Extracted from:

Exploring Graphs with Elixir

Connect Data with Native Graph Libraries and Graph Databases

This PDF file contains pages extracted from *Exploring Graphs with Elixir*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2022 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Exploring Graphs with Elixir

Connect Data with Native Graph Libraries and Graph Databases

Tony Hammond

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

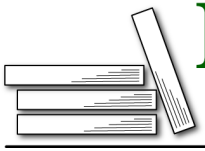
Exploring Graphs with Elixir

Connect Data with Native Graph Libraries and Graph Databases

Tony Hammond

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2022 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-840-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2022

CHAPTER 7

Graphing Globally with RDF

While property graphs focus on solving a range of immediate tactical problems, the RDF graph model focuses on the longer-term strategic goals of large-scale data integration, especially public data integration. This has been one of the driving forces behind the semantic web vision.

The Resource Description Framework (RDF) is a data model standardized by the W3C for describing resources on the web. But note that it's the descriptions that are on the web, not necessarily the resources that are being described. This means that we can use RDF to describe anything, whether it's online or not. Put simply, we might say anything that can be named. In fact, we can also describe unnamed things too. So, "anything" can be described. And these descriptions are built up using the common web-naming convention—the URI.

Of course, we can also use property graphs to describe things, but here's the real kicker. Without any common naming convention, it's difficult to exchange any meanings except by some prior arrangement, which means an application has to keep track of what goes with what, and what means what. With a common naming system, however, we can freely share our descriptions of things with others—with no ambiguity. And we can share our meanings too, which is a crazy big win.

So how does all this work?

The RDF data model describes resources in terms of statements which can be interpreted as a basic graph structure—a directed labeled graph. The labels used in RDF graphs are URIs. This is that little bit of magic that allows one RDF graph to be added to another RDF graph. This leads to an incredibly simple way of doing data integration. Just add your RDF datasets together—the names, or labels, that are used will take care of joining the data elements together.

• Click HERE to purchase this book now. discuss

By the way, the title of this chapter implies the use of URIs as global names, which allows for building global data structures. In practice, this also allows for building out a global graph, which is kind of awesome. More awesome though is the fact that this global graph already exists in the form of the linked open data (LOD) cloud[1] and is essentially a growing graph of graphs of open data.

Web Names—A Crib Sheet

URL—Uniform Resource Locator
A web document address

URI—Uniform Resource Identifier
A web identifier for a resource, which may also be located

IRI—Internationalized Resource Identifier
An international form for a URI

RDF is a data model and not a data format, which means that we can write out an RDF graph in many different ways. There are a number of standard serializations by the W3C, and these allow for a high degree of interoperability in reading and writing RDF data. We thus have a standard means of sharing RDF graphs and can input and output them with ease.

But there is more to RDF than this. We can also define and express standard schema languages in RDF, and then layer this schema RDF on top of an existing RDF dataset. This allows us to query over a richer graph of data—one that has now been augmented with the actual data model that the data conforms to. And as the schema languages encode a given logic, we can then make inferences over the data and add these new inferred statements back into our graph.

What's Different About RDF?

RDF is about sharing data—essentially sharing descriptions of things. And those descriptions are expressed as graphs.

Two of the most notable features of the RDF graph model seen from a graph perspective are global identity and inference. The first greatly simplifies data integration and the second helps in building out knowledge graphs.

Let's talk about each of these.

1. https://lod-cloud.net/

Integrating Data at Scale

RDF uses web names, or URIs, for identifying the nodes and edges in its data model. What does that mean? It means everything.

The URI has been a key development in the ongoing integration of the global telecoms network. It builds on the DNS system for naming computers as nodes on the (inter)network. By extending it with a network protocol scheme and a local address on the host system we can identify (and retrieve) documents using the web. The same URI pattern that we use for documents can also be used to identify data points within an RDF graph. The thinking here is that descriptions of things (that is, documents) can be sent in place of the actual things themselves, which may not be so easy to transmit without Star Trek transporters to beam them down. So, in principle, information about any resource (be it physical or abstract) can be returned. We can build out a global information network.

There's another benefit to using URIs—namespacing. With namespacing based on DNS names, we get naming authorities, branding, and trust, and with the DNS name as the namespace root, we get guarantees of uniqueness. It follows that we effectively have a commons for developing a shared semantics with types and properties all globally namespaced. There is now no confusion as to where names come from.

OK, so far, we've talked about sharing data and the semantics for that data, but we haven't talked about data integration. Let's look at the way data integration happens. If user *A* makes statements about a subject *S*, and user *B* also makes statements about a subject *S*, then those sets of statements can be simply added together because the subject *S* is the same in both cases. And we know the subject *S* is the same because we are using a global name. What we effectively have with RDF are self-joining datasets based on the use of URIs, or global names.

Extracting Knowledge from Graphs

Graphs are an excellent choice for representing knowledge bases as they allow easy and arbitrary connections to be set up between the data items. This naturally leads to the notion of knowledge graphs.

But knowledge graphs are more than fixed data stores. They generally follow an open-world model that allows new data to be added as required, and the shape of the data isn't constrained as is the case in a relational database.

• Click HERE to purchase this book now. discuss

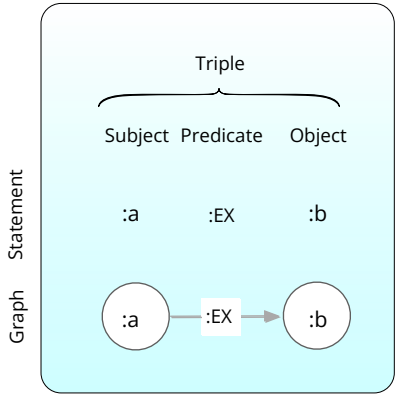
In a sense, they are programmable knowledge stores. New data can be added from the outside, new data can be generated from the inside, and new interpretations over the data can be made. They are more akin to knowledge machines.

RDF builds on common standards for naming, which allows for different datasets to be readily mixed together. Formal reasoning systems from the knowledge representation communities have been layered on top of the basic RDF model. RDF datasets can then be modeled according to RDF schemas (or "ontologies" as they are sometimes called) which are also expressed in RDF. These RDF schemas are built on a formal semantics and a system of logic. This means we can reason over the data, deduce logical inferences, and extract new facts, or statements, which can be added to the dataset. We can thus "grow" the dataset.

At this point, we should probably take a quick look at the RDF data model before we get some real experience with generating RDF from Elixir.

RDF Model

An RDF description is built up of a set of RDF statements where each RDF statement is comprised of a subject, a predicate (or property), and an object—or, as it is called, an RDF triple. Each RDF triple encodes an RDF statement. These RDF statements (or triples) are modeled in graph terms as a node, an edge, and a node as shown in this figure:



So, subjects link to objects via predicates (or properties). In our simple graph, subject `:a` links to object `:b` via the predicate (or property) `:EX`.

The terms here are all shown in the default namespace (indicated by the empty prefix before the : separator), but usually, each term will be taken from its own namespace. Suppose we have an RDF statement such as the following:

```
ns1:a ns2:EX ns3:b .
```

Here the namespace prefixes ns1:, ns2:, and ns3: are all paired off with URI namespaces. Writing that RDF statement out in its full form would give us this:

```
<http://ns1.com/a> <http://ns2.com/EX> <http://ns3.com/b> .
```

Here the <, > brackets mark out a URI.

This is the RDF triple. And an RDF dataset is a set of RDF triples. In fact, this RDF triple expresses a graph edge between two graph nodes.

Now this RDF triple shows the linkages between things. To add descriptions to these things, we add strings like this:

```
ns1:a ns2:EX_NAME "Example name" .
```

Or, in long form like this:

```
<http://ns1.com/a> <http://ns2.com/EX_NAME> "Example name" .
```

These descriptions are encoded as RDF triples, but with the object now as a string. This is how we add subject attributes in RDF.

Well, right about now, you might want to start coding. Let's crack on with that.

Creating the RDFGraph Project

To get some experience working with RDF graphs from Elixir, we'll set up an RDFGraph project under our umbrella application.

We're also going to need an RDF graph database for local experiments. We'll use the free version of Ontotext GraphDB.[2]

RDFGraph Project/Database Setup

See Appendix 1, Project Setups, on page ?, for details on retrieving a working project with code and data.

And see Appendix 2, Database Setups, on page ?, for help on setting up a local copy of GraphDB.

2. https://www.ontotext.com/products/graphdb/

We typically connect to RDF graph databases over the web using SPARQL endpoints, which is what we'll do here. SPARQL is the query language for RDF graphs, and we'll have more to say about this later.

We could use some help to make building and querying RDF from Elixir easier. When I first looked around for any hint of an RDF library in Elixir, I was excited to find the rdf[3] package from Marcel Otto,[4] which has exceptional support for working with RDF. He's also published the sparql[5] and sparql_client[6] packages for querying RDF, as well as the json_ld[7] package for serializing RDF. Check out the RDF on Elixir[8] page for more info.

Without further ado, let's create a new project RDFGraph. Go to the ExGraphsBook home project (see ExGraphsBook Umbrella, on page ?), cd down into the apps directory, and open up the new RDFGraph project:

```
$ mix new rdf_graph --module RDFGraph
```

Note that this time we use the --module option to override the default naming of the module.

We now have an apps directory that looks like this:

```
.
├── apps
│   ├── graph_commons
│   ├── native_graph
│   ├── property_graph
➤  │   └── rdf_graph
```

Let's cd into the rdf_graph directory:

```
.
├── lib
│   └── rdf_graph.ex
├── mix.exs
└── test
    └── ...
```

We'll declare a dependency on the sparql_client package (from the SPARQL.Client project) in the mix.exs file. (This will bring in the rdf and sparql package modules

3. https://hex.pm/packages/rdf
4. http://marcelotto.net/
5. https://hex.pm/packages/sparql
6. https://hex.pm/packages/sparql_client
7. https://hex.pm/packages/json_ld
8. https://rdf-elixir.dev/

from the RDF.ex and SPARQL.ex projects too.) We'll also use the hackney HTTP client in Erlang as recommended:

apps/rdf_graph/mix.exs

```
defp deps do
  [
    # graph_commons
    {:graph_commons, in_umbrella: true},

    # rdf graphs
    {:sparql_client, "~> 0.4"},
    {:hackney, "~> 1.17"}
  ]
end
```

As usual, use Mix to add in the dependency:

```
$ mix deps.get; mix deps.compile
```

We also need to add in the HTTP client:

```
config :tesla, :adapter, Tesla.Adapter.Hackney
```

Add these lines to the umbrella config.exs file in the main project directory or to an environment-specific import (for example, dev.exs).

Finally, let's wire our graph storage into the RDFGraph module with these use/2 macros:

apps/rdf_graph/lib/rdf_graph.ex

```
use GraphCommons.Graph, graph_type: :rdf, graph_module: __MODULE__
use GraphCommons.Query, query_type: :rdf, query_module: __MODULE__
```

Well, that's our setup. Our plan here is to spend some time first in this chapter looking at building RDF models, which may not be so familiar, and then to get into querying RDF graphs with SPARQL in Chapter 8, Querying RDF with SPARQL, on page ?.