seen, hypermedia isn't only about hypertext media types, but is also about hyperdata and, more importantly, interoperability of data. This goal is shared by the Semantic Web, which was also initiated by Tim Berners-Lee.

In this section we first give a brief overview of the Semantic Web and its relationship to REST via the new Linked Data initiative, which is a great illustration of hyperdata. Then we introduce RDF, the core standard for semantic representations, and explain how to expose, consume, and browse linked data with Restlet. This is also a good time to introduce FOAF, an RDF vocabulary to describe social relationships.

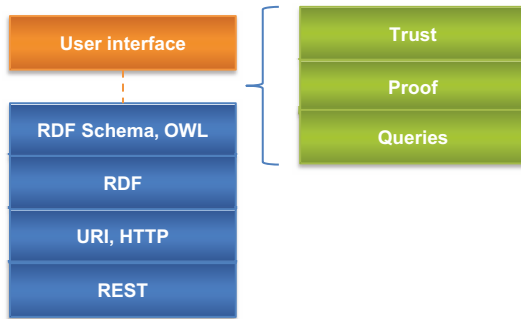
### 10.2.1 REST and the Semantic Web

The Semantic Web is an ambitious initiative that was publicly launched in 2001 by a now-famous *Scientific American* article [17] that resulted in great expectations. The work on its foundations involved many researchers across the globe and took a significant amount of time. During this time, developers and companies were left wondering how they could best take advantage of the specifications that had started to come out of the W3C, such as RDF, RDF Schema, OWL, and SPARQL. An impression of excessive complexity started to emerge along with the perceived lack of real-world use cases where the Semantic Web could shine.

That's when Tim Berners-Lee introduced his Linked Data idea, as an application of the Semantic Web that would allow browsing of semantically linked resources. Instead of storing semantic data in large specialized databases and requiring a special language to interact with them, the idea was to use the web and its HTTP protocol as a way to interact directly with the graph of semantic data, using hyperlinks to jump from one hyperdata document to another, just as we do with hypertext documents.

Linked Data finally offers a pragmatic and operational approach to the Semantic Web that's also perfectly in line with REST principles, including HATEOAS. This lighter approach was the foundation of Restlet support for the Semantic Web, available in the `org.restlet.ext.rdf` extension.

Concretely, Linked Data relies on URIs (which are HTTP URLs) to identify important data, in the same way that they identify documents on the regular web. It also relies on HTTP to retrieve, create, update, or delete the data, as with other RESTful web APIs.



**Figure 10.2** The Linked Data technological stack

### Restlet and its semantic roots

When the Restlet Framework launched in 2005, it was the result of extracting a generic piece of code from a website project called Semalink, aimed at facilitating the adoption of the Semantic Web by closing the gap with the regular web of documents. Later on, the success of the Linked Data initiative and its support in Restlet via the RDF extension were in a way a return to the project roots.

As illustrated in figure 10.2, Linked Data relies on the RDF language (on top of the REST, URI, and HTTP lower layers) to represent those data resources and their relationships with other resources and their attributes. Finally we have RDF Schema and its richer cousin, Ontology Web Language (OWL), which are languages (also expressed in RDF) used to define valid RDF graph structures called *ontologies*, or *meta-models* if you're more familiar with model-driven engineering.

In the long term you can expect other layers of the Semantic Web vision to find their way inside this stack to solve issues such as distributed queries, proof, and trust. There are already proposals to address these needs, such as SPARQL to query RDF databases. Some of these were defined before the Linked Data initiative and will probably have to be rethought to fit better with the regular web in order to reach a broader use level.

Regarding trust, there's a WebID protocol [3] in the W3C Incubator which aims at using HTTPS and client SSL certificates to build a web of trust in a pragmatic way that can nicely complement Linked Data. We've mentioned RDF several times, so now it's time to have a closer look at it and see how to use it for resource representations.

#### 10.2.2 Using RDF in representations

RDF is the acronym for Resource Description Framework, where a resource has the same meaning as in the REST architecture style—something of interest that can be addressed by a URI. As its name implies, RDF provides a way to describe and represent web resources in a precise and interoperable manner. To help you understand how RDF works, we first present the RDF data model, explaining the main concepts involved and how they relate to REST, and discuss the serialization media types available for this data model.

To make this discussion more concrete, we'll use examples from the FOAF language, which lets you express social links between people, a much simpler but open and semantic variant of Facebook or LinkedIn data sets.

### RDF DATA MODEL

In RDF all the data is defined as a graph, where nodes are either resources identified by a URI or literals (like a string or an integer) and where links connect either one resource to another or a resource and a literal defined as an attribute value. Those links are also frequently called statements, triples, or properties.

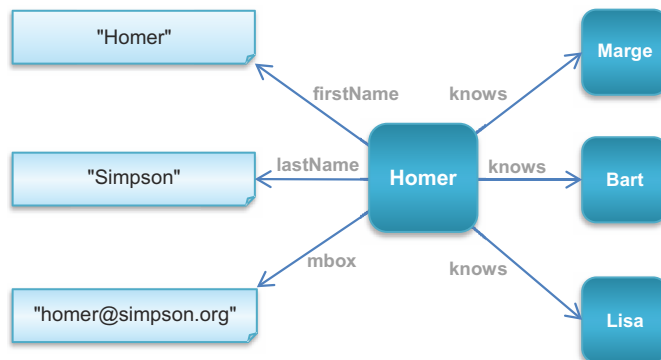
Figure 10.3 partially describes the Homer Simpson resource as a graph with a central resource node, three literal nodes on the left, and three related resource nodes on the right.

As you can see, the links have labels defining how Homer relates to the Marge, Bart, and Lisa resources and the meaning of the "Homer," "Simpson," and "homer@simpson.org" literals. Obviously, Homer knows his wife and children, and we could have added more links to express the exact familial relationships, such as father and husband. But in this case we decided to follow the FOAF vocabulary [18], which is less interested in family relationships than in generic links between people.

With the previous example, you almost have a valid RDF graph. The piece that's missing is unambiguous information telling you that the links are related to FOAF and not to another vocabulary. For this purpose, RDF relies again on URIs to precisely and uniquely define the meaning of those links.

This is the most important difference with hyperlinks found in HTML documents: better interoperability and the ability to have several links between the same pair of resources. In this case, the exact value of *knows* is <http://xmlns.com/foaf/0.1/knows>, the value of *mbox* is <http://xmlns.com/foaf/0.1/mbox>, and so on.

Let's create this example RDF graph using the RDF extension of the Restlet Framework available in the `org.restlet.ext.rdf.jar` file. As illustrated in the following listing, the translation is straightforward, because you're able to reuse the `Reference` class from the `org.restlet.data` package to define URI references.



**Figure 10.3** Example RDF graph partially describing Homer Simpson

## Listing 10.1 Creating an RDF graph with Restlet RDF extension

```

import org.restlet.data.Reference;
import org.restlet.ext.rdf.Graph;
import org.restlet.ext.rdf.Literal;

public class FoafExample {

    public static void main(String[] args) throws IOException {
        String FOAF_BASE = "http://xmlns.com/foaf/0.1/";
        Reference firstName = new Reference(FOAF_BASE + "firstName");
        Reference lastName = new Reference(FOAF_BASE + "lastName");
        Reference mbox = new Reference(FOAF_BASE + "mbox");

        Reference homerRef = new Reference(
            "http://www.rmep.org/accounts/chunkylover53/");
        Reference margeRef = new Reference(
            "http://www.rmep.org/accounts/bretzels34/");
        Reference bartRef = new Reference(
            "http://www.rmep.org/accounts/jojo10/");
        Reference lisaRef = new Reference(
            "http://www.rmep.org/accounts/lisa1984/");

        Graph example = new Graph();
        example.add(homerRef, firstName, new Literal("Homer"));
        example.add(homerRef, lastName, new Literal("Simpson"));
        example.add(homerRef, mbox, new Literal(
            "mailto:homer@simpson.org"));
        example.add(homerRef, knows, margeRef);
        example.add(homerRef, knows, bartRef);
        example.add(homerRef, knows, lisaRef);
    }
}

```

**FOAF ontology constants**

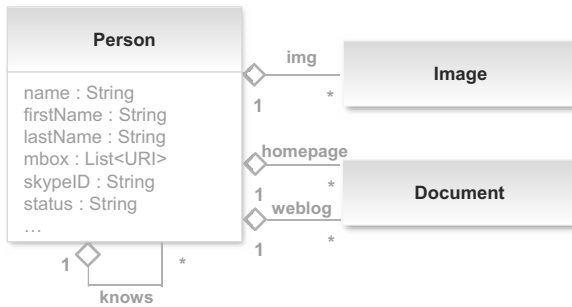
**Linked Simpson resources**

**Example RDF graph**

As introduced in figure 10.2, it's also possible to define the structure of valid RDF graphs as ontologies using the RDF Schema and OWL languages. *Ontology* might sound like a daunting term, but in this setting it refers to a set of object classes where the relations between classes and attributes are typed using a URI rather than a potentially ambiguous and imprecise label.

Figure 10.4 illustrates how to visualize a part of the FOAF ontology as a regular UML class diagram. If you read appendix D on the ROA/D methodology, you should be able to see how remarkably this diagram can complement figure D.15 and the Account resource class. An RDF class defined as Linked Data can be exactly the same as a REST resource class defined in a web API (see section D.4.3, “Identifying and classifying the resources,” for details), but adding further information about its attributes and relationships with other resource classes, such as other persons, images, and documents in the FOAF case.

Let's move beyond the abstract RDF data model and see how to serialize RDF graphs and use them as representations.



**Figure 10.4** Person class in the FOAF vocabulary

### RDF REPRESENTATION VARIANTS

Contrary to XML vocabularies such as Atom or XHTML, RDF doesn't force you to use a single serialization format. Even though the primary format is XML-based, there are others available:

- RDF/XML, a comprehensive XML serialization format for RDF
- Notation 3 (or n3), a compact alternative to RDF/XML also able to express rules
- Turtle, a subset of n3, simple and human-readable
- N-Triples, an even simpler subset of Turtle, useful for storing and exchanging RDF

To make those formats more concrete, you'll now serialize the example FOAF graph built with the Restlet extension. Let's add the following lines of code to the code in listing 10.2:

```

System.out.println("\nRDF/XML format:\n");
example.getRdfXmlRepresentation().write(System.out);

System.out.println("\nRDF/n3 format:\n");
example.getRdfN3Representation().write(System.out);

System.out.println("\nRDF/Turtle format:\n");
example.getRdfTurtleRepresentation().write(System.out);

System.out.println("\nRDF/NTriples format:\n");
example.getRdfNTriplesRepresentation().write(System.out);

```

The `getRdf*Representation()` methods create an instance of the `RdfRepresentation` class, passing it the `Graph` instance and the proper media type constant. Now, if you run this code, you'll first serialize the graph into RDF/XML. The key XML element is `rdf:Description`, which contains all the properties related to the Homer resource identified by the XML attribute `rdf:about`. Note also how a prefix `__NS1` was declared for the FOAF ontology URI:

```

<?xml version="1.0" standalone='yes'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:__NS1="http://xmlns.com/foaf/0.1/">
  <rdf:Description rdf:about="http://www.rmep.org/accounts/chunkylover53/">
    <__NS1:firstName>Homer</__NS1:firstName>
    <__NS1:lastName>Simpson</__NS1:lastName>
    <__NS1:mbox>mailto:homer@simpson.org</__NS1:mbox>
  </rdf:Description>
</rdf:RDF>

```

```

    <__NS1:knows>
      <rdf:Description rdf:about="http://www.rmep.org/accounts/bretzels34/"
    /></__NS1:knows>
    <__NS1:knows>
      <rdf:Description rdf:about="http://www.rmep.org/accounts/jojo10/"
    ></__NS1:knows>
    <__NS1:knows>
      <rdf:Description rdf:about="http://www.rmep.org/accounts/lisa1984/"
    ></__NS1:knows>
  </rdf:Description>
</rdf:RDF>

```

The second and third serializations are for n3 and Turtle formats and produce the same result. Note in the following snippet that the graph is all written in one line starting with `<http://www.rmep.org/accounts/chunkylover53/>`. In addition you can use namespace as illustrated in the first lines:

```

@prefix #: <:>.
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix type: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdf: <http://www.w3.org/2000/01/rdf-schema#>.
@keywords a, is, of, has.
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/firstName> "Homer";
<http://xmlns.com/foaf/0.1/lastName> "Simpson";
<http://xmlns.com/foaf/0.1/mbox> "mailto:homer@simpson.org";
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/bretzels34/>,
<http://www.rmep.org/accounts/jojo10/>,
<http://www.rmep.org/accounts/lisa1984/>.

```

Let's see how the fourth and simplest format, N-Triples, serializes the graph. For each link there's a new line, and no prefixes or factorizations are used. The result is more verbose but also straightforward to write, read, and understand:

```

<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/firstName> "Homer".
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/lastName> "Simpson".
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/mbox> "mailto:homer@simpson.org".
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/bretzels34/>.
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/jojo10/>.
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/lisa1984/>.

```

In addition to those pure RDF serializations, it's also possible to embed RDF inside other documents such as XHTML pages to enrich them with semantic data. The W3C-supported way to do this is RDFa, but similar efforts have been proposed, such as

Microformats and HTML 5 Microdata [19]. In the following snippet you can see one way to express the example graph as an XHTML page with RDFa special attributes:

```
<div xmlns:foaf="http://xmlns.com/foaf/0.1/"
  about="http://www.rmep.org/accounts/chunkylover53/">
  <span property="foaf:firstName">Homer</span>
  <span property="foaf:lastName">Simpson</span>
  <a rel="foaf:mbox" href="mailto:homer@simpson.org">
homer@simpson.org</span>
  <a rel="foaf:knows" href="http://www.rmep.org/accounts/bretzels34/">
Marge </a>
  <a rel="foaf:knows" href="http://www.rmep.org/accounts/jojo10/">
Bart</a>
  <a rel="foaf:knows" href="http://www.rmep.org/accounts/lisa1984/">
Lisa</a>
</div>
```

The mixed approach illustrated by RDFa makes it easy to embed semantic data in web pages but also makes it harder to extract the data back into proper RDF. To solve these issues W3C proposed GRDDL as a standard way to extract the RDF data by applying XSLT stylesheets to the XHTML documents.

### **The Schema.org initiative**

In June 2011, Google, Bing, and Yahoo! launched the Schema.org initiative to facilitate semantic annotation of data in regular web pages. They provide a way to express semantic data using HTML Microdata, including both common ontologies for things such as Persons, Places, Organizations, and so on, and support for their extraction in the most popular web search engines. A mapping to RDFa is also specified, making it a great option to consider for mixing RDF and web pages.

At this point, you should understand how powerful the RDF data model is—capable of modeling anything in a precise way—and how flexible it can be used in representation formats depending on the use context. In the next section you use RDF and its support in the Restlet Framework to expose the mail accounts as Linked Data.

## **10.3 Exposing and consuming Linked Data with Restlet**

In this section you restore the example mail application and expose two variants of the account resources, one in XML and one in RDF, using the FOAF vocabulary. Finally we look at the client side and see how to consume Linked Data and navigate from one resource to another by following RDF links.

### **10.3.1 Exposing RDF resources**

As you saw in listing 10.1, the RDF extension of Restlet comes with a DOM-like API composed of the following classes:

- Graph contains links and can produce an RDF representation.



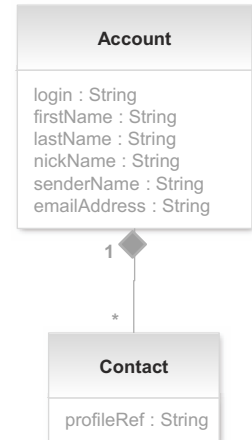
- Link is equivalent to an RDF statement or a triple composed of a source resource URI reference, a link type URI reference, and a target value, either literal or resource.
- Literal provides target values along with a datatype URI reference and language.

Once you've built a Graph instance, you can return it directly via a Restlet resource with annotated methods. To put this into practice in the example application, you first need to complete the account resource to expose a richer domain model than in previous chapters, where you only used a simple string.

Figure 10.5 shows enriched Account and Contact classes with new properties and relationships. Note how it resembles figure D.8 in appendix D (covering the ROA/D methodology, applied to our mail example), presenting only a part of the whole domain model but with more detail on the available properties.

Beyond the obvious properties in Account, the profileRef property in Contact is supposed to contain a URI reference to the FOAF profile of the contact. If the contact is also managed by the RESTful mail application, the URI will refer to the related Account resource (which will have a FOAF variant exposed).

The following listing initializes the domain model with four user accounts—for Homer Simpson and the three other members of his family: Marge, Bart, and Lisa.



**Figure 10.5** RESTful mail example domain object model

### Listing 10.2 Setting up the domain model with user accounts

```

public MailServerApplication() {
    setName("RESTful Mail API application");
    setDescription("Example API for 'Restlet in Action' book");
    setOwner("Restlet SAS");
    setAuthor("The Restlet Team");

    Account homer = new Account();
    homer.setFirstName("Homer");
    homer.setLastName("Simpson");
    homer.setLogin("chunkylover53");
    homer.setNickName("Personal mailbox of Homer");
    homer.setSenderName("Homer");
    homer.setEmailAddress("homer@simpson.org");
    homer.getContacts().add(new Contact("/accounts/bretzels34/"));
    homer.getContacts().add(new Contact("/accounts/jojo10/"));
    homer.getContacts().add(new Contact("/accounts/lisa1984/"));
    getAccounts().put("chunkylover53", homer);

    Account marge = new Account();
    marge.setFirstName("Marjorie");
    marge.setLastName("Simpson");
    marge.setLogin("bretzels34");
    marge.setNickName("Personal mailbox of Marge");
  
```

← **Homer user account**

← **Marge user account**

```

marge.setSenderName("Marge");
marge.setEmailAddress("homer@simpson.org");
marge.getContacts().add(new Contact("/accounts/chunkylover53/"));
marge.getContacts().add(new Contact("/accounts/jojo10/"));
marge.getContacts().add(new Contact("/accounts/lisa1984/"));
getAccounts().put("bretzels34", marge);

Account bart = new Account();
    bart.setFirstName("Bartholomew");
    bart.setLastName("Simpson");
    bart.setLogin("jojo10");
    bart.setNickName("Personal mailbox of Bart");
    bart.setSenderName("Bart");
    bart.setEmailAddress("bart@simpson.org");
    bart.getContacts().add(new Contact("/accounts/chunkylover53/"));
    bart.getContacts().add(new Contact("/accounts/bretzels34/"));
    bart.getContacts().add(new Contact("/accounts/lisa1984/"));
    getAccounts().put("jojo10", bart);

Account lisa = new Account();
    lisa.setFirstName("Lisa");
    lisa.setLastName("Simpson");
    lisa.setLogin("lisa1984");
    lisa.setNickName("Personal mailbox of Lisa");
    lisa.setSenderName("Lisa");
    lisa.setEmailAddress("lisa@simpson.org");
    lisa.getContacts().add(new Contact("/accounts/chunkylover53/"));
    lisa.getContacts().add(new Contact("/accounts/bretzels34/"));
    lisa.getContacts().add(new Contact("/accounts/jojo10/"));
    getAccounts().put("lisa1984", lisa);
}

```

← **Bart user account**

← **Lisa user account**

Let's now enhance the AccountResource annotated interface to expose the FOAF variant representation. As illustrated in listing 10.3, our first choice would have been to use a generic AccountRepresentation bean, to offer automatic conversion with XML, JSON, and similar formats using the XStream, Jackson, and similar Restlet extensions, as detailed in chapter 4. But for RDF representations we don't yet have an integrated and automated solution to annotate beans to produce proper RDF, even though existing open source projects [20] could be integrated to Restlet to achieve this. As a workaround you declare another getFoafProfile() method annotated with @Get but this time specifying with the "rdf" annotation value that this only returns RDF variants, either in RDF/XML, RDF/n3, Turtle, or N-Triples format.

### Listing 10.3 Enhanced Account annotated resource interface

```

public interface AccountResource {
    @Get
    public AccountRepresentation represent();

    @Get("rdf")
    public Graph getFoafProfile();
}

```

← **HTTP GET for XML, JSON**

← **HTTP GET for RDF variant**

Let's continue by providing the implementation of this enhanced resource via the `AccountServerResource` class, shown in listing 10.4. The implementation of both annotated methods is similar in the sense that they both populate a representation, using simple Java properties in the first case and URI properties in the second case, based on the FOAF ontology. Both approaches have benefits and drawbacks in terms of simplicity and interoperability.

The first approach is simpler because regular POJOs are used, but the interoperability is limited because the clients must have knowledge of those properties and their meanings to interpret and make good use of them.

The second approach is more complex because you need to type each property using a URI based on a parent ontology, which is less natural for Java developers. The benefit is that interoperability is improved because clients will be able to interpret the data at a higher level than raw XML. If they understand the ontology (such as the one defined by FOAF), they can precisely interpret the resulting semantic representations.

One of the advantages of using Restlet is that you can use both approaches at the same time, without additional development cost, as illustrated in the following listing.

#### Listing 10.4 Enhanced Account server resource

```
public class AccountServerResource extends ServerResource implements
    AccountResource {

    private Account account;

    public Map<String, Account> getAccounts() {
        return ((MailServerApplication) getApplication()).getAccounts();
    }

    @Override
    protected void doInit() throws ResourceException {
        String accountId = getAttribute("accountId");
        this.account = getAccounts().get(accountId);
    }

    public void remove() {
        getAccounts().remove(this.accountId);
    }

    public AccountRepresentation represent() {
        AccountRepresentation result = null;

        if (account != null) {
            result = new AccountRepresentation();
            result.setEmailAddress(account.getEmailAddress());
            result.setFirstName(account.getFirstName());
            result.setLastName(account.getLastName());
            result.setLogin(account.getLogin());
            result.setNickName(account.getNickName());
            result.setSenderName(account.getSenderName());

            for (Contact contact : account.getContacts()) {
                result.getContactRefs().add(contact.getProfileRef());
            }
        }
    }
}
```

← The associated account

← Build representation bean

```

        return result;
    }


    public Graph getFoafProfile() {
        Graph result = null;

        if (account != null) {
            result = new Graph();
            result.add(getReference(), FoafConstants.MBOX,
                new Literal("mailto:" + account.getEmailAddress()));
            result.add(getReference(), FoafConstants.FIRST_NAME,
                new Literal(account.getFirstName()));
            result.add(getReference(), FoafConstants.LAST_NAME,
                new Literal(account.getLastName()));
            result.add(getReference(), FoafConstants.NICK,
                new Literal(account.getNickName()));
            result.add(getReference(), FoafConstants.NAME,
                new Literal(account.getSenderName()));

            for (Contact contact : account.getContacts()) {
                result.add(getReference(), FoafConstants.KNOWS,
                    new Reference(getReference(),
                        contact.getProfileRef().getTargetRef()));
            }
        }

        return result;
    }
}

```


**Build RDF  
graph**

When you return the `Graph` instance from the `getFoafProfile()` method, the `ConverterService` takes over the process. If you've correctly added the `org.restlet.ext.rdf.jar` in your classpath, the `RdfConverter` will automatically wrap the `Graph` instance into an `RdfRepresentation` using the correct RDF media type based on client preferences.

At this point we encourage you to launch the server component and retrieve both representations from a web browser using the following URIs:

```

http://localhost:8111/accounts/chunkylover53/?media=xml
http://localhost:8111/accounts/chunkylover53/?media=rdf

```

As a result, you should obtain XML and FOAF documents similar to the one presented in sections 10.1.4 and 10.2.2. The mail example application is now not only RESTful but also part of the Linked Data!

### **SAX-LIKE RDF PROCESSING**

In addition, the RDF extension is also capable of handling large RDF representations. The issue with the DOM-like `Graph` approach is that all links are stored in memory before being serialized or after being parsed.

For this purpose, the extension comes with a `GraphHandler` abstract class that's similar in spirit to the `SAX ContentHandler` interface. You can use this class for both RDF parsing and writing purposes, and it contains a callback method for RDF processing events such as `startGraph()`, `endGraph()`, and `link(...)`.

We won't cover this SAX-like RDF feature in detail here, but if you want to experiment with it, you should look at the `GraphBuilder` class provided, which is a subclass of `GraphHandler` creating a `Graph` instance when parsing an RDF representation. There's also `RdfRepresentation` and its `parse(GraphHandler)` and `write(GraphHandler)` methods, which you can override to provide custom handling.

#### TEMPLATE RDF GENERATION

Remember that it's always possible to produce RDF representations using template engines like `FreeMarker` and `Velocity`, if that fits your use case. There's nothing wrong with this approach; it can easily give you precise control over RDF formatting.

Let's move to the client side and see how you can consume—and, more important, browse—your Linked Data mail application.

### 10.3.2 Consuming linked data with Restlet

In this section we'll explain how Restlet can consume RDF representations and we'll address the two main challenges you face when consuming linked data.

The first challenge is the need to support several RDF formats, such as `RDF/XML` and `RDF/n3`. For this purpose, the Restlet extension for RDF gives you an abstraction layer with the `RdfRepresentation` class, which is capable of using the correct parser based on the media type of the RDF representation returned by an origin server.

The second challenge is to have the ability to easily navigate among hyperlinked resources based on your use case. Again the RDF extension has a handy solution thanks to its `RdfClientResource`, which adds class methods to `ClientResource` for retrieving linked RDF resources and literal-valued properties.

Another challenge is to add knowledge to your clients of the RDF vocabularies (also known as ontologies) commonly used in Linked Data, and sometimes to map from one ontology to another. Here the best practice is to use existing ontologies such as `RDF Schema`, `OWL`, `Dublin Core` or `FOAF` as much as possible.

Let's get back to coding and try to consume the account resources using RDF. You could use the annotated `AccountResource` interface, but let's assume that the back-end isn't necessarily written in Restlet. After all, a web API should be accessible from any kind of HTTP client. Listing 10.5 only provides the URI of Homer's account, and still you're able to display the literal value of each of its properties, as well as the properties of each of its contacts, by navigating to the linked FOAF profile based on the `FoafConstants.KNOWS` URI (`http://xmlns.com/foaf/0.1/knows`).

#### Listing 10.5 Generic FOAF browser

```
public class FoafBrowser {

    public static void main(String[] args) {
        displayFoafProfile("http://localhost:8111/accounts/chunkylover53/");
    }

    public static void displayFoafProfile(String uri) {
        displayFoafProfile(new RdfClientResource (uri), 1);
    }
}
```

**Launch  
FOAF  
browsing**



```

public static void displayFoafProfile(RdfClientResource foafProfile,
    int maxDepth) {
    Set<Couple<Reference, Literal>> literals = foafProfile.getLiterals();

    if (literals != null) {
        for (Couple<Reference, Literal> literal : literals) {
            System.out.println(literal.getFirst().getLastSegment() + ": "
                + literal.getSecond());
        }
    }
    System.out.println("-----");

    if (maxDepth > 0) {
        Set<RdfClientResource> knows = foafProfile
            .getLinked(FoafConstants.KNOWS);

        if (knows != null) {
            for (RdfClientResource know : knows) {
                displayFoafProfile(know, maxDepth - 1);
            }
        }
    }
}

```

**Recursive FOAF display**

If you launch this FoafBrowser after starting the server, you should see the following output in the console, plus some log messages related to the HTTP connector launch:

```

lastName: Simpson
firstName: Homer
nick: Personal mailbox of Homer
name: Homer
mbox: mailto:homer@simpson.org
-----
lastName: Simpson
mbox: mailto:lisa@simpson.org
name: Lisa
firstName: Lisa
nick: Personal mailbox of Lisa
-----
lastName: Simpson
firstName: Bartholomew
mbox: mailto:bart@simpson.org
nick: Personal mailbox of Bart
name: Bart
-----
firstName: Marjorie
nick: Personal mailbox of Marge
name: Marge
mbox: mailto:homer@simpson.org
lastName: Simpson
-----

```

Although that approach is convenient and expressive, it requires you to load all of the DF representations in memory, in a DOM-like way. If you need to load larger representations or have a finer-grained control of your RDF client, you can also use the

`RdfRepresentation` class directly, in a SAX-like way. For this purpose you can use the constructor that takes a `Representation` parameter and then invoke the `parse` (`GraphHandler`) method to consume its content as a series of `Link` instances, one link at a time.

As you’ve seen in this section, consuming linked resources using RDF representations is as easy as exposing them with the Restlet extension for RDF. It provides a solution for most of the challenges you’ll face with only a few additional classes, mainly `RdfClientResource` and `RdfRepresentation`.

## 10.4 Summary

This chapter covered a lot of ground, including advanced topics that many web API developers aren’t initially aware of. Awareness of these topics is increasingly important for the future of RESTful web APIs.

Hypermedia, Linked Data, and above all the Semantic Web are topics that are much too large to cover in this chapter (or even in this book). We tried to present the important parts of the RESTful web in a pragmatic and Restlet-centric way to give you a sense of both the importance and the power of concepts such as HATEOAS and hyperdata, while giving concrete examples of implementation using Restlet on both client and server sides.

We presented the RDF extension of Restlet, which is capable of handling most common RDF serialization formats, such as RDF/XML, RDF/n3, Turtle, and N-Triples, and provides a convenient abstraction layer for both reading and writing, using either a DOM-like or SAX-like approach (to draw a parallel with common XML-processing techniques).

You’re now approaching the end of this book, with a final chapter that steps back a little to see what Restlet has to offer beyond what’s been covered so far, such as extra extensions available, resources provided by the community that extend or make use of the Restlet Framework, and the roadmap for the next Restlet version.

# Restlet IN ACTION

Louvel • Tempplier • Boileau



In a RESTful architecture any component can act, if needed, as both client and server—this is flexible and powerful, but tricky to implement. The Restlet project is a reference implementation with a Java-based API and everything you need to build servers and web clients that integrate with most web and enterprise technologies.

**Restlet in Action** introduces the Restlet Framework and RESTful web APIs. You'll see how to easily create and deploy your own web API while learning to consume other web APIs effectively. You'll learn about designing, securing, versioning, documentation, optimizing, and more on both the server and client side, as well as about cloud computing, mobile Android devices, and Semantic Web applications.

## What's Inside

- Written by the creators of Restlet!
- How to create your own web API
- How to deploy on cloud and mobile platforms
- Focus on Android, Google App Engine, Google Web Toolkit, and OSGi technologies

The book requires a basic knowledge of Java and the web, but no prior exposure to REST or Restlet.

The authors are founders and technical leads of Restlet Inc. and Restlet SAS. **Jérôme Louvel** is the creator of the Restlet Framework.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit [manning.com/RestletinAction](http://manning.com/RestletinAction)

“Broad, deep,  
and example-driven.”

—From the Foreword by  
Brian Sletten  
Bosatsu Consulting

“Accurate, informative,  
and extremely useful.”

—Dustin Jenkins, National  
Research Council of Canada

“A must-have for RESTful  
web services Java developers.”

—Fabián Mandelbaum  
NeoDoc SARL

“A broad reach for a perfectly  
minimalist framework.”

—Tal Liron, Three Crickets

“Thoroughly recommended ...  
helps the reader come  
to grips with REST.”

—Dave Pawson  
Pawson Software Services, Ltd

ISBN 13: 978-1-935182-34-4  
ISBN 10: 1-935182-34-X



9 781935 182344